# Usage-based Testing for Event-driven Software

Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von

Steffen Herbold
aus Bad Karlshafen

Göttingen
im Juni 2012

Referent: Prof. Dr. Jens Grabowski,
Georg-August-Universität Göttingen.

Korreferent: Prof. Dr. Stephan Waack,
Georg-August-Universität Göttingen.

Korreferent: Prof. Atif Memon, Ph.D.
University of Maryland, MD, USA

Tag der mündlichen Prüfung: 27. Juni 2012

**Abstract**

Most modern-day end-user software is Event-driven Software (EDS), i.e., accessible through Graphical User Interfaces (GUIs), smartphone apps, or in form of Web applications. Examples for events are mouse clicks in GUI applications, touching the screen of a smartphone, and clicking on links in Web applications. Due to the high pervasion of EDS, the quality assurance of EDS is vital to ensure high-quality software products for end-users. In this thesis, we explore a usage-based approach for the testing of EDS. The advantage of a usage-based testing strategy is that the testing is focused on frequently used parts of the software, while seldom used parts are only tested sparsely. This way, the user-experienced quality of the software is optimized and the testing effort is reduced in comparison to traditional software testing.

The goal of this thesis is twofold. On the one hand, we advance the state-of-the-art of usage-based testing. We define novel test coverage criteria that evaluate the testing effort with respect to usage. Furthermore, we propose three novel approaches for the usage-based test case generation. Two of the approaches follow the traditional way in usage-based testing and generate test cases randomly based on the probabilities of how the software is used. With our third test case generation approach, we take a different direction and define a heuristic that generates a test suite to optimize our usage-based coverage criteria. We evaluate our contributions to usage-based testing in two large-scale case studies that show the value of our work.

On the other hand, we provide a platform-independent framework for usage-based testing that allows the application of usage-based testing to different EDS platforms. Examples for EDS platforms are concrete GUI frameworks like Qt or concrete Web application platforms, e.g., PHP-based websites running on an Apache Web server. In the past, research effort for usage-based testing has always focused on one specific EDS platform and its results were either transferred to other EDS platforms through re-implementation or not at all. Through our framework, we provide a remedy for this issue and allow an easy transfer of results between different EDS platforms. Our approach is the application of the usage-based testing techniques to an abstract notion of events and the translation between platform-specific usage events and the abstract events to gain platform independence. We provide a proof-of-concept implementation to show the feasibility of our framework in practice.

**Zusammenfassung**

Der Großteil von moderner Endnutzer Software ist eventgetrieben, das heißt durch graphische Benutzeroberflächen (GUIs), Smartphone Apps oder Webanwendungen. Beispiele für Events sind Mausklicks in GUI Applikationen, das Berühren des Touchscreens eines Smartphones und Klicks auf Links in Webanwendungen. Aufgrund der hohen Durchdringung von eventgetriebener Software ist die Qualitätssicherung dieser Art von Software entscheidend um qualitativ hochwertige Endnutzersoftware zu erstellen. In dieser Arbeit betrachten wir eine benutzungsbasierte Strategie für das Testen von eventgetriebener Software. Der Vorteil von benutzungsbasierten Strategien ist, dass der Testfokus auf häufig benutzten Teilen der Software liegt, während selten benutzte Teile der Software nur sporadisch getestet werden. Hierdurch wird die durch den Benutzer wahrgenommene Softwarequalität optimiert und der Testaufwand ist reduziert im Vergleich zum intensiven Testen der kompletten Software.

Das Ziel dieser Arbeit ist zweigeteilt. Auf der einen Seite wollen wir den State-of-the-Art des benutzungsbasierten Testen vorantreiben. Wir definieren neue Überdeckungskriterien für die Evaluierung des Testaufwands in Bezug auf die Benutzung. Weiterhin schlagen wir drei neue benutzungsbasierte Testfallgenerierungsansätze vor. Zwei dieser Ansätze verfolgen die klassische Art und Weise der benutzungsbasierten Testfallerstellung. Das bedeutet das Testfälle randomisiert aufgrund der Benutzungswahrscheinlichkeiten abgeleitet werden. Unser dritter Ansatz verfolgt eine andere Richtung. Wir definieren eine Heuristik, die eine Testsuite ermittelt, welche die unsere benutzungsorientierten Überdeckungskritierien optimiert. Wir evaluieren diese Erweiterungen des benutzungsbasierten Testens in zwei großen Fallstudien und demonstrieren dadurch den Wert unserer Arbeit.

Auf der anderen Seite stellen wir ein plattformunabhängiges Framework zum benutzungsbasierten Testen, welches die Applikation von benutzungsorientierten Techniken auf verschiedenen eventgetriebenen Softwareplattformen erlaubt. Beispiele für eventgetriebene Softwareplattformen sind Frameworks für GUIs, wie zum Beispiel Qt oder Webanwendungsplattformen, wie zum Beispiel PHP basierte Webseiten, welche auf einem Apache Webserver laufen. In der Vergangenheit war die Forschung an benutzungsorientierten Testmethoden auf einzelne eventgetriebene Softwareplattformen beschränkt. Die Resultate mussten entweder von Hand durch Neuimplementierungen auf andere Plattformen portiert werden oder die Ergebnisse wurden bisher nicht übertragen. Unser Framework stellt eine Lösung für dieses Problem dar und erlaubt einen einfachen Transfer der Ergebnisse auf verschiedene eventgetriebene Softwareplattformen. Der Ansatz, den wir verfolgen, basiert auf der Idee, die benutzungsbasierten Testtechniken auf abstrakte Events anzuwenden und zwischen abstrakten Events und plattformspezifischen Events zu übersetzen. Um die Umsetzbarkeit unseres Frameworks zu demonstrieren, haben wir eine Proof-of-Concept Implementierung des Frameworks entwickelt.

**Acknowledgements**

During the course of my PhD studies, I was influenced by many people in various ways. I would like to use this opportunity to acknowledge and thank all. People are listed in no particular order.

First, I want to thank my advisors, Prof. Dr. Jens Grabowski and Prof. Dr. Stephan Waack. Without their comments and suggestions, this thesis would not have been possible. I also want to thank Prof. Atif Memon for the opportunity to visit the University of Maryland and his comments on my work.

My colleagues in Göttingen always provided an enjoyable and scientiffically inspiring environment: Lennart Obermann, Mehmet Gültas, Thomas Rings, Gunnar Krull, Benjamin Zeiss, Patrick Harms, Roman Asper, Philip Makedonski, Heike Jachinke, and Annette Kadziora. I am especially grateful for the effort that Thomas Rings put into the proof reading of this thesis. I would also like to thank the remaining members of my defense committee: Prof. Dr. Carsten Damm, JProf. Dr. Konrad Rieck, and Prof. Dr. Anita Schöbel.

I would like to thank all my friends and family, whose constant support and never ending inquiries as to the progress of my thesis helped me finishing it.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Listings

# Acronyms

**AEP** Asymptotic Equipartition Property

**API** Application Programming Interface

**AWT** Abstract Window Toolkit

**CIS** Complete Interaction Sequence

**CSV** Comma Separated Value

**DFA** Deterministic Finite Automaton

**DLL** Dynamic Link Library

**EIG** Event Interaction Graph

**ESG** Event Sequence Graph

**EDS** Event-driven Software

**EFG** Event-flow Graph

**ESIG** Event-Semantic Interaction Graph

**FA** Factor Analysis

**FSM** Finite State Machine

**GUI** Graphical User Interface

**HTML** Hypertext Markup Language

**HTTP** Hypertext Transfer Protocol

**ISTQB** International Software Testing Qualifications Board

**JFC** Java Foundation Classes

**JVM** Java Virtual Machine

**MM** Markov Model

**MFC** Microsoft Foundation Classes

**MSC** Message Sequence Charts

**PCA** Principle Component Analysis

**PoO** Point of Observation

**PID** Process ID

**PPM** Prediction by Partial Match

**SUT** System Under Test

**SaaS** Software as a Service

**TTCN-3** Testing and Test Control Notation 3

**UML** Unified Modeling Language

**URI** Uniform Resource Identifier

**W3C** World Wide Web Consortium

**VFSM** Variable Finite State Machine

**WWW** World Wide Web

**WSDL** Web Service Description Language

**XML** eXtensible Markup Language

# 1. Introduction

Software systems play a large role in our everyday lives and their quality directly impacts many aspects of our lives, e.g., communication at work with colleagues, with friends, and relatives through e-mail and social networks. Therefore, users demand high quality software products that perform their duties without fail, the development of which requires effective software quality assurance measures. In our research, we focus on *data centric quality assurance*, i.e., we collect empirical data about the aspect of a software product we want to consider and base the quality assurance on the collected data. One such approach is the *usage-based testing*.

Usage-based testing is an approach to focus the quality assurance effort. The assumptions of usage-based testing are that not all features of a software are equally important for the users and that users use software in a specific way and not randomly. This leads to recurring patterns that describe the usage of the software. Based on these patterns, the testing effort is focused such that highly used parts and often occurring patterns are tested intensively, while the seldom used parts are only tested sparsely. The means used to describe the usage patterns are *usage profiles*, i.e., models of the software that incorporate information about the usage.

The target our our investigations is event driven end-user software, i.e., software that reacts to interactions with the user, e.g., mouse clicks, keyboard input, clicking on a link in a browser, or touching the screen of a smart phone. We refer to these kinds of interactions to drive a software as *events*, which coins the term Event-driven Software (EDS). Therefore, the quality assurance for end-user software needs to be assessed by of testing EDS. The large issue of software quality assurance is that exhaustive testing is impossible, due to the infeasibly large amount of possible input values. Therefore, software testers have to decide on a test strategy for the selection of input values, in our case event sequences, e.g., a series of mouse clicks to be performed. Furthermore, the available time and budget restrain the possible amount of software testing during the software development and maintenance.

We utilize usage profiles we infer about the software as means for focusing the testing efforts. We use the usage profiles to derive test cases and analyze the completeness of a test suite by considering the event combinations covered with respect to the usage profile. Our investigations include multiple important EDS platforms, i.e., Windows and Java Graphical User Interface (GUI) applications and PHP-based Web applications.

## 1.1. Scope

For the definition of the scope of this thesis, we consider three main aspects of usage-based testing separately: placement in the software life-cycle; methods for usage profile inference; and testing techniques.

**Software Life-Cycle:** Usage-based testing is primarily applied during regression testing, i.e., the re-execution of tests for new software versions. During regression testing, there are often significant time and resource constraints that limit the possible testing effort. Usage-based testing provides the means to limit the effort by focusing on frequently used parts of the software. It is possible to use usage-based testing during earlier development phases, but such approaches built on assumptions about the systems usage and are, therefore, possibly inaccurate. Furthermore, automated interference of usage profiles is not possible in earlier phases of the life-cycle due to the lack of usage data. The techniques presented in this thesis are based on usage data. Hence, our solutions are for regression testing and we do not consider usage-based testing in other phases of the software life-cycle.

**Usage Profile Inference:** There are different data sources from which usage profiles can be inferred: knowledge about a system's users, measured usage data, and available models of a system. Usage profiles can be inferred automatically or defined by an expert. The resulting usage profiles can be qualitative or quantitative. Qualitative usage profiles describe the usage in terms like "often", "sometimes", and "seldom". Quantitative usage profiles are in the literature and throughout the remainder of this work referred to as probabilistic usage profiles, because they describe the usage in terms of probabilities. We consider automated inference of probabilistic usage profiles from measured usage data. We do not consider how existing models of a software can be extended with usage information, expert-driven definition of usage profiles, and sources other than measured usage data for the model inference.

**Testing Techniques:** Usage-based methods can be applied for a variety of tasks, e.g., as part of the project risk analysis to determine how likely error-prone features are to be executed, and similarly estimate the reliability of a software based on usage patterns and known failures. In usage-based testing, the usage information is employed to either generate new test cases from a usage profile or prioritize existing test cases based on the usage. We focus on test case generation and usage-based test suite evaluation.

## 1.2. Goals and Contributions

Within the scope of this thesis, we work on answering the following research questions.

- Which stochastic processes are well-suited for usage profiles?
- What are good strategies to derive test suites from usage profiles?
- How can test suites be evaluated with respect to usage?

Based on the research questions, the goal of this thesis is two-fold. On the one hand, we advance the state-of-the-art of usage-based EDS testing in order to provide answers for the underlying research questions. We define and evaluate stochastic processes with the Prediction by Partial Match (PPM) property for usage-based testing and evaluate their capabilities in comparison to Markov Models (MMs). Additionally, we define test coverage criteria that analyze a test suite with respect to its usage and usage-based test case generation mechanisms. On the other hand, we provide the means for usage-based EDS testing independent of concrete EDS platforms. To this aim, we define a framework for usage-based testing of EDS in general, independent of the platform. Other researchers and software test practitioners can adapt our framework and, thereby, broaden the scope of usage-based testing for EDS and evaluate our findings in a variety of settings to improve their validity. In order to instantiate our framework for different platforms, we develop an approach for capture/replay GUI testing that is not solely based upon observing user interactions, but also the internal communication of an application. The concrete contributions of this thesis are the following.

1. A versatile and platform-independent framework for usage-based testing of EDS (Chapter 4).

2. The definition of usage-based coverage criteria for EDS (Section 3.3) and a detailed comparison with non-usage based coverage criteria for EDS (Section 6.2).

3. A method for randomized usage-based test case generation (Section 3.4.2).

4. A heuristic for optimized test case generation with respect to usage-based coverage criteria (Section 3.4.3).

5. A PPM model as a usage profile designed specifically for the purpose of usage-based testing (Section 3.1.3).

6. A capture/replay approach for Windows Microsoft Foundation Classes (MFC) GUIs that provides an internal deployable lightweight usage monitor. The approach combines the observation and usage of internal application communication and external input information, e.g., mouse clicks, to achieve coordinate-independent replays (Section 5.2).

7. Lightweight deployable usage monitors for Java Foundation Classes (JFC) GUI applications (Section 5.3) and PHP-based Web applications (Section 5.4).

We evaluate all of the contributions in case studies or through proof-of-concept implementations.

## 1.3. Impact

The results of this dissertation and further data centric research that has been performed as offspring of this work have been published in one scientific journal article, four peer-reviewed international conference and workshop proceedings, and one book chapter.

### Journal Article

- Empirical Software Engineering, Vol. 16(6): *Calculation and optimization of thresholds for sets of software metrics*, Steffen Herbold, Jens Grabowski, Stephan Waack, Springer, 2011.

### Conferences and Workshops

- DFF 2010: *Retrospective Analysis of Software Projects using k-Means Clustering*, Steffen Herbold, Jens Grabowski, Helmut Neukirchen, Stephan Waack
- VALID 2011: *Retrospective Project Analysis Using the Expectation-Maximization Clustering Algorithm*, Steffen Herbold, Jens Grabowski, Stephan Waack
- TESTBEDS 2011: *Improved Bug Reporting and Reproduction trough Non-intrusive GUI Usage Monitoring and Automated Replaying*, Steffen Herbold, Uwe Bünting, Jens Grabowski, Stephan Waack.
- MVV 2011: *A Model for Usage-based Testing of Event-driven Software*, Steffen Herbold, Jens Grabowski, Stephan Waack.

### Book Chapter

- Advances in Computers, Vol. 85: *Deployable Capture/Replay Supported by Internal Messages*, Steffen Herbold, Uwe Bünting, Jens Grabowski, Stephan Waack, Elsevier, 2012.

Furthermore, the author identified the topics for and supervised the following two Bachelor's theses.

- Kathrin Becker: *Detection and Analysis of Dependencies Between TTCN-3 Modules*, Bachelor Thesis, 2010
- Jeffrey Hall: *Erweiterung des EventBench Projektes um Test-Assertions mit Fokus auf GUI Testen*, Bachelor Thesis, 2011

## 1.4. Structure of the Thesis

The remainder of this thesis is structured as follows. In Chapter 2, we present the foundations on which this work is built. First, we present the mathematical foundations of stochas-

tic processes and information theory, upon which we build our software testing approach. Then, we discuss the principles of EDS and the concrete EDS platforms we consider in this thesis. Afterwards, we present the foundations from software testing, discuss how EDS is tested and introduce the foundations of usage-based testing. In Chapter 3 we present our contributions to the usage-based testing. In Chapter 4, we define our framework for usage-based EDS testing and describe the structure and components of the framework in detail. Afterwards, we present our instantiation and implementation of the framework in Chapter 5. As part of the instantiation, we discuss a novel capture/replay approach for Windows MFC applications. Based on our instantiation, we performed three case studies, which we present and discuss in Chapter 6. We define and evaluate research hypotheses for each case study as means for the analysis of our approach. Then, we discuss our work in Chapter 7. We put our work in context of related work and discuss the strengths and limitations of our approach, as well as threats to the validity of our work. Finally, we conclude this thesis in Chapter 8.

# 2. Foundations

In this chapter, we describe the foundations on which this work is build. The foundations are split into four parts. First, we introduce the concept of time-discrete stochastic processes. Afterwards, we give a brief introduction into the field of information theory, with the focus on entropy. In the third part, we switch the focus from the mathematical foundations to software engineering and explain the concept of event-driven software. In the fourth part, we introduce the concepts from the field of software testing.

## 2.1. Probabilities and Stochastic Processes

The concept of stochastic processes plays an important role in many types of research, e.g., to describe the brownian motion of particles [29], modeling the behavior of stocks in financial markets [66], and the analysis of genes to detect functional important parts [120]. In this work, we use stochastic processes to model the behavior of software users. In this section, we first give a brief introduction into probabilities. Second, we introduce the notations that are required to work with stochastic process and the types of stochastic process we apply in our work. The introduction is based on [23, 33, 92, 117].

### 2.1.1. Notations and Properties

The first example used to describe probabilities is often the coin flip, where a coin is tossed and the result will be heads or tails. This is an example of an *experiment* with an uncertain outcome that can only be described in terms of probabilities. The following two definitions formalize this concept.

**Definition 2.1 (Sample Space)** *The* sample space *$S$ is the space of all possible outcomes of an experiment.*

**Definition 2.2 (Event (Stochastic))** *An* event *$A$ is a subset of the sample space, i.e., a collection of possible outcomes of an experiment.* $\mathbf{Pr}\{A\}$ *is the probability that the outcome of the experiment is included in event $A$.*

In the coin flip example, the sample space is $S = \{heads, tails\}$. We can split this sample space into two events $A = \{heads\}$ and $B = \{tails\}$. In case of a fair coin, the probabilities of the events are $\mathbf{Pr}\{A\} = \mathbf{Pr}\{B\} = 0.5$.

When it comes to practical applications, e.g., computer simulations of experiments, usually only numbers can be generated, e.g., a uniformly distributed real value in the interval $[0,1]$. For the simulation of a fair coin flip with a random number generator, the concept of random variables is a powerful tool.

**Definition 2.3 (Random Variable)** *A* random variable $X$ *is a function* $X : S \to \mathscr{X}$ *that assigns a value of the* alphabet $\mathscr{X}$ *to each sample* $s \in S$.

**Definition 2.4 (Induced Events)** *Random variables implicitly induce events on S. Each value* $x \in \mathscr{X}$ *induces an event*

$$A_x = \{s \in S : X(s) = x\}. \tag{2.1.1}$$

To show how random variables can be applied, we describe the simulation of a fair coin flip by drawing a random number uniformly from the interval $[0,1]$. We define

$$X(s) = \begin{cases} heads & s \leq 0.5 \\ tails & s > 0.5. \end{cases} \tag{2.1.2}$$

This induces two events $A_{heads}$ and $A_{tails}$, with probabilities $\mathbf{Pr}\{A_{heads}\} = \mathbf{Pr}\{A_{tails}\} = 0.5$.

**Lemma 2.1** *1)* $A_x \cap A_y = \emptyset$ *if* $x \neq y$ *and 2)* $\bigcup_{x \in \mathscr{X}} A_x = S$.

**Proof** *1) Let* $x, y \in \mathscr{X}$, $x \neq y$. *Then* $\forall s \in A_x : X(s) = x \neq y$ *and* $\forall s \in A_y : X(s) = y \neq x$. *Thus* $A_x \cap A_y = \emptyset$.
*2) Using the definition of* $A_x$, *we get*

$$\bigcup_{x \in \mathscr{X}} A_x = \bigcup_{x \in \mathscr{X}} \{s \in S : X(s) = x\}.$$

$\bigcup_{x \in \mathscr{X}} \{s \in S : X(s) = x\}$ *is a way to define the domain of X, which is S.* □

As Lemma 2.1 shows, the events $A_x$ induced by a random variable partition the sample space. This partitioning is called *event space*. In the remainder of this work, we do not consider the sample space and events directly. Instead, we work with random variables and the events space they induce. For convenience, we use the following notations when working with random variables.

**Notation 2.1** *Let* $X, Y$ *be random variables.*

- *We use the upper case* $X, Y$ *when speaking of the random variables and the lower case* $x, y$ *if we mean a concrete value.*
- *We use the notation* $\{X = x\}$ *instead of* $A_x$, *i.e., we write* $\mathbf{Pr}\{X = x\}$ *instead of* $\mathbf{Pr}\{A_x\}$. *For simplicity, we also refer to the concrete values* $x, y$ *as events.*

- *If we do not define otherwise, $\mathscr{X}$ denotes the alphabet of $X$, $\mathscr{Y}$ denotes the alphabet of $Y$.*
- *We write $X \sim p(x)$, $Y \sim p(y)$, meaning that $X$ and $Y$ are random variables with probability mass function $p(x) = \mathbf{Pr}\{X = x\}, x \in \mathscr{X}$ and $p(y) = \mathbf{Pr}\{Y = y\}, y \in \mathscr{Y}$, respectively.*
- *We use the terms probability mass function and probability distribution interchangeably.*
- *We write $(X,Y) \sim p(x,y)$ to denote the* joint distribution *of the two variables, with the joint probability mass function $p(x,y) = \mathbf{Pr}\{X = x, Y = y\}, x \in \mathscr{X}, y \in \mathscr{Y}$.*
- *We write $\mathbf{Pr}\{Y = y|X = x\}$ to denote the* conditional probability *that the value of $Y$ is $y$ given that the value of $X$ is $x$.*

We use the notion of random variables to define stochastic processes, which is one of the main principles on which our work is based. In general, stochastic processes are an index sequences of random variables. In this thesis, we work only with so called *time-discrete* stochastic processes. Therefore, we omit general definitions of stochastic processes, that also allow for continuous time and directly define the type we need. For a general and detailed introduction, the reader is referred to the literature, e.g., [92, 117].

**Definition 2.5 (Time-discrete Stochastic Process)** *A* time-discrete stochastic process *is an indexed sequence of random variables $X_1, X_2, \ldots$ and is characterized by the joint probability mass function $p(x_1, x_2, \ldots, x_n) = \mathbf{Pr}\{(X_1, X_2, \ldots, X_n) = (x_1, x_2, \ldots, x_n)\}$ with $(x_1, x_2, \ldots, x_n) \in \mathscr{X}^n$ for $n \in \mathbb{N}$.*

**Notation 2.2** *Let $X_1, X_2, \ldots$ a time discrete stochastic process.*

- *When we refer to concrete values of random variables of a stochastic process, we refer to them as the* state *of the process.*
- *We refer to $X_n$ as the* current state *of the process, and $X_{n-1}, \ldots, X_1$ as the past or history of the process.*
- *We also denote a stochastic process as $\{X_i\}$.*

A simple example of a time-discrete stochastic process is a series of coin flips. We then have random variables $X_i$ over the alphabet $\mathscr{X} = \{heads, tails\}$ with $\mathbf{Pr}\{X_i = heads\} = \mathbf{Pr}\{X_i = tails\} = 0.5$ for all $i \in \mathbb{N}$. A concrete simulation results in a sequence of heads and tails, e.g., "heads, heads, tails, heads, tails, heads, tails, ...". This example also exhibits one important property of stochastic processes. The probabilities are not influenced by the index, i.e., the time of the coin flip does not influence the outcome. This time invariance is formalized by the following definition.

**Definition 2.6 (Stationary Process)** *A stochastic process* $\{X_i\}$ *is called* stationary *if its joint distribution is invariant to timeshifts, i.e.,*

$$\mathbf{Pr}\{X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n\} = \mathbf{Pr}\{X_{1+l} = x_1, X_{2+l} = x_2, \ldots, X_{n+l} = x_n\} \quad (2.1.3)$$

*for all* $l \in \mathbb{Z}$.

**Remark 2.1** *Stationary processes have a* stationary distribution $\mu$, *such that* $\mu(x) = \lim_{t\to\infty} p(X_t = x)$ *for all* $x \in \mathscr{X}$.

To this point, we have considered stochastic processes without any restrictions on if and how the random variables are related over time, e.g., whether $X_n$ depends on $X_{n-1}$ or even $X_{n-13}$. This leads to complexity problems, because all of the history has to be known during simulations and can influence the probability of the next state. This leads to an exponential growth of complexity with $n$. Consider the following example, with $m = |\mathscr{X}|$. Then there are $m^n$ different configurations $(x_1, \ldots, x_n) \in \mathscr{X}^n$. Thus, to be able to calculate $\mathbf{Pr}\{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_1 = x_1\}$ for $x_{n+1} \in \mathscr{X}$, one either needs an algorithmic way to calculate this probability or store the probability for all $m^n$ combinations. Since this information is required for all possible $x_{n+1} \in \mathscr{X}$, we need access to $m^{n+1}$ values total. To deal with this exponential growth, when all of the history of the process is taken into account, we restrict the reliance on the past.

### 2.1.2. First-order Markov Models

The first type of stochastic process only has access to the immediate past, i.e., the next state of the process only depends on the current state $X_n$ of the process. This concept is formalized by definitions 2.7 and 2.8.

**Definition 2.7 (Markov Property)** *A discrete stochastic process* $X_1, X_2, \ldots$ *is said to possess the* Markov property *if*

$$\mathbf{Pr}\{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_1 = x_1\} = \mathbf{Pr}\{X_{n+1} = x_{n+1} | X_n = x_n\} \quad (2.1.4)$$

*for all* $n \in \mathbb{N}$ *and all possible* $x_1, \ldots, x_{n+1} \in \mathscr{X}$.

**Definition 2.8 (First-order Markov Model)** *A discrete stochastic process that posses the Markov property is a* First-order MM, *also known as* Markov chain.

In non-mathematical terms, the Markov property is a statement, that the past does not influence the future, only the current state does. Thus, first-order MMs are also called *memoryless*. From a technical point of view, this means that only one state, i.e., the current one needs to be known to infer the probability distribution of the next state. Therefore, the complexity is independent of $n$ and we only need access to $m^2$ values.

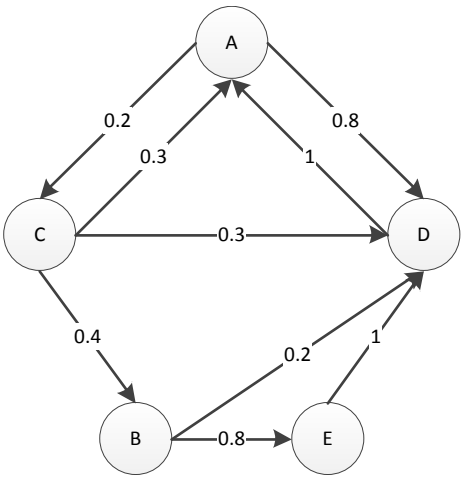



Figure 2.1.: A first-order MM over the alphabet $\mathscr{X} = \{A, B, C, D, E\}$.

It is possible to visualizes first-order MM in a convenient way as a directed graph. The nodes of the graph are the symbols of the alphabet $\mathscr{X}$. The edges starting in symbol $x \in \mathscr{X}$ and ending in symbol $x' \in \mathscr{X}$ are labelled with the probability $\mathbf{Pr}\{X_{n+1} = x' | X_n = x\}$. In case the probability is zero, the edge is omitted. In Figure 2.1, we depict a first-order MM with alphabet $\mathscr{X} = \{A, B, C, D, E\}$.

A useful tool to store the probabilities and also use them for mathematical analysis is to create a $|\mathscr{X}| \times |\mathscr{X}|$ matrix of the transition probabilities.

**Definition 2.9 (Transition Matrix)** *Let $\{X_i\}$ a first-order MM. Let the alphabet be enumerated, i.e., let $\mathscr{X} = \{x^1, \ldots, x^m\}$. The* transition matrix $P \in \mathbb{R}^m$ *is defined as*

$$P_{ij} = \mathbf{Pr}\{X_{n+1} = x^j | X_n = x^i\} \tag{2.1.5}$$

*for $i, j = 1, \ldots, m$.*

In less formal terms, each row $i$ contains the probability distribution of the next state, given that the current state is $x^i$.

For some applications, we work with first-order MMs that are stationary and calculate their stationary distribution. For this, the models need to fulfill additional properties. We define these properties in definitions 2.10 to 2.14. For all these definitions, we assume $\{X_i\}$ to be a first-order MM with alphabet $\mathscr{X} = \{x^1, \ldots, x^m\}$. Build on these foundations, we state the required properties, as well as the method for computation of the stationary distribution in Theorem 2.1.

**Definition 2.10 (k-step Transition Probability)** *The probability that the process is in state*

*$x^j$ after k steps, given that the current state of the process is $x^i$ defined as*

$$P_{ij}^{(k)} = \mathbf{Pr}\{X_{n+k} = x^j | X_n = x^i\} \tag{2.1.6}$$

*for $i, j = 1, \ldots, m$ and $k \in \mathbb{N}$.*

**Definition 2.11 (Recurrent)** *A state $x^i$ is* recurrent*, if*

$$\sum_{n=0}^{\infty} P_{ii}^{(n)} = \infty. \tag{2.1.7}$$

**Definition 2.12 (Period)** *The* period *$d_i$ of a state $x^i$ is the greatest common divisor of all $n \in \mathbb{N} : P_{ii}^{(n)} > 0$.*

**Definition 2.13 (Aperiodic)** *A recurrent state $x^i$ is* aperiodic *if its period is $d_i = 1$ and* periodic *if $d_i > 1$. If all states of the model are aperiodic, we say that the model is aperiodic.*

**Definition 2.14 (Irreducible)** *A first-order MM $\{X_i\}$ is* irreducible *if every symbol $x \in \mathscr{X}$ can be reached from every other symbol $x' \in \mathscr{X}$ in a finite number of steps.*

**Theorem 2.1** *Let $\{X_i\}$ be an irreducible, aperiodic first-order MM with all states recurrent and $\mathscr{X} = \{x^1, \ldots, x^m\}$. Then the limit*

$$\mu_j = \lim_{n \to \infty} P_{ij}^n \tag{2.1.8}$$

*exists and $\mu_j$ is the unique non-negative solution of*

$$\mu_j = \sum_{i=1}^{m} \mu_i P_{ij} \tag{2.1.9}$$

$$\sum_{j=1}^{m} \mu_j = 1. \tag{2.1.10}$$

**Proof** *See [33], pp. 393-394.*

Through Theorem 2.1 we have clear requirements on first-order MMs. If the MM should be stationary, we have to show that the three properties (irreducible, aperiodic, recurrent) are fulfilled. The following lemma simplifies this task.

**Lemma 2.2** *Let $\{X_i\}$ be an irreducible first order MM. Let $x \in \mathscr{X}$ be aperiodic. Then all $x' \in \mathscr{X}$ are aperiodic.*

**Proof** *See [33], p. 391.*

Thus, if we have already shown that our model is irreducible, we only need to show that one state is aperiodic and because of Lemma 2.2 follows that all states are aperiodic. Since the period is defined as the greatest common divisor of all path-lengths that can be used to reach a state from itself, it follows that if there is a self-loop, i.e., $P_{ii} > 0$, the period of a state is 1. Hence, it is sufficient to show that $P_{ii} > 0$ for any $i = 1, \ldots, m$.

### 2.1.3. High-order Markov Models

While the memorylessness of the first-order MM is helpful for the analysis of the models and a good way to limit the complexity of the models, it is often not reflected in the reality of the processes that are modelled. Often more than the current state influence the probability of the next state. To cope with longer memories, we generalize definitions 2.7 and 2.8 to introduce *k-th order MMs*.

**Definition 2.15 (*k*-th order Markov property)** *A discrete stochastic process $X_1, X_2, \ldots$ is said to possess the k-*th order Markov property *if*

$$\mathbf{Pr}\{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_1 = x_1\} = \mathbf{Pr}(X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_{n-k+1} = x_{n-k+1}\} \tag{2.1.11}$$

*for all $n \in \mathbb{N}$.*

**Definition 2.16 (*k*-th order Markov Model)** *A discrete stochastic process that posses the k-th order Markov property is a k-*th order MM.

**Notation 2.3** *Let $\{X_i\}$ a k-th order MM.*

- *We write $\mathbf{Pr}_{MM^k}$ to denote that the probability is calculated using a k-th order MM.*
- *We say that k is the Markov order of the model.*

The *k*-th order MMs provide a compromise between the memory length and the complexity. The complexity is bound by $m^{k+1}$ and, thus, still polynomial in $m$. However, it increases exponentially with $k$, thereby indirectly also bounding the possible memory length due to the complexity long memories impose. In Section 2.2, we introduce concepts that are only applicable to first-order MMs. The following theorem allows us, to apply these concepts to *k*-th order MMs as well.

**Theorem 2.2** *Let $\{X_i\}$ a k-th order MM. Then there exists a first-order MM $\{Y_i\}$ that is equivalent to $\{X_i\}$.*

**Proof** *To proof this theorem, we construct $\{Y_i\}$ from $\{X_i\}$. First we construct the alphabet $\mathscr{Y}$ such that it can replace the "memory" of the k-th order MM:*

$$\mathscr{Y} = \overbrace{\mathscr{X} \times \mathscr{X} \times \ldots \times \mathscr{X}}^{k \text{ times}} = \{(x_1, x_2, \ldots, x_k) : x_i \in \mathscr{X}, i = 1, \ldots, k\} \tag{2.1.12}$$


(a) Without split state C (b) With split state C

Figure 2.2.: Three states A, B, C exemplifying how splitting state C can serve as memory of the previous state. Probabilities are omitted.

*We interpret $y = (x_1, \ldots, x_k) \in \mathscr{Y}$ as follows: $x_k$ is the latest drawn symbol, $x_{k-1}$ is the symbol drawn one step before, ..., $x_1$ is the symbol drawn $k-1$ steps before. Thus, the memory of the latest k steps is encoded into y.*

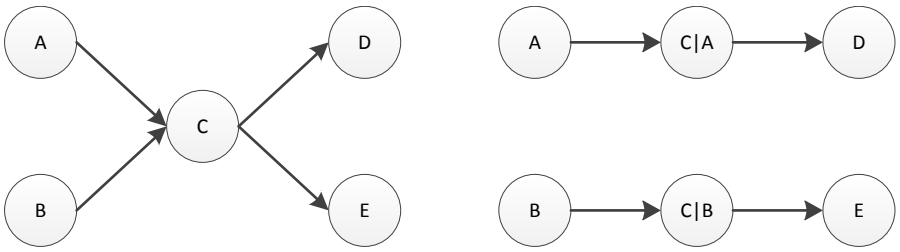
*To ensure that the probabilities of the next symbol mimic the behavior of $\{X_i\}$, we define them as follows:*

$$\begin{aligned}&\mathbf{Pr}\{Y_{n+1} = (x_1, \ldots, x_k) | Y_n = (x'_1, \ldots, x'_k)\} \\ &\quad = \begin{cases} \mathbf{Pr}\{X_{n+1} = x_k | X_n = x_{k-1}, \ldots, X_{n-k+1} = x_1\} & \forall i \in \{1, \ldots, k-1\} : x_i = x'_{i+1} \\ 0 & \exists i \in \{1, \ldots, k-1\} : x_i \neq x'_{i+1} \end{cases}\end{aligned} \tag{2.1.13}$$

*The first case ensures that if we have a pair of elements $y = (x_1, \ldots, x_k), y' = (x'_1, \ldots, x'_k) \in \mathscr{Y}$, where the "memory" of y, i.e., $x_1, \ldots, x_{k-1}$, matches the "current state" as defined by $y'$, i.e., $(x'_2, \ldots, x'_k)$ the next symbol, i.e., $x_k$ is drawn with the same probability it would have in the k-th order MM $\{X_i\}$. The second case sets the probability to zero for all invalid pairs of elements $y, y' \in \mathscr{Y}$, i.e., pairs where the "memory" does not match the current state, thereby ensuring that only valid pairs can be drawn.*

*Furthermore, Equation 2.1.13 describes the probability mass function of $\{Y_i\}$. The probabilities are exactly the same as in $\{X_i\}$: everything that is possible in $\{X_i\}$ is also possible in $\{Y_i\}$ and for everything that is not described by $\{X_i\}$ the probability is zero. Therefore, Equation 2.1.13 describes a valid probability mass function if the probability mass function of $\{X_i\}$ is valid.* □

While the Theorem 2.2 itself only declares the existence of an equivalent first-order MM, the proof shows how such a model can be obtained. The technique we use in the proof is known as *state splitting* and illustrated in Figure 2.2.




(a) Without split state C (b) With split state C

Figure 2.3.: Two small MMs, where state C in the model on the left hand is split to show how the splitting removes randomness. Probabilities are omitted.

### 2.1.4. Prediction by Partial Match

While longer memories are often desirable, as they lead to more precise models and better decisions, they also tend to make the models static, removing a lot of randomness from the decisions. Figure 2.3 exemplifies how a longer memory can remove randomness from the decisions. Consider an example, where the two sequences $A,C,D$ and $B,C,E$ are known when the model is defined. In Figure 2.3a, we show a first-order MM that models this scenario. The model also allows sequences $A,C,E$ and $B,C,D$, i.e., sequences that were not known when defining the model. In Figure 2.3b, we split state $C$ int $C|A$ and $C|B$, thereby effectively making the model a 2nd-order MM. The 2nd-order MM only allows the two initial sequences $A,C,D$ and $B,C,E$. Thus, longer memories have the advantage of modeling the available knowledge more precisely, but prevent the models from generating sequences that have not been known or considered during the creation of the model.

PPM models provide a way to resolve this conflict. They belong to the *variable-order MMs*, i.e., the Markov order is not fixed but can vary. In the PPM approach, the models are in principle normal $k$-th order MM with an additional "opt-out" probability. The opt-out offers the possibility to use a $(k-1)$-th order MM instead, from which an opt-out is also possible. We formalize this concept by the definition of the *PPM* property.

**Definition 2.17 ($k$-th order Prediction by Partial Match Property)** *A stochastic process $\{X_i\}$ is said to possess the k-th order* Prediction by Partial Match (PPM) property *if*

$$\begin{aligned}&\mathbf{Pr}_k\{X_{n+1}=x_{n+1}|X_n=x_n,\ldots,X_{n-k+1}=x_{n-k+1}\}\\&\quad=\begin{cases}\hat{\mathbf{Pr}}_k\{X_{n+1}=x_{n+1}|X_n=x_n,\ldots,X_{n-k+1}=x_{n-k+1}\} & \text{if not opt-out}\\ \hat{\mathbf{Pr}}_k\{escape|X_n=x_n,\ldots,X_{n-k+1}=x_{n-k+1}\}\\ \quad\cdot\mathbf{Pr}_{k-1}\{X_{n+1}=x_{n+1}|X_n=x_n,\ldots,X_{n-k+2}=x_{n-k+2}\} & \text{if opt-out}\end{cases}\end{aligned} \tag{2.1.14}$$

The concept of PPM comes from the field of data compression, where it is important to know probabilities of sequences to assign codewords to them. In that area, the opt-

out of PPM is used to deal with previously unseen data, i.e., to calculate probabilities for sequences that have not been part of the training of the PPM. Hence, the opt-out is defined as "$(x_{n-k},\ldots,x_n)$ has not been observed yet". Two examples for PPM instantiations for data compression are defined and compared by [77].

## 2.2. Information Theory

The field of information theory is closely related to communication theory and data compression. The most important achievements of information theory include an upper bound on the capacity of communication channels and a lower bound for possible data compression. In this thesis, we utilize the concepts of *entropy rates* for stochastic process and *typical sequences*. This section gives a quick tour through the information theoretic concepts. We outline the entropy and define the entropy rate for stochastic processes, and what typical sequences are. The presented foundations are based on [23, 59].

### 2.2.1. Entropy

The core definition of information theory is the entropy, a measure for *uncertainty* of a random variable.

**Definition 2.18 (Entropy)** *Let $X \sim p(x)$ a random variable. The* entropy *of $X$ is defined as*

$$H(X) = -\sum_{x\in\mathscr{X}} p(x)\log p(x). \tag{2.2.1}$$

The following example shows how the entropy measures the uncertainty. Consider a random variable X over the alphabet $\mathscr{X} = \{1,2,3,4\}$ uniformly distributed. Since the distribution is uniform, every value is equally possible, i.e., $p(x) = 1/4$ for all $x \in \mathscr{X}$. The entropy of $X$ is $H(X) = -\left(\frac{1}{4}\log\frac{1}{4}+\frac{1}{4}\log\frac{1}{4}+\frac{1}{4}\log\frac{1}{4}+\frac{1}{4}\log\frac{1}{4}\right) = 2$. Now consider a random variable $Y$ over the alphabet $\mathscr{Y} = \{1,\ldots,4\}$, with a probability distribution $p(1) = \frac{3}{4}$ and $p(2) = p(3) = p(4) = 1/12$. The entropy of $Y$ is $H(Y) = -\left(\frac{3}{4}\log\frac{3}{4}+\frac{1}{12}\log\frac{1}{12}+\frac{1}{12}\log\frac{1}{12}+\frac{1}{12}\log\frac{1}{12}\right) \approx 1.207$.

In our example, the entropy of $Y$ is significantly lower than the entropy of $X$, meaning the $Y$ is less uncertain than $X$. This reflects the intuition when both variables are considered. In case of $X$, one is completely unsure whether the result will be 1, 2, 3, or 4. In case of $Y$, the result is 1 in 75% of the time. Hence one is fairly certain of the chance of drawing 1, instead of 2, 3, or 4. Or the other way around: One is less uncertain about $Y$ than about $X$.

Similar to how the joint distribution of multiple random variables can be considered, and how one random variable can be conditioned on another random variable, we define the joint entropy and the conditional entropy in the following.

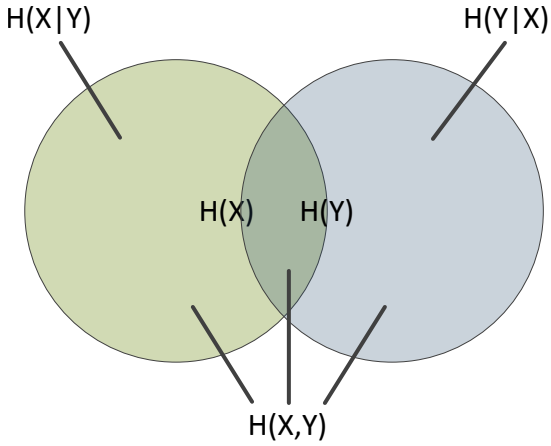



Figure 2.4.: Venn diagram that visualizes the relationship between the entropy, conditional entropy and joint entropy.

**Definition 2.19 (Joint Entropy)** *Let $X \sim p(x)$, $Y \sim p(y)$ a pair of random variables with a joint probability mass function $p(x,y)$. The* joint entropy *is then defined as*

$$H(X,Y) = -\sum_{x\in\mathscr{X}}\sum_{y\in\mathscr{Y}} p(x,y)\log p(x,y). \tag{2.2.2}$$

**Definition 2.20 (Conditional Entropy)** *Let $X \sim p(x)$, $Y \sim p(y)$ a pair of random variables with a joint probability mass function $p(x,y)$. The* conditional entropy *is then defined as*

$$H(Y|X) = -\sum_{x\in\mathscr{X}}\sum_{y\in\mathscr{Y}} p(x,y)\log p(y|x). \tag{2.2.3}$$

The intuitive interpretation of the joint entropy is, that it is the uncertainty, when one considers both $X$ and $Y$ at the same time and of conditional entropy that it is the uncertainty of $Y$, when one already knows the value of $X$. In Figure 2.4 we use a Venn diagram to visualize the relationship between the entropy, the joint entropy and the conditional entropy.

### 2.2.2. Entropy Rates

While the entropy is a useful tool when we work with random variables, it cannot be directly applied to stochastic processes, where we have a sequence of random variables. The entropy rate resolves this problem.

**Definition 2.21 (Entropy Rate)** *The* entropy rate *of a stochastic process $X_1, X_2, \ldots$ is defined as*

$$H(\mathscr{X}) = \lim_{n\to\infty} \frac{1}{n} H(X_1, X_2, \ldots, X_n) \tag{2.2.4}$$

*when the limit exists.*

In less formal terms, the entropy rate is the average entropy per symbol, i.e., we do not calculate the entropy of a single random variable, but rather of the joint entropy of the whole state $X_1, \ldots, X_n$ of a stochastic process and use this value to get the average entropy. The following lemma states an alternative definition of the entropy for stationary process.

**Lemma 2.3** *For stationary processes, the entropy rate can also be defined as*

$$H(\mathscr{X}) = H'(\mathscr{X}) = \lim_{n\to\infty} H(X_n|X_{n-1}, X_{n-2}, \ldots, X_1) \tag{2.2.5}$$

*and the limits for $H(\mathscr{X})$ and $H'(\mathscr{X})$ exist.*

**Proof** *See [23], pp. 64-65.* □

Using this alternate definition, we show the calculation of the entropy rate of stationary first-order MM, which is our objective when it comes to applying information theory in this work.

**Theorem 2.3** *Let $\{X_i\}$ a stationary first-order MM with stationary distribution $\mu$ and transition matrix P. The entropy rate of the process is*

$$H(\mathscr{X}) = -\sum_{i,j} \mu_i p_{ij} \log p_{ij}. \tag{2.2.6}$$

**Proof** *See [23], p. 66.* □

Using the results of Theorem 2.3 we can calculate the entropy rate of a stationary first-order MM with a relatively simple formula (Equation 2.2.6), if we know the transition matrix and the stationary distribution. The transition matrix of a first-order MM is always available or otherwise easy to calculate by enumerating the alphabet. As for the stationary distribution, Theorem 2.1 shows the requirements on the MM (aperiodic, irreducible), as well as the means for calculating the distribution. Hence, we have the required tools to calculate the entropy rate of aperiodic, irreducible, stationary first-order MM.

### 2.2.3. The AEP and Typical Sequences

A fundamental result of information theory is the Asymptotic Equipartition Property (AEP), also known as the Shannon-McMillan-Breimann Theorem. It is the information theoretic counterpart to the law of large numbers, which states that the average of the sum of independent, identically distributed (i.i.d.) random variables $\frac{1}{n}\sum_{i=1}^{n} X_i$ converges to the expected value $EX$ for large $n$. Similarly, the AEP states that the average encoding rate of an optimal encoding $\frac{1}{n}\log p(X_1, \ldots, X_n)$ converges to the entropy rate $H(\mathscr{X})$ of the underlying stochastic process.

**Theorem 2.4 (AEP: Shannon-McMillan-Breimann Theorem)** *Let $\{X_i\}$ a finite valued, stationary, aperiodic, irreducible stochastic process with entropy rate $H(\mathscr{X})$. Then*

$$-\frac{1}{n}\log p(X_0,\ldots,X_{n-1}) \to H(\mathscr{X}) \tag{2.2.7}$$

*with probability 1.*

**Proof** *See [23], pp. 475-479.* □

A consequence of the AEP is that it is possible to divide the possible sequence into typical and atypical sequences, where the typical sequences have some useful properties.

**Definition 2.22 (Typical Sequence)** *Let $\{X_i\}$ a stochastic process. A sequence $(x_1,\ldots,x_n)$ is called* $\varepsilon$-typical sequence *if*

$$2^{-n(H(\mathscr{X})+\varepsilon)} \leq p(x_1,\ldots,x_n) \leq 2^{-n(H(\mathscr{X})-\varepsilon)} \tag{2.2.8}$$

*for $\varepsilon > 0$ and $n \in \mathbb{N}$.*

**Definition 2.23 (Typical Set)** *Let $\{X_i\}$ a stochastic process. The* $\varepsilon$-typical set *of length n sequences is defined as the set of all $\varepsilon$-typical sequences of length n, i.e.,*

$$A_\varepsilon^{(n)} = \{(x_1,\ldots,x_n) \in \mathscr{X}^n : 2^{-n(H(\mathscr{X})+\varepsilon)} \leq p(x_1,\ldots,x_n) \leq 2^{-n(H(\mathscr{X})-\varepsilon)}\} \tag{2.2.9}$$

*for $\varepsilon > 0$ and $n \in \mathbb{N}$.*

**Theorem 2.5** *Let $X_i$ a finite valued, stationary, aperiodic, irreducible stochastic process with entropy rate $H(\mathscr{X})$. Then the typical set has the following properties.*

1. *For $(x_1,\ldots,x_n) \in A_\varepsilon^{(n)} : H(\mathscr{X}) - \varepsilon \leq -\frac{1}{n}\log p(x_1,\ldots,x_n) \leq H(\mathscr{X}) + \varepsilon$.*
2. *$\mathbf{Pr}\{A_\varepsilon^{(n)}\} > 1-\varepsilon$ for n sufficiently large.*
3. *$(1-\varepsilon)2^{n(H(\mathscr{X})-\varepsilon)} \leq |A_\varepsilon^{(n)}| \leq 2^{n(H(\mathscr{X})+\varepsilon)}$.*

**Proof** *See [23], pp. 51-53.* □

The reason why we introduced the AEP and typical sets are the properties 2 and 3 that Theorem 2.5 states. Property 3 bounds the size of the typical set, thereby also bounding the number of typical sequences to about $2^{nH(\mathscr{X})}$ sequences. If the entropy is significantly lower than $log|\mathscr{X}|$, only very few typical sequences exist. Property 2 states that atypical sequences are really improbable and nearly all of the probability mass is allocated by typical sequences. Hence, properties 2 and 3 in combination state that very few sequences allocate nearly all of the probability mass. While we do not strictly apply these results in the latter parts of this thesis, we use the intuition that we gain by these properties as motivation for our model assumptions.

## 2.3. Event-driven Software

The software testing concepts that we propose in this thesis are designed for EDS. There are various types of EDS, two of which we consider in thesis: GUIs and Web application. While we do not consider other types, e.g., embedded systems, we define the concepts based on the abstract notion of events that is not for any specific EDS platform.

**Definition 2.24 (Event (Software))** *An* event *e is an observable action of a software that is characterized by its* type *and its* target*, i.e.,* $e = \{type(e), target(e)\}$.

**Definition 2.25 (Event Space (Software))** *The event space E of a software is defined as the set of all possible events.*

The effect of an event is usually a change of the state of the software. Therefore, Deterministic Finite Automatons (DFAs) are often used for the specification of EDS, e.g., [8, 62, 131, 134].

**Definition 2.26 (Deterministic Finite Automaton (DFA))** *A* Deterministic Finite Automaton (DFA) *is a quintuple* $(Q, \Sigma, \delta, q_0, F)$*, where*

- $Q$ *is a set of states,*
- $\Sigma$ *the alphabet,*
- $\delta : (Q \times \Sigma \to Q)$ *a transition function,*
- $q_0$ *the initial state of the automaton,*
- *and* $F \subset Q$ *a set of final (or accepting) states.*

In EDS, the states $Q$ are the states of a system, and the alphabet $\Sigma$ is the event space. The transition function describes which events are allowed and which events are disallowed for each state. The initial state is usually the start of the System Under Test (SUT) and the final states are the termination of the SUT. Figure 2.5 shows a simple vending machine that starts with an *Initial* state from which it is created with or without soda in stock and changes its state to *Stocked* or *Empty*, respectively. Afterwards, it is possible to get soda if the machine is in the *Stocked* state. If the stock is empty afterwards, the state is changed to *Empty*.

In this work, we only work with approximative and possible imprecise models of the SUT that we infer directly from observing events sent to the SUT. We do not observe the actual internal state of the SUT and do not assume that we know these states from another source, e.g., an existing model of the SUT. Instead, we assume that the state is accurately described by the recent events, i.e., $Q = E^k$, where $k$ is the number of recent events used to describe the states and the accuracy increases with $k$. In the following, we describe the scenario for $k = 1$.

We define $Q = \Sigma = E$ as the events and that the SUTs state is the last observed event. Mathematically, this means we restrict $\delta$ such that only transitions $(e, e') = e'$ are allowed

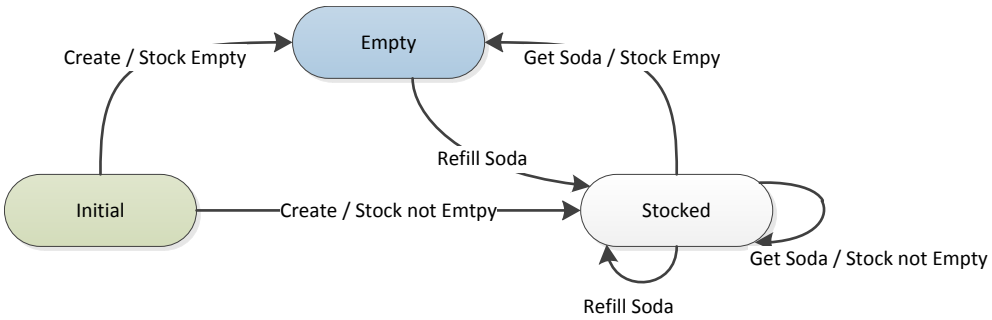



Figure 2.5.: A small automaton describing a vending machine. The initial state is colored green, the final state is colored blue.

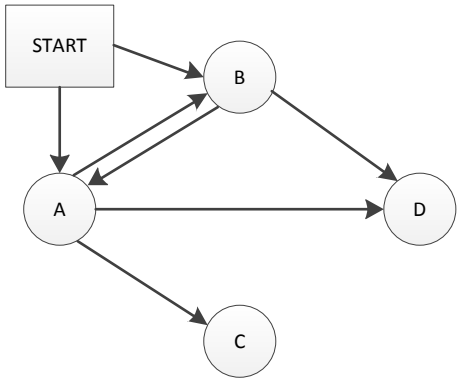



Figure 2.6.: A small EFG with events A, B, C, D.

and we say that $e'$ follows $e$. Due to this, we can remove the labels from the edges, when we graphically represent a DFA. While a SUT can only have one initial state, we can have multiple initial events. Therefore, $Q = E$ is a conflict with Definition 2.26, which only allows one single initial symbol $q_0$. To deal with this difficulty, we add a *START* symbol to $Q$ and transitions $(START, e) = e$ for all feasible initial events of the SUT. This kind of model for EDS is also known as Event-flow Graph (EFG) [62]. Figure 2.6 depicts an abstract EFG over the event space $\{A, B, C, D\}$.

### 2.3.1. Graphical User Interfaces

A GUI is a means for users to interact with software. The Linux Information Project defines a GUI as follows [57].

**Definition 2.27 (Graphical User Interface (GUI))** *A Graphical User Interface (GUI) is a* human-computer interface *(i.e., a way for humans to interact with computers) that uses*

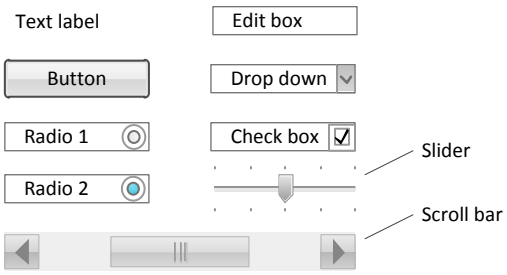



Figure 2.7.: A selection of common icons.


Figure 2.8.: A small dialog with a menu bar, a "Hello World" text label, and a "Close" button.

windows, icons *and* menus *and which can be manipulated by a mouse (and often to a limited extent by a keyboard as well).*

In the definition, the term icon is very general, and refers to anything from icons on the desktop to buttons in an application. Figure 2.7 shows a selection of icons that are common in GUIs. A window is usually a collection of icons, e.g., the main window of an application or a dialog. Additionally, most windows have a menu. Figure 2.8 shows a simple dialog to exemplify the concepts of Definition 2.27.

For the most part, we do not make distinctions between windows, items, and menus in this thesis. Instead, we use the generic and synonymous terms *GUI object* and *widget*. As Figure 2.8 shows, some widgets are themselves collections of widgets. In the example, the dialog "Window" is a collection of the menu, text label, and button widgets. This results in a parent/child relationship between widgets.

**Definition 2.28 (Parent/Child Relationship (1))** *A widget $A$ is the parent of widgets $B_1, B_2, \ldots$ if $A$ is a collection of $B_1, B_2, \ldots$.*

In Figure 2.8 the dialog "Window" is the parent and the other widgets are its children. Collections of widgets are not the type of parent/child relationships between widgets.

**Definition 2.29 (Parent/Child Relationship (2))** *A dialog B is a child of widget A if the existence of B depends on the existence of A (i.e., if A is destroyed, B is destroyed, too) and A has created B.*

An example for a child dialog is the "*File Open*" dialog that pops up when the user clicks on the open button in a menu or task bar. This dialog is then the child of the main window of the application. Child dialogs also have a *modality*.

**Definition 2.30 (Modal)** *Let B be a child dialog of a widget A. If B blocks the execution of A as long as it is open, we call A* modal. *If the execution of A is not blocked, we call B* non modal.

The "File Open" dialog is an example of a modal dialog. While you select the file you want to open, it is not possible to use the main window of the application. "*Help*" dialogs are usually non modal, i.e., the help dialog can be used concurrently to the main application.

While most widgets in an application have a parent, at some point there has to be a widget without a parent, because otherwise there would be circular relationships or an infinite depths of widgets.

**Definition 2.31 (Top-level Widget)** *A widget that does not have a parent is called a* top-level widget.

Most applications have exactly one top-level widget, i.e., the main windows of the application. However, complex software products can also have multiple top-level widgets.

As a result of the parent/child relationship between widgets, we define a *GUI* hierarchy. The top-level widgets reside in the highest layer of the hierarchy, then the children of the top-level widgets, and so on. Figure 2.9 shows an example of a GUI hierarchy. The hierarchy resembles a tree-like structure. Therefore, the GUI hierarchy defines a *widget tree* or *widget forest*, if there is one or multiple top-level windows, respectively.

Besides the structure of GUIs, the events are also indirectly part of Definition 2.27, as the latter part says "manipulated by a mouse (and often to a limited extent by a keyboard as well)". These manipulations are events. In practical terms, the manipulations include clicking with the left mouse button, scrolling the mouse wheel, and pressing a key. Hence, the type of a GUI event, according to Definition 2.24 is which mouse button is clicked, key is pressed, or where the mouse is moved. The target is *where* the GUI event takes place, i.e., on which widget.

The principles we discussed in this section are valid for all GUI platforms. In sections 2.3.1.1 and 2.3.1.2, we introduce two concrete GUI platforms in detail.

#### 2.3.1.1. Microsoft Foundation Classes (MFC)

The MFC comprise a GUI platform for Microsoft Windows developed by Microsoft [72]. MFC has a long history and exists since 1992. In general, it is a C++ wrapper for the

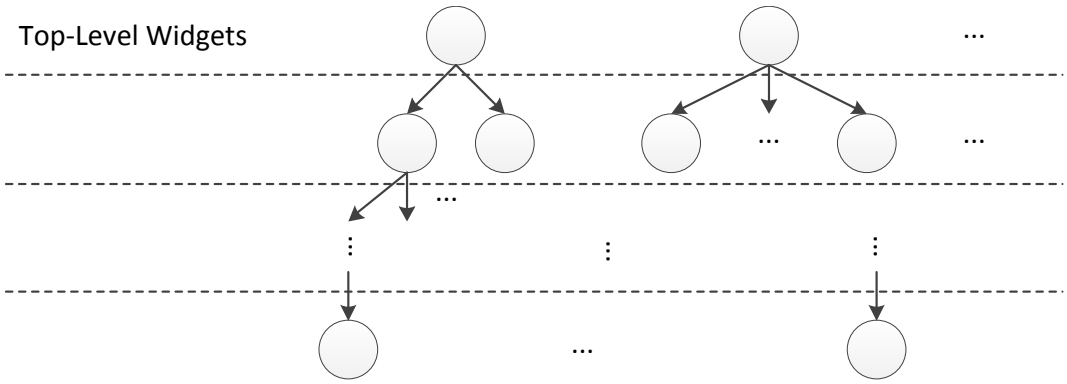



Figure 2.9.: An abstract example for a GUI hierarchy. The circles represent widgets and the arrows parent/child relationships.

Windows Application Programming Interface (API). Hence, MFC shares many important characteristics with the Windows API, even though they are often not exposed directly. The addressing of widgets as well as the communication between the widgets is the same as in the Windows API, even though MFC tries to hide the complexities to some degree.

To identify and address widgets a *window handle* referred to as *HWND* exists [73]. The HWND is unique, i.e., no two widgets in the system have the same HWND. Furthermore, the HWND is assigned during the creation of a widget and cannot be changed. After a widget is destroyed, its HWND is freed and can be re-used for a new widget. There is no guarantee that the same widget will have the same HWND it had before, when it is recreated. It is extremely unlikely that a widget will have the same HWND when it is recreated. Hence, the HWND is a reliable identifier for a widget during its existence, but cannot be used to identify the same widget when it is created at a different time.

For the communication with MFC widgets, *messages* are used [67]. The message-based communication is not only used for inter-widget communication, but also for the communication of the operating system with widgets, the business logic, and the GUI of an application. The messages are addressed to the widgets using their HWND. The target of a messages does not have to be in the same process as the sender, i.e., widgets from other processes can be addressed without limitations. The sender does not even need to know, to which process an addressed widget belongs. The message is addressed to the HWND and the operating system takes care of the message delivery.

The content of a message is a *type* and two parameters. The type defines the purpose of a message, e.g., `WM_SETFOCUS` to set the keyboard focus to a widget, `WM_SETTEXT` to change a widget's text, and `WM_LBUTTONDOWN` as a notification that the left mouse button has been pressed down on a widget. In this work, we always refer to types using symbolic names, like `WM_SETFOCUS`. Strictly speaking, the types are integer values and the symbolic names are C++ pre-processor definitions, i.e., `#define WM_SETFOCUS 7`. The two parameters

of a message are called LPARAM and WPARAM. The information they carry depends on the message type. For example, the WPARAM of the WM_SETFOCUS contains the HWND of the widget that lost the keyboard focus, and the LPARAM is not used. A second example is the WM_SETTEXT message, where the LPARAM contains a pointer to the new text of the widget and the WPARAM is not used. In Table 2.1, we list and describe the Windows messages that we require in the later parts of this thesis.

The MFC try to abstract from the message processing by providing GUI building blocks that already know how to interpret a message. To customize the reactions to messages, *message handler* are used. The message handlers are inheritance based. To customize a message handler of an MFC widget, a child class of the widget must be created that overrides the default message handler. Even though there is a layer of abstraction in between, the messages are still omnipresent and can be utilized.

The GUI events of MFC applications are a subset of the Windows messages. The messages with types WM_*BUTTONDOWN[1], WM_*BUTTONUP, WM_*BUTTONDBLCLK, WM_NC*BUTTONDOWN, WM_NC*BUTTONUP, WM_NC*BUTTONDBLCLK, WM_MOUSEWHEEL, WM_MOUSEMOVE, WM_KEYDOWN, and WM_KEYUP represent the GUI events. The type of the GUI event is the message type and the target widget of the GUI event is known through HWND the message is addressed to.

MFC widgets have three more properties we utilize in this thesis, in addition to the GUI hierarchy and the modality. The first property is the *window class* [68], which indicates the type of a widget, i.e., what kind of icon or window (Definition 2.27) a widget is. Examples for window classes are Button for a button, ToolBar32 for toolbars, and #32770 for dialogs. The second is the text of the widget. While not every widget has a text, many widgets, e.g., buttons and text labels contain a string that is displayed. This displayed string is the widget's text. The third is the *resource Id*, an identifier for the associated control of the widget [71, 69]. Not all widgets have an resource Id and multiple widgets can have the same resource Id, i.e., the same control.

#### 2.3.1.2. Java Foundation Classes (JFC)

The popular software platform Java [83] also contains a library for creating GUIs, the JFC. The JFC can be considered as a collection of three GUI libraries: the Abstract Window Toolkit (AWT) [82], Swing [85], and Java 2D [84]. We are only interested in the AWT and Swing, which provide actual GUI components and the infrastructure for their communication and behavior. The library Java 2D is for working with 2D graphics and imaging and, therefore, out of the scope of this thesis.

The Swing library is build on top of the AWT to provide more building blocks for widgets and greater comfort when working with them. The basic principles of both are the same. To communicate between widgets, *AWT events* are send. The AWT events are the JFC

[1] * is a placeholder for L, R, M, which stand for left, right, and middle mouse button, respectively.

| Message type | Description |
|---|---|
| WM_*BUTTONDOWN | Mouse button * is pressed down. |
| WM_*BUTTONUP | Mouse button * is released. |
| WM_*BUTTONDBLCLK | Send instead of second WM_*BUTTONDOWN in case of a double-click. |
| WM_NC*BUTTONDOWN | Mouse button * is pressed down in the non-client area of the application. |
| WM_NC*BUTTONUP | Mouse button * is released in the non-client area of the application. |
| WM_NC*BUTTONDBLCLK | Send instead of second WM_NC*BUTTONDOWN in case of a double-click in the non-client area of the application. |
| WM_MOUSEWHEEL | The mouse wheel is scrolled. |
| WM_MOUSEMOVE | The mouse is moved. |
| WM_KEYDOWN | Key is pressed down. |
| WM_KEYUP | Key is released. |
| WM_SYSKEYDOWN | System key is pressed down. |
| WM_SYSKEYUP | System key is released. |
| WM_CREATE | Receiving widget is created. |
| WM_DESTROY | Receiving widget is destroyed. |
| WM_SETTEXT | Text of the receiving widget is changed. |
| WM_SETFOCUS | Keyboard focus is changed to the receiving widget. |
| WM_COMMAND | A command is executed by the receiving widget. |
| WM_SYSCOMMAND | A system command is to be executed. |
| WM_HSCROLL | Send when a scroll bar or slider is moved horizontally. |
| WM_VSCROLL | Send when a scroll bar or slider is moved vertically. |

Table 2.1.: Selection of Windows messages required in this thesis. In the top half of the table, we show the messages related to GUI events, in the bottom half the messages related to the internal communication of MFC applications.

| Defining Class | Event type | Description |
|---|---|---|
| `WindowEvent` | `WINDOW_OPENED` | Send when a widget is created. |
| | `WINDOW_CLOSED` | Send when a widget is closed. |
| `FocusEvent` | `FOCUS_GAINED` | Send when a widget gains the keyboard focus. |
| `MouseEvent` | `MOUSE_CLICKED` | Send when the mouse is clicked. |
| | `MOUSE_DRAGGED` | Send when the mouse is moved while a button is pressed down. |
| | `MOUSE_MOVED` | Send when the mouse is moved. |
| | `MOUSE_PRESSED` | Send when a mouse button is pressed down. |
| | `MOUSE_RELEASED` | Send when a mouse button is released. |
| `KeyEvent` | `KEY_PRESSED` | Send when a key is pressed. |
| | `KEY_RELEASED` | Send when a key is pressed. |

Table 2.2.: Selection of JFC events required in this thesis.

counterpart to the Windows messages of MFC. They have a type, which is an integer, but has also a symbolic name achieved through a definition, e.g., `public static void KEY_PRESSED = 401`. All AWT events are a subclasses of a generic `java.awt.AWTEvent` class and the event types are classified by the classes they are defined in. For example, the class `KeyEvent` defines the `KEY_PRESSED` event type as well as all other keyboard related events.

Since Java is strictly object-oriented, there is no need for something like the HWND to address widgets. Instead, all widgets are objects and the events are addressed to objects. The widgets have *event handler*, which are the JFC counterpart of the MFC message handler. The event handler provides a layer of abstraction from the events to allow more comfortable programming of GUIs.

In Table 2.2, we list the AWT events that we require in this work. The AWT events defined in `MouseEvent` and `KeyEvent` are the GUI events. The type and target of the GUI events are the same as of the AWT events.

JFC widgets have three properties that we utilize in addition to the GUI hierarchy and the modality. The first two are the widget class and the widget text, similar to MFC (Section 2.3.1.1). The only difference is, that in case of JFC the class is literally the class of the widget, i.e., the Java class. The third property is the *index* of a widget. The index is an extension of the parent/child relationship. In JFC, the child widgets are stored in an ordered list by the parent. Since most of the time all widgets are created in the same order, the index is an indicator to differentiate between widgets.


Figure 2.10.: Communication scheme of Web applications.

### 2.3.2. Web Applications

With the growth of the Internet, websites evolved from simple static Hypertext Markup Language (HTML) pages that provided information mostly in a table-based layout, to large websites that do not only display content, but allow for an ever growing amount of interactions with the Web users. At some point during this evolution, most web sites changed from being static to dynamic representations that calculate the content of the website dynamically. This was the start of the Web applications. Nowadays, the term Web application is used in a very broad spectrum, meaning anything from PHP-based web sites [112] with a database back end via Web services [119] to cloud-based Software as a Service (SaaS) [60] applications. One definition of a Web application is provided by the World Wide Web Consortium (W3C) [118].

**Definition 2.32 (Web Application)** *A Web application is defined as a Hypertext Transfer Protocol (HTTP) based application whose interactions are amenable to machine processing.*

In Web applications, there is always a client and a server. Since we want to analyze usage, we assume the client side to be a Web browser operated by a user. The Web server is located somewhere in the Internet. Web applications work through the exchange of information between the client and the server. We outline the general communication scheme of Web applications in Figure 2.10. Depending on the Web application, the programming logic is executed on the client side, server side, or both. In this thesis, we only consider applications that run strictly the server side and are based on PHP.

In PHP-based Web applications, the clients sends a HTTP request to the Web server. Together with this request, the client can send variables. There are two types of variables: *GET* and *POST*. The GET variables are included in the Uniform Resource Identifier (URI) of the HTTP request. An example for a URI with GET variables `getVar1`, `getVar2`, ..., `getVarN` with values `getVal1`, `getVal2`, ..., `getValN` is `http://hostname.com/website.php?getVar1=getVal1&getVar2=getVal2&...&getVarN=getValN`. The POST variables are not included in the URI, but in a separate part of the HTTP request. Other than

that, they are the same as the *GET* variables. The client can send no variables, only GET variables, only POST variables, or send both GET and POST variables. The Web server processes the HTTP request by executing the PHP script and passing the GET and POST variables. It replies with the result of the PHP script, usually a HTML web page.

In this setting, the events are the HTTP requests. For an HTTP request $e$, we define the $target(e)$ of the event as the PHP script that is executed and the $type(e)$ of the event as the variables that are sent with the request. Hence, the type is *how* the target is executed, i.e., with which set of variables in mind. The event type does not include the values of the GET and POST variables. The rational for this is as follows. The variables are often used to pass session specific information to the PHP script, e.g., session or user IDs. This would disallow for events from different sessions to be equal. Furthermore, websites often use the variable values to distinguish between the displayed data, e.g., information about which lecture is displayed. In the context of testing, this information can be disregarded, as testbeds have their own data, which is usually different from the published versions of the website. Finally, POST variables often contain privacy critical information, e.g., passwords entered into a form of a website.

## 2.4. Software Testing

The field of software testing is crucial for the quality assurance of software. Myers defines software testing as follows [78].

**Definition 2.33 (Software Testing)** *Software testing is a process, or series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended.*

Software testing can be performed on any level of abstraction and any time during the software life cycle. Contrary to the popular opinion, the purpose of software testing is not to prove that there are bugs in a software. The purpose of software testing is to find bugs. Hence, a software test is effective if it finds bugs. However, while testing does not prove that there are no bugs in a software, it provides confidence in the quality of the software. If tests designed to find bugs do not find any bugs, it is an indicator for good software quality.

There are many different approaches to software testing, for different applications and levels of abstraction. In the following, we discuss the terminology of software testing and outline software testing of EDS. Afterwards, we introduce the notion of usage-based testing, the type of software testing we apply in this thesis.

### 2.4.1. Terminology

Software tests are designed to test one specific software system, the SUT. The actual tests of a software are organized in form of *test cases*. The IEEE defines test cases as follows [1].

**Definition 2.34 (Test Case)** *A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.*

Since one test case is usually not sufficient to test software, test cases are organized in test suites. The following definition of a test suite is taken from the International Software Testing Qualifications Board (ISTQB) [50].

**Definition 2.35 (Test Suite)** *A set of several test cases for a component or SUT, where the post condition of one test is often used as the precondition for the next one.*

As Definition 2.34 states, test cases contain an expected result, which is determined by the *test oracle* [50].

**Definition 2.36 (Test Oracle)** *A source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), other software, a user manual, or an individual's specialized knowledge, but should not be the code.*

By comparing the actual result of executing a test case against the SUT to the expected result, we calculate the *test verdict* associated with the test case. Usually, two test verdicts are used: *pass* and *fail*. The verdict pass is assigned when the actual result is equal to the expected result and the verdict fail if there is a deviation or the SUT crashes during the execution of the test case.

To measure the completeness of a test suite, we use *test coverage* measures. The test coverage is defined by the IEEE as follows [1].

**Definition 2.37 (Test Coverage)** *The degree to which a given test or set of tests addresses all specified requirements for a given system or component.*

The concrete coverage measures depend on the SUT and the level of abstraction used in testing. For example, when testing single software units, coverage metrics like the *statement coverage* are used [78].

Test cases can be *automated*, i.e., automatically executed by a computer. Automated test cases often compare the actual results to the expected results during the test execution. These comparisons are implemented in form of *assertions*. An assertion is a boolean expression that compares the actual results to the expected results. An assertion fails if the result of the boolean expression evaluates to false. If an assertion of a test case fails, it means that there is a deviation from the expected result and the verdict fail is assigned.

A software is usually not tested exactly once, but every time there is a change in the SUT to validate the quality of the system. These repeated executions are called *regression tests* and defined by the IEEE as follows [1].

**Definition 2.38 (Regression Test)** *Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.*

### 2.4.2. Event-driven Testing

To test event driven software, test cases are defined in terms of the initial state of the system, a sequence of events to be executed, the expected result after the execution of the event sequence, and, optionally, the results between events during the execution. Therefore, we define an EDS test case as follows.

**Definition 2.39 (EDS test case)** *An EDS test case consists of the initial state of the SUT, a sequence of events $s = (e_1, \ldots, e_n)$ and a sequence of expected results $expected(e_1), \ldots, expected(e_n)$, where $expected(e_i)$ is the expected result after event $e_i$. $expected(e_i)$ may be undefined, which allows any result.*

In case an EDS is specified using a EFG, the model can be used to directly infer test cases for the system by *walking* through the EFG. By walking, we mean that we start in the initial state of the system and then select the next state based on the transition function, and so on, until a final state is reached. There are many approaches to systematically walk through an DFA for test case generation, e.g., some of which we discuss in the related work (Section 7.1.

A second approach for the generation of test cases is the *capture/replay* technique [49, 50, 86], which is often used to create automatically executable test cases. The technique consists of three parts. First, the SUT is manually executed. During this execution, a *capture tool* records all interactions with the system, i.e., a sequence of the events. In an often optional second step, a test engineer modifies the recorded sequence and adds expected results. This results in a complete EDS test case. Thirdly, this test case is automatically executed against the SUT.

To measure the completeness of an EDS test suite, the *event coverage* is used [7, 8, 65, 81].

**Definition 2.40 (Event Coverage)** *Let $S_d$ be the set of all length $d$ event sequences of an SUT that are a subsequence of any valid sequence of events. Let $S_{test}$ be a test suite and $S_{test,d}$ be all length $d$ event sequences covered by $S_{test}$. The* depth $d$ event coverage *of $S_{test}$ is defined as*

$$Cov_d(S_{test}) = \frac{|S_{test,d}|}{|S_d|}. \tag{2.4.1}$$

**Notation 2.4** *Depth one event coverage is also known as* event coverage*, depth two event coverage is also known as* transition coverage.

To exemplify the event coverage, consider the test suite $S_{test} = (START, B, A, B, D)$ for the abstract EFG shown in Figure 2.6. The test case covers the depth two sequences $(START, B)$, $(B, A)$, $(A, B)$, and $(B, D)$ and it does not cover the sequences $(START, A)$, $(A, C)$, and $(A, D)$. Thus, $S_{test}$ has a depth two event coverage of $Cov_2(S_{test}) = \frac{4}{7} \approx 57\%$. Figure 2.11 visualizes this example.

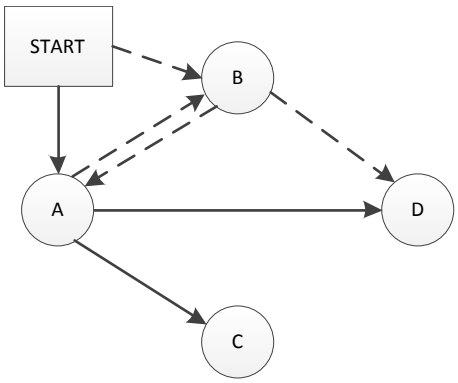



Figure 2.11.: The dashed edges visualize the depth-2 coverage of the test suite $S_{test} = (START, B, A, B, D)$.

### 2.4.3. Usage-based Testing

In an ideal world, we would test software exhaustively, i.e. "*all combinations of input values and preconditions*" [50]. However, the amount of possible testing of a SUT is limited by time and budget, making exhaustive testing in almost all scenarios impossible. Consider a scenario of an application that takes a 32-bit integer value $I$ as input and outputs binary representation of $I$. To test this application exhaustively, all $2^{32} = 4,294,967,296$ possible input values need to be tested, making the exhaustive testing of even such a small example nearly impossible. Thus, the testing effort needs to be focused.

Traditional software testing strategies deal with this problem by systematically reducing the combinations of input values and preconditions that need to be tested. For example, in *control-flow based* testing the criterion is that a predetermined amount of the control-flow, i.e., execution paths through the SUT are covered by the test suite. In EDS testing, the event coverage can be used to focus the effort to fulfill, e.g., only the depth two event coverage instead of all possible event sequences. With large-scale software, it is often not even possible to fulfill such a limiting criteria fully. Thus, further focusing of the testing effort is required. This is often referred to as test priorization. In usage-based testing, the testing effort is prioritized to optimize the user-based quality [50]:

**Definition 2.41 (User-based Quality)** *A view of quality, wherein quality is the capacity to satisfy the needs, wants, and desires of the user(s). A product or service that does not fulfill user needs is unlikely to find any users. This is a context dependent, contingent approach to quality since different business characteristics require different qualities of a product.*

When we apply this definition to the testing of a system's functionality, it means that the functions that are important for the user should be tested first. Usage-based testing assumes that the functions most probably used are the most important ones for a user. Thus,

usage-based testing focuses the effort such that functionality used with high probability is tested intensively and functionality used with low probability is only tested sparsely. *Usage profiles* are used to model which functionality is us used how often.

**Definition 2.42 (Usage Profile)** *A* probabilistic usage profile *is a discrete stochastic process over the observed event space, i.e.,* $\mathcal{X} = E_U$ *with* $E_U \subset E$.

In principle, a usage profile is a model of the SUT like an EFG. Nevertheless, there are crucial differences between usage profiles and other models. Most importantly, usage profiles are only defined over a subset of the event space, i.e., the observed event space $E_U$. Accordingly, usage profiles describe an *incomplete* SUT and it is possible that partial functionalities of a system are not included in a usage profile, if the related events are not part of $E_U$.

However, we argue that even if $E' = E \setminus E_U$ is unknown, i.e., we do not known which events are unknown in the usage profile exist, the usage profile is still a complete usage profile of the SUT. A usage profile is based on the observed usage of a system. Hence, all observed usage is part of the profile. This means that the missing parts have never been observed and have probability zero. From a stochastic point of view, it does not matter if an event has always probability zero or does not exist, therefore, the profile is complete.

In order to obtain an accurate usage profiles of a SUT, we require empirical data about the SUT's usage. For EDS, this means we need to monitor event sequences during the SUT's execution by users. This way, we gain *user sessions*.

**Definition 2.43 (User Session)** *A* user session $s$ *is an ordered sequence of events* $s = (e_1, e_2, \ldots, e_m)$ *with* $e_1, e_2, \ldots, e_m \in E$ *that has been observed during the execution of a system by a user* $u \in U$. $S_U$ *is the collection of all user sessions observed for the users* $U$.

Based on the gathered data, the probability mass function of the stochastic processes underlying is estimated. For example, consider training a first-order MM with the user sessions $S_U = \{(A,D);(A,B,D);(B,A,D);(B,D);(B,D)\}$ gathered from the SUT described by the EFG shown in Figure 2.6. The result of the training is depicted in Figure 2.12. The figure shows an END event, that is missing in the EFG. The END event symbolizes that its predecessor is a final event, i.e., the last event in a user session. Depending on the kind of EDS that is modelled, the final event can be important, e.g., in GUI applications it means that the application is terminated afterwards.

To train the MM, we modify the sessions by prepending a START symbol and appending a END symbol, i.e., $S'_U = \{(START,A,D,END);(START,A,B,D,END);(START,B,A,D,END);(START,B,D,END);(START,B,D,END)\}$. Then, we calculate the empirical probabilities of each transition, e.g., we have two times the transition $START \rightarrow A$ and three times the transitions $START \rightarrow B$, thus, $\mathbf{Pr}\{X_{n+1} = A | X_n = START\} = \frac{2}{5} = 0.4$ and $\mathbf{Pr}\{X_{n+1} = B | X_n = START\} = \frac{3}{5} = 0.6$. By repeating this procedure for all transitions, we obtain the MM depicted in Figure 2.12. This example also shows the difference between

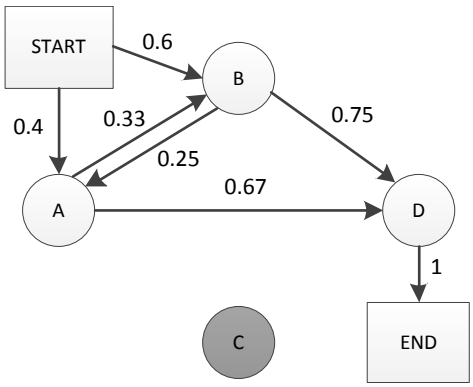



Figure 2.12.: A first-order MM trained from the sequences $S_U = \{$*(A,D); (A,B,D); (B,A,D); (B,D); (B,D)*$\}$.

the observed event space $E_U$ and the event space $E$. The event $C$ is never observed, i.e., not part of the user sessions $S_U$. Therefore, $C$ is only part of the EFG, which is based on the event space, but not of the MM, which is based on the event space.

To generate test cases from a usage profile, the underlying stochastic process is simulated, i.e., a sequence of events is drawn according to the probability distribution of the stochastic profile. To this aim, we set the initial symbol of the process $X_0 = START$, then determine $X_1$ according to the probability mass function $\mathbf{Pr}\{X_1 = x_1 | X_0 = START\}$, and so on. This procedure is repeated until we reach the END event. In case we need test cases of a predefined length, we have two options. On the one hand, if we do not care whether a test case ends with a valid end symbol, i.e., with a symbol that is a predecessor of *END*, we just discard everything in the sequence that is longer than the length we require. On the other hand, if we require a valid end, we just discard any drawn test case that does not match the length and re-draw until a test case of the desired length is drawn. Algorithms 1 and 2 describe the random walk procedure with and without a valid end symbol, respectively.

**Input** : Usage profile $\{X_i\}$, desired test case length $l$
**Output**: Sequence $(START, e_1, \ldots, e_l, END)$
1 $e_0 \leftarrow START$;
2 $i \leftarrow 0$;
3 **while** $e_i != END$ **do**
4 | Draw $e_{i+1}$ according to $\mathbf{Pr}\{X_{i+1} = e_{i+1} | X_i = e_i, \ldots, X_0 = e_0\}$;
5 | $i \leftarrow i + 1$;
6 **end**
7 **if** $i - 1 \neq l$ **then**
8 | Goto line 1;
9 **end**

**Algorithm 1**: Random walk with valid end.

**Input** : Usage profile $\{X_i\}$, desired test case length $l$
**Output**: Sequence $(START, e_1, \ldots, e_l)$
1 $e_0 \leftarrow START$;
2 $i \leftarrow 0$;
3 **while** $e_i != END$ **do**
4 | Draw $e_{i+1}$ according to $\mathbf{Pr}\{X_{i+1} = e_{i+1} | X_i = e_i, \ldots, X_0 = e_0\}$;
5 | $i \leftarrow i + 1$;
6 **end**
7 **if** $i \leq l$ **then**
8 | Goto line 1;
9 **end**

**Algorithm 2**: Random walk without valid end.

# 3. Advancement of Usage-based Testing

In this section, we present novel usage-based testing techniques. We outline the usage-profiles including their advantages and disadvantages. As part of this, we present a novel usage profile based on a PPM model with properties desirable for usage-based testing. Subsequently, we discuss how we analyze the usage of software through the analysis of the data and the calculation of stationary distributions of the stochastic processes underlying the usage profiles. Furthermore, we define a new coverage metric for usage-based testing. Finally, we introduce novel usage-based test case generation techniques.

## 3.1. Usage Profiles

The model builder generates SUT testing models from abstract event sessions $S_U$, which are available from the SUT. In our case, this means the training of a usage profile, i.e., a stochastic process over the observed events (see Definition 2.42).

In this thesis, we consider three different types of stochastic processes to implement usage profiles: first-order MMs, higher order MMs (i.e., $k$-th order MMs for $k > 1$), and PPM models. The MMs have already been used in the literature. We discuss them here for two reasons. First, all of the profile types have various advantages and disadvantages, which have not been discussed and compared to each other with respect to usage based testing. Second, we only found one contribution where higher order MMs have been used (Section 7.1.4). Our extensive case studies with usage profiles based on these models will improve the understanding of the implications that higher Markov orders have for usage-based testing. In the following, we discuss each type of process separately and how the properties of each type affects generated test suites.

### 3.1.1. First-order Markov Models

The first-order MMs are the most popular choice for usage profiles in the literature (Section 7.1). They have a close relationship to DFAs and are relatively easy to implement. Because of the convenient representation as a graph, the models can be visualized without difficulty, which is important for experts who analyze usage profiles manually. Moreover, the complexity is with $O(|E_U|^2)$ the lowest of the stochastic processes that we consider.

For the generation of test cases, the memorylessness of the first-order MMs is both an advantage and disadvantage, depending on the SUT. As described in Section 2.3, we do not consider the actual state of the SUT and work with possible imprecise models, where we

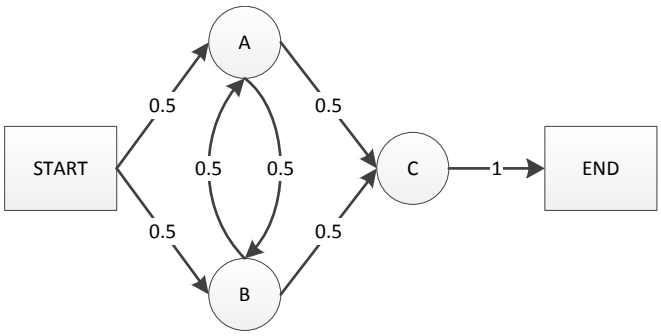



Figure 3.1.: A first-order MM as usage profile trained with sequences $s_1 = (A,B,C)$ and $s_2 = (B,A,C)$ (adapted from [47]).

assume that the last events can accurately describe the internal state of the SUT. With first-order MMs, the current state of the SUT is only described by the most recent event, which can lead to imprecisions. Consider a SUT with three events $A$, $B$, and $C$, where the order of $A$ and $B$ is undefined, but both $A$ and $B$ are precursors for $C$, i.e., $A$ and $B$ need to be executed, before $C$ is executed. Then $s_1 = (A,B,C)$ and $s_2 = (B,A,C)$ are valid and $s_3 = (A,C)$ is not allowed. Figure 3.1 depicts a usage profile with a first-order MM trained with $s_1, s_2$. The sequence $s_3$ is allowed in the profile. Hence, if a SUT has more requirements than *follows*-relations between events, first-order MMs are insufficient to model all requirements correctly, leading to the possibility of invalid sequences.

Consider the same example, without the condition that $A$ and $B$ are both precursors for $C$. Then the sequence $s_3$ is valid and can be generated by the usage profile, because the usage profile generalizes from the training data. This is the advantage provided by the memorylessness. There is a high degree of randomness, compared to higher order MMs, which are not able to generate $s_3$.

### 3.1.2. Higher order Markov Models

Higher-order MMs provide a solution for the definition of usage profiles that accurately model precursors. Here, the underlying models have a memory length of $k, k > 1$. In general, conditions that rely on events that are up to $k$ events in the past can be modeled. Everything that is out of this reach cannot be modeled. However, there are several drawbacks of higher order MMs. They are more complex to implement, since the transitions cannot simply be written down in a $|E_U| \times |E_U|$ transition matrix. Additionally, there is no easy concept for the visualization of higher order MMs, which makes the manual analysis of usage profiles by experts difficult. Furthermore, the complexity of higher-order models is in $O(|E_U|^{k+1})$, i.e., it grows exponentially with the memory lengths. Therefore, the scalability of the memory length is limited as the model complexity cannot be handled for indefinably high values of $k$.

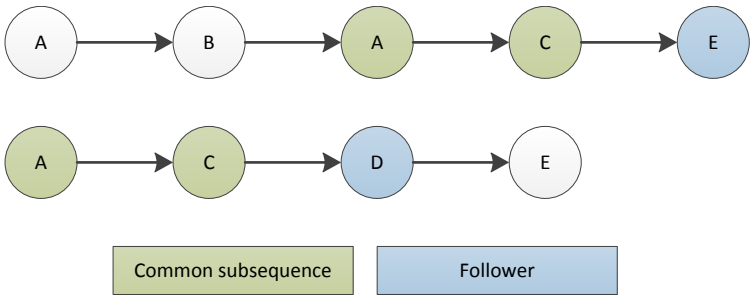



Figure 3.2.: Two training sequences with a common subsequence of length two with different following symbols.

A further disadvantage of long memory length is the loss of generalization such that the usage profiles becomes static instead of probabilistic. Consider the extreme case, where the memory length equals the length of the longest training sequences. Then, the usage profile becomes a collection of all training sequences, without any generalization. When a model is specifically tailored to the training data without generalizing from the data, we speak of *overfitting*. Albeit this example is an extreme case, it shows the adverse effect of long memories. Generally, in order to produce randomness in the model, the memory must not be longer than the length of the longest common subsequence with a different following event. Figure 3.2 visualizes the meaning of this statement. In the example, the longest common subsequence is $(A,C)$. In the first sequence the following event is $E$ and in the second sequence the following event is $D$. Hence, given $(A,C)$ have been observed, there is a chance that the following event is either $D$ or $E$. Hence, for $k = 2$, there is a random element in the model. For $k \geq 3$, there is no common subsequence with a different following symbol in the two sequences and the model degenerates into a non-probabilistic collection of sequences.

### 3.1.3. Prediction by Partial Match Models

Usage profiles based on PPM models provide a way to combine MMs with a low order and higher order MMs in order to mitigate the drawbacks of both types of models. However, we do not use a PPM variant from data compression, but rather define our own PPM variant, tailored to the needs of usage-based testing. The rational for this decision is that the opt-out used in data compression (a subsequence has not been observed yet, see Section 2.1.4) does not make sense for software testing. Instead, we use a fixed probability $0 < \varepsilon < 1$ for all sequences. Because of the fixed probability, an opt-out is always possible and we are able to write the definition of our PPM variant as a closed expression without cases, like in Definition 2.17.

**Definition 3.1 (Prediction by Partial Match, Variant 1)** *Let $k_{max} \in \mathbb{N}$ the PPM order and $\varepsilon \in (0,1)$ the fixed opt-out probability. The probability mass function of PPM, Variant 1 is then defined as*

$$\begin{aligned}
&\mathbf{Pr}_{PPM}\{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_{n-k+1} = x_{n-k+1}\} \\
&= \left( \sum_{i=k}^{1} \varepsilon^{k-i} \mathbf{Pr}_{MM^i}\{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_{n-i+1} = x_{n-i+1}\}(1-\varepsilon) \right) \\
&+ \varepsilon^k \mathbf{Pr}_{MM^0}\{X_{n+1} = x_{n+1}\}.
\end{aligned} \tag{3.1.1}$$

**Lemma 3.1** *Definition 3.1 describes a model that possesses the $k_{max}$-th order PPM property.*

**Proof** *Since we always opt-out, we only need to show, that Equation 3.1.1 matches the second case of Equation 2.1.14. We define*

$$\begin{aligned}
&\hat{\mathbf{Pr}}_k\{escape | X_n = x_n, \ldots, X_{n-k+1} = x_{n-k+1}\} \\
&= \varepsilon^{k_{max}-k} \mathbf{Pr}_{MM^k}\{X_{n+1} = x_{n+1} | X_n, \ldots, X_{n-k+1}\}(1-\varepsilon) \\
&+ \mathbf{Pr}_{k-1}\{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_{n-(k-1)+1}\}
\end{aligned} \tag{3.1.2}$$

*and*

$$\hat{\mathbf{Pr}}_0\{X_{n+1} = x_{n+1}\} = \varepsilon_{max}^k \mathbf{Pr}_{MM^0}\{X_{n+1} = x_{n+1}\} \tag{3.1.3}$$

*We expand Equation 3.1.2 for $k = k_{max}$ by inserting $\mathbf{Pr}_{k-1}, \mathbf{Pr}_{k-2}, \ldots$ and get*

$$\hat{\mathbf{Pr}}_{k_{max}}\{escape | X_n = x_n, \ldots, X_{n-k_{max}} = x_{n-k_{max}+1}\} \tag{3.1.4}$$
$$= \varepsilon^{k_{max}-k_{max}} \mathbf{Pr}_{MM^k_{max}}\{X_{n+1} = x_{n+1} | X_n, \ldots, X_{n-k_{max}+1}\}(1-\varepsilon) \tag{3.1.5}$$
$$+ \varepsilon^{k_{max}-(k_{max}-1)} \mathbf{Pr}_{MM^{k_{max}-1}}\{X_{n+1} = x_{n+1} | X_n, \ldots, X_{n-(k_{max}-1)+1}\}(1-\varepsilon) \tag{3.1.6}$$
$$+ \mathbf{Pr}_{k_{max}-2}\{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_{n-(k_{max}-2)+1}\} \tag{3.1.7}$$
$$= \ldots \tag{3.1.8}$$
$$= \varepsilon^{k_{max}-k_{max}} \mathbf{Pr}_{MM^k_{max}}\{X_{n+1} = x_{n+1} | X_n, \ldots, X_{n-k_{max}+1}\}(1-\varepsilon) \tag{3.1.9}$$
$$+ \varepsilon^{k_{max}-(k_{max}-1)} \mathbf{Pr}_{MM^{k_{max}-1}}\{X_{n+1} = x_{n+1} | X_n, \ldots, X_{n-(k_{max}-1)+1}\}(1-\varepsilon) \tag{3.1.10}$$
$$+ \ldots \tag{3.1.11}$$
$$+ \varepsilon^{k_{max}-(k_{max}-(k_{max}-1)} \mathbf{Pr}_{MM^{k_{max}-(k_{max}-1)}}\{X_{n+1} = x_{n+1} | X_n, \ldots, X_{n-(k_{max}-(k_{max}-1))+1}\}(1-\varepsilon) \tag{3.1.12}$$
$$+ \mathbf{Pr}_{k_{max}-k_{max}}\{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_{n-(k_{max}-k_{max})}\} \tag{3.1.13}$$

$$= \mathbf{Pr}_{MM^k_{max}}\{X_{n+1} = x_{n+1} | X_n, \ldots, X_{n-k_{max}+1}\}(1-\varepsilon) \tag{3.1.14}$$
$$+ \varepsilon \mathbf{Pr}_{MM^{k_{max}-1}}\{X_{n+1} = x_{n+1} | X_n, \ldots, X_{n-k_{max}+2)}\}(1-\varepsilon) \tag{3.1.15}$$
$$+ \ldots \tag{3.1.16}$$
$$+ \varepsilon^{k_{max}-1} \mathbf{Pr}_{MM^1}\{X_{n+1} = x_{n+1} | X_n\}(1-\varepsilon) \tag{3.1.17}$$
$$+ \mathbf{Pr}_0\{X_{n+1} = x_{n+1}\} \tag{3.1.18}$$
$$= \left( \sum_{i=k_{max}}^{1} \varepsilon^{k_{max}-i} \mathbf{Pr}_{MM^i}\{X_{n+1} = x_{n+1} | X_n, \ldots, X_{n-i+1}\}(1-\varepsilon) \right) \tag{3.1.19}$$
$$+ \mathbf{Pr}_0\{X_{n+1} = x_{n+1}\} \tag{3.1.20}$$
$$= \left( \sum_{i=k_{max}}^{1} \varepsilon^{k_{max}-i} \mathbf{Pr}_{MM^i}\{X_{n+1} = x_{n+1} | X_n, \ldots, X_{n-i+1}\}(1-\varepsilon) \right) \tag{3.1.21}$$
$$+ \varepsilon^k_{max} \mathbf{Pr}_{MM^0}\{X_{n+1} = x_{n+1}\} \tag{3.1.22}$$

□

As Equation 3.1.1 shows, the $k_{max}$-th order MM has the highest influence. The influence of the lower order models drops exponentially, i.e., the probabilities of the $(k_{max} - i)$-th order MM is only $\varepsilon^i$. This leads to the fact that with probability $\varepsilon^k$ the 0-th order MM is used, which is random drawing without any account for the history at all. Hence, if $\mathbf{Pr}_{MM^0}\{X = x\} > 0$ for all $x \in \mathscr{X}$, every sequences $x_1, x_2, \ldots$ with $x_i \in \mathscr{X}, i = 1, 2, \ldots$ has a positive probability and is, therefore, possible. In usage profiles $\mathbf{Pr}_{MM^0}\{X = x\} > 0$ is always the case, as only observed events are part of $E_U$ and every observed event has a positive probability. However, for testing it is not desirable that every event sequence is possible. On the contrary, it would mean that the usage profile does not account for the SUT's state at all. To counter this effect, we use the two orders $k_{max}$ and $k_{min}$ to define a second PPM variant.

**Definition 3.2 (Prediction by Partial Match, Variant 2)** *Let $k_{max}, k_{min} \in \mathbb{N}$, $k_{min} < k_{max}$ and $\varepsilon \in (0,1)$ the fixed opt-out probability. The probability mass function of PPM, Variant 2 is then defined as*

$$\begin{aligned} &\mathbf{Pr}_{PPM}\{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_{n-k} = x_{n-k_{max}}\} \\ &\quad = \left( \sum_{i=k_{max}}^{k_{min}} \varepsilon^{k-i} \mathbf{Pr}_{MM^i}\{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_{n-i} = x_{n-i}\}(1-\varepsilon) \right) \\ &\quad + \varepsilon^{k_{max}-k_{min}} \mathbf{Pr}_{MM^{k_{min}}}\{X_{n+1} = x_{n+1}\}. \end{aligned} \tag{3.1.23}$$

**Lemma 3.2** *Definition 3.2 describes a model that posses the $k_{max}$-th order PPM property.*

**Proof** *The proof is analog to the proof of Lemma 3.1, except that we define*

$$\mathbf{Pr}_{k_{min}} = \varepsilon^{k_{max}-k_{min}} \mathbf{Pr}_{MM^{k_{min}}} \{X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_{n-k_{min}+1} = x_{n-k_{min}+1}\} \quad (3.1.24)$$

*instead of Equation 3.1.3.* □

The second PPM variant is the same as the first, except that we built in a restriction $k_{min}$ that is used to abort the opt-out, thereby ensuring a minimal Markov order of $k_{min}$ to prevent the model from allowing a probability that the next event is randomly drawn. In terms of complexity, the PPM internally maintains MMs with orders $k_{min}, \ldots, k_{max}$, i.e., models with complexities $O(|E_U|^{k_{min}+1}), \ldots, O(|E_U|^{k_{max}+1})$. Because $O(|E_U|^{k_{min}+1}) \in O(|E_U|^{k_{min}+2}) \in \ldots \in O(|E_U|^{k_{max}+1})$ the overall complexity of the second PPM variant is $O(|E_U|^{k_{max}+1})$.

In the remainder of this thesis, we refer to PPM, variant 2 when we speak of PPM. The reason for this is that the second variant has the exact properties that we desire for a stochastic process that combines higher and lower order MMs. The parameter $k_{max}$ defines the maximum Markov order, i.e., the maximal length of preconditions that are captured. The parameter $k_{min}$ defines the minimal Markov order, i.e., the minimal length of subsequences with different followers required to allow randomness. The parameter $\varepsilon$ is a steering parameter that defines probability of the lower orders of the model.

In general, the PPM has the same the drawbacks of lower and higher order MM. For example, all sequences possible in a $k_{min}$-th order MM are possible in a PPM. This means, invalid sequences due to missed preconditions in the $k_{max}$-th order MM are also possible in the PPM. Furthermore, the PPM has the high complexity of the $k_{max}$-th order MM, but without the guarantee for the preconditions. However, the big advantage of PPM models is that the strength and weaknesses of the different MM orders are combined and their influence is steered by the selection of PPM parameters. For example, a very small value for $\varepsilon$ means that almost always the $k_{max}$-th order MM is used and the $k_{min}$-th order MM is very improbable. Hence, such a PPM is similar to a $k$-th order MM, but the probability of overfitting is mitigated because with probability $\varepsilon$ a lower order MM is used, where more sequences are possible. A further example is a very high value for $\varepsilon$. In this case, the lower order models are more probable, leading to a usage profile in which preconditions are sometimes respected and sometimes ignored, which is beneficial if the length of preconditions varies very much or is unknown.

## 3.2. Usage Analysis

A task that is not directly testing, but rather related to the development of the SUT in general, is the usage analysis of the SUT. This includes the evaluation how often features of the SUT are used, which features are ignored, or how diverse the usage of the SUT is. The usage data that we gather and the usage profiles that we calculate are well suited for this analysis. We observe the usage of a system's functionality indirectly through the events that are executed.

To this aim, we propose two different ways for the usage analysis. The first is based on the observed data. Here, we count for each observed event, how often it occurs. The higher the count, the more often the feature associated with the event is used. The second approach is based on usage profiles. The stationary distribution of the underlying stochastic process describes the probability that an event is executed if the amount of time the SUT is executed goes to infinity. Hence, the higher the probability of an event execution in the stationary distribution, the more often its associated feature is used.

While we are not able to determine the stationary distribution for arbitrary stochastic processes, we are able to do so for a specific kind of first-order MM (see Theorem 2.1). Our usage profiles have a non-recurrent state with the *START* symbol. However, we change our usage profile into a recurrent one as follows: we add a transition from *END* to *START* with probability 1. Since every state is reachable from *START*, and *END* is reachable from any state with positive probability in a finite number of steps because of the training, we have an irreducible first-order MM. Otherwise, our model would allow an execution that does not terminate or does not start at the beginning, both of which are impossible. From a software modeling point of view, the additional transition from *END* to *START* is not a problem. We interpret it as a user starting a new session after finishing the last one, e.g., in case of GUI applications closing the application and starting it again at a later point in time. We cannot state in a generalized fashion if our usage profiles are aperiodic. This has to be determined for every usage profile individually. However, if this is the case, we can calculate the stationary distribution of our usage profile.

## 3.3. Usage-based Coverage Criteria

Coverage criteria are a means to analyze the completeness of test suites. In the literature, the depth $d$ event coverage is used for this purpose (Section 2.4.2). For the calculation of the depth $d$ event coverage, we need the possible subsequences of length $d$ $S_d$ that are allowed by the usage profile and the subsequences covered by test suite $S_{test,k}$. The latter can be straight forward extracted from the test suite by analyzing the subsequences contained in each test case. For the determination of $S_d$, we use the fact that a subsequence is possible if it has a positive probability. Let $\{X_i\}$ be a usage profile and $d$ the coverage depth. Then, the possible subsequences of length $d$ are

$$\begin{aligned} S_d = \{(e_1,\ldots,e_d) : \\ &\mathbf{Pr}\{X_{n+1} = e_d | X_n = e_{d-1},\ldots,X_{n-d+1} = e_1\} > 0, \\ &e_1, e_d \in E_U \cup \{START, END\}\}. \end{aligned} \tag{3.3.1}$$

The drawback of using the event coverage for test suites determined with a usage-based approach is that the usage-based approach does not consider event coverage. In a usage-based approach often used components are more important than seldom used components

and should, therefore, have a higher weight when analyzing the coverage. However, the standard event coverage weights are the same for every event sequences, regardless of its usage. As a remedy, we propose *usage-based event coverage*. The idea of usage-based coverage criteria is to modify the coverage such that not all entities, in our case subsequences, are weighted equally. Instead, usage profiles determine the weight of each entity: the more probable the usage, the higher the weight. Based on this notion, the usage-based event coverage is defined as follows.

**Definition 3.3 (Usage-based Event Coverage)** *Let $\{X_i\}$ be a usage profile with joint probability mass function $p(e_1, \dots e_n)$. Let $S_d$ be the set of all length $d$ event sequence of an SUT that are a subsequence of any valid sequence of events. Let $S_{test}$ a test suite and $S_{test,d}$ all length $d$ event sequences covered by $S_{test}$. We define the* usage based depth $d$ event coverage *of $S_{test}$ as*

$$UsageCov_d(S_{test}) = \frac{\sum_{s_{cov} \in S_{test,d}} p(s_{cov})}{\sum_{s_{all} \in S_d} p(s_{all})} \tag{3.3.2}$$

As an example for the difference between standard and usage-based coverage, consider the test cases $(START, A, D, END)$ and $(START, B, D, END)$ for the usage profile depicted in Figure 2.12. Figure 3.3 shows the depth two coverage of the test cases. Both test cases have the same depth two standard event coverage, because both test cases cover three edges of the MM. However, the usage-based coverage of the test case $(START, B, D, END)$ is higher, because the probability mass of the covered edges is higher than the probability mass of the test case $(START, A, D, END)$. This is exactly the effect we desire for our usage-based coverage criteria. Event sequences that are more likely to be executed should have higher weights and, therefore, have a greater influence on the coverage.

The definition of usage-based coverage is both flexible and powerful. While we only consider MMs and PPM models in this work, the Definition 3.3 is defined for arbitrary stochastic processes. As long as the joint probability mass function of the process is available, we can calculate the usage-based coverage. The results of the usage-based coverage are valid for the complete SUT even if the usage profile is only defined over an incomplete subset of the SUT, i.e., only the observed events. All sequences, that are not possible in the incomplete SUT model described by the usage profile, but are possible in the SUT have a probability of zero in the usage profile. Therefore, these sequences do not influence the result of Equation 3.3.2 and consequently have no impact on the usage-based coverage. Therefore, even if the usage-based coverage is calculated over an incomplete model of the SUT, the coverage remains the same in relation to the (possibly unknown) complete model.

## 3.4. Session Generation

The generation of sessions to be used as test cases is one of the main tasks of usage-based testing. In the literature, this is done by random walks on the model (Section 2.4.3 for

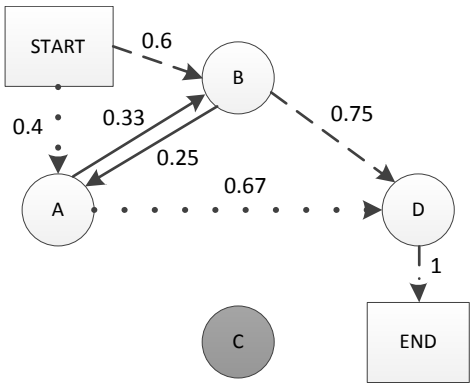



Figure 3.3.: A first-order MM. The dotted edges visualize the depth two coverage of the test case $(START, A, D, END)$ and the dashed edges of the test case $(START, B, D, END)$

.

a description of random walks and Section 7.1 for the related work). In this thesis, we define three novel approaches for usage-based test case generation, which we present in the following.

### 3.4.1. Drawing from All Possible Sequences

When a sequence of a given length is desired, random walks on the usage profile until a sequence of the desired length is found is inefficient. Instead, it makes more sense to consider only the sequences of the desired length, which are not possible with random walks. To this aim, the first step is to determine all sequences that are possible according to the usage profile. This is equivalent to the sequences that have a positive probability. Hence, for a usage profile $\{X_i\}$, the possible sequences of length $l$ that start with the *START* event and finish with the *END* event are defined as

$$\begin{aligned} S_{all,l} = \{(START, \hat{e}_1, \ldots, \hat{e}_l, END) : \\ \mathbf{Pr}\{X_{n+1} = END | X_n = \hat{e}_l, \ldots, X_{n-l} = \hat{e}_1, X_{n-l-1} = START\} > 0 \\ \hat{e}_1, \ldots, \hat{e}_l \in E_U\} \end{aligned} \tag{3.4.1}$$

and the sequences that start with the *START* event and finish with an arbitrary event are defined as

$$\begin{aligned} S'_{all,l} = \{(START, \hat{e}_1, \ldots, \hat{e}_l) : \\ \mathbf{Pr}\{X_{n+1} = \hat{e}_l | X_n = \hat{e}_{l-1}, \ldots, X_{n-l} = \hat{e}_1\} > 0 \\ \hat{e}_1, \ldots, \hat{e}_l \in E_U\} \end{aligned} \tag{3.4.2}$$

The second step is the selection of a concrete sequence from the set of all sequences. We utilize the joint probability mass function $p(e_1,\ldots,e_n)$ of $\{X_i\}$ to select sequences. The joint probability mass function $p(e_1,\ldots,e_n)$ of $\{X_i\}$ induces a probability distribution on $S_{all,l}$ and $S'_{all,l}$. We then select sequences by drawing according to this induced distribution. From the AEP follows, that we will draw a typical sequence with a high likelihood (Theorem 2.5). The test case generation procedure is formalized by algorithms 3 and 4. Because we generate all possible sequences and draw from them, we refer to this test case generation approach as the *all-possible approach*. In order to generate a test suite with the all possible approach, we apply Algorithm 5.

**Input** : Usage profile $\{X_i\}$, desired test case length $l$
**Output**: Sequence $(START, e_1, \ldots, e_l, END)$
1 Generate all sequences all valid sequences $S_{all,l}$ (Equation 3.4.1);
2 Draw $(START, e_1, \ldots, e_l, END)$ from $S_{all,l}$ according to the distribution $p(s), s \in S_{all,l}$;

**Algorithm 3**: Test case selection from all possible sequences with valid end.

**Input** : Usage profile $\{X_i\}$, desired test case length $l$
**Output**: Sequence $(START, e_1, \ldots, e_l)$
1 Generate all sequences all valid sequences $S'_{all,l}$ (Equation 3.4.2);
2 Draw $(START, e_1, \ldots, e_l)$ from $S'_{all,l}$ according to the distribution $p(s), s \in S_{all,l}$;

**Algorithm 4**: Test case selection from all possible sequences with an arbitrary end.

**Input** : Test suite size $m$, test case length $l$
**Output**: Test suite $S_{test}$
1 $m_{cur} \leftarrow 0$;
2 $S_{test} \leftarrow \emptyset$;
3 **while** $m_{cur} < m$ **do**
4  Generate session $s$ of length $l$ with the test case generation algorithm;
5  **if** $s \notin S_{test}$ **then**
6  $S_{test} \leftarrow S_{test} \cup \{s\}$;
7  $m_{cur} \leftarrow m_{cur} + 1$
8  **end**
9 **end**

**Algorithm 5**: Test suite generation algorithm

### 3.4.2. Hybrid Approach

The all-possible approach has one major drawback: the number of valid sequences grows exponentially in $l$. Thus, the approach is not scalable for larger memory lengths and, as our case studies show (see Section 6), it is not reasonably possible to even generate test cases of length five. We propose a test case generation approach that is a *hybrid* of the all-possible approach and random walks. The general idea is to generate all possible sequence up to an achievable possible length $l_{possible}$ and then finish the sequence by random walking to generate a test case of the desired length $l$. Algorithms 6 and 7 describe the hybrid approach.The algorithms directly call the algorithms of the all-possible approach. The second part of the algorithms, i.e., lines 5-12 are almost identical to the random walk algorithms 1 and 2, respectively. The only difference is that instead of starting at the beginning, the walk starts at the end of the sequence drawn with Algorithm 4.

When it comes to practical applications of the test case generation algorithms, the hybrid approach is an extension of the all-possible approach. For desired lengths $l \leq l_{possible}$ the hybrid approach is exactly like the all-possible approach. For $l > l_{possible}$, it is not possible to apply the all-possible approach. Therefore, in all instances where the all-possible approach is applicable, it behaves the same as the hybrid approach.

**Input** : Usage profile $\{X_i\}$, maximum achievable length to generate all sequences $l_{possible}$, desired test case length $l$
**Output**: Sequence $(START, e_1, \ldots, e_l, END)$

1 **if** $l_{possible} \geq l$ **then**
2 $\quad (START, e_1, \ldots, e_l, END) \leftarrow$ Algorithm 3$(\{X_i\}, l)$;
3 **else**
4 $\quad (START, e_1, \ldots, e_{l_{possible}}) \leftarrow$ Algorithm 4$(\{X_i\}, l_{possible})$;
5 $\quad i \leftarrow l_{possible}$;
6 $\quad$ **while** $e_i ! = END$ **do**
7 $\quad\quad$ Draw $e_{i+1}$ according to $\mathbf{Pr}\{X_{i+1} = e_{i+1} | X_i = e_i, \ldots, X_0 = e_0\}$;
8 $\quad\quad i \leftarrow i + 1$;
9 $\quad$ **end**
10 $\quad$ **if** $i - 1 \neq l$ **then**
11 $\quad\quad$ Goto line 5;
12 $\quad$ **end**
13 **end**

**Algorithm 6**: Hybrid test case generation method with a valid end.

**Input** : Usage profile $\{X_i\}$, maximum achievable length to generate all sequences $l_{possible}$, desired test case length $l$
**Output**: Sequence $(START, e_1, \ldots, e_l)$

```
1  if l_possible ≥ l then
2  |  (START, e_1, ..., e_l) ← Algorithm 4({X_i}, l);
3  else
4  |  (START, e_1, ..., e_{l_possible}) ← Algorithm 4({X_i}, l_possible);
5  |  i ← l_possible;
6  |  while e_i != END do
7  |  |  Draw e_{i+1} according to Pr{X_{i+1} = e_{i+1} | X_i = e_i, ..., X_0 = e_0};
8  |  |  i ← i + 1;
9  |  end
10 |  if i ≤ l then
11 |  |  Goto line 5;
12 |  end
13 end
```

**Algorithm 7**: Hybrid test case generation method with an arbitrary end.

### 3.4.3. Heuristic

The test case generation approach that where presented in the previous sections are all randomized and generate test cases according to the probabilities defined by the usage profile to achieve a distribution of test cases that reflects the usage and optimizes the user-based quality. The availability the of usage-based coverage criteria allows us to generate test cases in a target oriented manner. In this section, we define a heuristic to determine a test suite that achieves a desired usage-based coverage with minimal size.

Our heuristic uses a greedy strategy to determine a test suite $S_{test} \subset S_{all,l}$. Let $S_{test}$ be the current test suite and $d$ the coverage depth for which the test suite is optimized. We update the test suite with another sequence $s$, such that the increase in coverage $UsageCov_d(S_{test} \cup s)$ is maximized, i.e., $\arg\max_{s \in S_{all,l}} UsageCov_d(S_{test} \cup s)$. Hence, we select $s$ such that we gain a maximum of coverage. With the help of the definition of usage-based coverage

(Definition 3.3), we can reformulate this maximization problem as follows.

$$\begin{aligned}
\arg\max_{s \in S_{all,l}} UsageCov_d(S_{test} \cup s) &= \arg\max_{s \in S_{all,l}} \frac{\sum_{s_{cov} \in (S_{test,d} \cup s_d)} p(s_{cov})}{\sum_{s_{all} \in S_d} p(s_{all})} \\
&\overset{(1)}{=} \arg\max_{s \in S_{all,l}} \sum_{s_{cov} \in (S_{test,d} \cup s_d)} p(s_{cov}) \\
&\overset{(2)}{=} \arg\max_{s \in S_{all,l}} \sum_{s_{cov} \in S_{test,d}} p(s_{cov}) + \sum_{s_{cov} \in s_d \setminus S_{test,d}} p(s_{cov}) \\
&\overset{(3)}{=} \arg\max_{s \in S_{all,l}} \sum_{s_{cov} \in s_d \setminus S_{test,d}} p(s_{cov})
\end{aligned} \tag{3.4.3}$$

The simplifications (1) and (3) are valid, because $S_{all,l}$ and $S_{test,d}$ are fixed and, therefore, $\sum_{s_{all} \in S_d} p(s_{all})$ and $\sum_{s_{cov} \in S_{test,d}} p(s_{cov})$ are constant and do not influence the maximum. (2) follows from $S_{test,d} \cup s_d = S_{test,d} \cup (s_d \setminus S_{test,d})$ and the fact that $S_{test,d}$ and $(s_d \setminus S_{test,d})$ are disjunctive.

By Equation 3.4.3, we defined what exactly we select in a greedy way: a sequence such that its subsequences maximizes the gain in the covered probability mass. For the heuristic, we start with $S_{test} = \emptyset$ and continously select the next sequence using Equation 3.4.3 to expand $S_{test}$ until we have $UsageCov_d(S_{test}) \geq cov_{desired}$. Algorithms 8 and 9 describe the whole test suite generation procedure.

**Input** : Usage profile $\{X_i\}$, test case length $l$, coverage depth $d$, desired coverage $cov_{desired}$
**Output**: Test suite $S_{test}$

```
S_test ← ∅;
cov_cur ← 0;
Generate all sequences all valid sequences S_all,l (Equation 3.4.1);
while cov_cur < cov_desired do
    s ← arg max_{s∈S_all,l} Σ_{s_cov∈s_d\S_test,d} p(s_cov) with s_d, S_test,d the length d subsequences
    covered by s, respectively S_test;
    S_test ← S_test ∪ s;
    cov_cur ← UsageCov_d(S_test);
end
```

**Algorithm 8**: Greedy heuristic for test suite generation with a valid end

**Input** : Usage profile $\{X_i\}$, test case length $l$, coverage depth $d$, desired coverage $cov_{desired}$
**Output**: Test suite $S_{test}$

1 $S_{test} \leftarrow \emptyset$;
2 $cov_{cur} \leftarrow 0$;
3 Generate all sequences all valid sequences $S'_{all,l}$ (Equation 3.4.2);
4 **while** $cov_{cur} < cov_{desired}$ **do**
5 $\quad s \leftarrow \arg\max_{s \in S'_{all,l}} \sum_{s_{cov} \in s_d \setminus S_{test,d}} p(s_{cov})$ with $s_d, S_{test,d}$ the length $d$ subsequences covered by $s$, respectively $S_{test}$;
6 $\quad S_{test} \leftarrow S_{test} \cup s$;
7 $\quad cov_{cur} \leftarrow UsageCov_d(S_{test})$;
8 **end**

**Algorithm 9**: Greedy heuristic for test suite generation with an arbitrary end

# 4. A Framework for Usage-based Testing of Event-driven Software

In this chapter, we define a platform-independent framework for EDS testing, with focus on the applicability of the framework for usage-based testing. In the following, we start with the outline of the structure of the framework in Section 4.1, before we describe the details of the framework in sections 4.2 to 4.4.

## 4.1. Outline of the Framework

The general idea of our EDS testing framework is the usage of layers to abstract from specific target platforms and gain platform independence. To this aim, we define three layers: a *platform layer*, a *translation layer*, and an *event layer*. The platform layer contains everything that is platform specific, e.g., capture/replay tools designed for a specific platform. Hence, the platform layer works on platform events, e.g., Windows messages (Section 2.3.1.1). The event layer contains everything that is platform independent, e.g., test generation methods and the tools to train and work with usage profiles. To achieve platform independence, the event layer works on abstract events, as we defined in Definition 2.24. The task of the translation layer is to mediate between the event and the platform layer, i.e., to translate platform specific events into abstract events and vice versa. Figure 4.1 gives an overview of the layers and their components. Figure 4.2 depicts the framework in greater detail including the generated data and the interactions between the components.

The platform layer consists of two components that need to work together with the SUT. The *monitor* is responsible for observing the SUT and records its usage. As a result, the monitor generates *platform-specific event sessions*. The *executor* gets a *concrete test suite* as input and stimulates the SUT according to the test cases that are executed as part of the test suite.

On the *translation layer*, two components are responsible for the translation between the event layer and the platform layer. The *event parser* takes platform-specific event sessions as input and converts them into *abstract event sessions*. The *replay generator* takes *generated abstract event sessions* with or without assertions as input and translates them into a concrete test suite to be used by an executor on the platform layer.

The event layer has five components. The *model builder* creates a *SUT testing model* from abstract event sessions. In this thesis, this means that it creates the usage profiles. However, other testing models are also possible, albeit we do not consider them. The *usage*

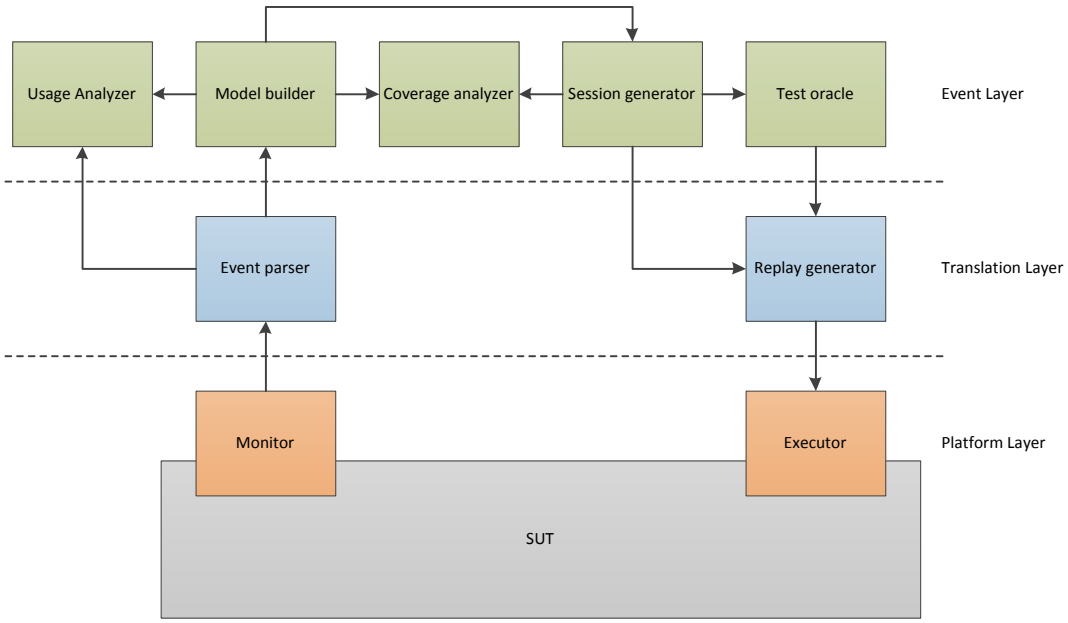



Figure 4.1.: This figure visualizes the three layers of our testing framework and the components of each layer.

*analyzer* uses abstract event sessions and SUTs testing models as input to generate usage reports, e.g., which events occur most often. The *session generator* uses the SUT testing model to *generate abstract event sessions*, which are the foundation of a generated test suite. The *coverage analyzer* calculates the coverage of the generated abstract event sessions with respect to the SUT testing model. With the *test oracle*, the generated abstract event sessions can be extended *with assertions*.

## 4.2. Platform Layer

The platform layer is at the bottom of the framework and it is the only layer with access to the SUT. Concrete implementations of platform layer components usually depend to a high degree on the API of the EDS platform on which the SUT is implemented. The only assumption we make about the platform layer components is that the monitor creates a log of platform-specific events in an accessible format and, vice versa, the executor uses a format for test suites that the replay generator can generate.

### 4.2.1. Monitoring of Event-driven Software

The monitor needs direct access to the SUT. There are three ways for monitoring SUTs.

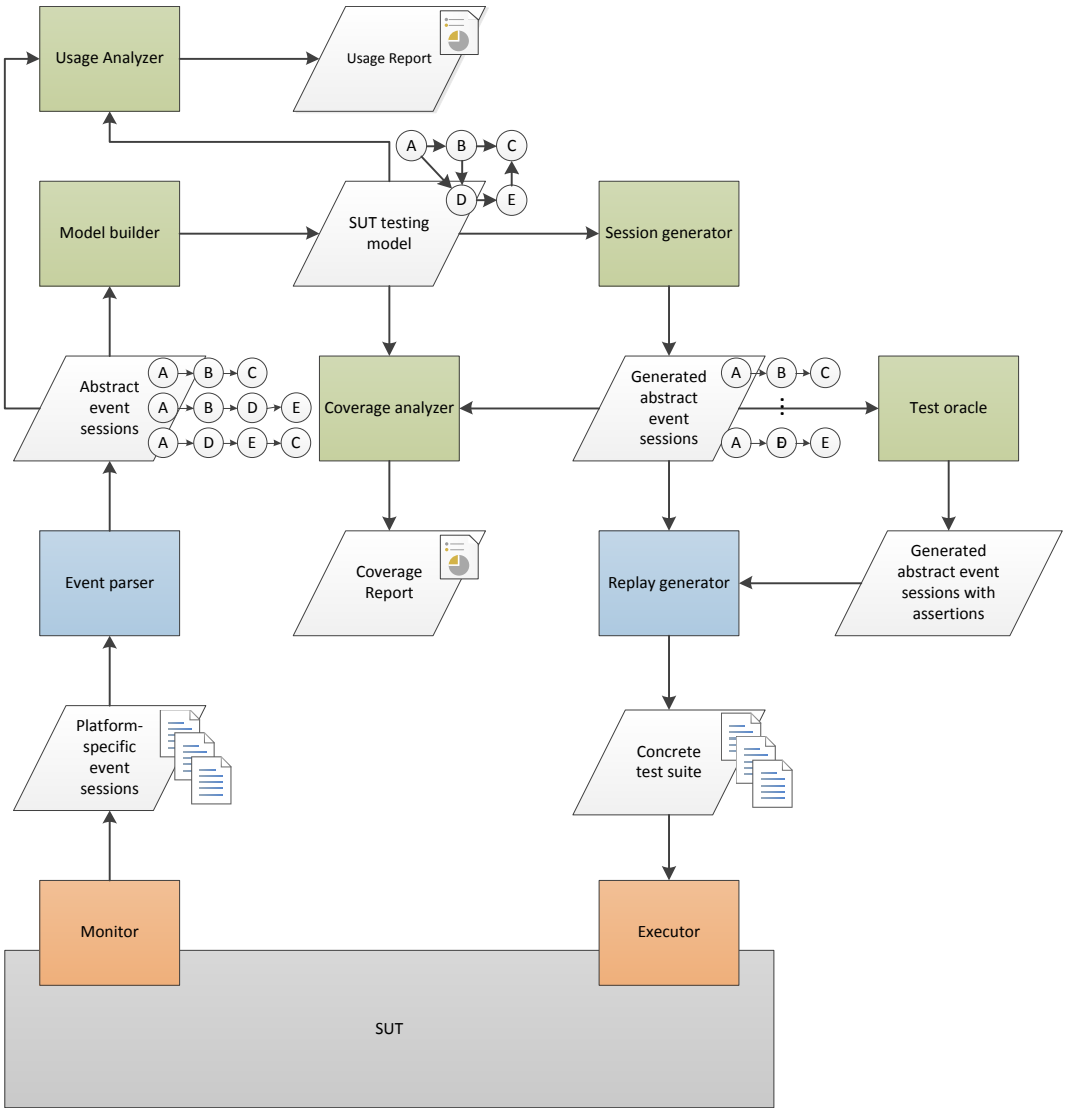
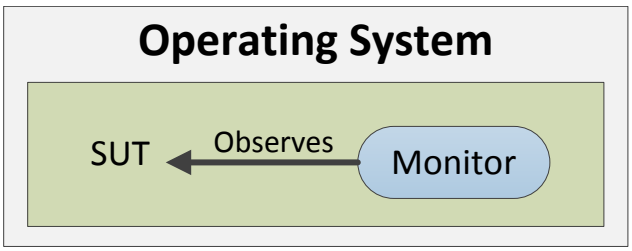
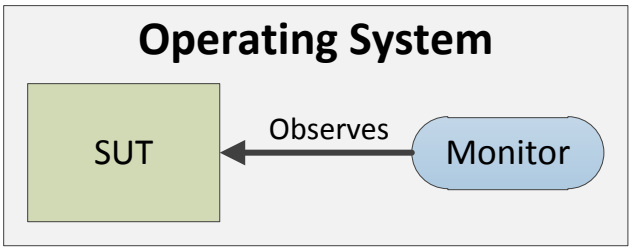



Figure 4.2.: This figures visualizes the components of the framework and the data they exchange. Colors of the components indicate their layer: green for the event layer, blue for the translation layer, and orange for the platform layer.




(a) Internal monitor.




(b) External monitor.

Figure 4.3.: Internal vs. external monitoring (from [46]).

1. The monitor is directly integrated into the SUT
2. The SUT provides a monitoring API that is accessible for the development of monitoring tools.
3. The EDS platform provides a monitoring API that is sufficiently powerful to observe the SUTs usage.

We differentiate between *internal* and *external* monitoring. We call tools that monitor the SUT from the outside external monitors and tools that are integrated into the SUT internal monitors. The first type of monitor we described is internal, while the second and third type are external. Figure 4.3 visualizes the difference between internal and external monitoring with the Point of Observation (PoO) of the monitor. In the internal monitor (Figure 4.3a), the PoO and the monitor are inside the SUT, while the PoO of an external monitor (Figure 4.3b) is outside of the SUT and the monitor completely separate from the SUT.

The major problem of usage-based testing approaches is to observe the SUTs usage by real users. To do this, we require that the monitor is deployable to the users and it needs to run during the execution of the SUT. With external monitors, this is often problematic. For example, if we use an industrial non-free capture/replay tool for the monitoring, which requires licenses for the capture/replay tool for all users. This is not only very expensive but also impractical, since it requires the users to install software they do not need. Even if this is not the case and we are able to deploy an external monitoring tool to the users, we still require the users to start the tool. This is an additional and optional step for the users and

is, therefore, prone to be omitted. Internal monitors have none of the above problems. If a monitor is internal, it must be deployable. Otherwise, the SUT itself could not be deployed. Furthermore, if the monitor is part of the SUT, it is automatically started by the SUT and, therefore, always runs when the SUT runs.

The advantage of internal monitors is also their drawback. They are highly invasive and raise several security and privacy concerns. A monitor that observes all user interactions including keystrokes potentially monitors sensitive textual information or passwords. Furthermore, the privacy of the users is greatly inhibited, as all their actions are recorded. Therefore, internal monitoring applications need to provide mechanisms to avoid these issues.

The other important difference between the three monitoring types is the flexibility of concrete monitoring tools. Monitoring tools of the second type are the least flexible and only applicable to a concrete SUT, as they depend on the SUT's API. Tools of the first type often suffer from the same problem. The third type is the most flexible. This is due to the fact, that the third type only depends on the EDS' API and is, therefore, applicable to all SUTs built on top of the EDS platform. In case tools of the first type only utilize the EDS' API, they can be as flexible as the third type. We present such a tool in Section 5.2.

### 4.2.2. Replaying of Events

In order to execute test cases, we need to execute events against the SUT. This duty is performed by the executor in our framework. In practice, this means that user actions such as mouse clicks or keyboard input are emulated in case of GUI applications and HTTP requests in case of Web applications.

We differentiate between the manual execution by a tester and the automated execution by a tool. Manual execution means that a tester stimulates the SUT by hand using the test case as a script. An example for the automated execution are capture/replay tools, where the events can be automatically executed against the SUT after the capturing. While we desire automated execution, we do not make assumptions about the executor. We only require a clearly defined test suite format that we can automatically generate from abstract events using the replay generator.

## 4.3. Translation Layer

The translation layer does not provide components required for the testing itself (e.g., replaying, test case generation). Instead, it is used to make the platform and event layer independent of each other. Therefore, the platform layer can use already existing tools, e.g., capture/replay tools can be re-used in our framework without requiring changes of the tools. The event layer can, therefore, work with different platforms and does not require changes or adaptations to work with a new platform. Instead, only appropriate translation layer com-

ponents are required. Thus, the translation layer provides our framework with flexibility in terms of the concrete test tooling that is used, both on the platform and event layer.

### 4.3.1. Event Parsing

The event parser works on sequences of platform-specific events and converts them into abstract events for the event layer, i.e., events that are uniquely identified by their type and their target. To this aim, the event parser needs to perform three tasks: noise filtering, abstract event type inference, and abstract event target inference. While especially the type and target inference seem trivial, all three tasks provide significant challenges.

For the specification of the noise filters, we have to define what noise in this context means: platform-specific events that are part of the monitored data but either redundant or not usage events. Both types of noise are generated because of insufficiencies of monitoring tools. Redundant events are multiple recordings for the same usage event, e.g, a *MOUSEDOWN*, *MOUSEUP* event pair and a *MOUSEPRESSED* event, both of which signify that a mouse button is pressed. If both were to be used to create abstract events, the mouse click would be duplicated and the resulting abstract event sequence is falsified. Messages that are not usage events are easier to deal with. These are messages that are part of the monitored information, e.g., to include the structure of a GUI in the monitored data. These messages can be filtered using the type of the platform-specific events.

After the insignificant messages are filtered, the types of the abstract events need to be inferred. Ideally, the type of a platform-specific event is the same as of the required abstract event. However, often this is not the case. Consider the above example of a *MOUSEDOWN*, *MOUSEUP* pair of platform-specific events to signify a mouse click. Directly translated, this leads to two abstract events *MOUSEDOWN*, *MOUSEUP*. While this is a feasible solution, it is desirable to have a *MOUSECLICKED* event instead. Therefore, the event parser needs to identify the pair *MOUSEDOWN*, *MOUSEUP* and convert it into an abstract event *MOUSECLICKED*.

The trivial approach for the target inference is the definition of the target of the abstract event as the target of the platform-specific events. While this works in some cases, e.g., Web applications, it is to simplistic in others. Consider Windows messages as platform specific events. Here, the target is the HWND of the addressed widget. The HWND is only valid for one existence of the widget and will change when it is recreated. Hence, the HWND is unsuited for identifying widgets between multiple sessions and, therefore, unsuited as target for the abstract events. Instead, the event parser needs to determine another description of the target. In case of GUI widgets this can be a combination of widget attributes, e.g., the title and the placement in the GUI hierarchy of the target.

While we require abstract events to be uniquely identified by their type and target, we allow them to have additional payload. This additional payload is ignored on the event layer. The event parser can use this payload to store important platform-specific information with

the abstract events. The replay generator can use the payload to generate a concrete test suite.

### 4.3.2. Replay Generation

The task and challenges of the replay generator are the direct opposite of the event parser. It converts sequences of abstract events into sequences of platform-specific events which define a concrete test suite. To this aim, the replay generator converts the abstract event types and targets back into platform-specific event types and targets as the are required by the executor.

The event type conversion is basically a mapping of an abstract event type to a sequence of platform events. The sequence can have length one, i.e., a one-to-one mapping, e.g., abstract event type "mouse clicked" to a platform event *MOUSEPRESSED*, but one-to-many mappings are also possible, e.g., "mouse clicked" to a pair *MOUSEDOWN*, *MOUSEUP*. Similarly, the abstract target has to be mapped to the target description required by the executor. This may include an extension of the information contained in the abstract target based on the platform-specific payload, pruning of the information of the abstract target, or conversion of the representation format of the abstract target. Whether the generation of platform-specific targets from the abstract target is possible, depends mainly on the event parser, i.e., if the information the event parser included in the abstract event is sufficient to describe the platform-specific event target.

Additionally, assertions that the test oracle potentially added to the abstract event sequences need to be converted into assertions that the executor can evaluate. If the executor is not able to evaluate the defined assertions, the replay generator ignores them and they are omitted from the concrete test suite.

## 4.4. Event Layer

The event layer contains the testing logic of the framework. It has the highest level of abstraction in the framework to make the defined testing approaches broadly available. While the lower layers have no direct relationship to usage-based testing, the event layer contains the components that train and exploit usage profiles. The model builder, usage analyzer, coverage analyzer, and session generator fulfill the tasks of usage-based testing we described in Section 3. Additionally, we provide a test oracle to allow the addition of assertions to test cases for the determination of test verdicts.

### 4.4.1. Test Oracles

The task of a test oracle is to determine the expected reactions of the SUT when test cases are executed. In the previous section, we defined methods to derive event sequences $(e_1, e_2, \ldots)$ from usage models. In order to expand the sequences to EDS test

cases according to Definition 2.39, the test oracle needs to determine the expected results $expected(e_1), expected(e_2), \ldots$. The simplest oracle is, that $expected(e_i)$ is undefined for all $i$. In this case, we speak of a *crash oracle*, meaning that the test case only evaluates if the software crashes in case an event sequence is executed or not.

The problem we face when we define more sophisticated test oracles on the event layer is that the evaluations of expected results, including what can and should be evaluated, depends to a large degree on the EDS platform and the executor that is used. This challenge is similar to the problem of translating between platform-specific and abstract events. Therefore, we suggest a similar solution. We define the abstract expected results that are translated into platform specific expected results by the replay generator. Concretely, we define *assertion events* as a special type of abstract events. Examples for possible assertion events are text comparisons and, for GUI applications, checks if a widget exists. We allow the implementation of any kind of assertion event and in comparison to normal abstract events, we allow that assertion events do not necessarily need a target. We use the allowed payload of abstract events to include the information about the expected result, e.g., concrete expected values.

On the event layer, we treat assertion events the same as other events. They can be placed at any point in a event sequence. Let $(e_1, e_2, e_3)$ be an event sequence and $ae$ be an assertion event. By inserting $ae$ into the sequence after $e_2$, i.e., $(e_1, e_2, ae, e_3)$ we implicitly define the expected reaction $expected(e_2) = ae$. It is possible, that multiple assertion event form an expected result, e.g., in the sequence $(e_1, e_2, ae_1, ae_2, e_3)$ we have $expected(e_2) = (ae_1, ae_2)$. How assertion events are treated for specific EDS platforms is decided by the replay generator (Section 4.3.2).

# 5. Framework Instantiation and Implementation

The framework described in the previous chapter describes how our approach for usage-based EDS testing looks like and which components are required. In order to show the feasibility of our approach, we instantiated the framework and implemented the required components. As part of this chapter, we present a capture/replay approach for Windows MFC GUI applications that we previously published [45, 46]. The description of the capture/replay approach is based on these publications.

## 5.1. Overview

For the instantiation of the framework, we require concrete implementations for each of the framework's components. However, we do not require that each component is implemented in a separate tool or library and allow bundling of components. We entitled our framework instantiation *EventBench* [38]. Altogether, we implemented six different software projects as part of EventBench. All of the projects are available as open source on the EventBench homepage. Figure 5.1 shows an overview of the implemented software projects.

On the platform layer, we provide monitoring components for three different platforms. The *MFCMonitor* [43] for Windows MFC applications, the *JFCMonitor* [41] for Java JFC applications, and the *PHPMonitor* [44] for PHP-based Web applications. Additionally, we provide a replaying tool for Windows MFC applications with *MFCReplay* [42].

The *EventBenchCore* [40] project is implemented as a Java library. The library bundles all functionality of the event layer. Through the implementation of a library, we allow the usage of our EDS testing techniques with any Java application. To access the event-layer functionality, we provide the Java application *EventBenchConsole* [39], which uses the EventBenchCore library internally. However, this is not the only task of the EventBenchConsole. It also contains all translation layer components we implemented. This includes translation layer components for the four platform layer components we implemented ourselves (MFCMonitor, JFCMonitor, PHPMonitor, MFCReplay) and a replay generator the generates output to be used with *curl* [24], a command line tool for sending HTTP requests. Furthermore, we provide a translation layer for the communication with *GUITAR* [63], a software for event-driven software testing.

In the following, we present the implementation in detail. We choose to present the components not layer-wise. Instead, we present the platform and translation layer components

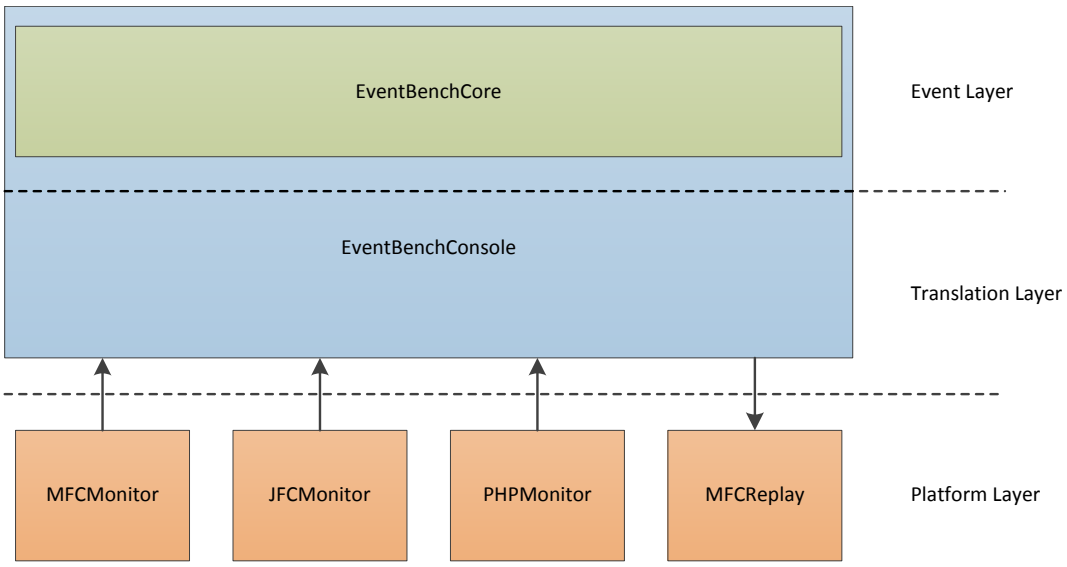



Figure 5.1.: Overview of the components we implemented to instantiate our usage-based EDS testing framework.

grouped by the specific platform for which they are implemented. To understand how our translation layer works, we need to introduce some technical details of our implementations, namely, how we implemented events.

Consider the event class hierarchy depicted in Figure 5.2 and the replayble class hierarchy depicted in Figure 5.3. The abstract events are defined by the generic class `Event<T>`, which has the type and the target of the event as attributes. All classes that implement `Event<T>` are compatible with the event layer. Therefore, the event parser has to generate instances of `Event<T>`. The replay generators are based on the class `ReplayableEvent<T extends IReplayable>` and the interface `IReplayable`. The replayable events contain a list of replayables that is used by the replay generator. Through implementing the interface `IReplayble` it is possible to define how platform and executor specific replays look like, e.g., the class `WindowsMessage` for replayables for MFCReplay.

By subclassing `Event<T>` appropriately it is possible to define events with platform-specific payload. An event that is the subclass of `ReplayableEvent` and with an appropriate implementation of the interface `IReplayable` means that the replay generator is available for this kind of event. The task of the event parser component is the creation of instances of `Event<T>` from platform-specific captures and the task of the replay generator is to query `ReplayableEvents` for their `IReplayables` to generate a concrete test suite.

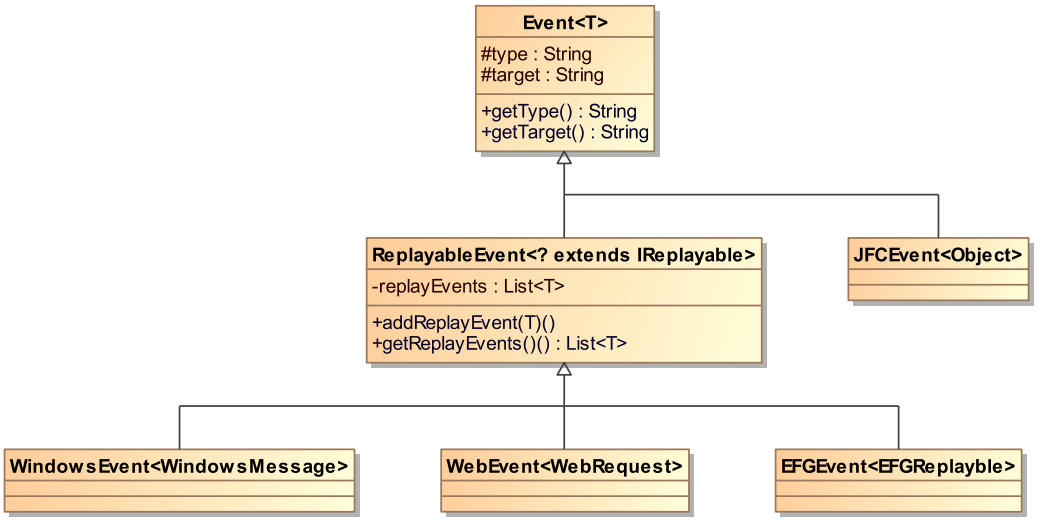



Figure 5.2.: Hierarchy of Java classes used by the EventBenchCore and the EventBench-Console for the representation of events.

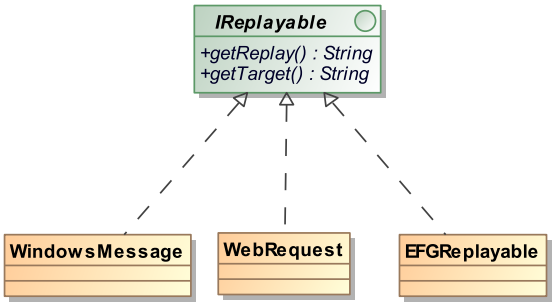



Figure 5.3.: Hierarchy of replayable objects used by the EventBenchCore and the Event-BenchConsole.

## 5.2. Platform and Translation Layer for Windows MFC GUIs

For the platform and translation layer implementation for Windows MFC we designed a complete capture/replay approach specifically for our purposes, i.e., the use in our usage-based framework. Additionally, our approach uses a novel technique for the abstraction of coordinates for coordinate independent capture/replay based on utilizing the *internal* message communication of the SUT. Altogether, we have the following requirements on our approach:

- Coordinate-independent: the captured information should allow coordinate-independent replaying of GUI actions.
- Utilize internal messages: to obtain coordinate independence, internal messages that are indirectly generated through user actions should be used.
- Deployable: the usage monitoring should be deployable with the software to allow users of the software to create bug reports that include the actual behavior patterns that triggered the faulty behavior.
- Easy-to-use: the adaptation of tools using our technique should be simple, and without much effort on the user's part.
- Non-intrusive: the adaptations of tools using our technique should only have a minimal impact on the software to be monitored.
- Usage monitoring: the captures produced by our techniques should be sufficient to infer usage data.
- Easy to disable: the capturing technique must be easy to disable to allow the usage of the software without monitoring.

Existing capturing tools mainly fail the criterion of deployability, which is crucial for the application of a capturing tool to monitor a SUT's usage. Most tools are stand-alone tools that run in parallel to the software that is captured. They therefore have only a loose relationship to the application to be monitored. Furthermore, the capturing tools need to be bought and licensed and cannot be deployed together with the software, unless the costumer also buys licenses for the capturing tools. Additionally, the capture/replay tools are often part of larger software development or quality assurance tool suites, which makes the deployment even more difficult.

### 5.2.1. Usage Monitoring Through the Observation of Windows Messages

For the capturing approach, we defined an internal usage monitor that we implemented as Dynamic Link Library (DLL) in the project MFCMonitor [43]. The DLL provides a lightweight interface to start and stop the monitoring. For the usage monitoring itself, we use the concept of *message hooks* [70]. The idea of hooks is the provision of a programmatic mechanism for the interception of Windows messages. Hooks are procedures that are placed

in the message processing queue of an application or operating system [67]. Generally speaking, there are two different kinds of hooks with different scopes. *System global* hooks intercept messages of all applications and processes, whereas *process hooks* only intercept messages of the process they are installed into. Furthermore, hooks do not monitor all messages. Instead, all hooks are of a type that defines which messages are monitored. Examples for hooks types are `WH_GETMESSAGE`, `WH_CALLWNDPROC`, and `WH_MSGFILTER`.

Hooks provide an elegant way for us to monitor the message communication in a non-intrusive way. We simply need to install a message hook and observe the communication without changing it to have access to the communication of the application. As long as our hook procedure logs messages and does not interfere with the message communication, it is non-intrusive with respect to the remainder of the software, both in terms of functionality and source code changes.

As for the scope of the hooks, we choose the process scope for usage monitoring because of various reasons. First of all, we believe that globally installed hooks would inhibit the deployability of the approach, as they raise many valid security concerns. For example, they can monitor other applications as well, which could mean that sensitive data entered in other applications is accidentally part of the capture and gets leaked. Another drawback of global hooks is that they receive messages from much more processes, i.e., they use a higher amount of computational power and possibly slow down the whole system. Furthermore, if the messages from other applications need to be filtered completely, because they serve no purpose for the capture. In comparison, hooks in the process scope can only monitor the process they have been installed into, thereby automatically preventing data leakage from other applications. Additionally, no filtering of messages from other processes is required as they are not received in the first place. The drawback of process hooks is that they need to be installed into each process of the application, which can increase the effort required to integrate the capturing into an application. If for some reason this is not possible, the capture will be incomplete.

Since hooks are not has been designed to receive exactly the messages that are relevant for capturing GUI executions, we need to select a collection of hooks that receives all required messages. In practice, this means that the collection of hooks will in fact receive a superset of the required messages. For our purpose, the hooks `WH_GETMESSAGE` and `WH_CALLWNDPROC` are necessary and sufficient. Both in combination receive all required messages. However, the two sets of received messages are not disjunctive, i.e., some messages are received twice. Thus, there must be a filter that assures uniqueness for each captured and logged message.

### 5.2.2. Message Filtering

As the hooks receive a superset of the required messages, there are two options: 1) include all messages in the capture; 2) filter the messages. To our minds, including all messages in the capture is infeasible, since the amount of logged data in the capture would explode due to

the overhead of non-relevant messages that are recorded and also unnecessary. An example for messages that should be excluded from the capture are the `WM_GETTEXT` messages. Most widgets are constantly polled for their name by using this message, which rapidly amounts to thousands of such messages. This polling is not relevant for the capture. In a small experiment, where we logged all messages without filtering, we started a test application and performed two mouse clicks. This very small experiment already produced 13 MB of log data. Furthermore, the application was slowed down to a degree, where its usage was infeasible.

We propose a lightweight message filter that utilizes the message type for the filtering. To this aim, we categorized the messages into four sets.

1. Messages that are related to the creation, destruction, or alteration of widgets.
2. Messages that are sent directly as a result of mouse or keyboard input.
3. Messages that are sent indirectly as a result of a user action to trigger the effect of the action internally.
4. Messages that are not required.

The message filter is realized by including messages of categories one to three and excluding category four.

The first category of messages includes information about the current state of the GUI in the capture. Thus, the capture includes which widgets exists, the parent/child relationships between the widgets, the modality of dialogs as well as general information about each widget, e.g., the widget type. Developers can use the information to analyze the life-cycle of widgets, e.g., whether they are created and destroyed properly. Furthermore, the detailed informations that are available about the GUI can be used to identify widgets when replaying the capture, to allow a coordinate independent replay. Examples for messages of this category are `WM_CREATE`, `WM_DESTROY`, and `WM_SETTEXT`, which are sent when a GUI object is created, destroyed, or had its text content changed, respectively.

The second category are the messages that the operating system sends to the application in case of mouse or keyboard activity, e.g., mouse movement, mouse clicks, or keyboard input. Mouse movement itself often has no effect, as most software is driven by mouse clicks and keyboard input only. Therefore, mouse movement is often irrelevant for capturing GUI executions. Furthermore, the replaying of mouse movement cannot be done without relying on screen coordinates, thereby violating one of our requirements. A major drawback is the amount of data required to capture mouse movement, which would drastically increase the size of log files. As a result, we deem capturing mouse movement optional and only suggest it in cases where it is absolutely necessary. Examples for messages of the second category are `WM_LBUTTONDOWN` for “left mouse button down”, and `WM_KEYDOWN` when a key is pressed. For the latter, the key, which is pressed, is one of the parameters.

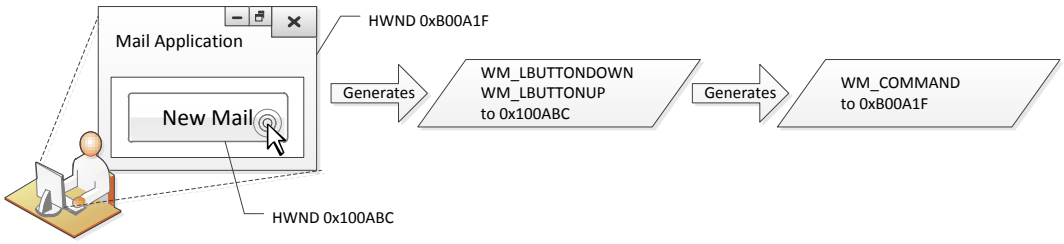



Figure 5.4.: Message generated by a button click (from [46]).

The third category are the reaction of MFC to the user actions, i.e., messages of the second category. The messages are used as internal means to communicate with the widgets and tell them how to react to a user action. For example, a `WM_COMMAND` may be sent as the result of a clicking on a menu item. The `WM_COMMAND` message is in turn handled by the application's command handlers to control its activity. Thus, a capture that includes these messages does not only record how the user interacted with the GUI, but also the internal reactions of the software to the actions. This is the strength of our approach. When generating a replay from our capture, we have the option to directly replay the internal action of the software, instead of the GUI interaction. While this may seem counter intuitive, we argue that our gain is a greater independence of screen coordinates. The second advantage is for developers who analyze the software. They can directly infer from the capture, whether the internal reaction of the software is as they expect, or whether the software reacts in unforeseen ways.

The forth category are the internal communications that are not beneficial for the capture. These are the bulk of messages, e.g., the already mentioned `WM_GETTEXT` message, the `WM_PAINT` message to redraw the widgets, and the `WM_KICKIDLE` message that simply tells the application that there is no apparent task at hand and it may perform some background activity.

Figure 5.4 exemplifies how messages of the second and third category are generated. In the scenario depicted in the figure, the user clicks with the left mouse button on the *New Mail* button. As reaction to this click, the operating system generates the messages `WM_LBUTTONDOWN` and `WM_LBUTTONUP` with the screen coordinates of the current mouse position and sends them to the widget on this position, i.e., to the *New Mail* button by addressing the messages to the HWND of the button. The MFC framework code for the widget translates the two messages into a click on the button (not depicted), and generates an internal `WM_COMMAND` messages that is sent to the parent of the *New Mail* button, i.e., the *Mail Application* dialog. The `WM_COMMAND` message is then processed by the command handler of the *Mail Application* dialog that executes the desired user action.

### 5.2.3. The MFC Capture Log Format

For the storage of the captures, we designed a flexible log format that we believe can be used to log any message-based communication. Our format is a dialect of the eXtensible Markup Language (XML). We choose XML for our log format due to its flexibility, especially when it comes to items of the same type that contain different kinds of information. This fits our requirements on the log format very well, as we only want to log items of the same type, i.e., messages, but the information we log depends on the kind of message. The drawback of using XML is that it has a relatively large overhead compared to, e.g., a Comma Separated Value (CSV) based format. However, if we wanted to use a log format with the same flexibility as XML allows us, we would have to introduce a possibly even larger overhead by ourselves. Another advantage of using an XML based log format is that it is easy to parse XML files with a variety of standard parsers available for many different technologies. This allows for any number of unforeseen exploitations and analysis methods on the captured data.

```
<!-- type 513 is WM_LBUTTONDOWN -->
<msg type="513">
 <param name="WPARAM" value="1"/>
 <param name="LPARAM" value="590009"/>
 <param name="window.hwnd" value="197330"/>
 <param name="point.x" value="194"/>
 <param name="point.y" value="11"/>
</msg>
<!-- type 514 is WM_LBUTTONUP -->
<msg type="514">
 <param name="WPARAM" value="0"/>
 <param name="LPARAM" value="590009"/>
 <param name="window.hwnd" value="197330"/>
 <param name="point.x" value="194"/>
 <param name="point.y" value="11"/>
</msg>
<!-- type 273 is WM_COMMAND -->
<msg type="273">
 <param name="WPARAM" value="57664"/>
 <param name="LPARAM" value="197330"/>
 <param name="window.hwnd" value="328394"/>
 <param name="command" value="57664"/>
 <param name="sourceType" value="0"/>
 <param name="source" value="197330"/>
</msg>
<!-- type 1 is WM_CREATE -->
<msg type="1">
 <param name="WPARAM" value="0"/>
 <param name="LPARAM" value="3138380"/>
 <param name="window.hwnd" value="66296"/>
 <param name="window.name" value="About TestProg"/>
 <param name="window.parent.hwnd" value="328394"/>
 <param name="window.class" value="#32770"/>
 <param name="window.ismodal" value="true"/>
</msg>
```

Listing 5.1: The log for messages generated by a mouse click.

Our XML dialect is straight forward. Every time a message is monitored and not filtered, a `msg`-node is added to the end of the log file that represents the monitored message. The type of the message is stored as an attribute of the node. To store information apart from the type of the message, each `msg`-node can have an arbitrary number of `param` child nodes. The `param`-nodes have two attributes: 1) the *name* of the parameter, e.g., `window.hwnd` for the HWND of the widget, to which the message is addressed to; 2) the *value* of the parameter. Both name and value are strings and therefore flexible when it comes to storing different kinds of values. The `param` nodes are the reason for the versatility of our log format, as they allow to store nearly arbitrary information together with a message, the only requirement being that the information can be expressed as a string.

Listing 5.1 shows an example of a capture. It shows the result of clicking with the left mouse button, where a widget is created as the result of the click. After the message `WM_LBUTTONDOWN` and `WM_LBUTTONUP` for the mouse click itself, a `WM_COMMAND` message is generated as result of a click, similar to the scenario depicted in Figure 5.4. As can be seen, the messages have different parameters. For each message, the LPARAM and WPARAM are stored. In addition, the coordinates of the click are stored with the `WM_LBUTTONDOWN` and `WM_LBUTTONUP` messages. For the `WM_COMMAND` message, the value of the command, as well as the source of the command and the type of the source are included in the log. They are important information for the analysis and exploitation of the command. With the `WM_CREATE` message, information about the widget that has been created is included in the parameters, i.e., the HWND of the widget, the HWND of its parent widget, the resource Id of the widget, the class of the widget, and its modality.

Some of the above described message specific parameters are already encoded in the LPARAM and WPARAM, e.g., the coordinates of a click are also encoded in the LPARAM of the `WM_LBUTTONDOWN` message params provide a means to assign names to parameters that are directly understandable without reading an API description. Other parameters are not part of the message and need to be determined otherwise, e.g., the parent HWND of a widget is determined with the `GetParent()` function of the Windows API.

### 5.2.4. Replaying Windows MFC GUIs Through Internal Messages

For the replaying of Windows MFC GUIs we provide the stand-alone application MFCReplay [42]. The replay of the applications is based on the replay of messages. The messages depend on the GUI events, which are inferred from the captured usage information on the translation layer (Section 5.2.5). The result of the replay generation of the translation layer is only a sequence of windows messages including their targets and assertions. The format is explained in detail with an example in Section 5.2.5.4. The main challenge remaining for the replay tool is the identification of the widgets, where a message is sent to based on the available information, i.e., a target string that includes the name, class, resource Id, and modality of the target widget and all its parent widgets (Section 5.2.5.1)

To this aim, we defined a heuristic approach that compares the widgets that currently exist in the system with the target string. We assume that the target string is *complete*, i.e., that the ancestors of the target widget are all part of the target string, up to the top-level window. This assumption is fulfilled if the capturing of the application is activated before the first widget is created. In the following, when we traverse widgets that exist in the system, we assume that the HWND of these widgets is known and, therefore, our search for the HWND is finished once we match a fitting widget in traversal.

We start by traversing the top-level windows of the system and match them to the first segment of the target string. Depending on how well they match, a score is assigned to each window. We then traverse all child widgets of the highest scoring widget and match them to the second segment of the target string in the same way. This is repeated until we either find the target widget or determine that is does not exist, i.e., at some point we have a score of zero. It is possible that we find multiple widgets with the same score. In case we did not reached the last segment of the target string, we iterate over the children of all highest scoring windows. In case of the last segment, i.e., the target widget is reached, there is a high possibility for performing a wrong replay, e.g., by selecting the wrong widget. In this case, we propose to either abort the replay or simply select the first one and issue a warning to the tester that the replay may have failed due to incorrect matching.

The scoring function is depicted by Table 5.1. The score depends on the attributes of the widget that we know from the target string, i.e., the resource Id, window class, name, and modality. The only attribute that needs to match in any case is the modality, because we can determine the modality reliably and it is always the same. The other attributes can either be absent (resource Id), generated dynamically at run-time (window class), or have changing parameters (name, e.g., if it includes the time).

Additionally, the score depends on the Process ID (PID) of the process, where the widget is located. At the beginning of the replay, the application under test is started by the replay tool itself and it is easy to determine the PID of the application that is started. In case the application has only one process, all widgets will be part of this process. Using the PID for the heuristic excludes accidentally matching widgets of other applications. In case the application has several processes, this will only improve the matching for widgets located in the start-up process. However, the matching of other widgets works as good as it would be without using the PID at all, because the scores will just be lower than for widgets in the same process. The weakness of the scoring are duplicate widgets, i.e., if widgets with equal resource Id, window class, and name exist. In this case, only the PID can be used to discern between the widgets. We consider scores below 5 problematic, because the match is very weak. In case of duplicate widgets or problematic scores, we propose to either abort the replay or at least issue a warning. Our implementation of the replay approach issues a warning.

The second task of the replay tool is the evaluation of assertions. Our replay implementation currently supports two assertions: *TextEqualsAssertion* and *FileEqualsAssertion*. The text equals assertion checks whether the text of a widget is equal to a pre-defined expected

| Score | Modality | PID | Resource Id | Name | Window Class |
|---|---|---|---|---|---|
| 14 | x | x | x | x | x |
| 13 | x | x | x | x | |
| 12 | x | x | x | | x |
| 11 | x | x | x | | |
| 10 | x | x | | x | x |
| 9 | x | | x | x | x |
| 8 | x | | x | x | |
| 7 | x | | x | | x |
| 6 | x | | x | | |
| 5 | x | | | x | x |
| 4 | x | x | | x | |
| 3 | x | x | | | x |
| 2 | x | | | x | |
| 1 | x | | | | x |
| 0 | | | | | |

Table 5.1.: Scoring function.

text. The widget is identified using a target string in the same manner as we described above to infer message targets. The second assertion compares two files for equality. The first file is the pre-defined expected result, the second file is generated by the SUT. If the two texts, respectively files are equal, the assertions are fulfilled.

The final verdict of a test case, i.e., whether it passed or failed is based on the results of finding the target widgets and the assertions. In case a target widget could not be found or an assertion failed, a test case failed, otherwise it passed.

### 5.2.5. The Event Parser for Windows MFC

We need to provide two functionalities on the translation layer for Windows MFC GUIs: the identification of the events from the capture and the definition with which Windows messages the events can be replayed. As an example for this, consider the two messages in the captures `WM_LBUTTONDOWN`, `WM_LBUTTONUP`, both addressed to a widget with window class `Button`. We need to translate this into an event $e$ with $type(e) =$ LeftMouseClick and $target(e) =$ TheButton. Then, we need to define how this event can be replayed, ideally without relying on the coordinate-dependent messages `WM_LBUTTONDOWN` and `WM_LBUTTONUP`.

#### 5.2.5.1. Event Target Description

In order to identify the event targets, i.e., the widgets throughout multiple executions, the HWND is insufficient as it is different in each program execution. We decided to define a string representation of the message target that includes vital information about the widget. The information should be the same in multiple executions of the application and, therefore, be usable as indicators to identity a widget. Furthermore, the complete information should be unique for each widget, as overlaps would result in imprecise message targets. We decided to use the name, class, resourceId, modality, and the same information about each ancestor of the widget in the GUI hierarchy as indicators. The string representation is recursively defined in Listing 5.2. `parent_target` is the `target` of the parent widget. If the widget has no parent, e.g., because it is a top-level window, `parent_target` is the empty word.

```
1 target = parent_target
2           <window name="..."
3                   class="..."
4                   resourceId="..."
5                   isModal="..."/>
```

Listing 5.2: Definition of the target string for Windows MFC GUIs

As the messages in the capture are addressed to HWNDs and not to target strings, we need a method to convert HWNDs to target strings. To achieve this, we utilize the message included in the capture of the first category, i.e., the messages related to the creation, destruction, and alteration of widgets. Using these messages, we maintain a widget tree in our application that reflects the status quo of the widgets at all times during the capture. The nodes of the tree are the widgets and the parent/child relationships of the tree nodes reflect the parent/child relationships of the widgets. To identity the widgets in the tree, the HWND is used. This is feasible, as the widget tree only reflects the state of the GUI during one execution and the HWND is, therefore, a unique and reliable identifier.

A node is included in the widget tree when a `WM_CREATE` message is found in the capture and removed if an `WM_DESTROY` message is found. With each node, the name, window class, resourceId, and modality of the widget is stored. The window class, resourceId, and modality cannot change during the existence of the widget and are set using parameters passed along with the `WM_CREATE` message. The name of the widget can change. It is initially set using a parameter of the `WM_CREATE` message and updated during the life-cycle of the widget by observing `WM_SETTEXT` messages. Figure 5.5 shows an example of a widget tree that describes the widgets included in a simple *Compose Mail* dialog. Listing 5.3 shows the target string of the *Send* button.

```
1 <window name="Compose Mail"
2         class="Dialog"
3         resourceId="1001"
4         isModal="true"/>
5 <window name="Send"
6         class="Button"
```

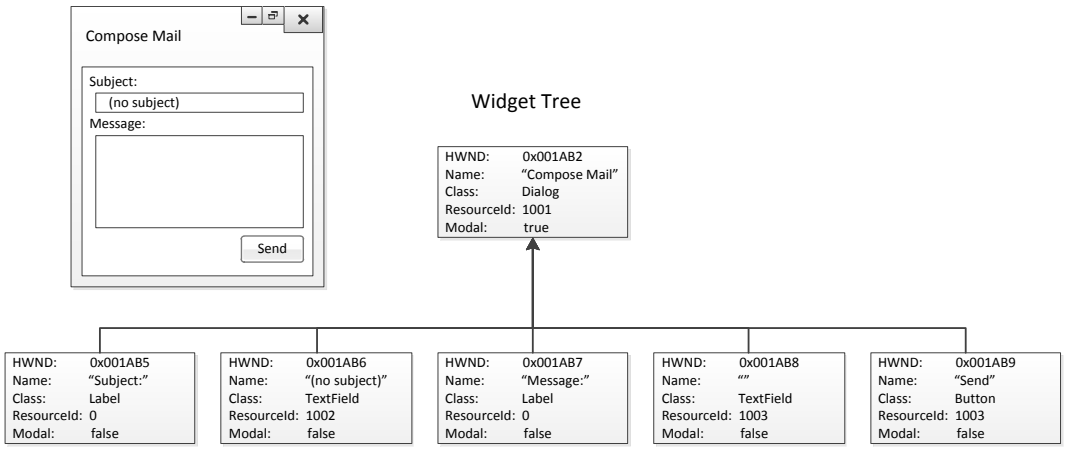



Figure 5.5.: A dialog and its widget tree representation (from [46]).

```
7         resourceId="1003"
8         isModal="false"/>
```

Listing 5.3: The target string of the Send button of the dialog shown in Figure 5.5.

#### 5.2.5.2. Rules for the Event Type Identification

We defined a set of rules for the identification of the event types included in a capture. The rules are matched against subsequences of the capture. The subsequences are determined based on the assumption that a user action starts with the user pressing the mouse or keyboard and ends with releasing it. Between pressing and releasing the mouse, internal messages may be send. Additionally, between two user actions, internal messages may also be send. This concept is exemplified in Listing 5.4. After the capture is split into subsequences of messages, where each subsequence represents one event, the events still need to be identified.

```
1  <!-- type 513 is WM_LBUTTONDOWN, begin of event 1 -->
2  <msg type="513">...</msg>
3  ...
4  <!-- sequence of internal messages -->
5  ...
6  <!-- type 514 is WM_LBUTTONUP, no *DOWN messages without a matching *UP message remaining -->
7  <msg type="514">...</msg>
8  ...
9  <!--sequence of internal messages -->
10 ...
11 <!-- type 513 is WM_LBUTTONDOWN, begin of event 2 because no open *DOWN messages remaining -->
12 <msg type="513">...</msg>
13 ...
```

Listing 5.4: A listing of messages in a capture and how they are split into events.

In our approach we define a set of rules for the event identification. The rules define the structure of the expected messages and are evaluated by the event parser. The rules are defined in XML. Listing 5.5 depicts an example for a rule that defines the message structure of a left click on a button. The syntax of these rules is as follows. Each `rule`-node has a *name* that describes the event associated with the rule, *LeftClickButton* in our example. The children of the `rule`-nodes are `msg`-nodes. These nodes match messages depending on their *type*, e.g., `WM_LBUTTONDOWN`. The message types are identified through their integer code, however, the names are available as XML entities. The messages must appear in the capture in the same order as in the rule. However, they do not need to be adjacent, i.e., there can be messages in between. If there are several messages in a row of the same type, the first one fitting the requirements of the `msg`-node is matched.

```
1  <rule name="LeftClickButton">
2   <msg type="&WM_LBUTTONDOWN;">
3    <!−− stores the message in the variable clicked −−>
4    <store var="clicked"/>
5   </msg>
6   <msg type="&WM_LBUTTONUP;">
7    <!−− checks if message is send to a button −−>
8    <equals>
9     <winInfoValue obj="this" winParam="class"/>
10    <constValue value="Button"/>
11   </equals>
12   <!−− checks if both messages are send to the same widget −−>
13   <equals>
14    <varValue obj="clicked" param="window.hwnd"/>
15    <varValue obj="this" param="window.hwnd"/>
16   </equals>
17  </msg>
18 </rule>
```

Listing 5.5: An example for an event parser rule.

Messages that have been matched by a `msg`-node can be stored with `store`-node children. In the example, the `WM_LBUTTONDOWN` message is stored in a variable called `clicked`. The variable names can be any string except `this`, which is reserved as an identifier for the current message.

It is possible to define constraints other than the type on the `msg`-nodes. These constraints are defined in form of value comparisons as `equals`-nodes. Each `equals`-node has two children, defining the terms that are compared. There are four different kinds of term nodes.

- `constValue`: a constant value, e.g., `<constValue value="Button"/>`.
- `paramValue`: the value of a message parameter, identified through its name, e.g., `<paramValue obj="this" param="window.hwnd"/>`.
- `winInfoValue`: information about the target of the message.
  - `<winInfoValue obj="this" winParam="class"/>` for the window class.
  - `<winInfoValue obj="this" winParam="hwnd"/>` for the HWND.

  - `<winInfoValue obj="this" winParam="resourceId"/>` for the resource Id.
  - `<winInfoValue obj="this" winParam="parentTarget"/>` for the target of the widget's parent.
- `msgInfoValue`: information about the type and target of the message.
  - `<msgInfoValue obj="this" msgParam="type"/>` for the type of the message (the integer value).
  - `<msgInfoValue obj="this" msgParam="target"/>` for the string representation of the message target.

The `obj`-attribute that each of the term node has, refers to the message to which the parameter value belongs. The value can be either a variable name of a stored message or `this`, which refers to the message for which the constraint is defined. In the example in Listing 5.5, two constraints are defined for the `WM_LBUTTONUP` message. The first checks whether the window class of the target of the `WM_LBUTTONUP` message is `Button`. The second compares the parameter `window.hwnd` of the `WM_LBUTTONUP` and `WM_LBUTTONDOWN` message.

It is also possible to match a sequence of multiple messages of the same type with the same constraints, by using the attribute `multiple` of the `msg`-node. For example, `<msg type="&WM_HSCROLL;" multiple="true"/>` matches a sequence of `WM_HSCROLL` messages. The messages in the sequence do not need to be adjacent. It is possible to define constraints for sequences in the same manner as for single messages, using `equals`-nodes. To store multiple sequences, `storeSeq` has to be used instead of `store`.

It is possible that two rules both match a subsequence of captured messages. Consider the rules `LeftClickButton` (Listing 5.5) and a rule `LeftClick` that matches any click with the left mouse (Figure 5.6). There is an overlap in the subsequences, where these two rules can be applied. In fact, the rule `LeftClick` can always be applied when `LeftClickButton` can be applied. To resolve such conflicts, the rules are prioritized. As means for the prioritization, the order of the rules in the XML file containing the rules is used. If multiple rules can be matched, the one that occurs first in the XML file has the highest priority and is applied.

```
1 <rule name="LeftClick">
2  <msg type="&WM_LBUTTONDOWN;" />
3  <msg type="&WM_LBUTTONUP;" />
4 </rule>
```

Listing 5.6: An event parser rule that matches any left mouse click.

#### 5.2.5.3. Rules for Replaying Events

Our approach for the generation of replays for events is to include descriptions for the replay into the rule set for the event matching. Thus, we have actually only discussed one half of

the rules in Section 5.2.5.2. A complete rule, including replay generation is depicted in Listing 5.7. The first part of the rule is the same as in Listing 5.5. The second part is for generation of a replay that executes a button click.

```
<rule name="LeftClickButton">
 <!-- event type matching part of the rule -->
 <msg type="&WM_LBUTTONDOWN;">
  <!-- stores the message in the variable clicked -->
  <store var="clicked"/>
 </msg>
 <msg type="&WM_LBUTTONUP;">
  <!-- checks if message is send to a button -->
  <equals>
   <winInfoValue obj="this" winParam="class"/>
   <constValue value="Button"/>
  </equals>
  <!-- checks if both messages are send to the same widget -->
  <equals>
   <varValue obj="clicked" param="window.hwnd"/>
   <varValue obj="this" param="window.hwnd"/>
  </equals>
 </msg>
 <!-- replay generation part of the rule -->
 <genMsg delay="100">
  <!-- defines the message type of the replayed message as BM_CLICK -->
  <type>
   <constValue value="&BM_CLICK;"/>
  </type>
  <!-- defines the target of the replayed message as the target of clicked -->
  <target>
   <msgInfoValue obj="clicked" msgParam="target"/>
  </target>
 </genMsg>
</rule>
```

Listing 5.7: A rule including replay generation.

Similar to the `msg`-nodes of the matching part of the rule, we defined `genMsg`-nodes in the replay generation part. Each `genMsg`-node represents one message that is sent to replay the event. There are two ways to generate these messages. The simplest is to resend a message exactly as it has been recorded. Let `var` be a message that has been stored with a `store`-node. Then it can be replayed directly with the following XML code as shown in Listing 5.8.

```
<genMsg delay="100">
 <storedVar obj="var"/>
</genMsg>
```

Listing 5.8: Replaying a recorded message.

The second way the manual definition of the message that is sent. This is the way, the message is defined in the example in Listing 5.7. In this case, the `genMsg`-node has the following children:

- `type`: describes the type of the message.

- `target`: describes the target of the message.
- `LPARAM`: describes the LPARAM of the message; this parameter is optional and has a default value of 0.
- `WPARAM`: describes the WPARAM of the message; this parameter is optional and has a default value of 0.

The values of all these nodes are defined using the same nodes for terms as for `equals`-nodes, i.e., `constValue`, `paramValue`, `winInfoValue`, and `msgInfoValue`. For the `LPARAM` and `WPARAM`, there is also the option to define the low-order and high-order bits of their values separately, as shown in Listing 5.9.

```
1 <LPARAM>
2   <HIWORD>
3     <!-- term-node -->
4   </HIWORD>
5   <LOWORD>
6     <!-- term-node -->
7   </LOWORD>
8 </LPARAM>.
```

Listing 5.9: Definition of a LPARAM with its HIWORD and LOWORD.

It is also possible to generate a sequence of messages based on a sequence of stored messages (stored with `storeSeq`). For this, the `genMsgSeq` is used. For each message in the stored variable, one message is generated. To achieve this, at least one of `type`, `target`, `LPARAM`, and `WPARAM` must not be constant, but defined with a term-node that uses the sequence variable.

A problem arises when the value passed as `LPARAM` or `WPARAM` of a message is in fact a HWND, and this HWND shall be the target of a message in the replay. An example for this is the `WM_HSCROLL` message, which is sent when a scrollbar is moved. The `WM_HSCROLL` message itself is not send to the scrollbar, but to the parent widget of the scrollbar. The `LPARAM` of this message contains the HWND of the scrollbar. However, to replay the scrolling we want to sent `TBM_SETPOS` messages to the scrollbar. Therefore, we want the target string for the widget that is defined by the HWND, which is contained in the `LPARAM`. To achieve this, we introduced `resolveHWND`-nodes into the matching part of the rules. The `resolveHWND`-nodes are child-nodes of `store` and `storeSeq`. They resolve the HWND to the target string of the widget and store it as a parameter of the message. An example for this is shown in Listing 5.10.

```
1  <msg ...>
2   ...
3   <storeSeq varSeq="scrolls">
4   <!-- interprets the LPARAM as HWND and resolve it to its target string -->
5   <resolveHwnd param="LPARAM"
6                storeParam="scrollBarTarget"/>
7   </storeSeq>
8   ...
9  </msg>
10 <genMsgSeq ...>
```

```
<!-- use the resolved target string as target for a replay message -->
<target>
 <seqValue seqObj="scrolls" param="scrollBarTarget"/>
</target>
...
</genMsgSeq>
```

Listing 5.10: Example for the resolution of a HWND into a target string and its re-use for a replay message.

#### 5.2.5.4. The Test Case Format

The concrete test suites for the replay generator do not require the user actions, but only the messages required for replaying. Listing 5.11 shows an example for the format. Each `msg`-node in this format defines a message to be send, including its type, LPARAM and WPARAM. The target string of the message is a child of the `msg`-node. In case the LPARAM or WPARAM also contain an HWND, the target string is defined as child of `LPARAM`, respectively `WPARAM` nodes. In our example, first a `WM_COMMAND` message is sent to replay a click on a toolbar. Then the text of a "Cancel" button is checked by using an assertion. This assertion implicitly checks, whether the widget exists. Afterwards, an "Ok" button is clicked.

```
<?xml version="1.0" encoding="UTF-8"?>
  <log>
    <session id="1">
      <!-- type 273 is WM_COMMAND -->
      <msg type="273" LPARAM="0" WPARAM="57664" delay="100">
        <!-- LPARAM contains a target string, which will be resolved into an HWND -->
        <LPARAM>
          <window name="TestProg - TestProg1" class="Afx:" resourceId="0" isModal="true"/>
          <window name="" class="ToolbarWindow32" resourceId="59392" isModal="false"/>
        </LPARAM>
        <!-- target widget of the message -->
        <window name="TestProg - TestProg1" class="Afx:" resourceId="0" isModal="true"/>
      </msg>
      <!-- assertion event that checks if the text of the target widget is "Cancel" -->
      <assertTextEquals expected="Cancel">
        <!-- target widget of the assertion -->
        <window name="TestProg - TestProg1" class="Afx:" resourceId="0" isModal="true"/>
        <window name="About TestProg" class="#32770" resourceId="0" isModal="true"/>
        <window name="Cancel" class="Button" resourceId="1" isModal="false"/>
      </assertTextEquals>
      <!-- type 245 is BM_CLICK -->
      <msg type="245" LPARAM="0" WPARAM="0" delay="500">
        <!-- target widget of the message -->
        <window name="TestProg - TestProg1" class="Afx:" resourceId="0" isModal="true"/>
        <window name="About TestProg" class="#32770" resourceId="0" isModal="true"/>
        <window name="OK" class="Button" resourceId="1" isModal="false"/>
      </msg>
    </session>
</log>
```

Listing 5.11: An example for a replay.

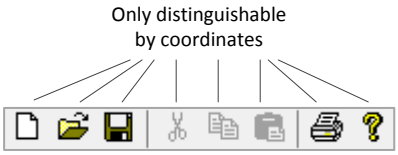



Figure 5.6.: A simple toolbar (from [46]).

#### 5.2.5.5. Limitations of the Event Parser and Replay Generator for Windows MFC GUIs

Our implementation of the rules is only prototypical and the rule set is incomplete. As a result, we cannot replay all usage events. However, we are able to replay a large subset of frequently occurring GUI interactions, including button clicks, keyboard input, selections with radio buttons and check boxes, text input in edit boxes, performing actions via the toolbar, moving scroll bars, and changing tabs. Additionally, we partially support interactions with menus. Actually opening a menu is not possible, however, replaying the action that is performed, e.g., open an about box by clicking on "Help" $\rightarrow$ "About" is possible.

All of these actions are replayed in a coordinate independent way. Most notably, actions on toolbars and dragging of scroll bars can be replayed in a coordinate independent way. The symbols in a toolbar are usually only distinguishable by their coordinates and do not have a HWND of their own to which messages can be addressed (Figure 5.6). Instead, the toolbar widget sends internal messages depending on the item that has been clicked. We defined a rule that replays this message and thereby circumvents using coordinates. The dragging of a scroll bar is an action that requires mouse movement. However, during the dragging, internal messages update its position. We capture these messages and move the scrollbar using these positions to replay the dragging, and remove the coordinate-dependent mouse movement.

## 5.3. Platform and Translation Layer for Java JFC GUIs

For Java JFC, we provide a platform layer implementation for the monitoring, the JFCMonitor [41], and an event parser for the monitor. We do not provide a replay generator or executor. The JFCMonitor can be used in combination with any JFC application that is packaged in an executable JAR archive. Our only requirement on the tool is that it can be successfully used for the usage monitoring of Java applications.

### 5.3.1. JFC Usage Monitoring with Event Listeners

In principle, the monitoring is similar to the concept of the MFCMonitor. Instead of the Windows messages, we observe the JFC events. Similar to Windows process hooks, the JFC API provides the interface `AWTEventListener`, which can be used to listen system-wide, i.e., in all threads of a Java Virtual Machine (JVM) to the JFC events being send.

Instead of the hook types in the Windows API, it is directly possible to configure which events an `AWTEventListener` receives. Since we can directly configure which messages we want to receive, we do not need message filtering. Furthermore, we ignore internal messages and only monitor the usage related events directly.

The relevant events are listed in Table 2.2. The events defined in the classes `MouseEvent` and `KeyEvent` are the usage events that we require. The events from the class `FocusEvent` are required to track the keyboard focus, since `KeyEvents` are often addressed to the JVM instead to a widget and the JVM takes care to redirect the input to the respective widget. In addition to the usage events, we have to track the GUI hierarchy, since it is not reliably possible to infer the ancestors of a widget at run-time via the JFC API. To this aim, we listen to the `WindowEvents` to maintain the GUI hierarchy internally during the capturing. In comparison to the `MouseEvents` and `KeyEvents`, we only listen to and do not log the `WindowEvents`.

### 5.3.2. The JFC Capture Log Format

The capture format that we use for the JFCMonitor is similar to the capture format of the MFCMonitor (Section 5.2.3). We use an XML dialect with the same `params` with `name`/`value` pairs to allow storing arbitrary information with events in a versatile way. However, there are differences between the XML dialects. In the JFCMonitor format, we use `event` instead of `msg` nodes for the monitored events. Furthermore, the GUI hierarchy is not part of the capture in form of create/destroy events. Instead, we include the information about the event target into each event in form of a `source`-node.

The `source`-node contains a sequence of `component`-nodes. Each `component`-node represents a widget in the GUI hierarchy. The order of the `component`-nodes reflects the placement in the hierachy, where the first node has the highest position in the GUI hierarchy (i.e., a top-level window) and the last node is the target widget. Each `component`-node provides basic information about the respective component, i.e., its title, class, icon, index, and a hash value that represents the ID of the actual Java object (Section 2.3.1.2). Listing 5.12 shows an example for the log produced for a mouse click on an “Ok” button.

```
1  <!-- id 500 is a MousePressed event -->
2  <event id="500">
3   <param name="X" value="8" />
4   <param name="Y" value="12" />
5   <param name="Button" value="1" />
6   <param name="Modifiers" value="16" />
7   <source>
8    <component>
9     <param name="title" value="Main Window" />
10    <param name="class" value="javax.swing.JFrame" />
11    <param name="icon" value="null" />
12    <param name="index" value="-1" />
13    <param name="hash" value="5585dc" />
14   </component>
15   <component>
```

```
16     <param name="title" value="Ok" />
17     <param name="class" value="javax.swing.JButton" />
18     <param name="icon" value="null" />
19     <param name="index" value="0" />
20     <param name="hash" value="1e064c" />
21    </component>
22   </source>
23  </event>
```

Listing 5.12: Example for the output of the JFCMonitor.

### 5.3.3. Event Parser for Java JFC GUIs

The event target identification and event type identification is straightforward. The reason for this is that the event types of Java JFC are already sufficiently abstract for the event layer. For example, there is a `MOUSE_CLICKED` event for mouse clicks and we do not need to construct a mouse click from mouse up and mouse down events. Furthermore, we already provide a detailed event target description with attributes of the target with each event as the `source`. Therefore, we define the abstract event type as the platform event type and the abstract event target as the `source` of the platform event.

## 5.4. Platform and Translation Layer for PHP-based Web Applications

For PHP-based Web applications, we provide a usage logging component, the PHPMonitor, an event parser for the data obtained by the PHPMonitor and a replay generator for an already existing tool called curl.

### 5.4.1. Usage Monitoring with a PHP script

The PHPMonitor requires that users are already tracked with cookies. We assume that the Apache HTTP server [3] is used. Then, the Apache add-on `mod_usertrack` configures the Apache such that it tracks users automatically with a cookie. Furthermore, the add-on allows the creation of a usage log that includes the cookie ID, a timestamp of the request, the URI, the referrer of the request, and the user agent for each HTTP request. However, this log is insufficient for our purposes because it is impossible to include information about the POST variables that are sent with the request.

The PHPMonitor itself is a small PHP script that needs to be executed with every PHP request of a user. To achieve this, the option `php_value_auto_prepend_file` of the PHP server configuration is used. By setting this option, the PHPMonitor script is automatically executed before each request. The logged information is similar to a log file the Apache HTTP server can create with the add-on `mod_usertrack`. It contains the cookie ID, a timestamp of the request, the URI, the referrer of the request, the user agent, and the POST

variable that are sent with the request. The log does not include the values of the POST variables that are sent. We only include the names of the POST variables because of the reasons we stated in Section 2.3.2. The addition of POST variables is the only difference between the log created with `mod_usertrack` and the PHPMonitor.

The drawback of using the PHPMonitor for the usage monitoring instead of the `mod_usertrack` add-on is that it only monitors requests that call PHP-based Web applications. Other calls, e.g., of simple HTTP Websites cannot be monitored. However, for our purpose, i.e., the monitoring of PHP based Web applications, the PHPMonitor is sufficient.

### 5.4.2. The PHP Capture Log Format

The log format of the PHPMonitor is a textual log file, where each line contains information about one PHP request. Each line starts with the cookie ID, followed by the timestamp, the URI, the referrer, and the names of the POST variables. Each value is framed by quotation marks and the values are separated by a white space. Listing 5.13 shows an example of a PHP log with four requests from the same user.

```
1 "172.20.0.9.1307611615198521" "2011-06-16 09:57:05" "/index.php?lang=de" "http://www.address.de/index.php?
      lang=de" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:2.0.1)" ""
2 "172.20.0.9.1307611615198521" "2011-06-16 09:57:05" "/contentABC.php?lang=de" "http://www.address.de/index.
      php?lang=de" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:2.0.1)" "postVar1"
3 "172.20.0.9.1307611615198521" "2011-06-16 09:57:05" "/contentABC.php?lang=de" "http://www.address.de/index.
      php?lang=de" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:2.0.1)" "postVar1 postVar2"
4 "172.20.0.9.1307611615198521" "2011-06-16 09:57:05" "/contentXYZ.php?lang=de" "http://www.address.de/index.
      php?lang=de" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:2.0.1)" "postVar2"
```

Listing 5.13: Example for a log produces by the PHPMonitor.

### 5.4.3. Event Parsing of PHP-based Web Applications

The event parsing for the data provided by the PHPMonitor provides a unique challenge, because the events are not ordered by their sessions. Instead, the usage sequences interleave, because multiple users can visit a website at the same time. Furthermore, we have events in our captured data that are not actual usage events, due to Web crawlers of search engines that index the website. Therefore, the main task of the event parser for PHP-based Web applications is the extraction of the usage sessions and not the inference of events and event targets.

The log produced by the PHPMonitor contains one request per line. We distill the usage sequences from this log with a three step procedure.

1. **Filter Web crawler.** We discard all log entries, whose user agent indicates that it is a Web crawler. Examples for such user agents are "*Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)*" for the crawler used by Google and "*Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)*" for the crawler used by Yahoo!. We use

a keyword-based filtering mechanism for this purpose. We provide a list of keywords, e.g., "Googlebot" and "Yahoo! Slurp" and discard all log entries where the user agent contains one of the key words. A complete list of the keywords that are used for the filtering is depicted in Listing A.3 in the appendix.

2. **Extract sessions.** We separate the log into clusters defined by the cookie IDs, such that one cluster contains the entries for one cookie ID. The obtained clusters can still contain multiple page visits by the same user, i.e., multiple usage sequences. We use periods of inactivity to determine whether an event starts a new usage session. If the time passed since the last request is larger than a timeout threshold, the request belongs to a new usage sequence. The timeout can be configured by the users of the event parser to match the needs of the PHP application.

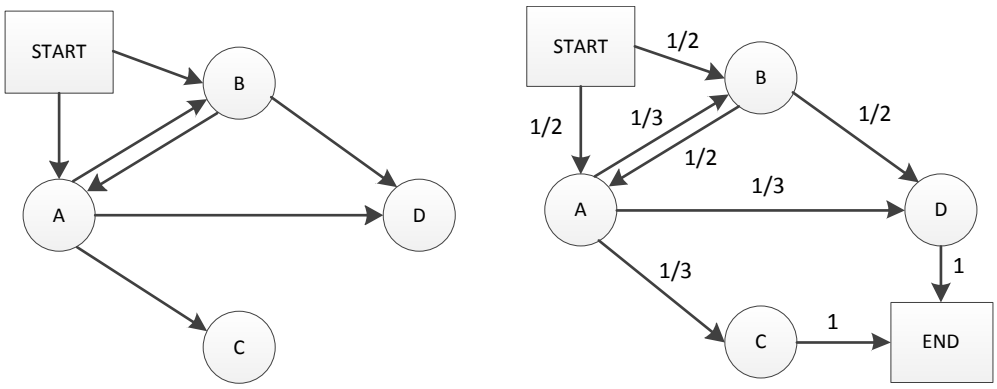
3. **Remove length one sessions.** After step two, we only have usage sequences left. However, we remove length one sessions, i.e., single page visits. The rational for this is that everybody that does not accept cookies produces only length one sessions. This holds true both for crawlers and users. Therefore, removing length one sessions provides a safety net against bots and also removes invalid usage sessions of users that do not accept cookies.

Once we have distilled the usage sequences from the log, we convert the log entries into abstract events. The target of the events is the PHP script that is executed and, therefore, can be extracted from the URI. For the type, we need the GET and the POST variables that are sent without their values. The GET variables can be extracted from the URI and the POST variables are part of the log. We denote the type as follows: $type(e) = GET[getVar1, getVar2, \ldots] + POST[postVar1, postVar2, \ldots]$

We also provide a replay generator that can be used with the command line tool curl. The events are translated into *abstract* curl calls. Let $e$ be an event with $target(e) = /content.php$ and $type(e) = GET[getVar1, getVar2] + POST[postVar1, postVar2]$. The curl call then is `curl -data "postVar1=DATA_POSTVAR1_DATA&postVar2=DATA_-POSTVAR2_DATA" /content.php?getVar1=DATA_GETVAR1_DATA&getVar2=DATA_-GETVAR2_DATA`. This is an abstract curl call, because the generated call does not include the test data, but placeholders for the data, e.g., `DATA_POSTVAR1_DATA` for the data of `postVar1`. We require a test data generator to replace the placeholders with actual data to create concrete curl calls.

## 5.5. Translation layer for GUITAR

GUITAR is a framework for EDS testing similar to the one we propose in Section 4. It provides tooling for model-driven EDS testing, for the most part based on EFGs. For a comparison of our work with GUITAR, the reader is referred to Section 7.1. Since both




(a) A small EFG with events A, B, C, D.

(b) The same EFG as random EFG, i.e., probabilities, such that each follower is equally likely. (needs to be redrawn)

Figure 5.7.: Example for the conversion of an EFG into an random EFG.

frameworks and their respective implementations provide functionality for EDS testing, interoperability between the frameworks is desirable. To achieve this to some degree, we provide a translation layer that can be used to convert EFGs into *random EFGs*, i.e., first-order MMs in which each following transition is equally likely. Figure 5.7 depicts an example for the conversion. Based on a random EFG, we are able to apply our randomized test case generation to models generated with GUITAR.

The conversion is based on parsing GUITAR test cases and treat them as training sequences. To make the complete EFG part of the test suite, we need depth two event coverage. Then, all transitions that are possible in the EFG are part of the test suite and if we train a first-order MM with the test suite as usage sequence, the graph representation of the MM is equivalent to the graph representation of the EFG. If we then omit the probabilities from the MM and instead assume that all following symbols are equally likely, we have a *random* EFG.

To generate test cases for GUITAR, we directly write in the GUITAR test case format. To achieve this, we add all the GUITAR information as payload into the abstract event as `EFGReplayables`.

## 5.6. Event Layer

In our instantiation of the framework, we implement all of the event layer components depicted in Figure 4.1. The components themselves are implemented in the Java library EventBenchCore to allow their re-use in other EDS testing projects. However, the EventBenchCore library does not provide any user interface feature, i.e., neither a GUI nor console

based method to access the components. Such a feature is provided by the EventBenchConsole, which uses the EventBenchCore library and provides both a GUI and a console based user interface.

Most event layer components are centered around the usage profiles, i.e., the stochastic processes. The model builder creates the usage profiles, the coverage analyzer and the session generator use the profiles for the generation of coverage reports and session generation. The usage analyzer also works partially on the usage profiles for the calculation of stationary distributions and the other application of the usage analyzer is straightforward counting of event occurrences. The description of all of these components is, therefore, complete if a usage profile is added. Hence, we only describe how we implemented usage profiles and our test oracle.

### 5.6.1. Implementation and Training of Usage Profiles

The implementation of usage profiles is basically the implementation of the underlying processes such that they can work with events, i.e., the `Event<T>` class. Hence, in this section we describe the implementation of the stochastic processes first-order MM, higher order MM, and PPM models. We use the concept of *tries* as a common method for the implementation of all our models.

Generally speaking, a trie is a tree-structure with counters that is used to count how often a subsequences occurs. A trie has a designated *root* node that is the empty word. A nodes of a trie is a tuple $(e,i)$, where $e \in E_U$ and $i \in \mathbb{N}$ is a counter. The counter $i$ of a trie node is inferred from a set of training sequences $S$. A path from the root of the trie to one of its nodes stands for the sequence defined by the path, e.g., the path $(root,(e_1,i_1),\ldots,(e_n,i_n))$ stands for the sequence $(e_1,\ldots e_n)$. The respective counter $i_n$ is the number of occurrences of the sequence $(e_1,\ldots,e_n)$ as a subsequence in the training sequences and we define $count(e_1,\ldots,e_n) = i_n$. Additionally, we define the depth of a trie node as the distance of the node from the root node. Hence, the depth of a node is equivalent to the length of the sequence its counter describes. A trie has depth $d$ if all length $d$ subsequences with non-null counters are part of the trie. Furthermore, we define $V_d$ as the set that contains all nodes with depth $d$.

The complexity of a depth $d$ trie is bound by the number of possible sequences as follows. The worst case is that a trie contains non-null counters for all possible sequence. Then, the complexity is $\sum_{i=1}^{d} |E_U|^i \leq 2|E_U|^d \in O(|E_U|^d)$.

We use the empirical probabilities as estimator for the probabilities of the next symbol. We can use the trie to calculate the empirical probabilities of the next symbol in a $k$-th order MM as follows:

$$\overline{\mathbf{Pr}}_{MM^k}\{X_{n+1} = e_{n+1} | X_n = e_n,\ldots,X_{n-k} = e_{n-k}\} = \frac{count(e_{n-k},\ldots,e_{n+1})}{\sum_{\hat{e}_{n+1}\in E_U} count(e_{n-k},\ldots,e_n,\hat{e}_{n+1})} \tag{5.6.1}$$

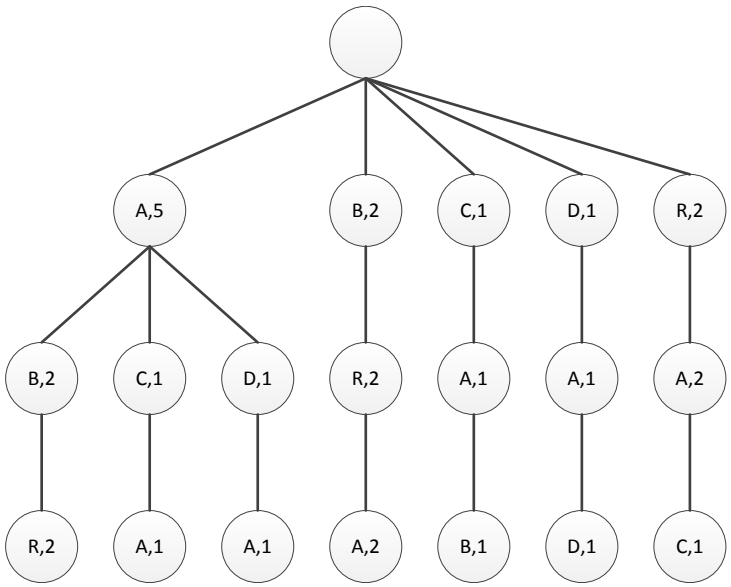



Figure 5.8.: A trie with depth three constructed from the sequence *(A, B, R, A, C, A, D, A, B, R, A)* (taken from [6]).

From Equation 5.6.1 follows directly that we need a trie of depth $d = k+1$ to calculate the probabilities for a $k$-th order MM. The complexity of a depth $k+1$ trie is $O(|E_U|^{k+1})$, which is the same as the complexity of a $k$-th order MM. Therefore, depth $k+1$ tries are not only sufficient for the calculation of empirical $k$-th order MM probabilities, but also their asympotic complexity is equal to the asymptotic complexity that is minimaly required.

As for PPM models, we require the calculation of all Markov orders $k_{min}, \ldots, k_{max}$. As we showed above, a depth $k_{max}+1$ trie is sufficient for the calculation of the probabilities of a $k_{max}$-th order MM. More precisely, a depth $k_{max}+1$ trie is sufficient for the calculation of any $\hat{k}$-th order MM with $\hat{k} \leq k_{max}$ and, therefore, sufficient for the calculation of the probabilities for all MMs with Markov orders $k_{min}, \ldots, k_{max}$. For the complexity, the same statement as above holds true.

Figure 5.8 shows an example of a trie taken from [6]. The trie has a depth three and is trained with the sequence $(A,B,R,A,C,A,D,A,B,R,A)$. The depth one level contains the counters for all length one subsequences, i.e., how often each symbol occurs. The depth two level contains the counters for the length two sequences, e.g., the node on the left-most side of the figure shows the counter for the sequence $(A,B)$. We calculate the first-order MM probability $\mathbf{Pr}\{X_{n+1} = B | X_n = A\}$ using Equation 5.6.1 as $\mathbf{Pr}\{X_{n+1} = B | X_n = A\} = \frac{count(A,B)}{\sum_{\hat{e}_{n+1} \in E_U} i} = \frac{2}{0+2+1+1+0} = \frac{1}{2}$.

The fact that tries are sufficient for MMs and PPM models for the calculation of empirical probabilities allows us to unify the training of all three stochastic process types. Instead of

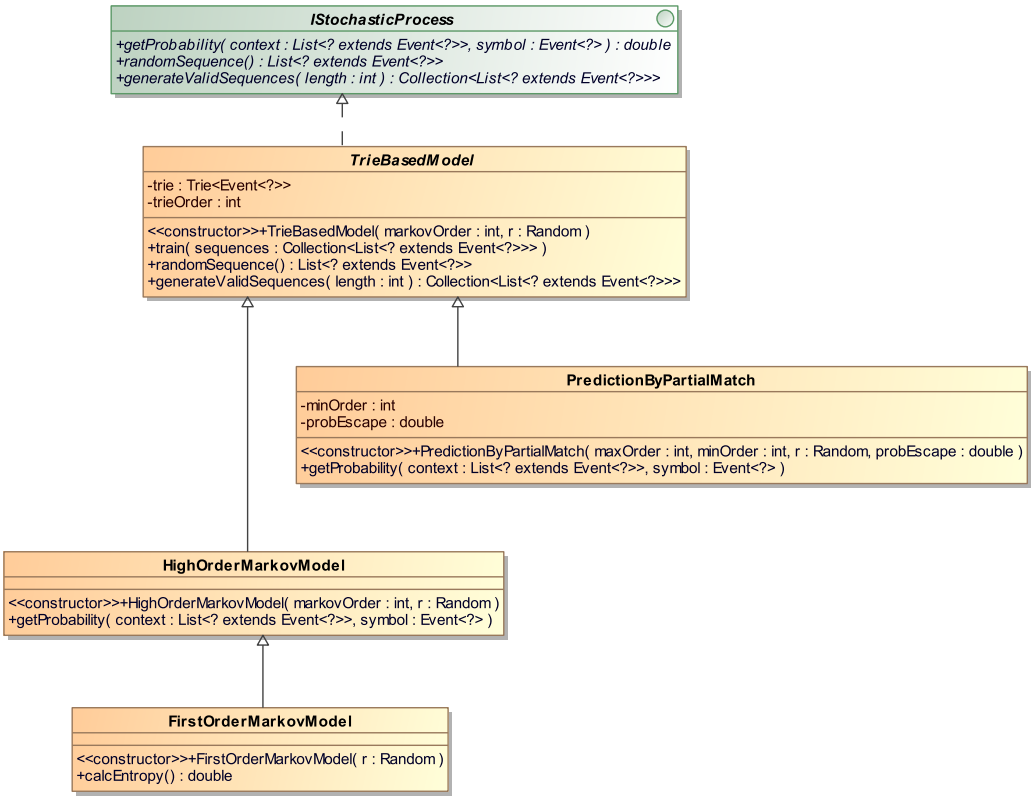



Figure 5.9.: Hierarchy of Java classes used for the implementation of stochastic processes.

training procedures for each of the stochastic process, we only need to generate a trie of the required depth from the usage sequences that we observed.

For the usage-based coverage and the test case generation methods that we defined in sections 3.3 and 3.4, the type of usage profile, i.e., the underlying stochastic process is irrelevant. Therefore, our implementation provides the means to apply these methods independent of the stochastic process. To this aim, we provide a Java interface `IStochasticProcess` that all stochastic processes have to implement. The class `TrieBasedModel` implements the `IStochasticProcess` interface and provides a skeletal implementation of stochastic processes that can calculate their probabilities based on a trie. Our stochastic processes are implemented as children of this class. Figure 5.9 shows the internal of the stochastic process in the EventBenchCore.

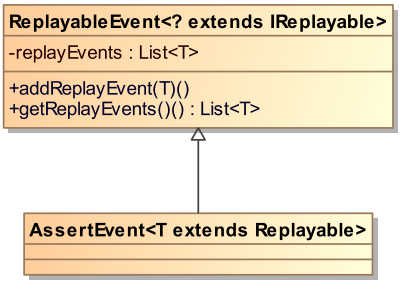



Figure 5.10.: Hierarchy of Java classes used by the EventBenchCore for the representation of assertion events.

### 5.6.2. Test Oracles

Our idea for implementing a test oracle is to include assertions in event sessions as events themselves, so called *assertion events* (Section 4.4.1). For our implementation, this means the integration of the assertions into the event hierarchy as depicted in Figure 5.2. Because the assertions are only relevant for the execution of a concrete test suite, we add an `AssertEvent` class as subclass of `ReplayableEvent` (see Figure 5.10). This way, we indirectly define that assertions are replayable and to be used by the replay generator. The `AssertEvents` themselves do not provide functionality. The type of an assertion is defined by adding instances of the `IReplayable` interface that contain value checks.

This concept is best explained using examples. In this thesis, we define two assertion events. The first is the *"text equals" assertion event*. This assertion compares the expected text of its target to an expected string that is provided as payload of the assertion event, i.e., a `TextEqualsReplay` that implements `IReplayable`. The second example is the *"file equals" assertion event*. This assertion event has no target. Instead, its payload contains two file paths: the path to an expected file and the path to the actual file. The expected file is pre-defined by a tester and contains the expected output of the SUT. The actual file is generated by the SUT as part of a test case. The payload is implemented in the `FileEqualsReplay` class that implements the `IReplayble` interface. Figure 5.11 depicts the hierarchy of replayables for the assertions that we defined.

As for the test oracle itself, we assume that no automated way to infer assertion events is available. Instead, a test engineer has to fulfill the role of the test oracle and manually add appropriate assertion events to event sequences. To this aim, the EventBenchConsole provides a GUI that allows the addition of assertion events to abstract event sequences.

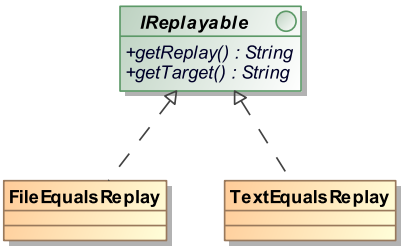



Figure 5.11.: Hierarchy of Java classes used by the EventBenchCore for the represenation of replayable objects for assertions.

# 6. Case Studies

In this chapter, we present three case studies that we conducted as part of this thesis. In these case studies we evaluate if the internal capture/replay approach for MFC we defined in Section 5.2 is applicable for real world software without large modifications. Furthermore, we evaluate the usage-based coverage criteria that are defined in Section 3.3 and test case generation methods that are defined in Section 3.4.

## 6.1. Case Study 1: Integration of the MFCMonitor into Software Products

In the first case study, we evaluate our capture/replay for Windows MFC. The results of this case study have previously been published and the description of this case study is based on these previous publications [45, 46].

### 6.1.1. Goals and Hypotheses

The goal of our first case study is to determine if the capture/replay approach that we defined in Section 5.2 fulfills our requirements. To evaluate our approach, we determine whether to accept or reject the following hypotheses.

- H1: Indirect messages can be used to gain coordinate independence.
- H2: The message filter that is implemented for the capturing is necessary.
- H3: The internal capturing is scalable with respect to the size of software product.
- H4: The approach is easy to adapt and applicable for GUI testing.
- H5: The approach is applicable to platforms other than MFC.
- H6: The captured data is sufficient for usage-based testing.

In the following, we describe the metrics that we gathered during the case studies to evaluate the hypotheses. Then, we present the case study methodology and the results of our experiments. Afterwards, we evaluate our hypotheses with respect to the results of the case study.

### 6.1.2. Evaluation Criteria

To evaluate the hypotheses H1, H5, and H6 we use qualitative results from the experiments that we conduct in our case study as indicators whether the hypotheses should be accepted or rejected. We accept H1 if our case study shows that through indirect messages coordinate independence can be gained for user actions that usually require screen coordinates and reject H1 if we cannot achieve this. H5 is accepted if the case study shows that the approach is applicable to a platform other than MFC and rejected if the results show evidence that this is not the case. H6 is accepted if the gathered data fulfills the criteria required for usage monitoring, i.e., the data can be used to identify the user actions.

For the hypotheses H2 and H3, we use quantitative metrics that are gathered from the case studies for the evaluation. To evaluate H2 the number of messages received with and without a message filter are used. H2 is accepted if the amount of messages without a message filter is very large and significantly smaller if a message filter is in place. For H3, we measure the size of log files generated in relation to the number of user actions that took place. Furthermore, the capturing must not influence the run-time performance negatively by blocking the SUT for prolonged amounts of time. We accept H3 if the amount of log data and consumed run-time are reasonably small for all applications used in the case study, regardless of the application size.

H4 is evaluated based on both qualitative and quantitative measures. On the one hand, we measure the amount of work required for the integration of the capturing into a software product through the lines of code. On the other hand, we evaluate if it is difficult for new users to use the techniques and tools designed to write GUI test cases. We accept this hypothesis if the amount of work required for the inclusion is very small and it is feasible to efficiently write GUI test cases without much extensive training.

### 6.1.3. Methodology

The methodology of our case study was straightforward. First, we integrated the MFC-Monitor into several software products. Then, for each of these products, we analyzed the captures that were produced, to determine if they included the desired information. Then we tried to convert what we captured into automatable replays of the execution. In case of positive results, we identified the limitations of the replay capabilities.

#### 6.1.3.1. Integration into Software Products

The first step of our case study is the integration of the MFCMonitor into several software products. The first application is a small MFC application that contains mostly generated code and has no logic or real application. This laboratory application has the sole purpose to provide an easy playground to test the usage monitoring without custom-made constructs or business logic of the application interfering. Still, the lab application is a good tool to test

the capabilities to replay specific types of widgets without coordinates. The lab application is available as open source [38].

The second product is *MarWin* that an industrial software product to which we have access as part of a cooperation with the Mahr GmbH Göttingen (Germany), which is the company that developed MarWin. MarWin is a large scale industrial software platform that is designed to be the basis of software products for both existing and future measuring devices in the field of dimensional metrology. It is written in C++ and contains many MFC based GUIs. MarWin is a very large and complex software with more than two million lines of code and is subject to continuous development and long-term evolution. Several hundreds of products based on MarWin are shipped each year.

The third product is based on the *Qt framework* [80] to evaluate if the MFCMonitor also works with GUI applications that are not based on MFC. We selected a small tool for Windows, which converts between different CSV formats.

We deem the integration into a software product successful if the amount of required work is low, bears no risk, and only a small portion of the software is changed. Furthermore, the execution of the application must not be slowed down considerably and the size of the capture should not be too large and, thus, consume large amounts of disk space in a short period of time.

#### 6.1.3.2. Analysis of the Captures

The second part of the case study is the analysis of the captures that we obtain from the MFCMonitor. Our criteria for the analysis are as follows.

- Does the capture include all information required to maintain the widget tree, i.e., the messages of the first category (e.g., `WM_CREATE`)?
- Does the capture include all user actions, i.e., the messages of the second category (e.g., `WM_LBUTTONDOWN`)?
- Does the capture include internal messages that can be utilized for replaying actions, i.e., the messages of the third category (e.g., `WM_COMMAND`)?
- How many messages that are not required are part of the capture?

All of the above criteria are related to the messages that are part of the capture and can be analyzed through manual inspection of the files.

#### 6.1.3.3. Replayability of the Capture

The third step is the determination of how well our prototype can actually replay the captured executions. We reduce this analysis to the subset of actions that we have implemented and extrapolate from that. We deem the replay successful if there are no glitches and no manual action is required.

#### 6.1.3.4. Applicability in the Industry

Finally, we wanted to know if our approach is feasible for GUI testing in real-life industrial settings. To determine this, we wanted to go beyond the experiments and integrate proof-of-concept test cases in an industrial testing scenario. We deem our approach ready for such usage if it is possible to generate a replay from a capture with one click, if it is possible to add assertions with reasonably low amount of work, and if the generated test cases run regularly and fully automated.

### 6.1.4. Results

In the following, we present the results of using the internal capturing approach, as well as how well our replaying methodology works. We discuss our experience for each of the three software products separately, as they differ and because we had a different focus in each of the experiments.

#### 6.1.4.1. Lab Application

The integration the MFCMonitoring into the lab application was successful without any problems. The work required was the addition of 10 lines of code to load the DLL and start the capturing at the beginning of the program execution, and 5 lines of code to stop the capturing, and free the DLL when terminating the application. The code is programmed defensively and includes error handling. It is, therefore, as risk free as possible. The produced capture fulfills our criteria completely. It includes all messages that we desire and the overhead of other messages is very small.

In a first experiment, we started the application, performed 102 user actions, i.e., mouse clicks and keyboard inputs, and finally terminated the application. The disk space that was consumed by the capture is very low, only 227 Kilobytes for all 102 actions. The application was not slowed down at all. The replay of the captured actions works without problems, albeit we limit the user actions to the actions for which our replay already works (Section 5.2.5.5).

In a second test, we tested overlapping user actions, thereby violating our assumption that each user action starts with pressing the mouse or keyboard and ends with releasing it. Our first test was pressing shift while entering text in an edit box. While all keys were sent correctly to the SUT, the replay only produced small letters, i.e., typing while pressing shift is currently one of the limitations of the replay capabilities. A second test was the termination of the application using the popular Alt+F4 command. This experiment worked, however, the buttons were not replayed directly, instead the internal command to terminate the application was sent. In a third test, we held a mouse button down while typing. Due to the prioritization of the rule, the mouse click was matched and all keystrokes were lost.

#### 6.1.4.2. MarWin

The first step of the integration into MarWin is the same as for the lab applications. The code to load the DLL, start the capture, stop the capture, and unload the DLL are the same 15 lines of code we used for the lab application. In a second step, we integrated the logging of the MFCMonitor into an already existing tracing mechanism, such that the application only produces one log file that contains all information. In order to differentiate between the capture and the other logged information, we use the prefix "UL:"[2] for everything that is part of the capture. We created a patched version of the MFCMonitor, where we replaced the writing to a separate log file for the capture with the calls of the already existing tracing mechanism. The required changes in the patched version were 12 lines of code. Altogether, we needed 27 lines of code to fully integrate the MFCMonitor into MarWin.

Based on the integration, we measured the impact of the MFCMonitor on a large scale application like MarWin. We assume that the size and complexity of MarWin is correlated to the amount of internal message traffic of the application. Therefore, we use the experiments with MarWin as a stress test for the message filter of our approach. In an experiment, we started MarWin, performed 64 user actions. Then, we terminated the application. During this period, a total of 20,221 messages were received by our hook procedures. Of these messages, 19,715 messages were filtered and 506 included in the capture. The time consumed by the monitoring was 1361 milliseconds. In average, the time consumed was about 2.7 milliseconds per message and 21 milliseconds per action. The size of the generated log was 139 Kilobytes.

Afterwards, we tested if the replaying of user actions is affected by the higher complexity of MarWin in comparison to the lab application. After we fixed some glitches that had no impact on the lab application, we were able to replay the same actions as in the lab application. An example for such a glitch is that we have a TabView widget in our lab application and we can change the active tab with mouse clicks. However, the replay of changing tabs in MarWin did not work. The reason for this was that the tabs were not part of a TabView, but of a PropertySheet so that they require different internal messages for their replay. Once we fixed this and differentiated properly between TabViews and PropertySheets, the replay worked without problems.

The final step was the creation of GUI test cases for MarWin with the MFCMonitor and MFCReplay as capture/replay and integration of the test cases into Mahr's test cycle. We created four GUI test cases that all run without any trouble in the nightly build. The process for generating a test case is highly automated. Before recording a new test case, the old capture has to be removed, which is performed by a batch operation. Then, the test engineer records the user actions required for the test case. Afterwards the test engineer only needs to execute a batch file. The batch file transforms the capture into an event sequence and automatically opens a dialog were the test engineer can insert assertions into the event. Afterwards, the test case is finished.

[2] UL stands for usage log.

#### 6.1.4.3. Qt Application

To integrate the MFCMonitor into a Qt-based tool, the same 15 lines of code as for the lab application and MarWin can be used. The capturing of executions works only partially. We still capture the messages of the first and second category, i.e., we monitor the widget's life-cycle and user actions. However, the internal communication between widgets is different in Qt and MFC and cannot be monitored in the same way. Therefore, the messages of the third category, which are vital for our replaying approach, are not monitored.

### 6.1.5. Discussion

In this section, we discuss the results of case study 1 and evaluate the the research hypothesis that we postulated in Section 6.1.1.

*H1: Indirect messages can be used to gain coordinate independence.* There is strong evidence to support this hypothesis. We are even able to replay actions that require mouse movement, e.g., dragging of scroll bars. Furthermore, toolbar icons are usually only distinguishable by coordinates. In constrast, we use internal messages to replay clicks on a toolbar without resorting to using coordinates. Therefore, we accept this hypothesis.

*H2: The message filter implemented for the capturing is necessary.* The experiments that were performed as part of the integration in MarWin show that the total amount of messages explodes rather fast with over 20,000 messages for only 64 user actions. Capturing all these messages is infeasible, slows down the execution, and produces large amounts of data. Only 506 of these messages passed the message filter. This shows that the message filter is necessary and effective. Therefore, we accept this hypothesis.

*H3: The internal capturing is scalable with respect to the size of the software product.* In none of the experiments we experienced any trouble with both run time and amount of data produced. In a small application, a 227 Kilobyte capture file was produced for 102 user actions, i.e., about 2.2 Kilobyte per action. In a large scale application, a 139 Kilobyte capture file was produced for 64 user actions, i.e., about 2.1 Kilobyte per action. Thus, the evidence suggests that the amount of work required for the capturing is constant, independent of the size and complexity of the monitored software product. Therefore, we accept this hypothesis.

*H4: The approach is easy to adapt and usable for GUI testing.* The integration of our capturing prototype always requires the same 15 lines of code. Even the further integration with an already existing tracing mechanism required only 12 lines of code. Therefore, the required work to integrate the internal capturing is minimal and no obstacle for the adaptation of our techniques. Additionally, with batch operations, it is possible to automate the test case generation process to a high degree, thereby allowing a simple creation of test cases. Thus, the approach is easy to adapt and does not require intensive tutoring. In our experiments with MarWin, we have created four GUI test cases, thereby proving that the

techniques are feasible for industrial GUI testing. Based on this evidence, we accept this hypothesis.

*H5: The approach is applicable to platforms other than MFC.* In our case study, we integrate our prototype MFCMonitor, which we specifically built to work with MFC, into a Qt application. The results produced were mixed. The internal capturing works the same. However, we cannot obtain all the information that we require for our replaying approach. The reason for this is that Qt does not use Windows messages for internal communication, but Qt events. Thus, the information that we require is still available, albeit in different form. We believe that by writing capturing and replaying specifically for Qt similar results like the ones for MFC can be achieved. Furthermore, we argue that the same holds true for other GUI platforms, because they also have a concept similar to Windows messages, e.g., the X Window System on which most Linux GUIs are based. Therefore, we can neither accept nor reject this hypothesis at this point.

*H6: The captured data is sufficient for usage-based testing.* The replay method that we defined creates events from the capture using a set of rules. The events are exactly the kind of information required for usage based testing, as they are indeed the user actions and not just single messages. Therefore, we accept this hypothesis.

## 6.2. Case Study 2: Evaluation of Usage-Based Coverage Criteria and Randomized Test Case Generation Methods

In the second case study, we evaluate the usage-based coverage criteria and the randomized usage-based test case generation techniques.

### 6.2.1. Goals and Hypotheses

The goal of this case study is two-fold. On the one hand, we want to evaluate if the usage-based coverage criteria behave as we expect, i.e., yield higher values for test suites generated based on a usage profile and perform similar to standard coverage for other test suites. On the other hand, we evaluate the differences between the two randomized strategies for test case generation, i.e., the random walk and the hybrid approach. Additionally, we use this study to evaluate how PPM models perform in relation to MMs. To this aim, we define the following research hypotheses.

- H7: Usage-based coverage criteria yield better results than standard coverage criteria for test suites that are generated from probabilistic usage profiles.
- H8: Usage-based coverage criteria perform similar to standard coverage criteria for randomly generated test suites.
- H9: The random walk test case generation approach and the hybrid test case generation approach perform differently.

- H10: The usage-based coverage results of PPM models are significantly different from MMs.
- H11: The standard coverage results of PPM models are significantly different from MMs.

### 6.2.2. Evaluation Criteria

For the evaluation of the research hypotheses, we employ several metrics. We evaluate hypotheses H7-H9 according to the results yielded by the standard and usage-based coverage criteria. In the case study, we calculate the usage-based coverage and the standard coverage of depths one to four. We say that one coverage criteria yields better results if it reports consistently higher values than other coverage criteria. We say that one test case approach performs better than another one if it consistently yields test suites with higher coverage values. As for H10 and H11, we say that the PPM models are significantly different if the results are different from the MM with the same order as the one that impacts the PPM probability the most. If the opt-out probability is $\varepsilon \leq 0.5$, we say that the PPM is significantly different if the results are different from a $k_{max}$-th order MM.

Additionally, we use four metrics that are similar to coverage criteria:

1. observed covered: the percentage of observed sequences covered by the test suite;
2. observed covered, weighted: the probability mass of the observed sequences covered by the test suite;
3. new covered: the percentage of possible new sequences covered by the test suite
4. new covered, weighted: the probability mass of the possible new sequences covered by the test suite.

We collect each of these metrics for the depths two to four[3]. We use these metrics to evaluate the structure of the generated test suites in relation to the observed usage data. Metrics similar to variants 1) and 3) were also used by Sprenkle et al. [106], who use absolute values instead of percentages. The variants 2) and 4) indicate a preference towards highly probable sequences, which should occur in a usage-based approach. We expect the differences between groups of metrics 1), 3) and 2), 4) to mimic differences between the standard coverage and the usage-based coverage.

### 6.2.3. Data

We have three data sets, where each set is obtained from an application of a different domain and EDS platform. Furthermore, the collection method for each data set is different. Hence, our data sets provide a diverse sample of usage data.

[3]For depth one, 1) and 2) are the same as the event coverage and usage-based event coverage of depth 1. 3) and 4) are both zero, otherwise unobserved events would be part of the usage profile.

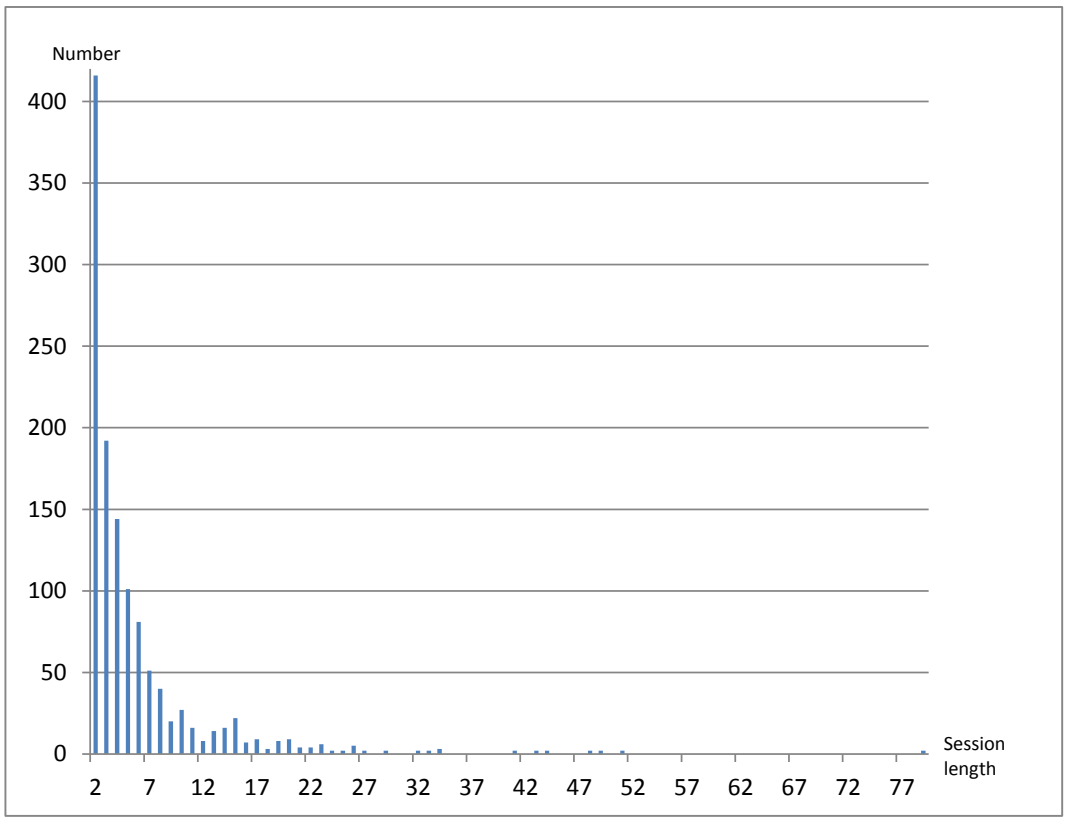



Figure 6.1.: Sequence lengths of the observed user sessions of the Web application data.

#### 6.2.3.1. Data Set 1: Web Application Data

The first data set has been obtained with the PHPMonitor. We monitored old website of the research group "*Software Engineering for Distributed Systems*"of the Georg-August-Universität Göttingen Germany [31] over a period of three months. We parse the gathered usage data with the event parser described in Section 5.4.3 and a session timeout of 60 minutes. We obtain 1,222 user sessions with an average of 5.65 events per sessions by 795 unique users over an event space of 67 observed events. Figure 6.1 visualizes the lengths of the user sessions. The amount of sessions drops exponentially with the lengths. Most of the observed sessions are very short. Over half of them have a length less than or equal to three, the longest session is 79 events long.

#### 6.2.3.2. Data Set 2: MFC Application Data

We collected the second data set in cooperation with an industrial partner, the Mahr GmbH Göttingen. We already described as part of the first case study how we integrated the MFC-

| Sequence lengths | 2 | 5 | 33 | 54 | 74 | 96 | 123 | 219 | 523 |
|---|---|---|---|---|---|---|---|---|---|

Table 6.1.: Sequence lengths of the captured MarWin Machine Monitor usage sessions.

| Sequence lengths | 1,584 | 1,655 | 1,824 | 2,332 | 2,407 | 2,654 |
|---|---|---|---|---|---|---|
| | 2,911 | 3,076 | 3,309 | 3,517 | 5,604 | |

Table 6.2.: Sequence lengths of the captured ArgoUML usage sessions.

Monitor into MarWin. We used the integrated monitor to obtain a usage data set. Concretely, we choose only a part of MarWin, the *Machine Monitor* as object for our study. The Machine Monitor is a tool for the observation and manual operation of measuring devices. Its functionalities include, e.g., moving the machine and changing measuring probes. For our data collection, we setup a machine with installed usage monitor and asked six employees of Mahr to operate the Machine Monitor to perform typical tasks. We obtained 9 user sessions with an average of 125 events over an event space of 153 observed events. The lengths of the observed sequences is very diverse. The shortest is only two events long, the longest has 523 events. All observed sequence lengths are listed in Table 6.1.

#### 6.2.3.3. Data Set 3: JFC Application Data

To collect our third data set, we performed an experiment in which users had to perform a specific task with the the SUT. Meanwhile, we monitored the execution of the SUT to observe how the users perform the task and, therefore, obtain usage data. The object under study was the Unified Modeling Language (UML) modeling tool ArgoUML [113]. The user group consisted of eleven computer science experts. One of the users was familiar with ArgoUML, the other ten users have never worked with ArgoUML before. The tasks for the users was the re-drawing of a UML class diagram with ArgoUML. The complete task description, as we handed it out to the participants of the study is depicted in Appendix A.3. We obtained 11 user sessions with an average of 2,806 events over an event space of 75 observed events. The lengths of the observed sequences varies between 1,584 and 5,604 events , depending on how efficient the users performed the given task. All observed sequence lengths are listed in Table 6.2.

### 6.2.4. Methodology

We evaluate each hypothesis by the training of different usage profiles of the SUT. We generate test suites from these profiles and then determine the values of both standard and usage-based coverage criteria of the generated test suites.

In some cases, we want to generate test suites with more test cases than possible by the usage profile. The number of possible sequences drops rapidly with higher Markov orders,

| Test case length | Random EFG 1st order MM PPM, $k_{min} = 1$ | 2nd order MM PPM, $k_{min} = 2$ | 3rd order MM PPM, $k_{min} = 3$ | 4th order MM PPM, $k_{min} = 4$ |
|---|---|---|---|---|
| 2 | 492 | 97 | 70 | 70 |
| 3 | 7,270 | 360 | 119 | 95 |
| 4 | 116,616 | 233 | 105 | 105 |
| 5 | - | 7,564 | 551 | 132 |
| 6 | - | 35,189 | 1,450 | 219 |
| 7 | - | 162,039 | 3,843 | 403 |
| 8 | - | - | 9,582 | 672 |

Table 6.3.: Number of possible sequences for the different usage profiles for the Web application data.

| Test case length | Random EFG 1st order MM PPM, $k_{min} = 1$ | 2nd order MM PPM, $k_{min} = 2$ | 3rd order MM PPM, $k_{min} = 3$ | 4th order MM PPM, $k_{min} = 4$ |
|---|---|---|---|---|
| 4 | 726 | 24 | 8 | 8 |
| 5 | 4,461 | 50 | 9 | 8 |
| 6 | 27,579 | 130 | 9 | 8 |
| 7 | 169,634 | 332 | 8 | 7 |
| 8 | - | 743 | 13 | 7 |
| 9 | - | 1,678 | 37 | 14 |
| 10 | - | 3,837 | 75 | 39 |

Table 6.4.: Number of possible sequences for the different usage profiles for the Mahr Machine Monitor data.

as tables 6.3 to 6.5 show. When the number of possible test cases is less than the desired test suite size, we simply take all possible test cases as the test suite instead of walking the model.

Because our test suite generation is probabilistic, randomness plays a role in our case study. Therefore, we perform ten replications of the experiments and evaluate the results of the case study on the mean values of the replications. Furthermore, we analyze the standard deviation between the replications. A large standard deviation indicates unstable results, which means that the results of the experiments cannot be generalized - even in the small scope of one case study. Instead, test case generation and evaluation depends to a large degree on chance.

There are three parameters for the test suite generation: 1) the usage profile from which we generate the test suite; 2) the test suite size, i.e., the number of test cases; and 3) the lengths of the generated test cases.

| Test case length | Random EFG 1st order MM PPM, $k_{min} = 1$ | 2nd order MM PPM, $k_{min} = 2$ | 3rd order MM PPM, $k_{min} = 3$ | 4th order MM PPM, $k_{min} = 4$ |
|---|---|---|---|---|
| 4 | 4,027 | 107 | 21 | 9 |
| 5 | 35,154 | 517 | 91 | 19 |
| 6 | 317,813 | 2,222 | 300 | 52 |
| 7 | - | 9,853 | 863 | 171 |
| 8 | - | 45,092 | 2,565 | 429 |
| 9 | - | 204,238 | 8,250 | 993 |
| 10 | - | - | 27,019 | 2,492 |

Table 6.5.: Number of possible sequences for the different usage profiles for the ArgoUML data.

#### 6.2.4.1. Usage Profiles

We need to evaluate the coverage criteria with results that were generated from different usage profiles to judge the potential impact of the type of the usage profile on coverage. To this aim, we compare twelve different usage profiles of the SUT for test case generation. We train four usage profiles based on MM with the orders one to four and PPM models with orders $(k_min, k_max)$ of $(1,2),(1,3),(1,4),(2,3),(2,4)$, and $(3,4)$ and an opt-out probability of $\varepsilon = 0.1$ from the available usage data. For the second data set, the MarWin Machine Monitor data, we only use the profiles with orders less than or equal to two, because the number of possible sequences is too low for the higher orders to allow a meaningful analysis of the results (see Table 6.4).

We evaluate hypothesis H7 by analyzing the coverage metrics of each of the test suites that were generated from these models. We also generate a random EFG from the observed data, i.e., a first-order MM, where each transition is equally likely. Therefore, the generated test cases are completely random. This random EFG can be viewed as a very weak usage profile, because it does not incorporate any probabilities for event sequences; nor is it a complete model, because it only includes event transitions that have actually been observed. Because all events are equally probable, the usage-based coverage criteria should perform similar to standard coverage criteria for test suites generated from this model. Hence, we use the random EFG model to evaluate hypothesis H8.

Finally, we generate a *complete EFG* that contains all transitions between events, without our knowledge if the transition is possible or not. To guarantee that the size of the complete model does not totally differ from the usage profiles, we assume that the observed events $E_U$ are equal to the possible events $E$. For PHP-based Web application, the inclusion of all transition is reasonable because HTTP requests can be sent in an arbitrary order. For the GUI applications, the assumptions only holds in single-dialog applications, where all widgets are always enabled. Both the MarWin Machine Monitor and ArgoUML fail this

criteria. Therefore, we overestimate the models in these two cases. We use this complete model to perform an additional comparison of the difference between the usage-coverage criteria and the standard coverage to verify that usage-based coverage remains valid for the complete model.

#### 6.2.4.2. Test Suite Size

We analyze if the coverage criteria behave different for small and large test suites, By evaluating test suites of different sizes. We chose six different sizes for the test suites: 1) a *tiny* test suite with 50 test cases; 2) a *very small* test suite with 100 test cases; 3) a *small* test suite with 500 test cases; 4) a *medium* test suite with 1,000 test cases; 5) a *large* test suite with 5,000 test cases; and 6) a *very large* test suite with 10,000 test cases.

#### 6.2.4.3. Test Case Length

We generate test suites with seven different test case lengths for all three data sets to evaluate the overall impact of the length. For the web application, we use the lengths 2, 3, . . . , 8 and for the GUI applications the lengths 4, 5, . . . , 10. We choose slightly longer lengths for the GUI applications for two reasons: 1) the observed sequences were significantly longer; 2) the first actions are often only the preparation of the actual interaction, e.g., changing the keyboard focus to the appropriate edit box.

For the hybrid approach, we need the maximum achievable length to generate all sequences for the creation of longer test cases. The maximal length that we were able to calculate can be inferred from tables 6.3 to 6.5. In case we did not enter a value in the table, we could not calculate it. Hence, the maximal achievable length for a first-order MM is four for the Web data set, seven for the MFC data set, and six for the JFC data set.

### 6.2.5. Results

The randomness in our test case generation process means that the generation of test suites of a given size subject to a user profile and test case length is not always possible. If the maximum test suite size is reached, results with a longer desired size remain the same, as the test suite remains the same. Even in these cases, we depict our results as if the test suites were different to show that increasing the desired test suite does not increase the coverage. Instead, other parameters, e.g., the test case length need to be changed. These cases of maximum size may be inferred from the number of possible sequences shown in tables 6.3 to 6.5.

We obtained stable results for all three data sets. The standard deviation between the ten replications of the experiments remains below 10% of the mean value of all replications, except in cases where the coverage values themselves approach zero (meaning that the standard deviation is also close to zero). For our evaluations, we consider the average values of

the ten replications. In the following, we show selective results that are representative for our findings. The complete results for all usage profiles, coverage depths, and both test case generations methods are depicted in Appendix A.4.

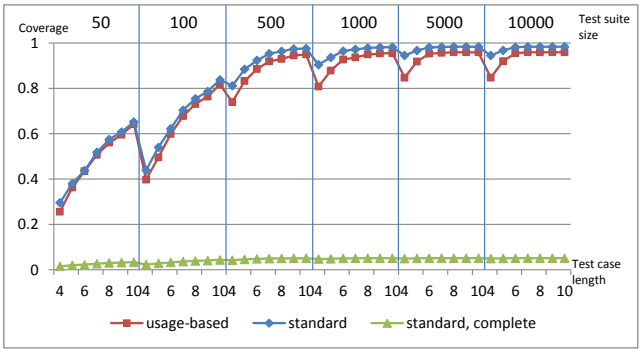
The results of the two test case generation methods are almost identical. We did not observe consistent and significant differences in the results. Over 95% of the usage-based coverage results and over 86% of the standard coverage results are within 3% deviation. We interpret a deviation that is this small as noise. We accept the larger differences in the standard coverage, because our test cases are generated with respect to usage-based coverage, which explains the higher, but still very small fluctuations in the standard coverage results. Due to this finding, we only use results of the random walk test case generation as example in the remainder of this section. However, the examples look nearly the same for the hybrid test case generation.

The coverages for all test suites generated from the random EFG show that the coverage remains nearly the same for standard and usage-based criteria. Figure 6.2 shows the results of the depth two coverage of the test suites generated with random walks. The results show that the values of the usage-based coverage (red) and standard coverage (blue) are within a few percent in all cases.

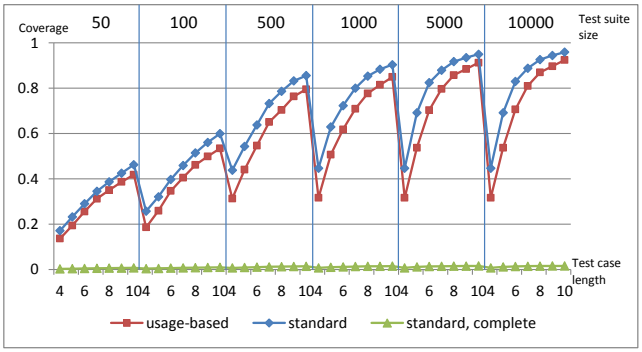
In relation to the complete model (green in all result figures), the coverages for depths two to four remain very low. In fact, coverage only converges to 100% for the coverage of depth 1, because we use the same events in both models. For all higher coverage depths, the coverage of all possible sequences in the complete model never achieves 15%. These observations hold for the MMs, PPM models, and the random EFG. Thus, even the weak usage profile that the random EFG describes leads to a big difference in the coverage.

Test suites generated from the MMs and PPM models report usage-based coverage consistently higher than standard coverage. This results in faster conversion to very high usage-based coverage values than for the standard coverage. For example, Figure 6.3 shows the depth two coverage for the test suites that are generated from the first-order MM. Especially for the JFC data set, the usage-based coverage converges to one rapidly. We observed a similar effect with the Web application data set. For the smaller test suite sizes this, effect is also observed for the MFC data set. For the larger test suite sizes, this effect vanishes because the test suite size approaches the number of possible sequences. We observe the smallest differences for the 2nd-order MM and the MFC data set. Here, the results of the usage-based coverage are almost equal (see Figure 6.4). However, this is due to the effect that the number of possible test cases is often close to or similar to the test suite size, which leads to the generation of all possible sequences and, thereby, dilutes the differences between the coverages.

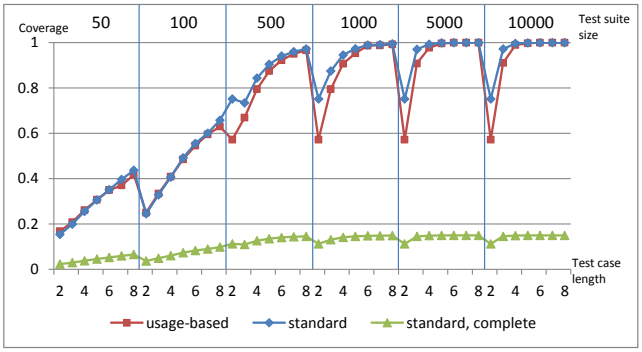
The performance of the PPM models and the MM models is comparable in terms of the usage-based coverage. The PPM models perform slightly better, i.e., the coverage converges a bit faster than for the MM, especially for short test cases. Figure 6.5 shows the comparison of a 2nd-order MM with a PPM model with $k_{min} = 1, k_{max} = 2$. In terms of the standard coverage, there is a significant difference between the PPM models and MMs.




(a) JFC data set.

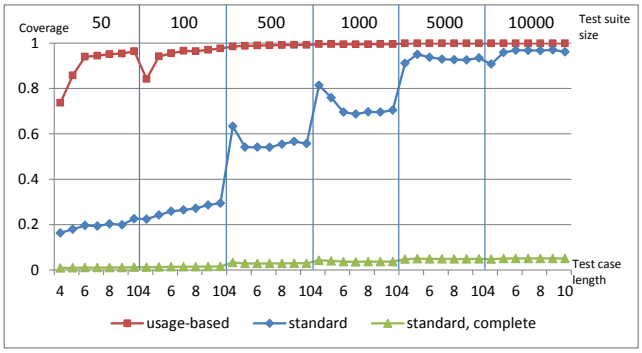



(b) MFC data set.

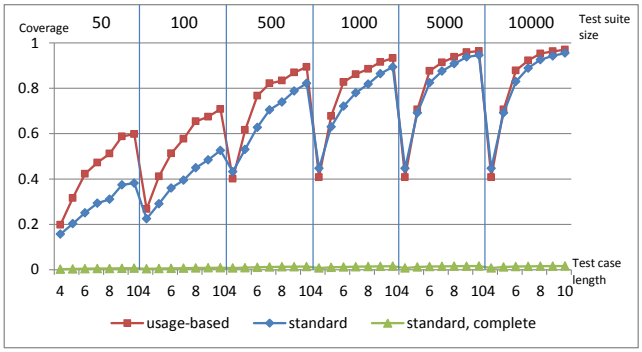



(c) Web application data set.

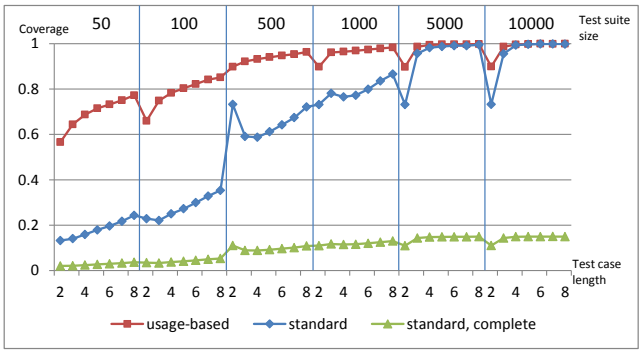
Figure 6.2.: Results for the depth two for the random walk test case generation with the random EFG.




(a) JFC data set.




(b) MFC data set.




(c) Web application data set.

Figure 6.3.: Results for the depth two for the random walk test case generation with the first-order MM.

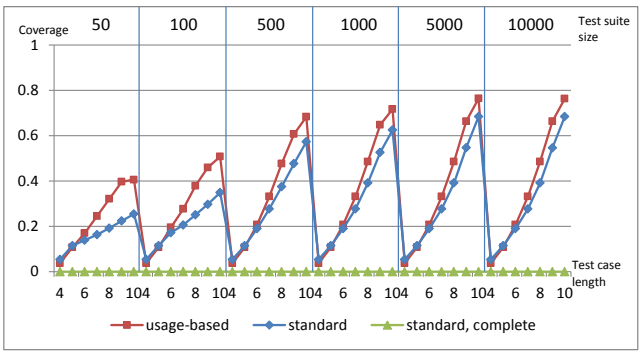



Figure 6.4.: Results for the depth three for the random test case generation with the 2nd-order MM and the MFC data set.

In general, the PPM models have a significant disadvantage in comparison to the MMs, because the number of possible sequences is that of the lowest allowed Markov order $k_{min}$. This disadvantage shows in most comparisons of MMs and PPM models, e.g., the one Figure 6.6 depicts. However, in some cases the PPM models perform better than the MMs, e.g., in the case Figure 6.5 depicts.

The generated test suites contain a mixture of observed and new subsequences, i.e., some sequences have the same structure as in the observed training data, while others have been newly generated. Figure 6.7 shows the depth two coverage of new and observed sequences for the first-order MM of the Web application. These results show that the weighting has the same effect as with the coverage criteria, in which the usage-based test generation favors new and observed sequences with a high probability.

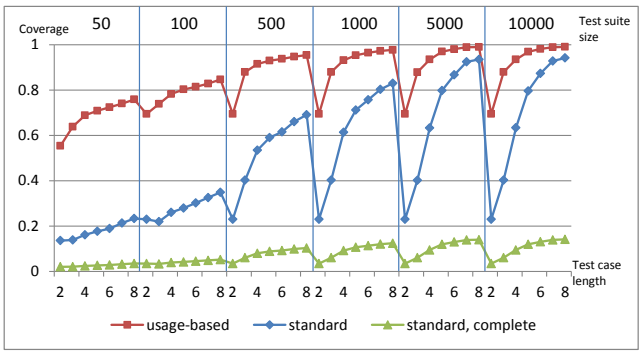
Furthermore, the results show that the coverage increases with both test case length and test suite size. Coverage increases non-linearly and slows with increasing length and size.

### 6.2.6. Discussion

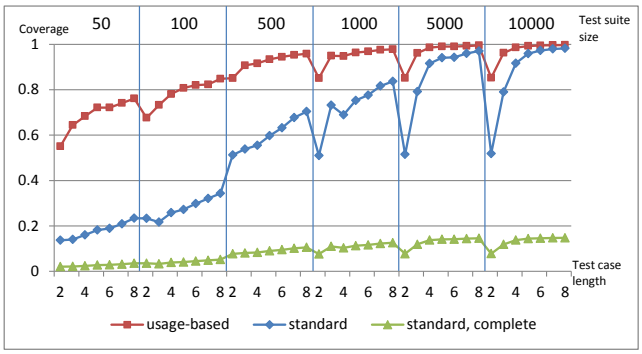
In the following, we discuss our research hypotheses with respect to the case study results.

*H7: Usage-based coverage criteria yield better results than standard coverage criteria for test suites generated from probabilistic usage profiles* In the experiments with MMs and PPM models, the usage-based coverage criteria yield better results in relation to both what is possible according to the usage profile and a complete model. Therefore, we accept this hypothesis.

*H8: Usage-based coverage criteria perform similar to standard coverage criteria for randomly generated test suites* The experiments with a random EFG for test case generation show that there exists no significant difference between standard and usage-based coverage criteria for randomly generated test suites. Therefore, we accept this hypothesis.

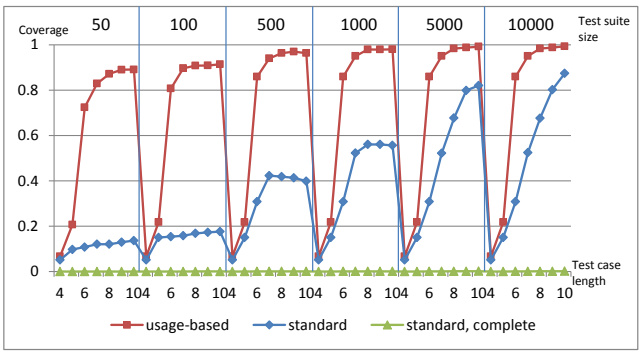



(a) Results for the depth two coverage for the random walk test case generation with the 2nd-order MM and the Web application data set.

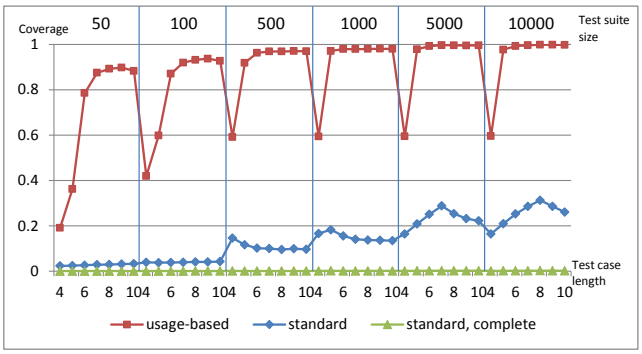



(b) Results for the depth two coverage for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 2$ and the Web application data set.

Figure 6.5.: Comparison of a 2nd-order MM amd a PPM model with $k_{min} = 1, k_{max} = 2$.

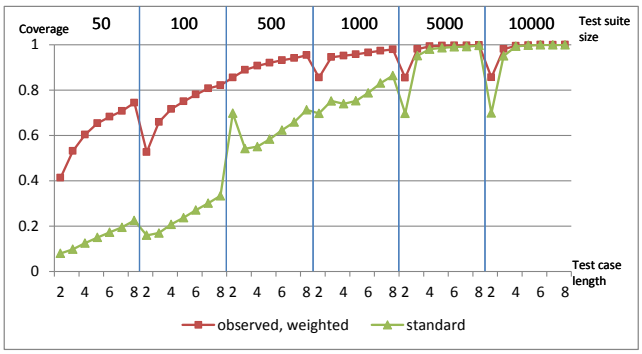



(a) Results for the depth three coverage for the random walk test case generation with the 3rd-order MM and the JFC data set.

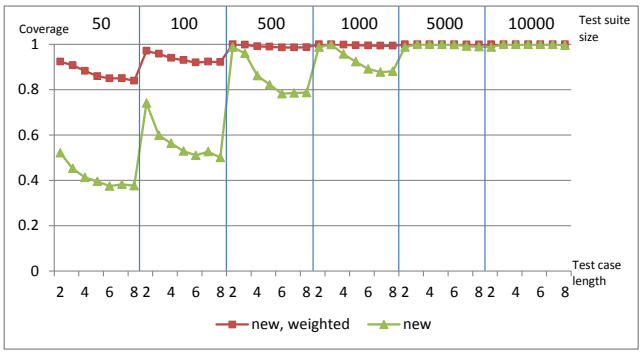



(b) Results for the depth three coverage for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 3$ and the JFC data set.

Figure 6.6.: Comparison of a 3rd-order MM amd a PPM model with $k_{min} = 1, k_{max} = 3$.




(a) Results for the depth two coverage of the observed sequences on the training data for the random walk test case generation with the first-order MM and the Web application data set.




(b) Results for the depth two coverage of the possible new sequences in relation to the training data for the random walk test case generation with the first-order MM and the Web data set.

Figure 6.7.: Example for the structure of the test suites in relation to the structure of the training data.

*H9: The random walk test case generation approach and the hybrid test case generation approach perform differently.* The results of both test case generation mechanism are almost equal. Therefore, we reject this hypothesis.

*H10: The usage-based coverage results of PPM models are significantly different from MMs.* The results of the usage-based coverage are similar for both models. The slightly higher results for the PPM models indicate that there may be a small advantage for the PPM models. However, the difference is too small to accept this hypothesis. Since we also cannot ignore the presence of these differences, we cannot clearly reject this hypothesis either. Therefore, we neither accept nor reject this hypothesis and state that additional experiments are required to determine the validity of this hypothesis.

*H11: The standard coverage results of PPM models are significantly different from MMs.* The results show a significant difference in the standard coverage results. For the most parts, the MMs models have higher standard coverage values. However there are also counterexamples, where the PPM models perform better. Thus, we cannot state in general which of the two models perform better. However, the findings clearly show that the models perform differently. Therefore, we accept this hypothesis.

## 6.3. Case Study 3: Evaluation of the Heuristic Test Case Generation

In the third case study, we evaluate the performance of the test case generation heuristic that we designed in Section 3.4.3.

### 6.3.1. Goals and Hypotheses

The primary goal of this case study is the evaluation of how well our heuristic performs in terms of generating test suites. This evaluation consists of three parts. First, we need to determine if the heuristic is indeed capable of generating test suites with a desired usage-based coverage. Second, we need to analyze if the heuristic scales for high coverage values. Third, we expect that significantly less test cases are required to achieve high coverage values in comparison to the randomized test case generation approaches that we evaluated in the previous case study. We postulate the following research hypothesis.

- H12: The heuristic is able to determine test suites with a desired usage-based coverage.
- H13: The heuristic is scalable in terms of desired usage-based coverage.
- H14: The test suites determined by the heuristic require less test cases to achieve the same amount of coverage like randomly generated test cases from usage profiles.

In addition to our primary goal, we also evaluate if we can estimate the size of the test suites that the heuristic generates with an entropy based-estimator. Our motivation for this

comes from the AEP theorem and the definition of typical sets. Through the properties of an $\varepsilon$ typical set (Theorem 2.5) we know that if we have a low entropy, relatively few sequences contain at least probability mass $1-\varepsilon$ of the probability for large sequence lengths $n$. The aim of our estimator is the application of this reasoning for the estimation of the test suite size. The problem we face is that we do not have a large $n$. Both the test cases that we generate and the coverage depths that we evaluate are relatively short. However, we argue that there still should be a correlation between the entropy and the number of test cases that we require. We evaluate this goal by answering the following research question based on the results of this case study.

- RQ1: Is it possible to use an entropy-based estimator for the required test suite size?

### 6.3.2. Evaluation Criteria

The evaluation of hypothesis H12 is straightforward. Either we achieve the desired coverages or we do not. In case we do not achieve the desired values, we investigate if it is because of a problem with the heuristic or the input values (e.g., insufficient test case length). For H13, we evaluate the run-time of the experiments that we perform. The calculation of Equation 3.4.3 is expensive and has to be performed repeatedly in the `while` loop of algorithms 8 and 9. The number of iterations depends on the size of the generated test suite, which grows with high coverage values. By measuring the run-time, we have a metric to evaluate the costs of the heuristic and its scalability. We do not evaluate the scalability in terms of the test case length, because the length depends on the usage profile, as we discussed in Section 3.4.2. We evaluate hypothesis H14 through the comparison of the test suite sizes the that heuristic achieves with test suites with the same coverage that are generated as part of case study 2.

### 6.3.3. Data

We use the same data as in case study 2, i.e., a Web application data set (Section 6.2.3.1), a MFC data set (Section 6.2.3.2), and a JFC data set (Section 6.2.3.3).

### 6.3.4. Methodology

Our methodology is similar to the methodology of case study 2. We generate test suites from different usage profiles for multiple test case lengths and multiple desired coverages.

#### 6.3.4.1. Usage Profiles

We use a subset of the usage profiles that we used in case study 2 (Section 6.2.4.1), i.e., the MMs. We do not use the random EFG because its purpose was to evaluate H8 and we do

not use the PPM models, because we focus on the heuristic and not the comparison between the usage profiles.

#### 6.3.4.2. Test Case Lengths

We use the same test case lengths as in case study 2 (Section 6.2.4.3). However, not all of the lengths are feasible for all usage profiles, because we are bound by the exponential increase of possible test cases. The lengths that we can achieve are the same as the maximal achievable lengths for the hybrid test case generation approach.

#### 6.3.4.3. Desired Coverages

For the desired coverage, we have two parameters: the coverage depth and the desired coverage percentage. We use the coverage depths one to four and four different values for the desired coverages: 80%, 90%, 95%, and 98%. This way, we analyze both the effects of higher coverage depths on the heuristic as well as possible problems with scalability for high coverage values that approach 100%.

### 6.3.5. Results

Whether or not the heuristic is able to determine test suites that achieve the desired coverage depends on the input values. Especially for short test case lengths and deep coverages, the heuristic often fails to achieve the desired coverage. However, especially for depth one coverage only very few test cases are required. For example, for the JFC data set, the depth one coverage of 80% is achievable with only one test case (see Table 6.6). The complete results are depicted in Appendix A.5.

In general, we require more test cases to achieve the desired coverage for the Web data set than for the JFC data. Furthermore, we often do not achieve the desired coverage for the MFC data. However, in all three cases, we reach the desired or maximal achievable coverage with significantly fewer test cases than with randomly generated test suites. For example, the coverage of a size 50 test suite with length eight test cases generated from the Web application data set with a first-order MM is below 80%. In comparison, the heuristic achieve a coverage of 80% with a size 30 test suite with length four test cases (see Table 6.7).

In all cases, where we do not reach the desired coverage, we observed three things. First, with growing test case lengths and, therefore, a growing number of possible sequences, the achieved coverages grow significantly and to such a degree that we argue that the coverage is always achieved if the test cases are sufficiently long. Second, in case we do not achieve the desired coverage, the randomized test case generation strategies also perform poorly, thereby indicating that the desired coverage is too ambitious for the allowed test case length. Third, the generated test suite is almost always smaller than the number of

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 4 | 1 | 86% | 2 | 92.9% | 3 | 96.1% | 6 | 98% |
| | 5 | 1 | 86.8% | | | 2 | 97.1% | | |
| | 6 | | | | 1 | 92.9% | | | |
| | 7 | | | | 1 | 95.8% | | | |
| | 8 | | | | 1 | 97.1% | | | |
| | 9 | | | | 1 | 97.4% | | | |
| 2 | 4 | | | | 61 | 59% | | | |
| | 5 | 3 | 84.5% | 5 | 91.7% | 8 | 95% | 45 | 98% |
| | 6 | 2 | 82.3% | 4 | 91.7% | 6 | 95.3% | 24 | 98% |
| | 7 | 2 | 87.3% | 3 | 92.3% | 5 | 95.3% | 15 | 98% |
| | 8 | 1 | 81.7% | 3 | 93.9% | 4 | 95.6% | 10 | 98% |
| | 9 | 1 | 82.5% | 2 | 91.4% | 4 | 95.6% | 9 | 98% |
| 3 | 4 | | | | 88 | 14.4% | | | |
| | 5 | | | | 190 | 59.1% | | | |
| | 6 | 6 | 81.7% | 12 | 90.6% | 27 | 95.2% | 155 | 98% |
| | 7 | 4 | 80.2% | 9 | 90.9% | 19 | 95.2% | 88 | 98% |
| | 8 | 3 | 80.1% | 7 | 90.7% | 14 | 95.2% | 54 | 98% |
| | 9 | 3 | 83% | 6 | 91.2% | 12 | 95.2% | 37 | 98% |
| 4 | 4 | | | | 107 | 0.5% | | | |
| | 5 | | | | 392 | 14.4% | | | |
| | 6 | | | | 762 | 59.1% | | | |
| | 7 | 12 | 80.3% | 29 | 90.1% | 96 | 95% | 476 | 98% |
| | 8 | 9 | 81.5% | 20 | 90.1% | 65 | 95% | 277 | 98% |
| | 9 | 7 | 81.3% | 16 | 90.2% | 48 | 95% | 179 | 98% |

Table 6.6.: Results of the heuristic test case generation with the 2nd-order MM and the JFC data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 2 | 7 | 81.7% | 11 | 91.5% | 15 | 95.5% | 25 | 98% |
| | 3 | 5 | 83.7% | 7 | 90.7% | 10 | 95.6% | 14 | 98% |
| | 4 | 4 | 85.7% | 5 | 96% | 8 | 96% | 11 | 98.2% |
| 2 | 2 | 109 | 80% | 440 | 90% | | 452 | 91.4% | |
| | 3 | 43 | 80% | 97 | 90% | 173 | 95% | 289 | 98% |
| | 4 | 30 | 80% | 65 | 90% | 110 | 95% | 166 | 98% |
| 3 | 2 | | | | 452 | 28.2% | | | |
| | 3 | 1,279 | 80% | | | 7,270 | 87.1% | | |
| | 4 | 414 | 80% | 1,120 | 90% | 2,835 | 95% | 6,416 | 96.2% |
| 4 | 2 | | | | 452 | 3.8% | | | |
| | 2 | | | | 7,270 | 26.7% | | | |
| | 4 | 14,322 | 80% | | | 116616 | 85.4% | | |

Table 6.7.: Results of the heuristic test case generation with the first-order MM and the Web data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

possible sequences. However, adding more test cases to the test suite does not change the coverage. Hence, the heuristic at least determines a subset of the possible sequences that covers as much as possible.

Another effect that we observed is that the test suites that do not reach the desired coverage are often significantly larger than the ones that do reach the desired coverage. This is due to the effect, that in case the desired coverage is not reached, all sequences with any gain are added, including those with only a very small gain. As soon as the desired coverage is reachable, the sequences with very small gain are dropped from the test suite and, thereby, the overall size of the test suite is reduced.

The runtime of the heuristic was unproblematic for the experiments. The experiments with the Web data set took the longest with 617 minutes, followed by the MFC data set with 114 minutes and the JFC data set with 98 minutes.

### 6.3.6. Discussion

In the following, we discuss our research hypotheses and research questions with respect to the results of this case study.

*H12: The heuristic is able to determine test suites with a desired usage-based coverage.* The results for this hypothesis are mixed. On the one hand, we achieve the desired coverage very often for the lower coverage depths and long test case lengths. On the other hand, we often fail to achieve the desired coverage, especially for coverage depth four and short test

case lengths. We conclude that our heuristic works as we expected in principle. However, the results depend to a great deal on the input parameters. One cannot expect our heuristic to generate a test suite with a coverage of 90% if all of the potential test cases (i.e., all test cases of a given length) together only achieve a coverage of 60%. Furthermore, our results show that with growing test case lengths, the achievable desired coverages grow. Based on these findings, we accept this hypothesis with the addition that the test case length needs to be sufficiently long.

*H13: The heuristic is scalable in terms of desired usage-based coverage.* Our runtime analysis did not reveal any problems with the scalability of the heuristic. Therefore, we accept this hypothesis.

*H14: The test suites determined by the heuristic require less test cases to achieve the same amount of coverage like randomly generated test cases from usage profiles.* Our results provide strong support for this hypothesis. Therefore, we accept this hypothesis.

*RQ1: is it possible to use an entropy-based estimator for the required test suite size?* The entropy of the first-order MMs are 2.37514 for the Web application data, 1.44054 for the MFC data, and 1.06598 for the JFC data. While there seems to be a correlation between the entropy, the number of possible depth $d$ sequences for the coverage, the test case length on the one side and the test suite size on the other side, we were not able to determine an estimator based on these quantities. For our analysis, we only used the Web application data and the JFC data, because the coverage results of the MFC data were to low in general, which leads to using the maximum number test cases without redundancies.

The entropy of the Web application data trained first-order MM models is more than twice as high as the entropy of the model trained with the JFC data. As per our intuition, high entropy indicates higher randomness and, therefore, more test cases are required to achieve a desired coverage, which is directly reflected in the results. We require more test cases for the Web application data than for the JFC data, even though the number of events is similar (67 events for the Web application data, 75 events for the JFC data). However, our attempts to determine how exactly a correlation looks like failed and we need to reevaluate this research question.

To our mind, the main problem is that the structure of the usage profile plays an important role for the estimator. As an example, consider the graph structure of first-order MMs. If the structure describes a complete graph, it is easy to select any subsequence that often occurs, because all sequences are possible. The more sparse the underlying graph structure is, the more difficulties arise to generate test cases with a specific high-probability. Therefore, we theorize that graph theoretic approaches that, e.g., incorporate the degree of nodes and the circumference of the underlying graph structure are required to provide an estimator. Furthermore, advanced correlation analysis techniques, e.g., Principle Component Analysis (PCA) [87] and Factor Analysis (FA) [53] should be used.

# 7. Discussion

In this chapter, we discuss our work in the context of related work. Furthermore, we discuss the limitations of our approach and the threats to the validity of our findings.

## 7.1. Related Work

We divide the discussion of the related work in four parts. First, we discuss how existing capture/replay approaches compare to the capture/replay approach that we defined for Windows MFC. Then, we discuss non-probabilistic model-based EDS testing approaches, whose models are similar to our usage profiles. Afterwards, we discuss how usage profiles are used in the context of Web usage mining. Finally, we discuss the previously presented approaches for the application of usage-based testing methods. We previously published a discussion of the work related to our capture/replay approach [46]. The discussion of this part of the related work is based on this previous publication.

### 7.1.1. Capture/replay GUI Testing

Capture/replay is an approach for GUI testing [49], where executions are recorded and afterwards replayed. The capture/replay approach for Windows MFC that we presented has many similarities with other capture/replay techniques. In this section, we will not compare our approach to every capture/replay technique that has been developed throughout the years, but limit ourselves to those that are similar to our approach. Techniques that are not directly related to our approach and we do not discuss in this section are, e.g., [2, 19, 54, 93, 110].

Steven et al. present the tool jRapture that uses a modified version of the Java API [109]. Their general idea is the replacement of parts of the Java API with a modified version that provides the same functionality, but monitors all available input. This includes not only user interaction with the GUI, i.e., mouse and keyboard events, but also file input. While this seems like an internal monitoring approach at first glance, it is external. The modified version of the Java API is provided by a modified JVM [56]. Thus, the JVM is the capturing tool and the monitoring is not part of the Java application itself. Furthermore, the approach is easily broken through API changes. For example, if a new class that offers further methods for input is introduced, the modified API can not monitor this input without being updated as well. In our work, we use only a select few elemental API functions and it is, therefore, more stable towards API changes.

An approach that is similar to the hooks that we use is presented as part of the GUI-TAR suite by Memon et al. [63, 79]. For the monitoring of Java widgets, the authors use Java's reflection mechanism to manipulate the event and action handlers of the widgets. The approach wraps own handlers around the original ones. The new handlers perform the monitoring and delegate the actual event and action handling to the original one. The concept is the same as the hooks that we use in our work. The events and action handling are not changed, but intercepted to be monitored and otherwise handled as usual. In contrast to our work, the monitoring is only performed for user generated events. The internal communication between the widgets is not part of the monitoring. Furthermore, our monitoring can also extract information about the GUI state and general structure, e.g., widget creation and parent/child relationships between widgets.

Orso and Brian developed a capturing technique that is deployable [86]. The authors use instrumentation to modify the deployed code. The technique does not monitor user actions, but method calls and field manipulations. The instrumentation modifies the generated binary code by modifying the method signatures and field manipulations. In comparison to our approach, the advantage of this approach is that it does not require modification of the source code. However, we only inject binary code that is otherwise separated from the remainder of the application, whereas Orso and Brian change the actual binary code of the complete application. Furthermore, we consider a different level of abstraction. While the windows messages are not on the level of abstraction that we desire, i.e., events, the method calls and field manipulations monitored by Orso and Brian are on an even lower level.

The capture/replay tool WinRunner is part of a larger suite for functional software testing [48]. It uses a *GUI Map* that is similar to our widget tree. The structure of the GUI Map and the window tree are almost the same. Nevertheless, there is a huge difference in how they are obtained and maintained. The GUI Map is created statically through inspection of the widgets of a software. Thus, it is created separately from the capturing and only represents how the structure of the GUI can be, i.e., which widgets could exist at some point during the execution of the application. In comparison, the widget tree is created from information in the capture and represents the state of the GUI dynamically as it changes during the capture.

The *GUI Forest* is another structure similar to our widget tree used by Memon and Soffa [64]. In principle, the properties and generation process of the GUI Forest are the same as that of the GUI Map. The differences between our widget tree and the GUI forest are also as stated above.

### 7.1.2. Model-based Event-driven Software Testing

There are several non-probabilistic EDS testing models similar to the probabilistic usage profiles that we employ. We already discussed the EFG by Memon [62] in Section 2.3. Xie and Memon built on the concept of the EFG to define the Event Interaction Graph (EIG) [129, 130, 131], which tries to account for the internal state of the SUT through the analysis of interdependencies between events. Yuan and Memon further extend this concept into

the Event-Semantic Interaction Graph (ESIG) to incorporate run-time information for the identification relationship between events. Both the EIG and the ESIG are to some degree related to higher order MMs. The higher order models identify relationships between up to their history length automatically. Relationships between events occurring within a distance larger than the Markov order cannot be identified. Furthermore, the EIG and ESIG try to identify only actual relationships, while the higher order MMs automatically assume a relationship within the history length.

The EFG and its variants are implemented in the GUITAR GUI testing suite. GUITAR contains components for the inference of EFGs through a process called GUI ripping as well as test case generation from the EFGs. Similar to our framework, GUITAR works on an abstract notion of events for model inference and test case generation to abstract from concrete EDS platforms. There are two important distinctions between GUITAR and our work. Our framework and GUITAR both provide the means for EDS testing independent of a concrete EDS platform. However, GUITAR does so in form of an implementation and does not provide an abstract framework that describes the structural components required to achieve this goal. Furthermore, GUITAR only considers model-based GUI testing[4], while we consider usage-based GUI testing.

Belli et al. define with the Event Sequence Graph (ESG) an approach similar to the EFG ESG [7, 8]. In comparison to the EFG, the ESG has a set of *entry* and *exit* nodes, which define valid start and end states of the model. This concept is basically the same as the *START* and *END* symbols that we incorporate in our models. In comparison to our work, the ESG does not allow for any histories at all and only accounts for the most recent event, whereas our higher order MMs and PPM models are able to account for the history to a certain degree.

For the testing of GUIs, numerous automaton-based testing strategies have been proposed, e.g., based on Finite State Machines (FSMs) [9, 20, 21] and adaptations of FSMs that deal with the complexity and scalability of the approaches, e.g., the Variable Finite State Machine (VFSM) by Shehady and Siewiorek [103] and Complete Interaction Sequence (CIS) as extension for the FSM by White and Almezen [124]. There are several important distinctions between these approaches and our work. We focus on EDS testing in general and not only on GUI testing. Furthermore, the presented models require manual modeling while we derive our models automatically from usage data.

### 7.1.3. Usage Profiles

While in this work we utilize usage profiles for software testing, the usage-based testing has rather little attention in comparison to another application of usage profiles, such as the *Web usage mining*. The work done in this field is extensive. We found over 300 notable publications on this topic since 1996. Therefore, we give only an overview of the most

[4]While there is a publication by Brooks et al. [12] that utilizes GUITAR for usage-based testing (Section 7.1.4), the developed prototypes are not part of GUITAR.

important papers. However, there are several literature surveys on this topic, e.g., [32, 55, 89, 101, 108].

With the growth of the World Wide Web (WWW) in the middle of the nineties, its usage became an interesting venue for researchers. This sparked the field of Web usage mining, whose basic idea is the identification of patterns of how users navigate a Web site. The first publications on this topic include work on usage pattern recognition and usage data mining by Chen et al. [15, 14], the development of a Web mining tool by Mobasher, Cooley et al. [22, 76], as well as first practical applications for the usage profiles for the generation of adaptive sites by Perkowitz and Billsus [88] and request prediction by Schechter et al. [102].

In the following years, researchers applied the Web usage mining results to various problems, e.g., request prediction [91, 137], pre-fetching [4, 17, 133], caching strategies [13, 132, 133], and usage-based personalization [30, 74, 75]. Of these applications, the request prediction is most closely related to our research. Our procedures for the randomized test cases are basically predictors for the next event. However, the application domains are different. We consider software testing for any type of EDS, while the request prediction in Web usage mining only focuses on predicting page calls for Web applications.

The most often applied type of stochastic process for usage modeling in the context of Web usage mining are MMs. First-order MMs (e.g., [51, 52, 100, 136, 135]), higher order (e.g., [90]), and variable order MMs (e.g., [11, 25, 27]) are all frequently used. Chen and Zhang [16], Shi et al. [104], and Ban et al. [4, 5] also propose PPM models for Web pre-fetching. In comparison to our work, all of the above consider the usage profiles only for one specific application, while we use the usage profiles as part of a versatile framework.

### 7.1.4. Usage-based Event-driven Software testing

The concept of utilizing the usage of a software for quality assurance is not novel. In the literature, the term *operational profile* is often used instead of usage profile. The usage modeling has first been suggested in context of user-oriented software reliability modeling by Littlewood [58] and Cheung [18]. In both cases, the systems are modeled as a first-order MM and the reliability is assessed based on a reliability assumption for each component and the probability that a component is called. The same modeling concept was later adapted by Whittaker et al. [125, 126], Walton et al. [121], and Wesslén and Wohlin [123] for the generation of test cases by randomly walking the usage profile. A slightly different model that allows the definition of conditions to restrict, which events are possible, was proposed by Woit [128, 127].

A different approach towards using first-order MMs for the software quality optimization has been taken by Gutjahr [37, 35, 36]. Here, the probabilities of the usage profile are adjusted such that not only often used parts of the software have high probabilities, but also functionally critical parts, independent of their usage. The aim of this study is to mitigate the weakness of usage-based testing that ignores the criticality of software parts in favor of

only considering the usage. Doerner and Gutjahr build upon this work to allow conditions on event transitions other than the first-order Markov property [26].

A tool-based approach for usage-based testing in context of the Testing and Test Control Notation 3 (TTCN-3) has been proposed by Dulz and Zhen [28] and Guen et al. [34]. They infere the structure of a first-order MM from available sequence information about the SUT, e.g., Message Sequence Charts (MSC) or UML diagrams. To reflect the actual usage of a SUT, the probabilities in this model need to be set by an expert. They are not automatically inferred from usage information.

There are three principle differences between our approach and all of the above works. First, all of the above works rely on someone pre-defining the model, including the usage probabilities, with the exemption of [28, 34], who infer the model structure (but also not the probabilities) automatically. Hence, the models are derived purely from expert knowledge and possibly do not reflect the actual usage of the SUT. In comparison, we automatically infer our models based on usage data obtained from observing the SUT. Second, the above approaches only generate abstract test cases that need to be either manually executed by a tester or converted into automatically executable test cases by a test engineer. In our work, we include this step in our model through the translation layer and thereby allow the generation of automatically executable test cases directly from the model. Third, the work presented above only works with first-order MM (or slight adaptations). The authors do not consider other stochastic process. In comparison, we also include higher order MMs and PPM models in our approach and its analysis.

With the growth of Web applications, their testing became more important leading to advanced Web testing techniques. This also led to considerations of usage-based Web testing. Tonella and Ricca defined a usage-based testing approach for Web applications [116, 114, 115] based on first-order MMs. Their testing procedure is completely automated, and includes the inference of the usage profile. In comparison to our work, Tonella and Ricca only work with first-order MMs, while we consider also other stochastic processes. Furthermore, they focus solely on Web applications, while we provide a framework for usage-based testing for all kinds of EDS.

Another approach towards usage-based Web testing was defined by Sampath et al. [94, 95, 96, 97, 98]. The authors use different concepts for the usage-based testing. They use a concept analysis clustering approach for the reduction of test suites and prioritization of test cases. The approach was empirically evaluated by Sprenkle et al. [107]. While the approach is usage-based, the methodology is completely different from our work and not based on stochastic processes. Additionally, the authors focus solely on Web applications and do not take other EDS software systems into account.

Sant et al. considered usage-based test case generation for Web applications based on the concept of $n$-grams [99]. In principle, a $n$-gram can be considered to be a $(n-1)$-th order MM. The authors conducted a study that analyses the effect on the order of the MM on the test case generation in terms of statement coverage. The test generation procedure they employed are random walks. Similar studies were conducted by Sprenkle et al. [106, 105].

In comparison to Sant et al.'s study, Sprenkle et al.'s studies have a larger scope, because they consider more applications and $n$-gram orders. Furthermore, they analyze the structure of the generated test cases in terms of how the generated test cases look like in comparison to the training data. These studies are similar to our second case study (Section 6.2). While we did not consider statement coverage directly, but rather event coverage, the two coverages are known to be closely related [65]. However, there are several distinctions between our studies. We considered with our hybrid approach a second test case generation method. Furthermore, we also considered GUI applications and not only web applications. Moreover, the main objective of our investigation was the analysis of the usage-based coverage and the standard coverage was only used a means for comparison.

To our knowledge, Brooks and Memon [12] conducted the only study that combines usage-based testing and GUI testing with automated model inference. The authors combine the EFG with usage information to generate higher order MMs in order to generate test cases. The developed MM closely resemble the ones that we generate. However, from their study, it is unclear how the usage data is obtained. Furthermore, the study is restricted to Java GUIs. While the underlying GUITAR software allows greater flexibility, the authors do not utilize it in their studies. Additionally, the authors focus solely on MMs, while we also incorporate PPM models in our studies. In another study, Memon observes the software to collect usage data for testing [61]. However, the collected data is not used to generate a usage profile but replayed directly as it has been observed.

There are also models that solely deal with the question of how efficient usage profiles for software testing look like. Takagi and Furukawa discuss the construction of high-order MMs software usage models and how they can be estimated, as well as how expensive this procedure is [111]. Bochmann et al. discuss how the complexity of a derived model can be reduced with state merging techniques [10]. In comparison to our work, both of the authors do not study the application of their models to software testing.

## 7.2. Strengths and Limitations

Our approach and the case studies that we conducted have several threats to validity and limitations. In this section, we first discuss the strengths of our approach. Then, we analyze the limitations of our framework and discuss the threats to the validity of our case studies.

### 7.2.1. Strengths of our Approach

The greatest strength of our approach is its versatility. Other research is focused on single event platforms. To our knowledge only the GUITAR tooling by Memon et al. provides similar capabilities for the abstraction of specific GUI platforms. Still, we do not only provide the capabilities for the abstraction, but provide a framework that describes the required components and defines interfaces for the implementation. Furthermore, in the context of

usage-based testing, there is no existing approach that allows the application of testing in multiple domains, i.e., Web applications as well as GUI applications. Additionally, the translation layer allows for compability of instantiations of our framework with already existing tooling, as we show through the translation layers for GUITAR and curl. Moreover, the usage-based coverage provides a novel view on the test coverage to assess the test process from a user quality point of view.

### 7.2.2. Limitations of the Framework and its Instantiation

The framework that we devised has several limitations in its current state. We only consider usage profiles that are inferred from training data that we obtained from a platform. We do not allow the manual definition or change of usage profiles. Therefore, the incorporation of already existing models of the SUT, e.g., in form of UML diagrams into the framework, is not possible. Furthermore, the instantiation is a research prototype. While we work with an industrial partner to test the framework in practice, we have not evaluated if it is indeed feasible in a large-scale industrial setting. Parts of the implementation at the current state are only feasible to proof our concept but not for practical applications. For example, we only implemented two assertions, which are insufficient for real-world test suites.

### 7.2.3. Threats to Validity

We acknowledge several threats to the validity of our work. We did not analyze the capability to detect errors in our case studies, e.g., by executing the test suites and analyzing the capabilities to detect faults that we seeded. However, we do not feel that this is necessary, since the fault finding capabilities of our approach are the same as of other usage based testing methods and EDS testing methods in general, i.e., all faults that are caused by specific event sequences can potentially be found. Studies of the fault finding capabilities have been conducted , e.g., by Sant et al. [99] for usage-based testing and Memon [62] for the EFG. Moreover, two of our three data sets were obtained in artificial settings and not during real-life software usage.

# 8. Conclusion

In this final chapter of the thesis, we summarize our findings and contributions. Furthermore, we give an outlook on future work in this research direction.

## 8.1. Summary

The general purpose of this work was to address problems from the field of usage-based testing in the context of EDS. Our first contribution to this aim is a framework for usage-based EDS testing, which allows the application of usage-based EDS testing technologies independent of concrete EDS platforms. Our framework consists of three layers: 1) a platform layer that contains the platform specific usage-monitors and test execution tools; 2) a translation layer that converts the platform specific information into abstract events; 3) an event layer that contains the usage-based testing logic and works on abstract events. We instantiated our framework and applied the usage-based testing methods implemented in the event layer to three different platforms: PHP-based Web applications, Windows MFC GUI applications, and Java JFC GUI applications. Furthermore, we use the translation layer to provide a bridge between our framework and the GUITAR framework for model-driven testing of EDS and curl.

As part of our implementation, we devised a novel capture/replay technique for Windows MFC applications. In comparison to existing capture/replay tools, our technique provides an internal monitoring component that can be integrated into the SUT. This allows the deployment of the monitoring to costumers in order to obtain usage data. Furthermore, our approach uses a novel method to abstract from screen coordinates to facilitate coordinate-independent replaying of executions. To this aim, we utilize not only the external stimuli (e.g., mouse clicks), but also the internal communication of the application.

We advance the state-of-the-art of usage-based testing through the definition of novel usage-based coverage criteria and the advancement of usage-based test case generation. Our coverage criteria are not based on the absolute percentage of a SUT model that is covered, but rather on the probability mass according to a usage profile. Hence, test cases that are very probable according to a usage profile increase the coverage more than those that are improbable. For the test case generation, we defined approaches based on selecting test cases from all possible test cases of a given length in addition to consideration of the random walk based test case generation that is commonly applied in the literature. On the one hand, we combine the probabilistic selection from all possible sequences with random

walks in a hybrid approach for the test case generation. On the other hand, we define a heuristic that selects test cases based on the probability mass that the test cases cover in order to achieve a desired usage-based coverage with a test suite.

We evaluated our work in three case studies. In our first case study, we showed the feasibility of the capture/replay approach that we defined both in a lab setting and together with an industrial partner with a large scale industrial software. In our second case study, we evaluated the usage-based coverage in comparison to non-usage-based coverage to show the differences in performances for usage-based test case generation. Furthermore, we compared the different probabilistic test case generation approaches, i.e., the random walk and our hybrid approach. We conclude that the usage-based coverage is an effective tool for the evaluation of usage-based test suites. Furthermore, our findings show that the two test case generation methods are almost equal, regardless of the differences in methodology. In the third case study, we evaluated our test suite generation heuristic in terms of runtime efficiency, the size of the generated test suites, and if we are able to achieve desired coverage values. Our findings show, that the results of the heuristic depend on the input values. If the allowed test case length is reasonable in comparison to the desired coverage, the heuristic is able to determine comparatively small test suites that achieve a predefined usage-based coverage.

## 8.2. Outlook

There are several open problems that provide opportunities for future research. The extension of the framework instantiation to additional EDS platforms is an important task for the applicability of our testing techniques in a broad spectrum. Important platforms, such as Web service platforms that use Web Service Description Language (WSDL) are not yet supported by our implementation. Additionally, some types of EDS were not yet part of our considerations, e.g., embedded systems. The extension to additional platform would further validate the platform-independence that we gain through our framework, as well as provide opportunities for further case studies.

A different direction is the extension of the framework itself. Currently, the framework only supports models that are trained directly from usage data. It is desirable to combine the usage model with existing models of the software that are created as part of the software development process, e.g., UML state machines and activity diagrams to improve the quality of the usage profiles. This also leads to the possibility of a testing approach that combines model-driven testing with usage-based testing, e.g., to determine test suites that achieve a given usage-based coverage but also other coverage properties in relation to a model of the software. Furthermore, the combination of probabilistic model inference and exact model inference techniques as they have been analyzed by Werner and Grabowski [122] could be a solution to improve the accuracy of the usage profiles.

A different way to extend the framework is its generalization from the testing of EDS to the testing of arbitrary software. To achieve this, research has to be conducted on how such an abstract representation in this scenario looks like and how the layers have to be adapted to achieve this representation. While this is a complex task that might even be impossible to achieve, the gain would be enormous. A practical step by step way to generalize the framework should be a feasible solution to this problem, i.e., start with the extension of the framework for another type of software, e.g., cloud computing systems.

In the future, we also plan to improve the usage-based testing methods, which are already implemented as part of our framework instantiation. In the current state, the test oracles are rather weak, i.e., they require that a tester defines the assertions without any support for automation. An automated test oracle would greatly improve the effectivity of automated test case generation, as the error finding capabilities of the test cases would be improved from a simple test oracle to complex evaluations of the SUT. Another venue for the improvement of our instantiation is to research further test case generation strategies. Our heuristic already shows that very small test suites are sufficient for usage-based testing. However, the heuristic follows a greedy strategy and does not yield the optimal result. While the exact calculation of the optimal result is with a high likelyhood infeasible due to the computational complexity, other heuristic strategies, e.g., ant algorithms or genetic programming could lead to better results.

Moreover, the extension of the usage profiles with information about known failures to provide usage-based reliability assessment of the software is a valuable extension of the framework. The usage profile are used as means to determine how probable it is that an event sequence that leads to a failure is actually executed by a user. This probability is the reliability of the system, given that the usage profile is accurate and all failures are known. The project management and the quality assurance team can use the results of the reliability analysis for the internal analysis of the software quality. Furthermore, the reliability is an important parameter for the estimation and steering of maintenance efforts. Moreover, the probability that a specific failure occurs is a good indicator for the software maintenance team which failures should be fixed first, i.e., the ones that often occur.

Another important task is a case study that evaluates the usage-based test suites with respect to its error finding probabilities. To this aim, a usage profile needs to be trained from a SUT that contains known failures. Afterwards, test suites generated from the usage profile need to be executed against the SUT in order to analyze which failures are found. The results should be compared to test suites of similar size that have been generated with a non usage-based approach.

# Bibliography


[1] IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology, 1990.

[2] Bowen Alpern, Ton Ngo, Jong-Deok Choi, and Manu Sridharan. DejaVu: deterministic Java replay debugger for Jalapeño Java virtual machine. In *Addendum to the 2000 Proceedings of the Conference on Object-oriented programming, systems, languages, and applications (Addendum)*, OOPSLA '00, pages 165–166, New York, NY, USA, 2000. ACM.

[3] Apache Foundation. Apache HTTP Server. `http://httpd.apache.org/`, June 2012.

[4] Zhijie Ban, Zhimin Gu, and Yu Jin. An online PPM prediction model for web prefetching. In *WIDM '07: Proceedings of the 9th annual ACM international workshop on Web information and data management*, pages 89–96, New York, NY, USA, 2007. ACM.

[5] Zhijie Ban, Zhimin Gu, and Yu 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*, pages 166–173, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order Markov models. *Journal of Artificial Intelligence Research*, 22:385–421, December 2004.

[7] Fevzi Belli. Finite state testing and analysis of graphical user interfaces. In *Proceedings of 12th International Symposium on Software Reliability Engineering (ISSRE)*, 11 2001.

[8] Fevzi Belli, Christof J. Budnik, and Lee White. Event-based modelling, analysis and testing of user interactions: approach and case study. *Software Testing, Verification and Reliability*, 16(1):3–32, 2006.

[9] Philip J. Bernhard. A reduced test suite for protocol conformance testing. *ACM Transactions on Software Engineering Methodologies*, 3(3):201–220, July 1994.

[10] Gregor v. Bochmann, Guy-Vincent Jourdan, and Bo Wan. Improved usage model for web application reliability testing. In *Proceedings of the 23rd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'11, pages 15–31, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Jose Borges and Mark Levene. Evaluating Variable-Length Markov Chain Models for Analysis of User Web Navigation Sessions. *IEEE Transactions on Knowledge and Data Engineering*, 19(4):441–452, 2007.

[12] Penelope Brooks and Atif M. Memon. Automated GUI Testing Guided by Usage Profiles. In *ASE '07: Proceedings of the 22nd IEEE international conference on Automated software engineering*, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Cheng-Yue Chang and Ming-Syan Chen. A new cache replacement algorithm for the integration of web caching and prefectching. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pages 632–634, New York, NY, USA, 2002. ACM.

[14] Ming-Syan Chen, Jong Soo Park, and Philip. S. Yu. Data mining for path traversal patterns in a web environment. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 385–392, May 1996.

[15] Ming-Syan Chen, Jong Soo Park, and Philip S. Yu. Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):209–221, 1998.

[16] Xin Chen and Xiaodong 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*, page 296, Washington, DC, USA, 2002. IEEE Computer Society.

[17] Xin Chen and Xiaodong Zhang. A popularity-based prediction model for web prefetching. *IEEE Computer*, 36(3):63–70, 2003.

[18] Roger. C. Cheung. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, 6(2):118–125, 1980.

[19] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '98, pages 48–59, New York, NY, USA, 1998. ACM.

[20] Tsun. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178 – 187, may 1978.

[21] James M. Clarke. Automated test generation from a behavioral model. In *in Proceedings of Pacific Northwest Software Quality Conference*. IEEE Press, 1998.

[22] Robert Cooley, Bamshad Mobasher, and Jaideep Srivastava. Web mining: Information and pattern discovery on the world wide web. *IEEE International Conference on Tools with Artificial Intelligence*, 0:0558, 1997.

[23] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., 2nd edition, 2006.

[24] Daniel Stenberg et al. curl. `http://curl.haxx.se/`, June 2012.

[25] Mukund Deshpande and George Karypis. Selective Markov models for predicting Web page accesses. *ACM Transactions on Internet Technology*, 4(2):163–184, 2004.

[26] Karl Doerner and Walter J. Gutjahr. Representation and optimization of software usage models with non-Markovian state transitions. *Information and Software Technology*, 42(12):873–887, 2000.

[27] Xing Dongshan and Shen Junyi. A new markov model for web access prediction. *Computing in Science and Engineering*, 4(6):34–39, 2002.

[28] Winfried Dulz and Fenhua Zhen. Matelo - statistical usage testing by annotated sequence diagrams, markov chains and ttcn-3. In *Proceedings of the Third International Conference on Quality Software*, QSIC '03, pages 336–, Washington, DC, USA, 2003. IEEE Computer Society.

[29] Albert Einstein. *Investigations on the Theory of Brownian Movement*. Dover Books on Physics. Dover Publications, 1956.

[30] Magdalini Eirinaki and Michalis Vazirgiannis. Web mining for web personalization. *ACM Transactions on Internet Technology*, 3(1):1–27, 2003.

[31] Software Engineering and Distributed Systems Group of the University Göttingen. Website of the Group for Software Engineering and Distributed Systems of the University Göttingen. `http://www.swe-old.informatik.uni-goettingen.de`, June 2011.

[32] Federico Michele Facca and Pier Luca Lanzi. Mining interesting knowledge from weblogs: a survey. *Data Knowledge Engineering*, 53(3):225–241, 2005.

[33] William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, Inc, 1971.

[34] Helene Le Guen, Raymond Marie, and Thomas Thelin. Reliability estimation for statistical usage testing using markov chains. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, pages 54–65, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Walter J. Gutjahr. Failure risk estimation via Markov software usage models. In *SAFECOMP 96, Proc. of the 15th International Conference on Computer Safety, Reliability and Security*, pages 183–192. Citeseer, 1997.

[36] Walter J. Gutjahr. Software dependability evaluation based on markov usage models. *Performance Evaluation*, 40(4):199–222, 2000.

[37] W.J. Gutjahr. Importance sampling of test cases in Markovian software usage models. *Probability in the engineering and informational sciences*, 11:19–36, 1997.

[38] Steffen Herbold. EventBench. `http://eventbench.informatik.uni-goettingen.de/`, June 2012.

[39] Steffen Herbold. EventBenchConsole. `http://eventbench.informatik.uni-goettingen.de/trac/wiki/Software/EventBenchConsole`, June 2012.

[40] Steffen Herbold. EventBenchCore. `http://eventbench.informatik.uni-goettingen.de/trac/wiki/Software/EventBenchCore`, June 2012.

[41] Steffen Herbold. JFCMonitor. `http://eventbench.informatik.uni-goettingen.de/trac/wiki/Software/JFCMonitor`, June 2012.

[42] Steffen Herbold. MFCReplay. `http://eventbench.informatik.uni-goettingen.de/trac/wiki/Software/MFCReplay`, June 2012.

[43] Steffen Herbold. MFCUsageMonitor. `http://eventbench.informatik.uni-goettingen.de/trac/wiki/Software/userlog`, June 2012.

[44] Steffen Herbold. PHPMonitor. `http://eventbench.informatik.uni-goettingen.de/trac/wiki/Software/PHPMonitor`, June 2012.

[45] Steffen Herbold, Uwe Bünting, Jens Grabowski, and Stephan 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.

[46] Steffen Herbold, Uwe Bünting, Jens Grabowski, and Stephan Waack. Deployable Capture/Replay Supported by Internal Messages. *Advances in Computers*, 85:327–367, 2012.

[47] Steffen Herbold, Jens Grabowski, and Stephan Waack. A Model for Usage-based Testing of Event-driven Software. In *3rd International Workshop on Model-Based Verification & Validation From Research to Practice*. IEEE Computer Society, June 2011.

[48] Hewlet-Packard Company. HP Functional Testing, August 2011.

[49] James H. Hicinbothom and Wayne W. Zachary. A Tool for Automatically Generating Transcripts of Human-Computer Interaction. In *Human Factors and Ergonomics Society 37th Annual Meeting*, volume 2 of Special Sessions, page 1042, 1993.

[50] International Software Testing Qualitifications Board (ISTQB). Standard glossary of terms used in Software Testing, Version 2.1, 4 2010.

[51] Søren Jespersen, Torben Bach Pedersen, and Jesper Thorhauge. Evaluating the markov assumption for web usage mining. In *WIDM '03: Proceedings of the 5th ACM international workshop on Web information and data management*, pages 82–89, New York, NY, USA, 2003. ACM.

[52] Dimitrios Katsaros and Yannis Manolopoulos. Prediction in wireless networks by markov chains. *IEEE Wireless Communications*, 16(2):56–63, 2009.

[53] Jae-on Kim and Charles W. Mueller. *Introduction to Factor Analysis: What It Is and How to Do It* . SAGE Publications Inc., 1978.

[54] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. Deterministic replay of distributed Java applications. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 219 –227, 2000.

[55] Raymond Kosala and Hendrik Blockeel. Web mining research: a survey. *ACM SIGKDD Explorations Newsletter*, 2(1):1–15, 2000.

[56] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[57] Linux Information Project. GUI Definition. `http://www.linfo.org/gui.html`, 10 2004.

[58] Bev Littlewood. A reliability model for systems with markov structure. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 24(2):172–177, 1975.

[59] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[60] Peter Mell and Tim Grance. The nist definition of cloud computing. `http://www.nist.gov/itl/cloud/upload/cloud-def-v15.pdf`, July 2009.

[61] Atif M. Memon. Employing user profiles to test a new version of a GUI component in its context of use. *Software Quality Journal*, 14(4):359–377, 2006.

[62] Atif M. Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[63] Atif M. Memon, Ishan Banerjee, Adithya Nagarajan, Bao N. Nguyen, Bryan Robbins, Xun Yuan, and Qing Yie. Guitar. `http://guitar.sourceforge.net`, June 2012.

[64] Atif M. Memon and Mary Lou Soffa. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 118–127, New York, NY, USA, 2003. ACM.

[65] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for gui testing. *SIGSOFT Software Engineering Notes*, 26:256–267, September 2001.

[66] Robert C. Merton. *Continuous Time Finance*. Wiley-Blackwell, 1992.

[67] Microsoft. About Messages and Message Queues. `http://msdn.microsoft.com/en-us/library/ms644927.aspx`, June 2012.

[68] Microsoft. About Window Classes. http://msdn.microsoft.com/en-us/library/windows/desktop/ms633574.aspx, June 2012.

[69] Microsoft. Common Control Parameters. `http://msdn.microsoft.com/en-us/library/windows/desktop/aa380902.aspx`, June 2012.

[70] Microsoft. Hooks. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms632589.aspx`, June 2012.

[71] Microsoft. Menus and Other Resources. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms632583.aspx`, June 2012.

[72] Microsoft. Microsoft Foundation Classes. `http://msdn.microsoft.com/en-us/library/d06h2x6e.aspx`, June 2012.

[73] Microsoft. Windows Data Types. `http://msdn.microsoft.com/en-us/library/aa383751.aspx`, June 2012.

[74] Bamshad Mobasher. Data mining for web personalization. *Lecture Notes in Computer Science*, 4321:90, 2007.

[75] Bamshad Mobasher, Honghua Dai, Tao Luo, and Miki Nakagawa. Discovery and Evaluation of Aggregate Usage Profiles for Web Personalization. *Data Mining Knowledge Discovery*, 6(1):61–82, 2002.

[76] Bamshad Mobasher, Namit Jain, Eui-Hong (Sam) Han, and Jaideep Srivastava. Web mining: Pattern discovery from world wide web transactions. Technical report, 1996.

[77] Alistair Moffat. Implementing the ppm data compression scheme. *IEEE Transactions on Communications*, 38(11):1917 –1921, nov 1990.

[78] Glenford J. Myers. *The art of software testing*. Business Data Processing: A Wiley Series. John Wiley & Sons, Inc, 2004.

[79] Adithya Nagarajan and Atif M. Memon. Refactoring Using Event-based Profiling. In *Proceedings of The First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

[80] Nokia Corporation. Qt SDK. `http://qt.nokia.com/products/`, June 2012.

[81] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[82] Oracle. Abstract Window Toolkit (AWT). `http://docs.oracle.com/javase/6/docs/technotes/guides/awt/`, June 2012.

[83] Oracle. Java. `http://www.oracle.com/us/technologies/java/overview/index.html`, June 2012.

[84] Oracle. Java 2D API. `http://www.oracle.com/technetwork/java/index-jsp-138693.html`, June 2012.

[85] Oracle. Swing. `http://docs.oracle.com/javase/6/docs/technotes/guides/swing/`, June 2012.

[86] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM.

[87] Karl Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572, 1901.

[88] Mike Perkowitz and Oren Etzioni. Adaptive sites: Automatically learning from user access patterns. In *Proc. 6th Int. World Wide Web Conf., Santa Clara, California*. Citeseer, 1997.

[89] Dimitrios Pierrakos, Georgios Paliouras, Christos Papatheodorou, and Constantine D. Spyropoulos. Web usage mining as a tool for personalization: A survey. *User Modeling and User-Adapted Interaction*, 13(4):311–372, 2003.

[90] Peter L. T. Pirolli and James E. Pitkow. Distributions of surfers' paths through the World Wide Web: Empirical characterizations. *World Wide Web*, 2(1-2):29–45, 1999.

[91] James E. Pitkow and Peter L. T. 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*, pages 13–13, Berkeley, CA, USA, 1999. USENIX Association.

[92] Sheldon M. Ross. *Probability Models for Computer Science*. Harcourt/Academic Press, 2002.

[93] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. *ACM SIGPLAN Notices*, 31:258–266, May 1996.

[94] Sara Sampath, Amie Souter, and Lori Pollock. Towards defining and exploiting similarities in web application use cases through user session analysis. In *Proceedings of the Second International Workshop on Dynamic Analysis*. Citeseer, 2004.

[95] Sreedevi Sampath, Renee C. Bryce, Gokulanand Viswanath, Vani Kandimalla, and A. Gunes Koru. Prioritizing user-session-based test cases for web applications testing. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 141–150, Washington, DC, USA, 2008. IEEE Computer Society.

[96] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 132–141, Washington, DC, USA, 2004. IEEE Computer Society.

[97] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, and Amie Souter. Analyzing clusters of web application user sessions. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM.

[98] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, and Amie Souter Greenwald. Applying concept analysis to user-session-based testing of web applications. *IEEE Transactions on Software Engineering*, 33(10):643–658, 2007.

[99] Jessica Sant, Amie Souter, and Lloyd Greenwald. An exploration of statistical models for automated test case generation. *SIGSOFT Software Engineering Notes*, 30:1–7, May 2005.

[100] Ramesh R. Sarukkai. Link prediction and path analysis using markov chains. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 377–386, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[101] Jost Schatzmann, Karl Weilhammer, Matt Stuttle, and Steve Young. A survey of statistical user simulation techniques for reinforcement-learning of dialogue management strategies. *The Knowledge Engineering Review*, 21(2):97–126, 2006.

[102] Stuart Schechter, Murali Krishnan, and Michael D. Smith. Using path profiles to predict http requests. *Computer Networks and ISDN Systems*, 30(1-7):457–467, 1998.

[103] Richard K. Shehady and Daniel P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 80 –88, jun 1997.

[104] Lei Shi, Zhimin Gu, Yunxia Pei, and Lin Wei. A ppm prediction model based on web objects' popularity. In *Proceedings of the Second international conference on Fuzzy Systems and Knowledge Discovery - Volume Part II*, FSKD'05, pages 110–119, Berlin, Heidelberg, 2005. Springer-Verlag.

[105] Sara Sprenkle, Camille Cobb, and Lori Pollock. Leveraging user-privilege classification to customize usage-based statistical models of web applications. In *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[106] Sara Sprenkle, Lori Pollock, and Lucy Simko. A study of usage-based navigation models and generated abstract test cases for web applications. In *IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, 2011.

[107] Sara Sprenkle, Sreedevi Sampath, Emily Gibson, Lori Pollock, and Amie Souter. An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 587–596, Washington, DC, USA, 2005. IEEE Computer Society.

[108] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web usage mining: discovery and applications of usage patterns from web data. *ACM SIGKDD Explorations Newsletter*, 1(2):12–23, 2000.

[109] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jRapture: A Capture/Replay tool for observation-based testing. *SIGSOFT Software Engineering Notes*, 25(5):158–167, 2000.

[110] Kuo-Chung Tai, Richard H. Carver, and Evelyn E. Obaid. Debugging Concurrent Ada Programs by Deterministic Execution. *IEEE Transactions on Software Engineering*, 17:45–63, January 1991.

[111] Tomohiko Takagi and Zengo Furukawa. Construction Method of a High-Order Markov Chain Usage Model. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2007.

[112] The PHP Group. PHP Hypertext Preprocessor. `http://www.php.net`, June 2012.

[113] Tigris.org. Argouml. `http://argouml.tigris.org/`, June 2012.

[114] Paolo Tonella and Filippo Ricca. Dynamic model extraction and statistical analysis of web applications. In *WSE '02: Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)*, page 43, Washington, DC, USA, 2002. IEEE Computer Society.

[115] Paolo Tonella and Filippo Ricca. Statistical testing of web applications. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2):103–127, 2004.

[116] Paolo Tonella and Filippo Ricca. Dynamic Model Extraction and Statistical Analysis of Web Applications: Follow-up after 6 years. In *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*, pages 3–10, 2008.

[117] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley & Sons, Inc., 2002.

[118] W3C. Web Application Description Language. `http://www.w3.org/Submission/wadl/`, June 2012.

[119] W3C. Web Services Glossary. `http://www.w3.org/TR/ws-gloss/`, June 2012.

[120] Stephan Waack, Oliver Keller, Roman Asper, Thomas Brodag, Carsten Damm, Wolfgang Fricke, Katharina Surovcik, Peter Meinicke, and Rainer Merkl. Score-based prediction of genomic islands in prokaryotic genomes using hidden markov models. *BMC Bioinformatics*, 7(1):142, 2006.

[121] Gwendolyn H. Walton, Jesse H. Poore, and Carmen J. Trammell. Statistical testing of software based on a usage model. *Software: Practice ant Experience*, 25(1):97–108, 1995.

[122] Edith Werner and Jens Grabowski. Model Reconstruction: Mining Test Cases. In *Third International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011).*, October 2011.

[123] Anders Wesslén and Claes Wohlin. Modelling and Generation of Software Usage. In *Proceedings Fifth International Conference on Software Quality*, pages 147–159, 1995.

[124] Lee White and Husain Almezen. Generating test cases for gui responsibilities using complete interaction sequences. In *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, pages 110 –121, 2000.

[125] James A. Whittaker and Jesse H. Poore. Markov analysis of software specifications. *ACM Transactions Software Engineering Methodologies*, 2(1):93–106, 1993.

[126] James A. Whittaker and Michael G. Thomason. A Markov Chain Model for Statistical Software Testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, 1994.

[127] Denise M. Woit. Specifying operational profiles for modules. *SIGSOFT Software Engineering Notes*, 18(3):2–10, July 1993.

[128] Denise M. Woit. Conditional-event usage testing. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '98, pages 23–. IBM Press, 1998.

[129] Qing Xie and Atif M. Memon. Rapid "crash testing" for continuously evolving gui-based software applications. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 473 – 482, sept. 2005.

[130] Qing Xie and Atif M. Memon. Model-based testing of community-driven open-source gui applications. In *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pages 145 –154, sept. 2006.

[131] Qing Xie and Atif M. Memon. Using a pilot study to derive a GUI model for automated testing. *ACM Transactions on Software Engineering and Methodologies*, 18(2):7:1–7:35, November 2008.

[132] Qiang Yang and Haining Henry Zhang. Web-log mining for predictive web caching. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1050–1053, 2003.

[133] Qiang Yang, Haining Henry Zhang, and Tianyi Li. Mining web logs for prediction models in www caching and prefetching. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 473–478, New York, NY, USA, 2001. ACM.

[134] Xun Yuan and Atif M. Memon. Generating event sequence-based test cases using gui run-time state feedback. *IEEE Transactions on Software Engineering*, 36(1):81–95, 2010.

[135] Jianhan Zhu, Jun Hong, and John G. Hughes. Using markov chains for link prediction in adaptive web sites. In *Soft-Ware 2002: Proceedings of the First International Conference on Computing in an Imperfect World*, pages 60–73, London, UK, 2002. Springer-Verlag.

[136] Jianhan Zhu, Jun Hong, and John G. Hughes. Using markov models for web site link prediction. In *HYPERTEXT '02: Proceedings of the thirteenth ACM conference on Hypertext and hypermedia*, pages 169–170, New York, NY, USA, 2002. ACM.

[137] Ingrid Zukerman, David W. Albrecht, and Ann E. Nicholson. Predicting users' requests on the www. In *UM '99: Proceedings of the seventh international conference on User modeling*, pages 275–284, Secaucus, NJ, USA, 1999. Springer-Verlag New York, Inc.

# A. Appendix

## A.1. Listings for the Capture/Replay Approach for Windows MFC GUIs

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <!DOCTYPE rules SYSTEM "ruleDoctype.dtd">
3  <rules xmlns="ul:rules" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
       ul:rules ruleSchema.xsd">
4
5    <!-- rules regarding mouse clicks -->
6    <rule name="LeftClickButton">
7      <msg type="&WM_LBUTTONDOWN;">
8        <store var="clicked"/>
9      </msg>
10     <msg type="&WM_LBUTTONUP;">
11       <equals>
12         <constValue value="Button"/>
13         <winInfoValue obj="this" winParam="class"/>
14       </equals>
15       <equals>
16         <paramValue obj="clicked" param="window.hwnd"/>
17         <paramValue obj="this" param="window.hwnd"/>
18       </equals>
19     </msg>
20     <genMsg delay="500">
21       <type>
22         <constValue value="&BM_CLICK;"/>
23       </type>
24       <target>
25         <msgInfoValue obj="clicked" msgParam="target"/>
26       </target>
27     </genMsg>
28   </rule>
29
30   <rule name="LeftClickListBox">
31     <msg type="&WM_LBUTTONDOWN;">
32       <equals>
33         <winInfoValue obj="this" winParam="class"/>
34         <constValue value="ListBox"/>
35       </equals>
36       <store var="clicked"/>
37     </msg>
38     <msg type="&WM_LBUTTONUP;">
39       <equals>
40         <paramValue obj="this" param="window.hwnd"/>
41         <paramValue obj="clicked" param="window.hwnd"/>
42       </equals>
```

```
43        <store var="up"/>
44      </msg>
45      <genMsg delay="500">
46        <type>
47          <constValue value="&LB_SETCURSEL;"/>
48        </type>
49        <target>
50          <msgInfoValue obj="clicked" msgParam="target"/>
51        </target>
52        <WPARAM>
53          <paramValue obj="up" param="scrollPos"/>
54        </WPARAM>
55      </genMsg>
56    </rule>
57
58    <rule name="TabChange">
59      <msg type="&WM_LBUTTONDOWN;">
60        <equals>
61          <constValue value="SysTabControl32"/>
62          <winInfoValue obj="this" winParam="class"/>
63        </equals>
64        <store var="clicked"/>
65      </msg>
66      <msg type="&WM_LBUTTONUP;">
67        <equals>
68          <paramValue obj="this" param="window.hwnd"/>
69          <paramValue obj="clicked" param="window.hwnd"/>
70        </equals>
71        <store var="up"/>
72      </msg>
73      <!-- tab change message for Tab Controls-->
74      <genMsg delay="100">
75        <type>
76          <constValue value="&TCM_SETCURSEL;"/>
77        </type>
78        <target>
79          <msgInfoValue obj="up" msgParam="target"/>
80        </target>
81        <WPARAM>
82          <paramValue obj="up" param="scrollPos"/>
83        </WPARAM>
84      </genMsg>
85      <!-- tab change message for Property Pages-->
86      <genMsg delay="500">
87        <type>
88          <constValue value="1125"/>
89        </type>
90        <target>
91          <winInfoValue obj="up" winParam="parentTarget"/>
92        </target>
93        <WPARAM>
94          <paramValue obj="up" param="scrollPos"/>
95        </WPARAM>
96      </genMsg>
97    </rule>
98
99  <!--
100   <rule name="ComboBox">
```

```
<msg type="&WM_LBUTTONDOWN;">
  <equals>
    <winInfoValue obj="this" winParam="class"/>
    <constValue value="ComboBox"/>
  </equals>
  <store var="clicked"/>
</msg>
<msg type="&WM_COMMAND;">
  <equals>
    <paramValue obj="this" param="window.hwnd"/>
    <paramValue obj="clicked" param="window.hwnd"/>
  </equals>
  <store var="cmd1">
    <resolveHwnd param="source" storeParam="sourceDesc"/>
  </store>
</msg>
<msg type="&WM_COMMAND;" multiple="true">
  <equals>
    <paramValue obj="this" param="source"/>
    <paramValue obj="clicked" param="window.hwnd"/>
  </equals>
  <storeSeq varSeq="cmds">
    <resolveHwnd param="window.hwnd" storeParam="msgTarget"/>
    <resolveHwnd param="source" storeParam="sourceDesc"/>
  </storeSeq>
</msg>
<msg type="&WM_LBUTTONUP;">
  <equals>
    <paramValue obj="this" param="window.hwnd"/>
    <paramValue obj="clicked" param="window.hwnd"/>
  </equals>
</msg>
<genMsg delay="100">
  <type>
    <constValue value="&WM_SETFOCUS;"/>
  </type>
  <target>
    <msgInfoValue obj="clicked" msgParam="target"/>
  </target>
</genMsg>
<genMsg delay="500">
  <type>
    <constValue value="&CB_SHOWDROPDOWN;"/>
  </type>
  <target>
    <msgInfoValue obj="clicked" msgParam="target"/>
  </target>
  <WPARAM>
    <constValue value="1"/>
  </WPARAM>
</genMsg>
<!-
<genMsg delay="100">
  <type>
    <constValue value="273"/>
  </type>
  <target>
    <msgInfoValue obj="cmd1" msgParam="target"/>
```

```
159       </target>
160       <LPARAM>
161         <paramValue obj="cmd1" param="sourceDesc"/>
162       </LPARAM>
163       <WPARAM>
164         <paramValue obj="cmd1" param="WPARAM"/>
165       </WPARAM>
166     </genMsg>
167     <genMsgSeq delay="50">
168       <type>
169         <constValue value="273"/>
170       </type>
171       <target>
172         <seqValue seqObj="cmds" param="msgTarget"/>
173       </target>
174       <LPARAM>
175         <seqValue seqObj="cmds" param ="sourceDesc"/>
176       </LPARAM>
177       <WPARAM>
178         <seqValue seqObj="cmds" param="WPARAM"/>
179       </WPARAM>
180     </genMsgSeq>-->
181   </rule>
182   <rule name="ComboLBox">
183     <msg type="&WM_LBUTTONDOWN;">
184       <equals>
185         <winInfoValue obj="this" winParam="class"/>
186         <constValue value="ComboLBox"/>
187       </equals>
188       <store var="clicked"/>
189     </msg>
190     <msg type="&WM_COMMAND;">
191       <equals>
192         <paramValue obj="this" param="window.hwnd"/>
193         <paramValue obj="clicked" param="window.hwnd"/>
194       </equals>
195       <store var="cmd1">
196         <resolveHwnd param="source" storeParam="sourceDesc"/>
197       </store>
198     </msg>
199     <msg type="&WM_COMMAND;" multiple="true">
200       <equals>
201         <paramValue obj="this" param="source"/>
202         <paramValue obj="clicked" param="window.hwnd"/>
203       </equals>
204       <storeSeq varSeq="cmds">
205         <resolveHwnd param="window.hwnd" storeParam="msgTarget"/>
206         <resolveHwnd param="source" storeParam="sourceDesc"/>
207       </storeSeq>
208     </msg>
209     <msg type="&WM_LBUTTONUP;">
210       <equals>
211         <paramValue obj="this" param="window.hwnd"/>
212         <paramValue obj="clicked" param="window.hwnd"/>
213       </equals>
214     </msg>
215     <genMsg delay="100">
216       <type>
```

```
217         <constValue value="&WM_SETFOCUS;"/>
218       </type>
219       <target>
220         <msgInfoValue obj="clicked" msgParam="target"/>
221       </target>
222     </genMsg>
223     <genMsg delay="500">
224       <type>
225         <constValue value="&CB_SHOWDROPDOWN;"/>
226       </type>
227       <target>
228         <msgInfoValue obj="clicked" msgParam="target"/>
229       </target>
230       <WPARAM>
231         <constValue value="1"/>
232       </WPARAM>
233     </genMsg>
234   </rule>
235
236   <rule name="LeftClickCommandComboLBox">
237     <msg type="&WM_LBUTTONDOWN;">
238       <equals>
239         <constValue value="ComboLBox"/>
240         <winInfoValue obj="this" winParam="class"/>
241       </equals>
242       <store var="clicked"/>
243     </msg>
244     <msg type="&WM_LBUTTONUP;">
245       <equals>
246         <paramValue obj="clicked" param="window.hwnd"/>
247         <paramValue obj="this" param="window.hwnd"/>
248       </equals>
249       <store var="up"/>
250     </msg>
251     <msg type="&WM_COMMAND;">
252       <equals>
253         <paramValue obj="clicked" param="window.hwnd"/>
254         <paramValue obj="this" param="source"/>
255       </equals>
256       <store var="cmd1">
257         <resolveHwnd param="source" storeParam="sourceDesc"/>
258       </store>
259     </msg>
260     <msg type="&WM_COMMAND;" multiple="true">
261       <equals>
262         <paramValue obj="this" param="source"/>
263         <paramValue obj="cmd1" param="window.hwnd"/>
264       </equals>
265       <storeSeq varSeq="cmds">
266         <resolveHwnd param="window.hwnd" storeParam="target"/>
267         <resolveHwnd param="source" storeParam="sourceDesc"/>
268       </storeSeq>
269     </msg>
270     <genMsg delay="100">
271       <type>
272         <constValue value="&CB_SETCURSEL;"/>
273       </type>
274       <target>
```

```
275         <msgInfoValue obj="up" msgParam="target"/>
276       </target>
277       <WPARAM>
278         <constValue value="1"/>
279         <!--<paramValue obj="up" param="scrollPos"/>-->
280       </WPARAM>
281     </genMsg>
282     <genMsg delay="100">
283       <type>
284         <constValue value="&CB_SHOWDROPDOWN;"/>
285       </type>
286       <target>
287         <winInfoValue obj="clicked" winParam="parentTarget"/>
288       </target>
289       <WPARAM>
290         <constValue value="0"/>
291       </WPARAM>
292     </genMsg>
293     <genMsg delay="100">
294       <type>
295         <constValue value="&WM_COMMAND;"/>
296       </type>
297       <target>
298         <winInfoValue obj="cmd1" winParam="parentTarget"/>
299       </target>
300       <LPARAM>
301         <paramValue obj="cmd1" param="sourceDesc"/>
302       </LPARAM>
303       <WPARAM>
304         <paramValue obj="cmd1" param="WPARAM"/>
305       </WPARAM>
306     </genMsg>
307     <genMsgSeq delay="100">
308       <type>
309         <constValue value="&WM_COMMAND;"/>
310       </type>
311       <target>
312         <seqValue seqObj="cmds" param="target"/>
313       </target>
314       <LPARAM>
315         <seqValue seqObj="cmds" param="sourceDesc"/>
316       </LPARAM>
317       <WPARAM>
318         <seqValue seqObj="cmds" param="WPARAM"/>
319       </WPARAM>
320     </genMsgSeq>
321   </rule>-->
322
323   <rule name="LeftClickCommand">
324     <msg type="&WM_LBUTTONDOWN;">
325       <store var="clicked"/>
326     </msg>
327     <msg type="&WM_LBUTTONUP;">
328       <equals>
329         <paramValue obj="clicked" param="window.hwnd"/>
330         <paramValue obj="this" param="window.hwnd"/>
331       </equals>
332     </msg>
```

```
333     <msg type="&WM_COMMAND;">
334       <equals>
335         <paramValue obj="clicked" param="window.hwnd"/>
336         <paramValue obj="this" param="source"/>
337       </equals>
338       <store var="cmd">
339         <resolveHwnd param="source" storeParam="sourceDesc"/>
340       </store>
341     </msg>
342     <genMsg delay="500">
343       <type>
344         <msgInfoValue obj="cmd" msgParam="type"/>
345       </type>
346       <target>
347         <msgInfoValue obj="cmd" msgParam="target"/>
348       </target>
349       <LPARAM>
350         <paramValue obj="cmd" param="sourceDesc"/>
351       </LPARAM>
352       <WPARAM>
353         <paramValue obj="cmd" param="WPARAM"/>
354       </WPARAM>
355     </genMsg>
356   </rule>
357
358   <rule name="LeftClickSysCommand">
359     <msg type="&WM_LBUTTONDOWN;">
360       <store var="clicked"/>
361     </msg>
362     <msg type="&WM_LBUTTONUP;">
363       <equals>
364         <paramValue obj="clicked" param="window.hwnd"/>
365         <paramValue obj="this" param="window.hwnd"/>
366       </equals>
367     </msg>
368     <msg type="&WM_SYSCOMMAND;">
369       <store var="cmd"/>
370     </msg>
371     <genMsg delay="500">
372       <storedVar obj="cmd"/>
373     </genMsg>
374   </rule>
375
376   <rule name="NCLeftClickSysCommand">
377     <msg type="&WM_NCLBUTTONDOWN;">
378       <store var="clicked"/>
379     </msg>
380     <msg type="&WM_LBUTTONUP;">
381       <equals>
382         <paramValue obj="clicked" param="window.hwnd"/>
383         <paramValue obj="this" param="window.hwnd"/>
384       </equals>
385     </msg>
386     <msg type="&WM_SYSCOMMAND;">
387       <equals>
388         <paramValue obj="clicked" param="window.hwnd"/>
389         <paramValue obj="this" param="window.hwnd"/>
390       </equals>
```

```
391       <store var="cmd"/>
392     </msg>
393     <genMsg delay="500">
394       <storedVar obj="cmd"/>
395     </genMsg>
396   </rule>
397
398   <rule name="LeftClickMenuItemCmd">
399     <msg type="&WM_LBUTTONDOWN;">
400       <store var="clicked"/>
401     </msg>
402     <msg type="&WM_LBUTTONUP;">
403       <equals>
404         <paramValue obj="clicked" param="window.hwnd"/>
405         <paramValue obj="this" param="window.hwnd"/>
406       </equals>
407     </msg>
408     <msg type="&WM_MENUSELECT;">
409       <equals>
410         <paramValue obj="clicked" param="window.hwnd"/>
411         <paramValue obj="this" param="window.hwnd"/>
412       </equals>
413     </msg>
414     <msg type="&WM_COMMAND;">
415       <equals>
416         <paramValue obj="this" param="sourceType"/>
417         <constValue value="0"/>
418       </equals>
419       <store var="cmd"/>
420     </msg>
421     <genMsg delay="500">
422       <storedVar obj="cmd"/>
423     </genMsg>
424   </rule>
425
426
427   <!-- rules involving mouse movement -->
428   <rule name="HScroll_TrackBar">
429     <msg type="&WM_LBUTTONDOWN;">
430       <equals>
431         <winInfoValue obj="this" winParam="class"/>
432         <constValue value="msctls_trackbar32"/>
433       </equals>
434       <store var="clicked"/>
435     </msg>
436     <msg type="&WM_HSCROLL;" multiple="true">
437       <equals>
438         <paramValue obj="this" param="scrollBarHandle"/>
439         <paramValue obj="clicked" param="window.hwnd"/>
440       </equals>
441       <storeSeq varSeq="scrolls">
442         <resolveHwnd param="scrollBarHandle" storeParam="scrollBarTarget"/>
443       </storeSeq>
444     </msg>
445     <msg type="&WM_LBUTTONUP;">
446       <equals>
447         <paramValue obj="this" param="window.hwnd"/>
448         <paramValue obj="clicked" param="window.hwnd"/>
```

```
449       </equals>
450     </msg>
451     <genMsgSeq delay="50">
452       <type>
453         <constValue value="&TBM_SETPOS;"/>
454       </type>
455       <target>
456         <seqValue seqObj="scrolls" param="scrollBarTarget"/>
457       </target>
458       <LPARAM>
459         <seqValue seqObj="scrolls" param="scrollPos"/>
460       </LPARAM>
461       <WPARAM>
462         <constValue value="1"/>
463       </WPARAM>
464     </genMsgSeq>
465   </rule>
466
467
468   <rule name="VScroll_TrackBar">
469     <msg type="&WM_LBUTTONDOWN;">
470       <equals>
471         <winInfoValue obj="this" winParam="class"/>
472         <constValue value="msctls_trackbar32"/>
473       </equals>
474       <store var="clicked"/>
475     </msg>
476     <msg type="&WM_VSCROLL;" multiple="true">
477       <equals>
478         <paramValue obj="this" param="scrollBarHandle"/>
479         <paramValue obj="clicked" param="window.hwnd"/>
480       </equals>
481       <storeSeq varSeq="scrolls">
482         <resolveHwnd param="scrollBarHandle" storeParam="scrollBarTarget"/>
483       </storeSeq>
484     </msg>
485     <msg type="&WM_LBUTTONUP;">
486       <equals>
487         <paramValue obj="this" param="window.hwnd"/>
488         <paramValue obj="clicked" param="window.hwnd"/>
489       </equals>
490     </msg>
491     <genMsgSeq delay="50">
492       <type>
493         <constValue value="&TBM_SETPOS;"/>
494       </type>
495       <target>
496         <seqValue seqObj="scrolls" param="scrollBarTarget"/>
497       </target>
498       <LPARAM>
499         <seqValue seqObj="scrolls" param="scrollPos"/>
500       </LPARAM>
501       <WPARAM>
502         <constValue value="1"/>
503       </WPARAM>
504     </genMsgSeq>
505   </rule>
506
```

```
507
508   <rule name="HScroll_ScrollBar">
509     <msg type="&WM_LBUTTONDOWN;">
510       <equals>
511         <winInfoValue obj="this" winParam="class"/>
512         <constValue value="ScrollBar"/>
513       </equals>
514       <store var="clicked"/>
515     </msg>
516     <msg type="&WM_HSCROLL;" multiple="true">
517       <equals>
518         <paramValue obj="this" param="scrollBarHandle"/>
519         <paramValue obj="clicked" param="window.hwnd"/>
520       </equals>
521       <storeSeq varSeq="scrolls">
522         <resolveHwnd param="scrollBarHandle" storeParam="scrollBarTarget"/>
523       </storeSeq>
524     </msg>
525     <msg type="&WM_LBUTTONUP;">
526       <equals>
527         <paramValue obj="this" param="window.hwnd"/>
528         <paramValue obj="clicked" param="window.hwnd"/>
529       </equals>
530     </msg>
531     <genMsgSeq delay="50">
532       <type>
533         <constValue value="&SBM_SETPOS;"/>
534       </type>
535       <target>
536         <seqValue seqObj="scrolls" param="scrollBarTarget"/>
537       </target>
538       <LPARAM>
539         <constValue value="1"/>
540       </LPARAM>
541       <WPARAM>
542         <seqValue seqObj="scrolls" param="scrollPos"/>
543       </WPARAM>
544     </genMsgSeq>
545   </rule>
546
547
548   <rule name="VScroll_ScrollBar">
549     <msg type="&WM_LBUTTONDOWN;">
550       <equals>
551         <winInfoValue obj="this" winParam="class"/>
552         <constValue value="ScrollBar"/>
553       </equals>
554       <store var="clicked"/>
555     </msg>
556     <msg type="&WM_VSCROLL;" multiple="true">
557       <equals>
558         <paramValue obj="this" param="scrollBarHandle"/>
559         <paramValue obj="clicked" param="window.hwnd"/>
560       </equals>
561       <storeSeq varSeq="scrolls">
562         <resolveHwnd param="scrollBarHandle" storeParam="scrollBarTarget"/>
563       </storeSeq>
564     </msg>
```

```
565     <msg type="&WM_LBUTTONUP;">
566       <equals>
567         <paramValue obj="this" param="window.hwnd"/>
568         <paramValue obj="clicked" param="window.hwnd"/>
569       </equals>
570     </msg>
571     <genMsgSeq delay="50">
572       <type>
573         <constValue value="&SBM_SETPOS;"/>
574       </type>
575       <target>
576         <seqValue seqObj="scrolls" param="scrollBarTarget"/>
577       </target>
578       <LPARAM>
579         <constValue value="1"/>
580       </LPARAM>
581       <WPARAM>
582         <seqValue seqObj="scrolls" param="scrollPos"/>
583       </WPARAM>
584     </genMsgSeq>
585   </rule>
586
587   <!-- Does not work correctly, if a scrollbar has no handle of its own, e.g., a standard scrollbar of a listbox -->
588   <rule name="VScrollNC">
589     <msg type="&WM_NCLBUTTONDOWN;">
590       <store var="clicked"/>
591     </msg>
592     <msg type="&WM_VSCROLL;">
593       <equals>
594         <paramValue obj="this" param="window.hwnd"/>
595         <paramValue obj="clicked" param="window.hwnd"/>
596       </equals>
597       <store var="scrolls"/>
598     </msg>
599     <genMsg delay="50">
600       <type>
601         <constValue value="&WM_VSCROLL;"/>
602       </type>
603       <target>
604         <msgInfoValue obj="clicked" msgParam="target"/>
605       </target>
606       <WPARAM>
607         <LOWORD>
608           <constValue value="4"/>
609         </LOWORD>
610         <HIWORD>
611           <paramValue obj="scrolls" param="scrollPos"/>
612         </HIWORD>
613       </WPARAM>
614     </genMsg>
615   </rule>
616
617
618   <!--<rule name="VScrollNC">
619     <msg type="&WM_NCLBUTTONDOWN;">
620       <store var="clicked"/>
621     </msg>
622     <msg type="&WM_VSCROLL;" multiple="true">
```

```
623       <equals>
624         <paramValue obj="this" param="window.hwnd"/>
625         <paramValue obj="clicked" param="window.hwnd"/>
626       </equals>
627       <storeSeq varSeq="scrolls">
628         <resolveHwnd param="window.hwnd" storeParam="scrollBarTarget"/>
629       </storeSeq>
630     </msg>
631     <genMsgSeq delay="20">
632       <type>
633         <constValue value="&WM_VSCROLL;"/>
634       </type>
635       <target>
636         <seqValue seqObj="scrolls" param="scrollBarTarget"/>
637       </target>
638       <WPARAM>
639         <LOWORD>
640           <constValue value="4"/>
641         </LOWORD>
642         <HIWORD>
643           <paramValue obj="scrolls" param="scrollPos"/>
644         </HIWORD>
645       </WPARAM>
646     </genMsgSeq>
647   </rule>−−>
648
649   <!−− Copy of set focus without kill−focus −−>
650   <rule name="LeftClickSetFocus">
651     <msg type="&WM_LBUTTONDOWN;">
652       <equals>
653         <winInfoValue obj="this" winParam="class"/>
654         <constValue value="Edit"/>
655       </equals>
656       <store var="clicked"/>
657     </msg>
658     <msg type="&WM_SETFOCUS;">
659       <store var="setfocus"/>
660     </msg>
661     <msg type="&WM_COMMAND;">
662       <equals>
663         <paramValue obj="this" param="source"/>
664         <paramValue obj="clicked" param="window.hwnd"/>
665       </equals>
666       <equals>
667         <paramValue obj="this" param="sourceType"/>
668         <constValue value="256"/>
669       </equals>
670       <store var="cmd"/>
671     </msg>
672     <msg type="&WM_LBUTTONUP;">
673       <equals>
674         <paramValue obj="this" param="window.hwnd"/>
675         <paramValue obj="clicked" param="window.hwnd"/>
676       </equals>
677     </msg>
678     <genMsg delay="20">
679       <type>
680         <constValue value="&WM_SETFOCUS;"/>
```

```
681       </type>
682       <target>
683         <msgInfoValue obj="setfocus" msgParam="target"/>
684       </target>
685       <WPARAM>
686         <constValue value="0"/>
687       </WPARAM>
688     </genMsg>
689     <genMsg delay="500">
690       <type>
691         <constValue value="&WM_COMMAND;"/>
692       </type>
693       <target>
694         <msgInfoValue obj="cmd" msgParam="target"/>
695       </target>
696       <LPARAM>
697         <msgInfoValue obj="setfocus" msgParam="target"/>
698       </LPARAM>
699       <WPARAM>
700         <paramValue obj="cmd" param="WPARAM"/>
701       </WPARAM>
702     </genMsg>
703   </rule>
704
705   <!-- Works only partially! -->
706   <rule name="LeftClickChangeFocus">
707     <msg type="&WM_LBUTTONDOWN;">
708       <equals>
709         <winInfoValue obj="this" winParam="class"/>
710         <constValue value="Edit"/>
711       </equals>
712       <store var="clicked"/>
713     </msg>
714     <msg type="&WM_KILLFOCUS;">
715       <store var="killfocus"/>
716     </msg>
717     <msg type="&WM_SETFOCUS;">
718       <equals>
719         <paramValue obj="this" param="WPARAM"/>
720         <paramValue obj="killfocus" param="window.hwnd"/>
721       </equals>
722       <store var="setfocus"/>
723     </msg>
724     <msg type="&WM_COMMAND;">
725       <equals>
726         <paramValue obj="this" param="source"/>
727         <paramValue obj="clicked" param="window.hwnd"/>
728       </equals>
729       <equals>
730         <paramValue obj="this" param="sourceType"/>
731         <constValue value="256"/>
732       </equals>
733       <store var="cmd"/>
734     </msg>
735     <msg type="&WM_LBUTTONUP;">
736       <equals>
737         <paramValue obj="this" param="window.hwnd"/>
738         <paramValue obj="clicked" param="window.hwnd"/>
```

```
739       </equals>
740     </msg>
741     <genMsg delay="100">
742       <type>
743         <constValue value="&WM_KILLFOCUS;"/>
744       </type>
745       <target>
746         <msgInfoValue obj="killfocus" msgParam="target"/>
747       </target>
748       <WPARAM>
749         <msgInfoValue obj="setfocus" msgParam="target"/>
750       </WPARAM>
751     </genMsg>
752     <genMsg delay="100">
753       <type>
754         <constValue value="&WM_SETFOCUS;"/>
755       </type>
756       <target>
757         <msgInfoValue obj="setfocus" msgParam="target"/>
758       </target>
759       <WPARAM>
760         <msgInfoValue obj="killfocus" msgParam="target"/>
761       </WPARAM>
762     </genMsg>
763     <genMsg delay="500">
764       <type>
765         <constValue value="&WM_COMMAND;"/>
766       </type>
767       <target>
768         <msgInfoValue obj="cmd" msgParam="target"/>
769       </target>
770       <LPARAM>
771         <msgInfoValue obj="setfocus" msgParam="target"/>
772       </LPARAM>
773       <WPARAM>
774         <paramValue obj="cmd" param="WPARAM"/>
775       </WPARAM>
776     </genMsg>
777   </rule>
778
779
780   <!-- Rules for keyboard input -->
781   <rule name="KeyDown">
782     <msg type="&WM_KEYDOWN;">
783       <store var="keydown"/>
784     </msg>
785     <genMsg delay="100">
786       <storedVar obj="keydown"/>
787     </genMsg>
788     <!--<idinfo>
789       <paramValue obj="keydown" param="key"/>
790     </idinfo>-->
791   </rule>
792
793   <rule name="KeyUp">
794     <msg type="&WM_KEYUP;">
795       <store var="keyup"/>
796     </msg>
```

```
797     <genMsg delay="500">
798       <storedVar obj="keyup"/>
799     </genMsg>
800     <!--<idinfo>
801       <paramValue obj="keyup" param="key"/>
802     </idinfo>-->
803   </rule>
804
805   <rule name="SysKeyDown">
806     <msg type="&WM_SYSKEYDOWN;">
807       <store var="keydown"/>
808     </msg>
809     <genMsg delay="100">
810       <storedVar obj="keydown"/>
811     </genMsg>
812     <idinfo>
813       <paramValue obj="keydown" param="key"/>
814     </idinfo>
815   </rule>
816
817   <rule name="SysKeyUp">
818     <msg type="&WM_SYSKEYUP;">
819       <store var="keyup"/>
820     </msg>
821     <genMsg delay="500">
822       <storedVar obj="keyup"/>
823     </genMsg>
824     <idinfo>
825       <paramValue obj="keyup" param="key"/>
826     </idinfo>
827   </rule>
828
829
830   <!-- What follows are coordinate-based rules. They are "hail mary" rules that try to salvage events that cannot
          be matched or replayed in a coordinate independent way (yet). -->
831   <rule name ="LeftClickCoordinates">
832     <msg type="&WM_LBUTTONDOWN;">
833       <store var="clicked"/>
834     </msg>
835     <msg type="&WM_LBUTTONUP;">
836       <equals>
837         <paramValue obj="clicked" param="window.hwnd"/>
838         <paramValue obj="this" param="window.hwnd"/>
839       </equals>
840       <store var="up"/>
841     </msg>
842     <genMsg delay="100">
843       <type>
844         <constValue value="&WM_LBUTTONDOWN;"/>
845       </type>
846       <target>
847         <msgInfoValue obj="clicked" msgParam="target"/>
848       </target>
849       <LPARAM>
850         <LOWORD>
851           <paramValue obj="clicked" param="point.x"/>
852         </LOWORD>
853         <HIWORD>
```

```
854             <paramValue obj="clicked" param="point.y"/>
855           </HIWORD>
856         </LPARAM>
857         <WPARAM>
858           <paramValue obj="clicked" param="WPARAM"/>
859         </WPARAM>
860       </genMsg>
861       <genMsg delay="500">
862         <type>
863           <constValue value="&WM_LBUTTONUP;"/>
864         </type>
865         <target>
866           <msgInfoValue obj="up" msgParam="target"/>
867         </target>
868         <LPARAM>
869           <LOWORD>
870             <paramValue obj="up" param="point.x"/>
871           </LOWORD>
872           <HIWORD>
873             <paramValue obj="up" param="point.y"/>
874           </HIWORD>
875         </LPARAM>
876         <WPARAM>
877           <paramValue obj="up" param="WPARAM"/>
878         </WPARAM>
879       </genMsg>
880     </rule>
881
882     <rule name ="NCLeftClickCoordinates">
883       <msg type="&WM_NCLBUTTONDOWN;">
884         <store var="clicked"/>
885       </msg>
886       <msg type="&WM_LBUTTONUP;">
887         <equals>
888           <paramValue obj="clicked" param="window.hwnd"/>
889           <paramValue obj="this" param="window.hwnd"/>
890         </equals>
891         <store var="up"/>
892       </msg>
893       <genMsg delay="100">
894         <type>
895           <constValue value="&WM_NCLBUTTONDOWN;"/>
896         </type>
897         <target>
898           <msgInfoValue obj="clicked" msgParam="target"/>
899         </target>
900         <LPARAM>
901           <LOWORD>
902             <paramValue obj="clicked" param="point.x"/>
903           </LOWORD>
904           <HIWORD>
905             <paramValue obj="clicked" param="point.y"/>
906           </HIWORD>
907         </LPARAM>
908         <WPARAM>
909           <paramValue obj="clicked" param="WPARAM"/>
910         </WPARAM>
911       </genMsg>
```

```
912     <genMsg delay="500">
913       <type>
914         <constValue value="&WM_LBUTTONUP;"/>
915       </type>
916       <target>
917         <msgInfoValue obj="up" msgParam="target"/>
918       </target>
919       <LPARAM>
920         <LOWORD>
921           <paramValue obj="up" param="point.x"/>
922         </LOWORD>
923         <HIWORD>
924           <paramValue obj="up" param="point.y"/>
925         </HIWORD>
926       </LPARAM>
927       <WPARAM>
928         <paramValue obj="up" param="WPARAM"/>
929       </WPARAM>
930     </genMsg>
931   </rule>
932
933   <rule name ="NCLeftClickCoordinates2">
934     <msg type="&WM_NCLBUTTONDOWN;">
935       <store var="clicked"/>
936     </msg>
937     <msg type="&WM_NCLBUTTONUP;">
938       <equals>
939         <paramValue obj="clicked" param="window.hwnd"/>
940         <paramValue obj="this" param="window.hwnd"/>
941       </equals>
942       <store var="up"/>
943     </msg>
944     <genMsg delay="100">
945       <type>
946         <constValue value="&WM_NCLBUTTONDOWN;"/>
947       </type>
948       <target>
949         <msgInfoValue obj="clicked" msgParam="target"/>
950       </target>
951       <LPARAM>
952         <LOWORD>
953           <paramValue obj="clicked" param="point.x"/>
954         </LOWORD>
955         <HIWORD>
956           <paramValue obj="clicked" param="point.y"/>
957         </HIWORD>
958       </LPARAM>
959       <WPARAM>
960         <paramValue obj="clicked" param="WPARAM"/>
961       </WPARAM>
962     </genMsg>
963     <genMsg delay="500">
964       <type>
965         <constValue value="&WM_NCLBUTTONUP;"/>
966       </type>
967       <target>
968         <msgInfoValue obj="up" msgParam="target"/>
969       </target>
```

```
970       <LPARAM>
971         <LOWORD>
972           <paramValue obj="up" param="point.x"/>
973         </LOWORD>
974         <HIWORD>
975           <paramValue obj="up" param="point.y"/>
976         </HIWORD>
977       </LPARAM>
978       <WPARAM>
979         <paramValue obj="up" param="WPARAM"/>
980       </WPARAM>
981     </genMsg>
982   </rule>
983
984   <rule name ="LeftClickCoordinatesTargetChanged">
985     <msg type="&WM_LBUTTONDOWN;">
986       <store var="clicked"/>
987     </msg>
988     <msg type="&WM_LBUTTONUP;">
989       <store var="up"/>
990     </msg>
991     <genMsg delay="100">
992       <type>
993         <constValue value="&WM_LBUTTONDOWN;"/>
994       </type>
995       <target>
996         <msgInfoValue obj="clicked" msgParam="target"/>
997       </target>
998       <LPARAM>
999         <LOWORD>
1000          <paramValue obj="clicked" param="point.x"/>
1001        </LOWORD>
1002        <HIWORD>
1003          <paramValue obj="clicked" param="point.y"/>
1004        </HIWORD>
1005      </LPARAM>
1006      <WPARAM>
1007        <paramValue obj="clicked" param="WPARAM"/>
1008      </WPARAM>
1009    </genMsg>
1010    <genMsg delay="500">
1011      <type>
1012        <constValue value="&WM_LBUTTONUP;"/>
1013      </type>
1014      <target>
1015        <msgInfoValue obj="up" msgParam="target"/>
1016      </target>
1017      <LPARAM>
1018        <LOWORD>
1019          <paramValue obj="up" param="point.x"/>
1020        </LOWORD>
1021        <HIWORD>
1022          <paramValue obj="up" param="point.y"/>
1023        </HIWORD>
1024      </LPARAM>
1025      <WPARAM>
1026        <paramValue obj="up" param="WPARAM"/>
1027      </WPARAM>
```

```
1028      </genMsg>
1029    </rule>
1030
1031    <rule name ="LeftClickCoordinatesTargetChanged2">
1032      <msg type="&WM_LBUTTONDOWN;">
1033        <store var="clicked"/>
1034      </msg>
1035      <msg type="&WM_NCLBUTTONUP;">
1036        <store var="up"/>
1037      </msg>
1038      <genMsg delay="100">
1039        <type>
1040          <constValue value="&WM_LBUTTONDOWN;"/>
1041        </type>
1042        <target>
1043          <msgInfoValue obj="clicked" msgParam="target"/>
1044        </target>
1045        <LPARAM>
1046          <LOWORD>
1047            <paramValue obj="clicked" param="point.x"/>
1048          </LOWORD>
1049          <HIWORD>
1050            <paramValue obj="clicked" param="point.y"/>
1051          </HIWORD>
1052        </LPARAM>
1053        <WPARAM>
1054          <paramValue obj="clicked" param="WPARAM"/>
1055        </WPARAM>
1056      </genMsg>
1057      <genMsg delay="500">
1058        <type>
1059          <constValue value="&WM_NCLBUTTONUP;"/>
1060        </type>
1061        <target>
1062          <msgInfoValue obj="up" msgParam="target"/>
1063        </target>
1064        <LPARAM>
1065          <LOWORD>
1066            <paramValue obj="up" param="point.x"/>
1067          </LOWORD>
1068          <HIWORD>
1069            <paramValue obj="up" param="point.y"/>
1070          </HIWORD>
1071        </LPARAM>
1072        <WPARAM>
1073          <paramValue obj="up" param="WPARAM"/>
1074        </WPARAM>
1075      </genMsg>
1076    </rule>
1077
1078
1079  </rules>
```

Listing A.1: Complete rule set for the MFC event type identification and replay generation.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <xs:schema targetNamespace="ul:rules"
3      elementFormDefault="qualified"
```

```
4     xmlns="ul:rules"
5     xmlns:xs="http://www.w3.org/2001/XMLSchema"
6  >
7
8    <xs:element name="rules">
9      <xs:complexType>
10       <xs:sequence minOccurs="0" maxOccurs="unbounded">
11         <xs:element name="rule" type="ruleType"/>
12       </xs:sequence>
13     </xs:complexType>
14   </xs:element>
15
16   <!-- rule type definition -->
17   <xs:complexType name="ruleType">
18     <xs:sequence>
19       <xs:element name="msg" type="msgType" minOccurs="0" maxOccurs="unbounded"/>
20       <xs:choice minOccurs="0" maxOccurs="unbounded">
21         <xs:element name="genMsg" type="genMsgType"/>
22         <xs:element name="genMsgSeq" type="genMsgSeqType"/>
23       </xs:choice>
24       <xs:element name="idinfo" type="varType" minOccurs="0" maxOccurs="1"/>
25     </xs:sequence>
26     <xs:attribute name="name" type="xs:string" use="optional"/>
27   </xs:complexType>
28
29   <!-- message type definition -->
30   <xs:complexType name="msgType">
31     <xs:sequence minOccurs="0" maxOccurs="1">
32       <xs:choice minOccurs="0" maxOccurs="unbounded">
33         <xs:element name="equals" type="equalsType"/>
34         <xs:element name="equalsSeq" type="equalsSeqType"/>
35       </xs:choice>
36       <xs:choice minOccurs="0" maxOccurs="unbounded">
37         <xs:element name="store" type="storeType"/>
38         <xs:element name="storeSeq" type="storeSeqType"/>
39       </xs:choice>
40     </xs:sequence>
41     <xs:attribute name="type" type="xs:int" use="required"/>
42     <!-- problem: should be strings like WM_LBUTTONDOWN, but program needs ints -->
43     <!--<xs:attribute name="type" type="xs:string" use="required"/>-->
44     <xs:attribute name="multiple" type="xs:boolean" use="optional"/>
45   </xs:complexType>
46
47   <!-- storage types for message nodes -->
48   <!-- a restriction forbidding "this" would be nice -->
49   <xs:complexType name="storeType">
50     <xs:sequence>
51       <xs:element name="resolveHwnd" type="resolveTargetType" minOccurs="0" maxOccurs="
             unbounded"/>
52     </xs:sequence>
53     <xs:attribute name="var" use="required" type="storVarName"/>
54   </xs:complexType>
55   <xs:complexType name="storeSeqType">
56     <xs:sequence>
57       <xs:element name="resolveHwnd" type="resolveTargetType" minOccurs="0" maxOccurs="
             unbounded"/>
58     </xs:sequence>
59     <xs:attribute name="varSeq" use="required" type="storVarName"/>
```

```
60  </xs:complexType>
61  <xs:complexType name="resolveTargetType">
62    <xs:attribute name="param" type="xs:string"/>
63    <xs:attribute name="storeParam" type="xs:string"/>
64  </xs:complexType>
65
66  <!-- equals type for message nodes -->
67  <xs:complexType name="equalsType">
68    <xs:sequence minOccurs="2" maxOccurs="2">
69      <xs:group ref="valueGroup"/>
70    </xs:sequence>
71  </xs:complexType>
72  <xs:complexType name="equalsSeqType">
73    <xs:sequence>
74      <xs:element name="seqValue" type="seqValueType"/>
75      <xs:choice>
76        <xs:group ref="seqValueGroup"/>
77      </xs:choice>
78    </xs:sequence>
79  </xs:complexType>
80
81  <!-- types for generating messages -->
82  <xs:complexType name="genMsgType">
83    <xs:choice>
84      <xs:sequence>
85        <xs:element name="type" type="varType"/>
86        <xs:element name="target" type="varType"/>
87        <xs:element name="LPARAM" type="PARAMtype" minOccurs="0" maxOccurs="1"/>
88        <xs:element name="WPARAM" type="PARAMtype" minOccurs="0" maxOccurs="1"/>
89      </xs:sequence>
90      <xs:element name="storedVar">
91        <xs:complexType>
92          <xs:attribute name="obj" type="xs:string"/>
93        </xs:complexType>
94      </xs:element>
95    </xs:choice>
96    <xs:attribute name="delay" type="xs:int" use="required"/>
97  </xs:complexType>
98  <xs:complexType name="genMsgSeqType">
99    <xs:choice>
100     <xs:sequence>
101       <xs:element name="type" type="seqType"/>
102       <!-- target must be a variable! -->
103       <xs:element name="target">
104         <xs:complexType >
105           <xs:sequence>
106             <xs:element name="seqValue" type="seqValueType"/>
107           </xs:sequence>
108         </xs:complexType>
109       </xs:element>
110       <xs:element name="LPARAM" type="seqPARAMtype" minOccurs="0" maxOccurs="1"/>
111       <xs:element name="WPARAM" type="seqPARAMtype" minOccurs="0" maxOccurs="1"/>
112     </xs:sequence>
113     <xs:element name="storedSeqVar">
114       <xs:complexType>
115         <xs:attribute name="seqObj" type="xs:string"/>
116       </xs:complexType>
117     </xs:element>
```

```
118     </xs:choice>
119     <xs:attribute name="delay" type="xs:int" use="required"/>
120   </xs:complexType>
121
122   <xs:complexType name="PARAMtype">
123     <xs:choice>
124       <xs:group ref="valueGroup"/>
125       <xs:sequence>
126         <xs:element name="LOWORD" type="varType"/>
127         <xs:element name="HIWORD" type="varType"/>
128       </xs:sequence>
129     </xs:choice>
130   </xs:complexType>
131
132   <xs:complexType name="seqPARAMtype">
133     <xs:choice>
134       <xs:group ref="seqValueGroup"/>
135       <xs:sequence>
136         <xs:element name="LOWORD" type="seqType"/>
137         <xs:element name="HIWORD" type="seqType"/>
138       </xs:sequence>
139     </xs:choice>
140   </xs:complexType>
141
142
143   <xs:complexType name="seqValueType">
144     <xs:attribute name ="seqObj" type="xs:string" use="required"/>
145     <xs:attribute name ="param" type="xs:string" use="required"/>
146   </xs:complexType>
147
148   <!-- values that can be used by equals and genMsg nodes -->
149   <xs:group name="valueGroup">
150     <xs:choice>
151       <xs:element name="paramValue" type="paramValueType"/>
152       <xs:element name="constValue" type="constValueType"/>
153       <xs:element name="winInfoValue" type ="winInfoValueType"/>
154       <xs:element name="msgInfoValue" type="msgInfoType"/>
155     </xs:choice>
156   </xs:group>
157   <xs:complexType name="varType">
158     <xs:group ref="valueGroup"/>
159   </xs:complexType>
160
161   <xs:group name="seqValueGroup">
162     <xs:choice>
163       <xs:element name="constValue" type="constValueType"/>
164       <xs:element name="seqValue" type="seqValueType"/>
165     </xs:choice>
166   </xs:group>
167   <xs:complexType name="seqType">
168     <xs:group ref="seqValueGroup"/>
169   </xs:complexType>
170
171   <xs:complexType name="paramValueType">
172     <xs:attribute name="obj" type="xs:string" use="required"/>
173     <xs:attribute name="param" type="xs:string" use="required"/>
174   </xs:complexType>
175   <xs:complexType name="constValueType">
```

```
176     <xs:attribute name="value" use="required"/>
177   </xs:complexType>
178   <xs:complexType name="winInfoValueType">
179     <xs:attribute name="obj" type="xs:string" use="required"/>
180     <xs:attribute name="winParam" type="winParamType" use="required"/>
181   </xs:complexType>
182   <xs:complexType name="msgInfoType">
183     <xs:attribute name="obj" type="xs:string" use="required"/>
184     <xs:attribute name="msgParam" type="msgParamType" use="required"/>
185   </xs:complexType>
186   <xs:simpleType name="winParamType">
187     <xs:restriction base="xs:string">
188       <xs:enumeration value="class"/>
189       <xs:enumeration value="resourceId"/>
190       <xs:enumeration value="hwnd"/>
191       <xs:enumeration value="parentTarget"/>
192       <xs:enumeration value="parentClass"/>
193     </xs:restriction>
194   </xs:simpleType>
195   <xs:simpleType name="msgParamType">
196     <xs:restriction base="xs:string">
197       <xs:enumeration value="type"/>
198       <xs:enumeration value="target"/>
199     </xs:restriction>
200   </xs:simpleType>
201
202
203   <xs:simpleType name="storVarName">
204     <xs:restriction base="xs:string">
205       <!-- this regex is formally not supported by XMLSchema, als look-arounds and \b are not supported :/ -->
206       <xs:pattern value="(?:(?!(?>\bthis\b))\w)+"/>
207     </xs:restriction>
208   </xs:simpleType>
209 </xs:schema>
```

Listing A.2: XML Schema for the MFC event type identification and replay generation.

## A.2. Listings for the Web Application Translation Layer

```
1  findlinks
2  discobot
3  Googlebot
4  Slurp
5  YandexBot
6  Spider
7  ScholarUniverse
8  Baiduspider
9  Exabot
10 Robot
11 MetaGer−Bot
12 YandexImages
13 Gigabot
14 SiteBot
15 bingbot
16 Ezooms
17 Jeeves/Teoma
18 msnbot
19 DotBot
20 changedetection.com/bot.html
21 FAST Enterprise Crawler 6
22 psbot
23 http://ws.daum.net/aboutWebSearch.html
24 NerdByNature.Bot
25 Sogou web spider
26 ssearch_bot
27 Purebot
28 http://www.icjobs.de
29 scoutjet
30 Netcraft Web Server Survey
31 TurnitinBot
32 ia_archiver
33 MJ12bot
34 Domnutch−Bot
35 Eurobot
36 GarlikCrawler
37 CMS Crawler
38 MSIECrawler
39 NaverBot
40 80legs
41 AhrefsBot
42 SISTRIX Crawler
43 NetcraftSurveyAgent
44 Search17Bot
45 Semager
46 YandexFavicons
47 heritrix
48 suggybot
49 Netluchs
50 Ocelli
51 PHPCrawl
52 SolomonoBot
53 Sosospider
54 Xerka WebBot
55 YahooCacheSystem
```

```
56 Xenu Link Sleuth
57 cmsworldmap
58 suchen.de
59 amaredo.com/de/suche.html
60 ibot
61 w3af.sourceforge.net
62 w3af.sf.net
63 yacybot
64 larbin2
65 t−h−u−n−d−e−r−s−t−o−n−e
66 sqlmap
```

Listing A.3: List of keywords used to filter Web crawlers based on their user agents.

## A.3. ArgoUML Data Gathering Description

The following three pages depict the description of how we collected usage date for ArgoUML. We gave the printed version of this task description to the eleven participants of our study.

GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN
INSTITUTE OF COMPUTER SCIENCE
Software Engineering for Distributed Systems
http://www.swe.informatik.uni-goettingen.de

**Case Study: UML with ArgoUML**


---

# Drawing Class Diagrams with ArgoUML

## Introduction

In one of our research projects, we investigate *usage-based* software testing. This means, that the testing of software products is based on how the software is actually used. In order to do this, we require *usage profiles* of the System Under Test (SUT), i.e., the software. In an ideal scenario, these profiles are based on monitoring user actions during the daily use of the SUT. Since this is a problem from a privacy point of view, we take a different approach to obtain usage profiles in this case study. We define a *use case* for the SUT, tell users to execute the use case with SUT and monitor this execution. This way, we gain insight of how users perform a specific use case with the SUT.

## Use Case: Drawing of Class Diagrams

The use case we defined is quite simple. Re-draw a UML class diagram with the tool ArgoUML (`http://argouml.tigris.org/`).

## What we monitor

We are interested in user actions, i.e., mouse clicks and keyboard inputs. In order to monitor these, we use an instrumented version of the SUT. In this case, this instrumentation is that ArgoUML is not started directly. Instead, the monitoring tool JFCMonitor (`http://eventbench.informatik.uni-goettingen.de/trac/wiki/Software/JFCMonitor`) executes ArgoUML indirectly. The tool produces a text file that contains the sequence of all user actions.

## What we do with the data

We train stochastic process, i.e., Markov chains. These chains have the user actions as states and the transitions between the states are *follows* relations between the user actions. If we have monitored a sufficient amount of users, our models will reliably predict the probability of the next user action.

## What don't do with the data

All collected data is handled anonymously. The files are only identified by the timestamps of when the logging started. We do not analyze single users.

## Instructions (CIP-Pool/AFS-Access)

1. Open the folder ~/SCRATCH/uml-case-study/argouml.
2. Start ArgoUML with the following command: `java -jar argouml.jar`
3. Make sure the language of ArgoUML is English. In case it is not, change the language settings to English (Edit → Settings → Appearance → Language)
4. Close ArgoUML.
5. Start ArgoUML with the following command: `java -jar jfcmonitor.jar argouml.jar`
6. Redraw the diagram shown in Figure 1 with ArgoUML. The design (e.g., location/size of the classes in the diagrams) does not have to be the same, only the content. Do not restart ArgoUML!
7. Save the result and close ArgoUML.

## Instructions (Local Copy)

REQUIREMENTS: Java 6; Java-binary folder must be in the path

1. Extract the archive argouml-jfcmonitor.zip and go the location where you extracted the data to.
2. Start ArgoUML with the following command: `java -jar argouml.jar`
3. Make sure the language of ArgoUML is English. In case it is not, change the language settings to English (Edit → Settings → Appearance → Language)
4. Close ArgoUML.
5. Start ArgoUML with the following command: `java -jar jfcmonitor.jar argouml.jar`.
6. Redraw the diagram shown in Figure 1 with ArgoUML. The design (e.g., location/size of the classes in the diagrams) does not have to be the same, only the content. Do not restart ArgoUML!
7. Save the result and close ArgoUML.
8. EMail the file jfcmonitor_%TIMESTAMP%.log to herbold@cs.uni-goettingen.de.

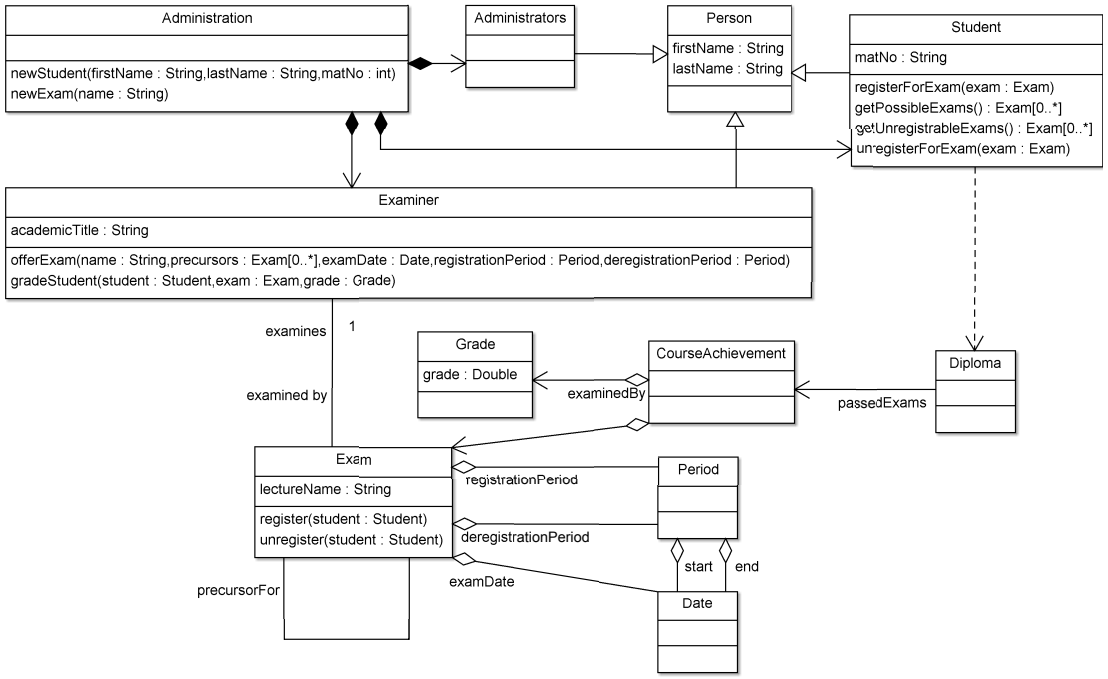
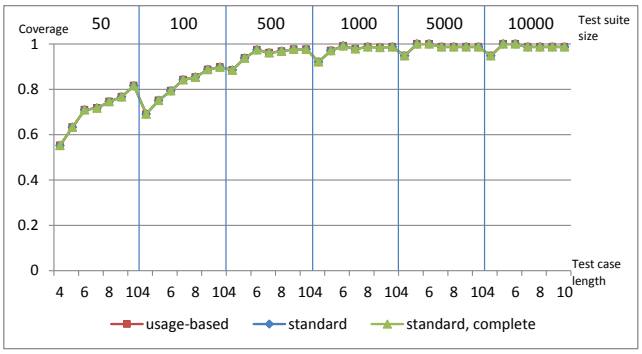


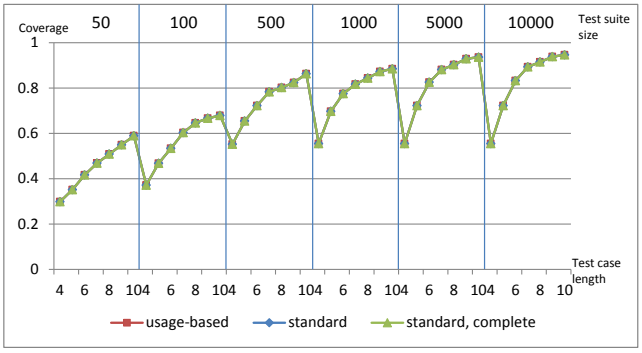

Figure 1: UML class diagram to re-draw.

## A.4. Coverage results for all experiments.

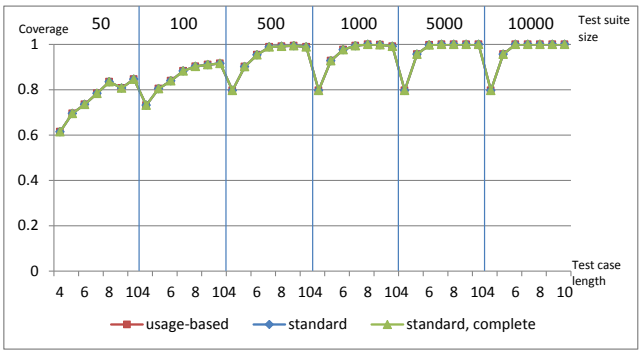
### A.4.1. Depth one for the hybrid.




(a) JFC data set.

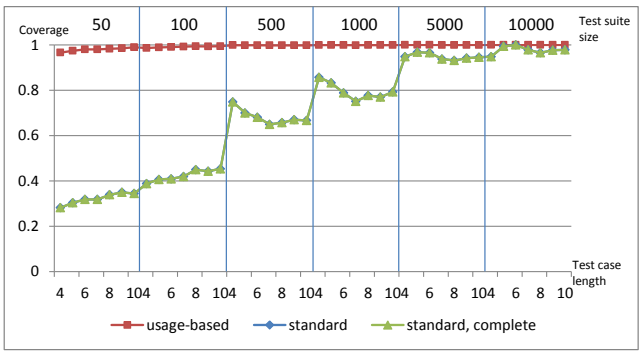



(b) MFC data set.

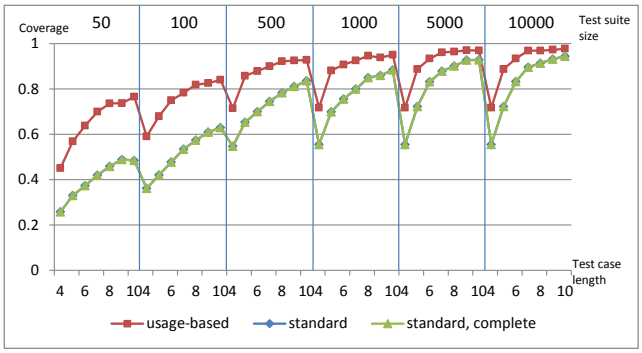



(c) Web application data set.

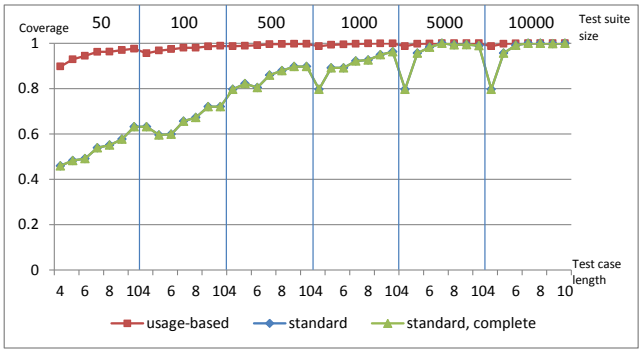
Figure A.1.: Results for the depth one for the hybrid test case generation with the random EFG.




(a) JFC data set.

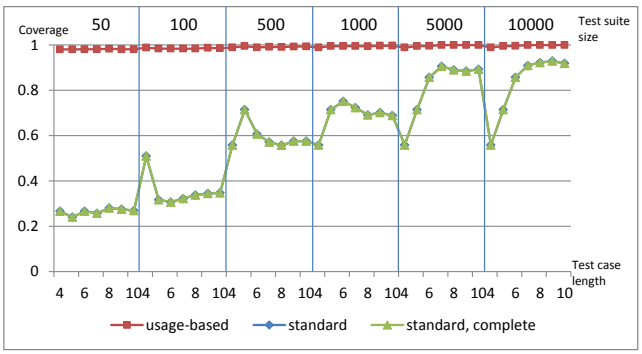



(b) MFC data set.

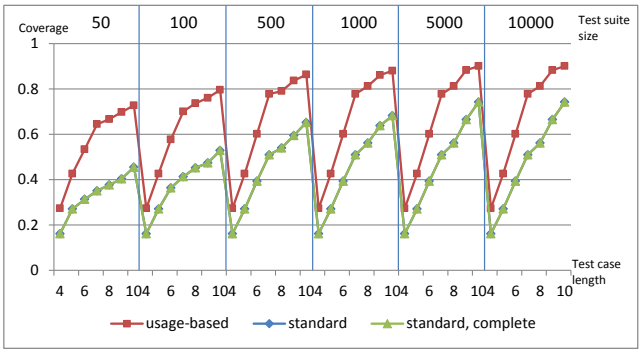



(c) Web application data set.

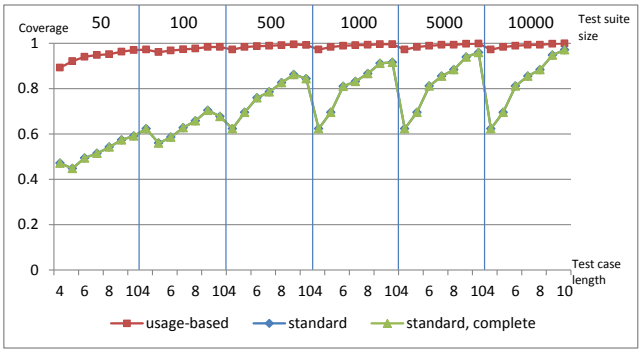
Figure A.2.: Results for the depth one for the hybrid test case generation with the first-order MM.




(a) JFC data set.

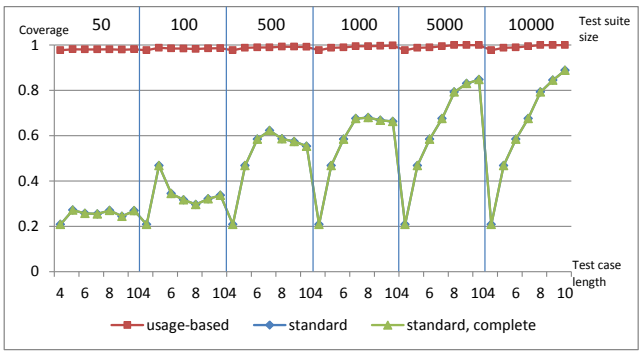



(b) MFC data set.

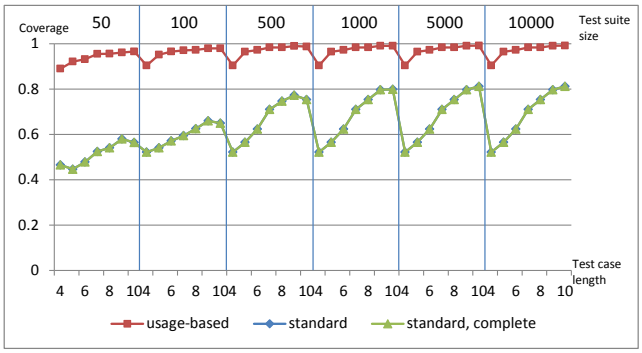



(c) Web application data set.

Figure A.3.: Results for the depth one for the hybrid test case generation with the 2nd-order MM.

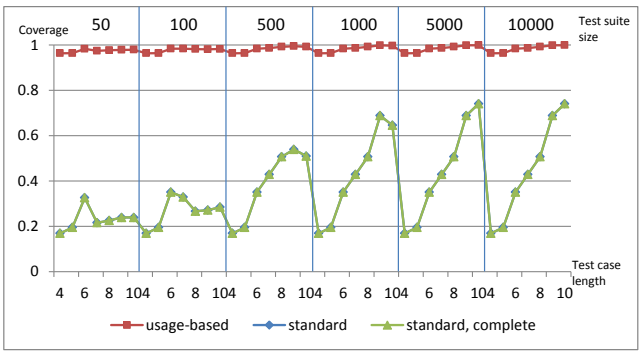



(a) JFC data set.

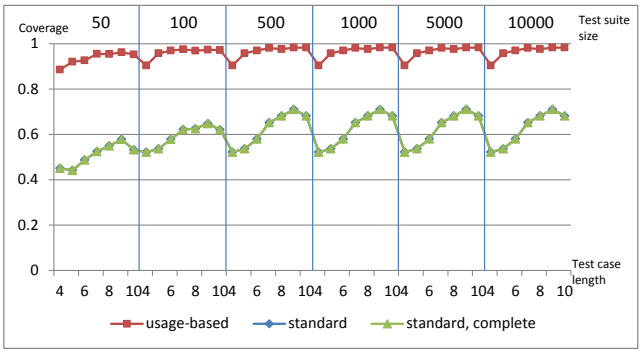



(b) Web application data set.

Figure A.4.: Results for the depth one for the hybrid test case generation with the 3rd-order MM.

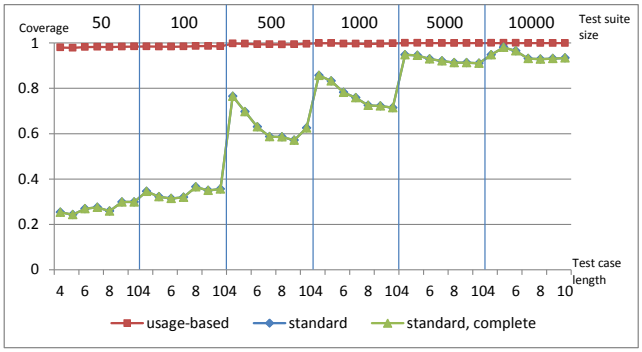



(a) JFC data set.

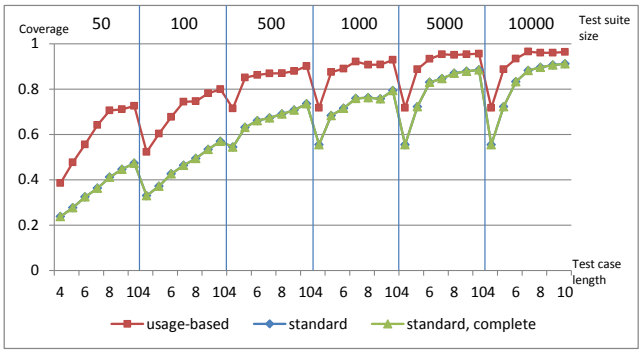



(b) Web application data set.

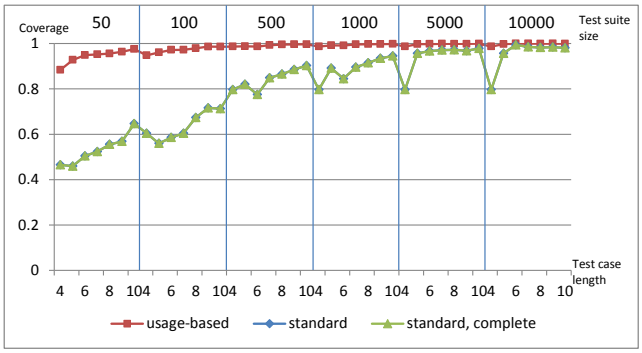
Figure A.5.: Results for the depth one for the hybrid test case generation with the 4th-order MM.




(a) JFC data set.

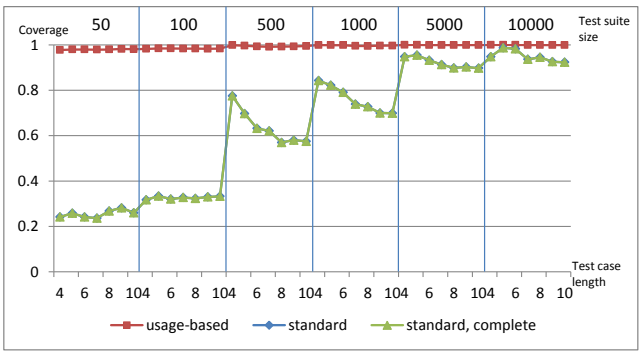



(b) MFC data set.

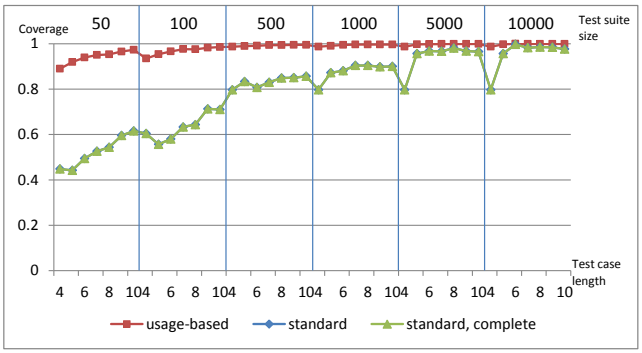



(c) Web application data set.

Figure A.6.: Results for the depth one for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 2$.

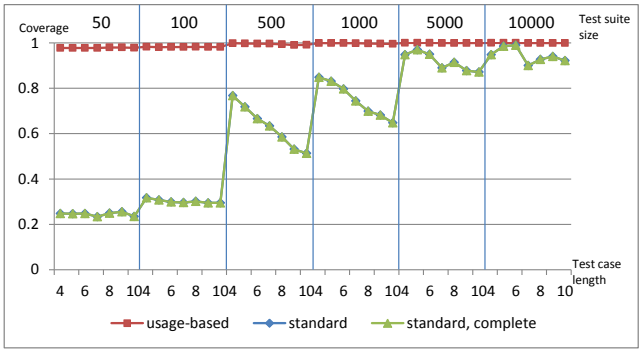



(a) JFC data set.

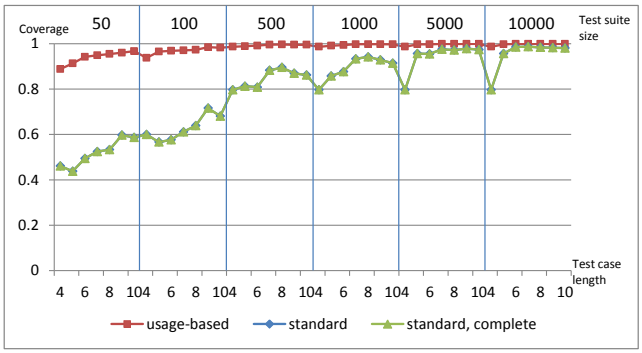



(b) Web application data set.

Figure A.7.: Results for the depth one for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 3$.

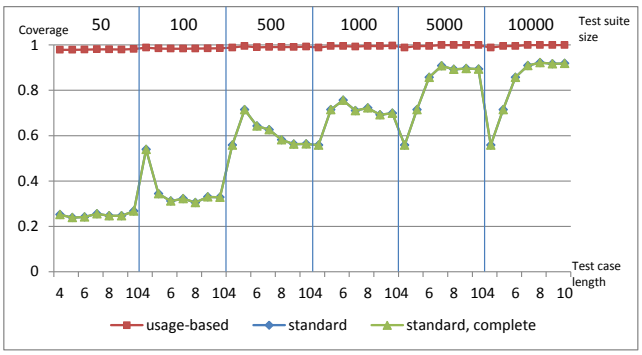



(a) JFC data set.

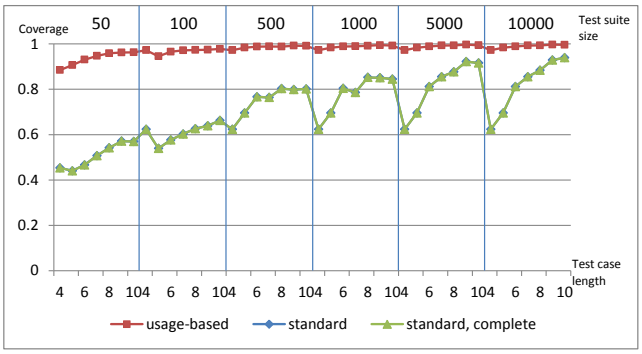



(b) Web application data set.

Figure A.8.: Results for the depth one for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 4$.

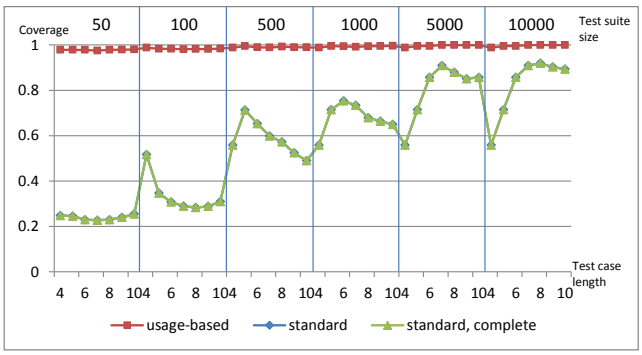



(a) JFC data set.

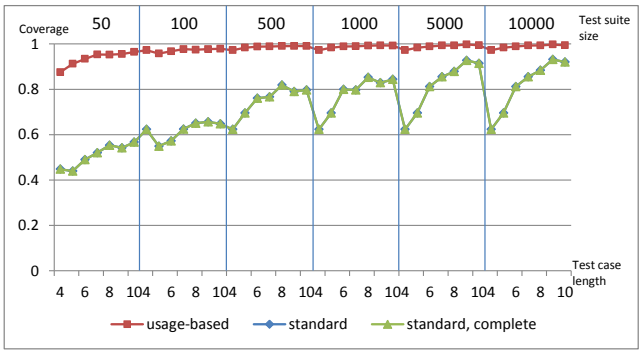



(b) Web application data set.

Figure A.9.: Results for the depth one for the hybrid test case generation with the PPM model with $k_{min} = 2, k_{max} = 3$.

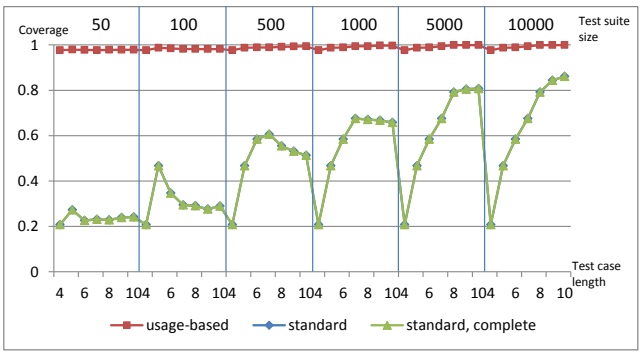



(a) JFC data set.

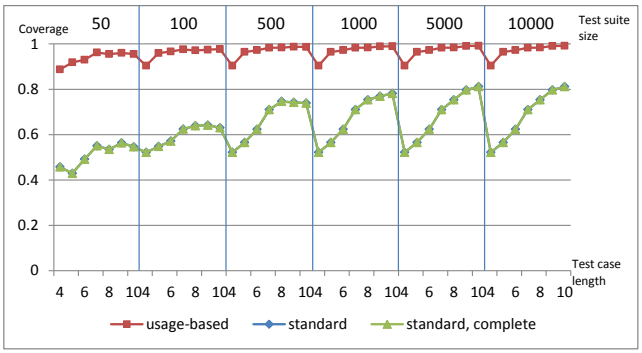



(b) Web application data set.

Figure A.10.: Results for the depth one for the hybrid test case generation with the PPM model with $k_{min} = 2, k_{max} = 4$.



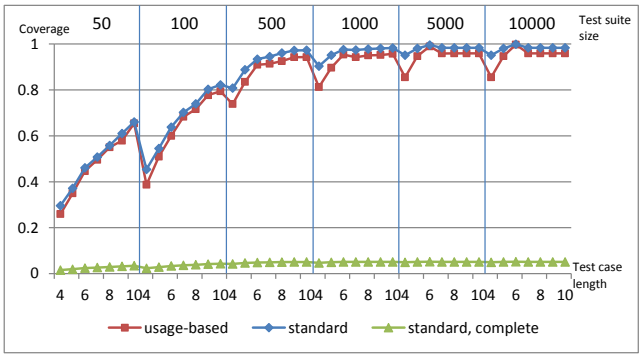

(a) JFC data set.



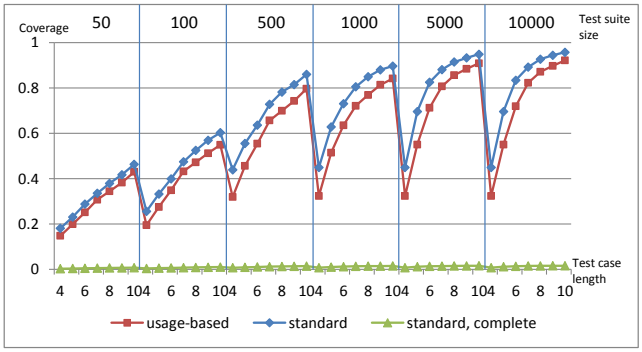

(b) Web application data set.

Figure A.11.: Results for the depth one for the hybrid test case generation with the PPM model with $k_{min} = 3, k_{max} = 4$.

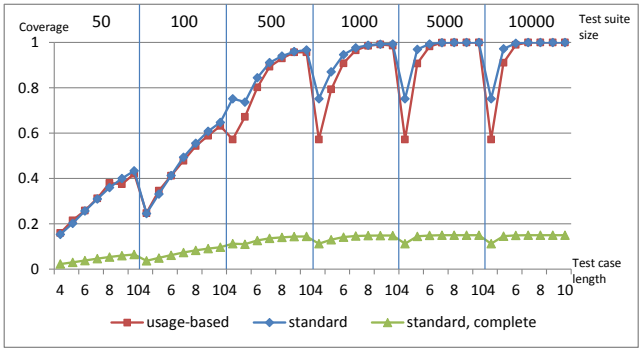
### A.4.2. Depth two for the hybrid.




(a) JFC data set.

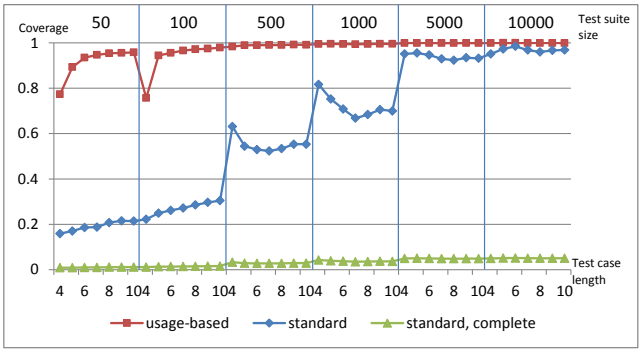



(b) MFC data set.

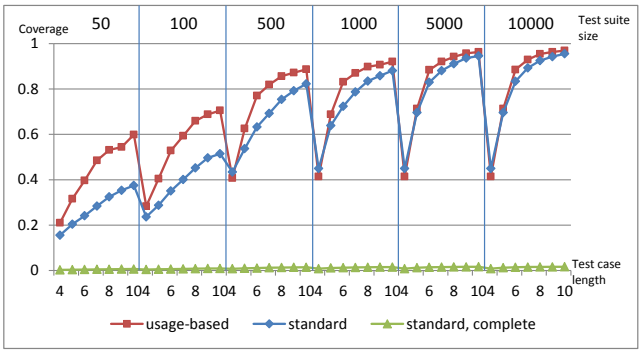



(c) Web application data set.

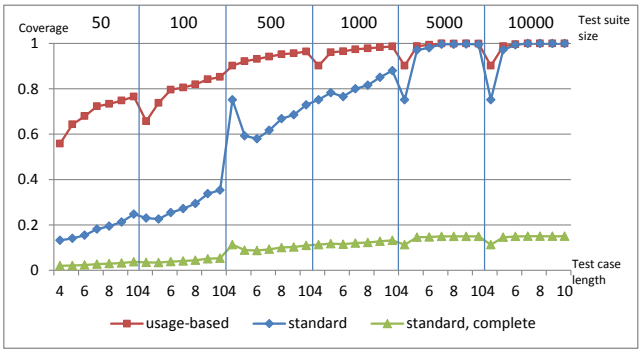
Figure A.12.: Results for the depth two for the hybrid test case generation with the random EFG.




(a) JFC data set.

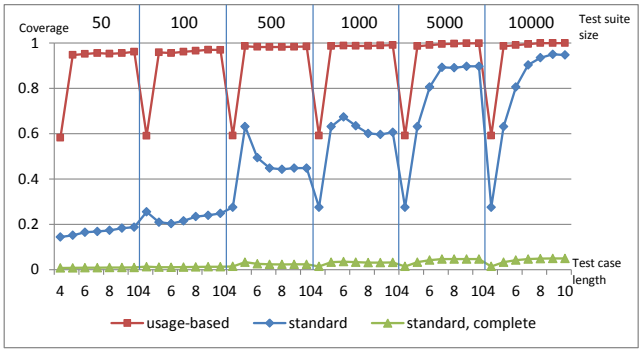



(b) MFC data set.

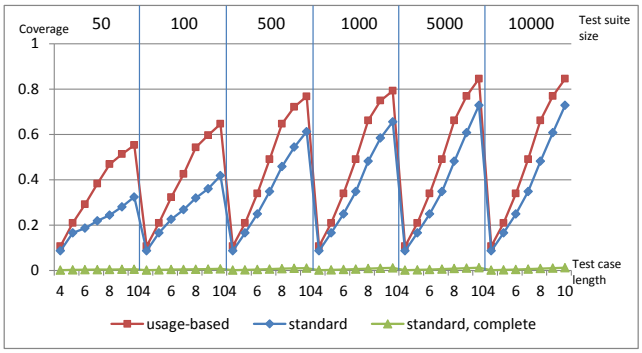



(c) Web application data set.

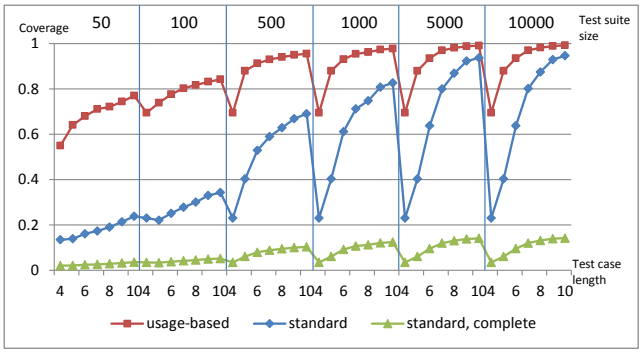
Figure A.13.: Results for the depth two for the hybrid test case generation with the first-order MM.




(a) JFC data set.

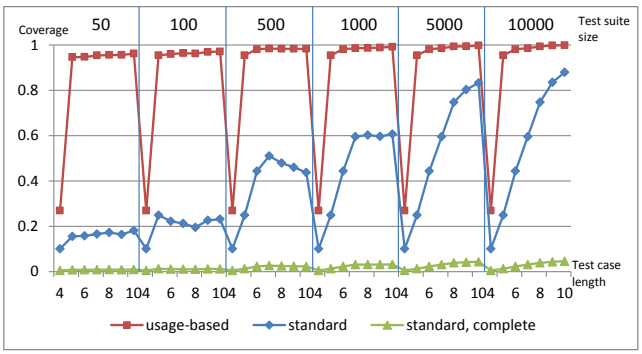



(b) MFC data set.

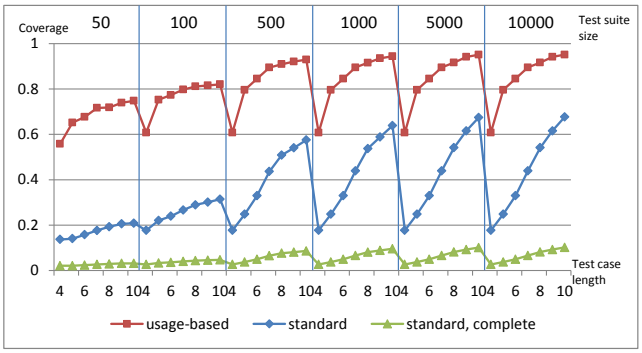



(c) Web application data set.

Figure A.14.: Results for the depth two for the hybrid test case generation with the 2nd-order MM.

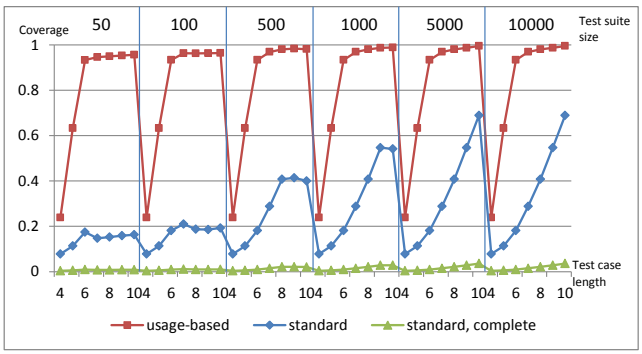



(a) JFC data set.

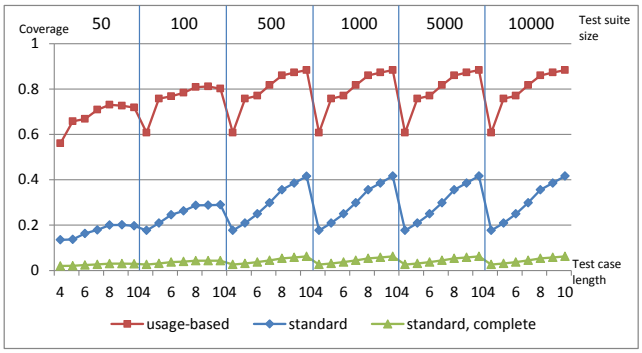



(b) Web application data set.

Figure A.15.: Results for the depth two for the hybrid test case generation with the 3rd-order MM.

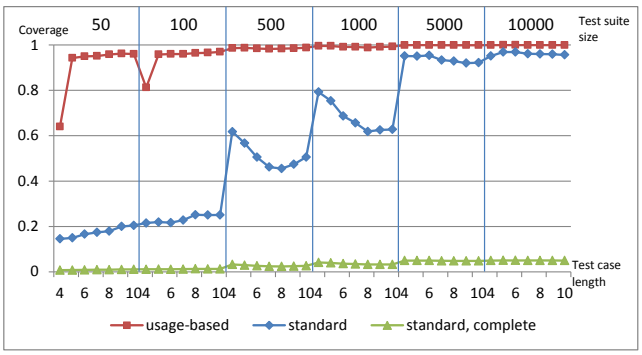



(a) JFC data set.

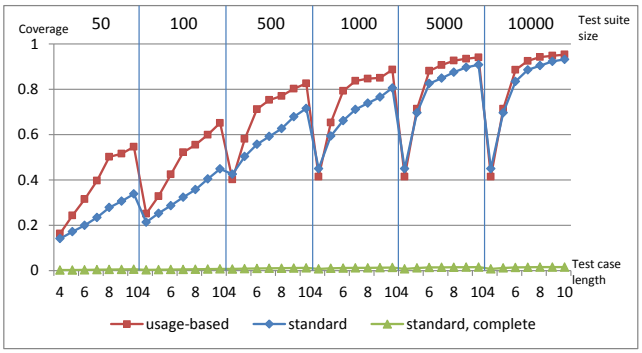



(b) Web application data set.

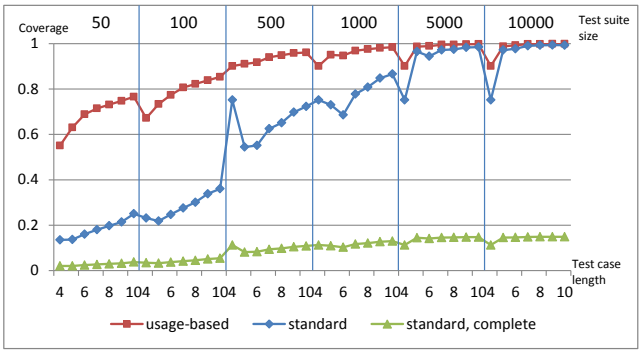
Figure A.16.: Results for the depth two for the hybrid test case generation with the 4th-order MM.




(a) JFC data set.

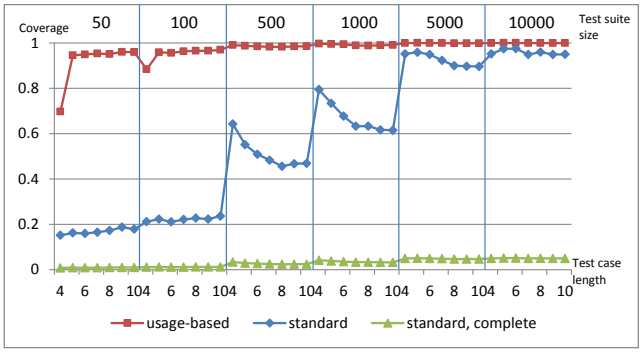



(b) MFC data set.

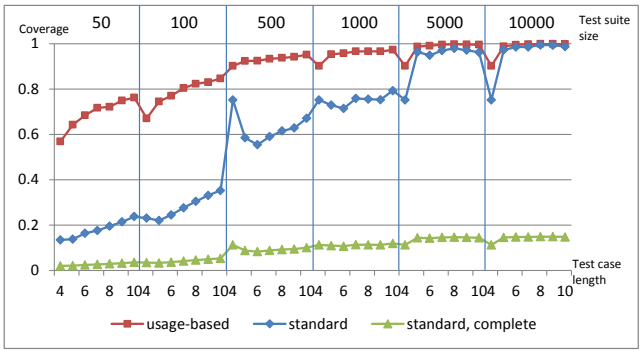



(c) Web application data set.

Figure A.17.: Results for the depth two for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 2$.

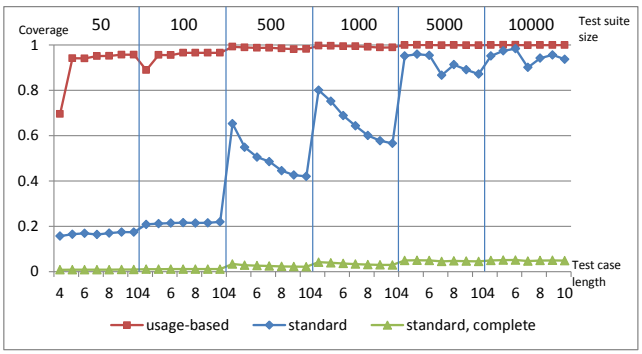



(a) JFC data set.

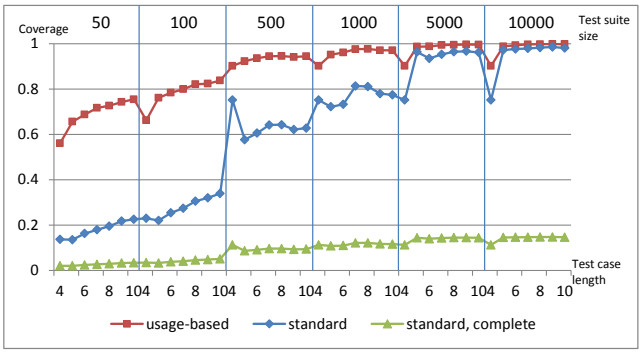



(b) Web application data set.

Figure A.18.: Results for the depth two for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 3$.

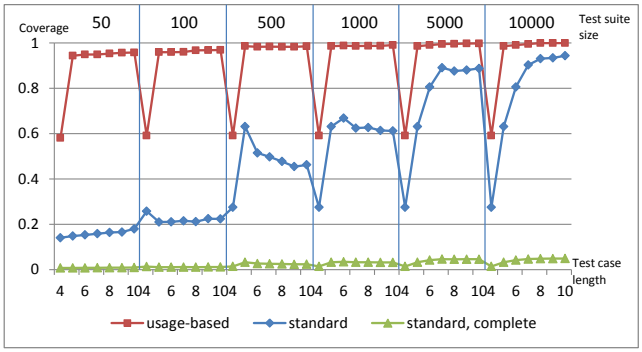



(a) JFC data set.

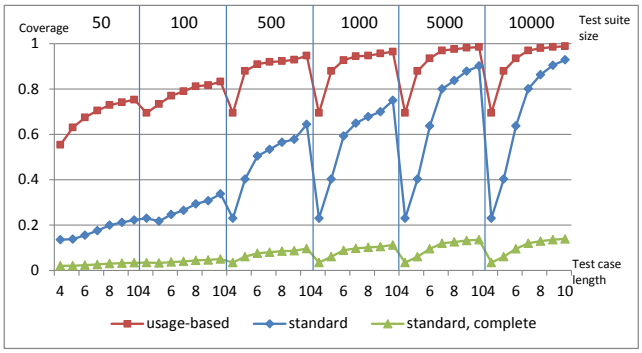



(b) Web application data set.

Figure A.19.: Results for the depth two for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 4$.

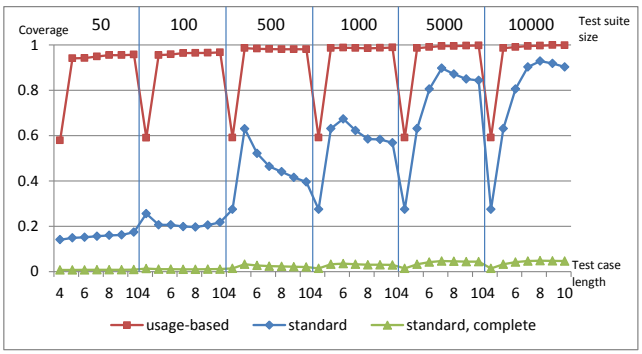



(a) JFC data set.

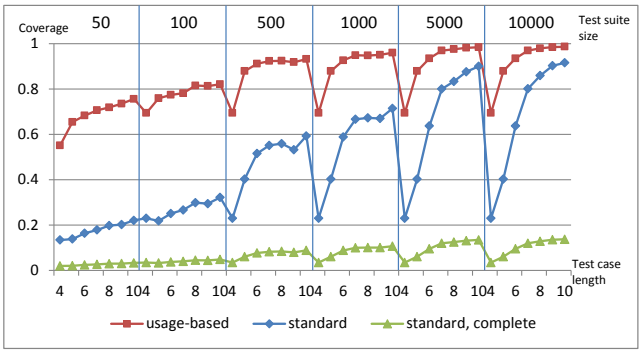



(b) Web application data set.

Figure A.20.: Results for the depth two for the hybrid test case generation with the PPM model with $k_{min} = 2, k_{max} = 3$.

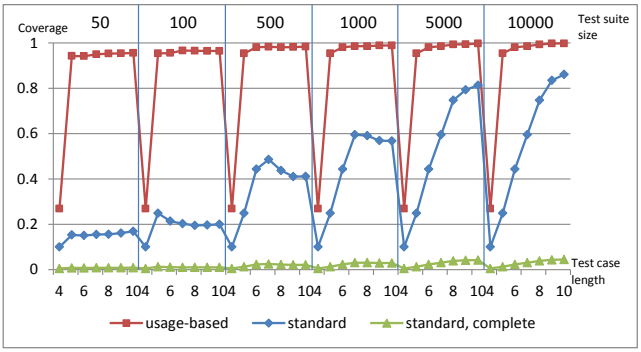



(a) JFC data set.

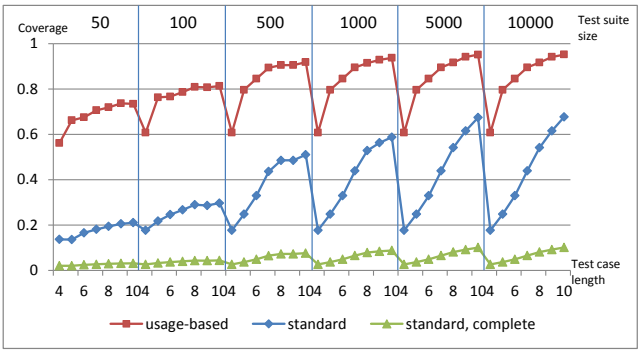



(b) Web application data set.

Figure A.21.: Results for the depth two for the hybrid test case generation with the PPM model with $k_{min} = 2, k_{max} = 4$.



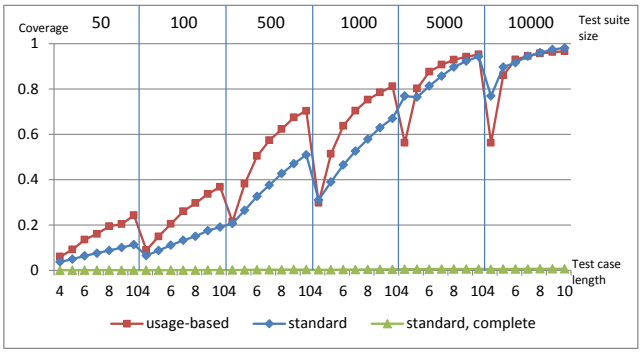

(a) JFC data set.



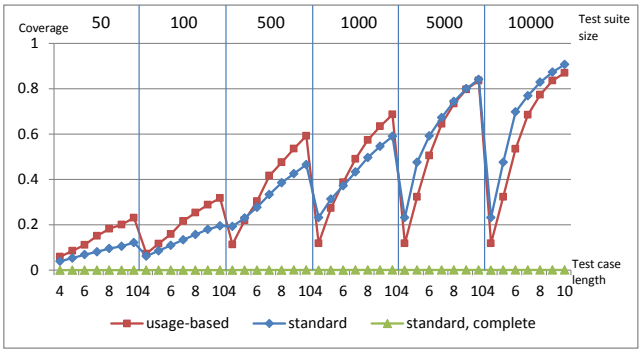

(b) Web application data set.

Figure A.22.: Results for the depth two for the hybrid test case generation with the PPM model with $k_{min} = 3, k_{max} = 4$.

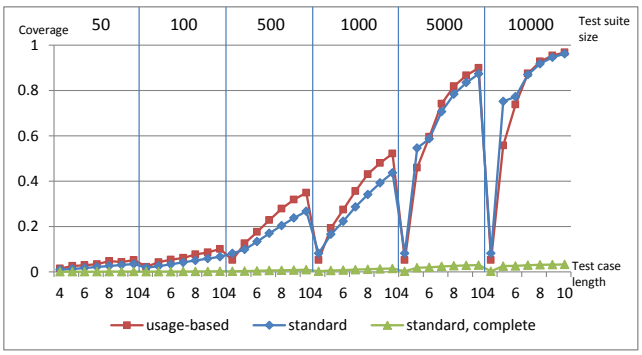
### A.4.3. Depth three for the hybrid.




(a) JFC data set.

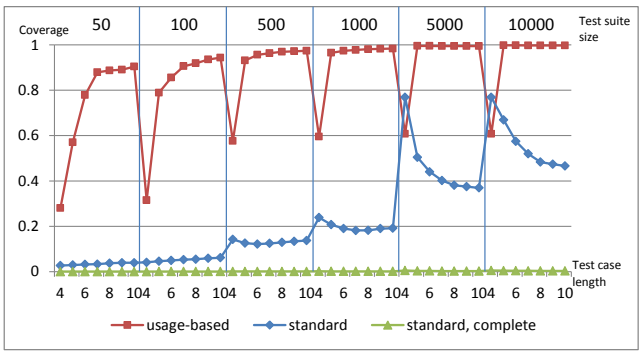



(b) MFC data set.

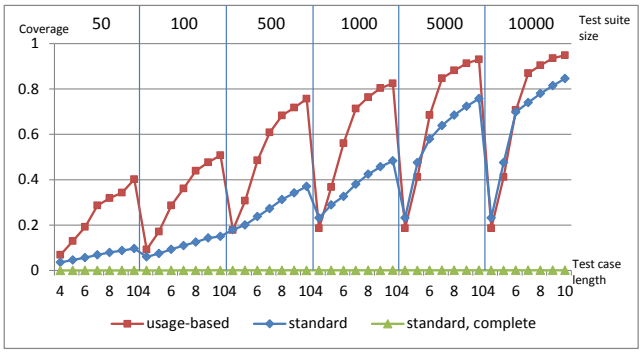



(c) Web application data set.

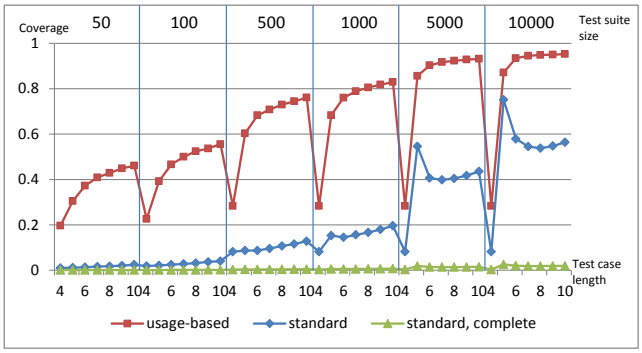
Figure A.23.: Results for the depth three for the hybrid test case generation with the random EFG.




(a) JFC data set.

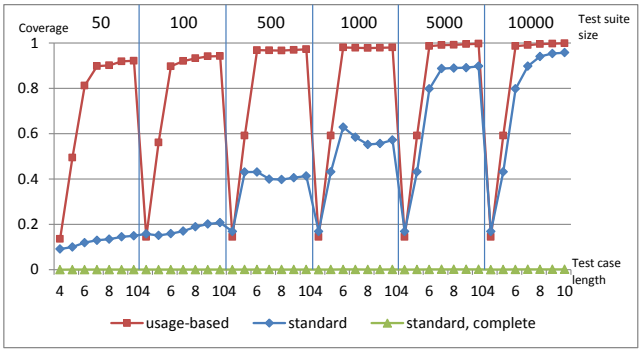



(b) MFC data set.

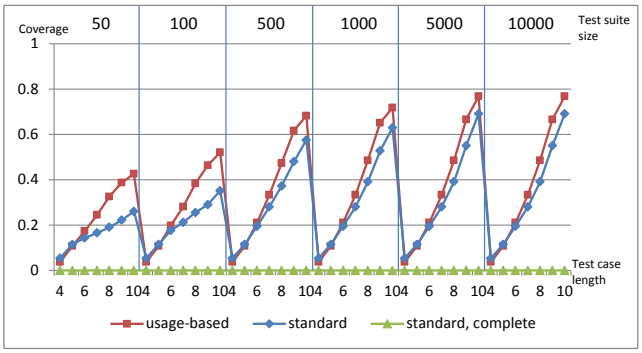



(c) Web application data set.

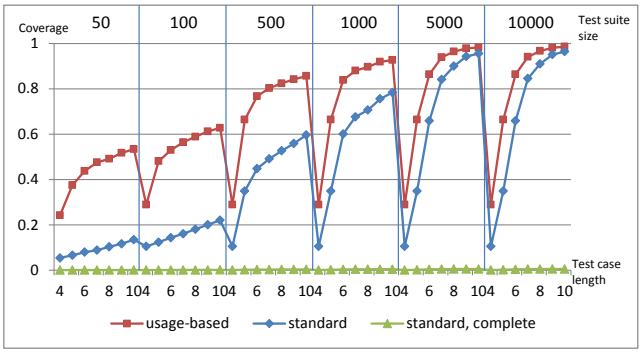
Figure A.24.: Results for the depth three for the hybrid test case generation with the first-order MM.




(a) JFC data set.

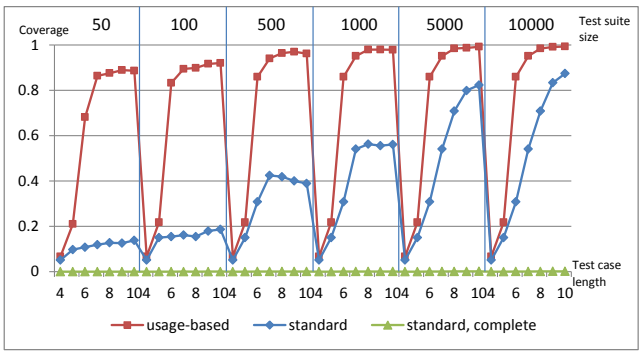



(b) MFC data set.

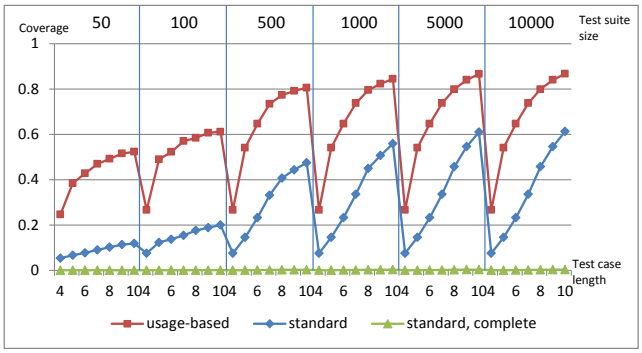



(c) Web application data set.

Figure A.25.: Results for the depth three for the hybrid test case generation with the 2nd-order MM.

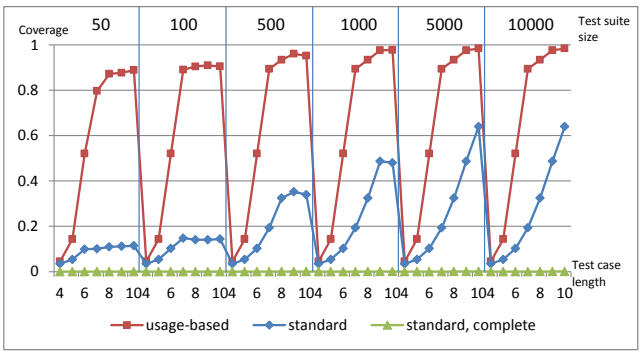



(a) JFC data set.

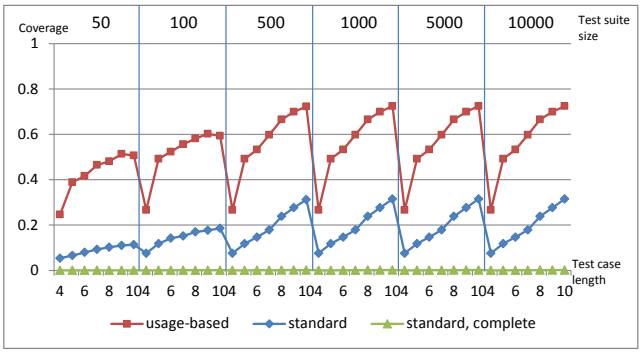



(b) Web application data set.

Figure A.26.: Results for the depth three for the hybrid test case generation with the 3rd-order MM.

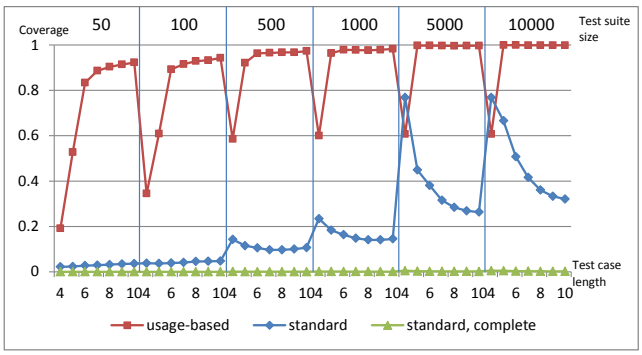



(a) JFC data set.

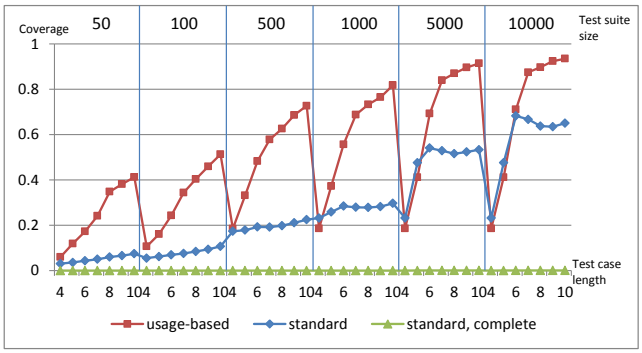



(b) Web application data set.

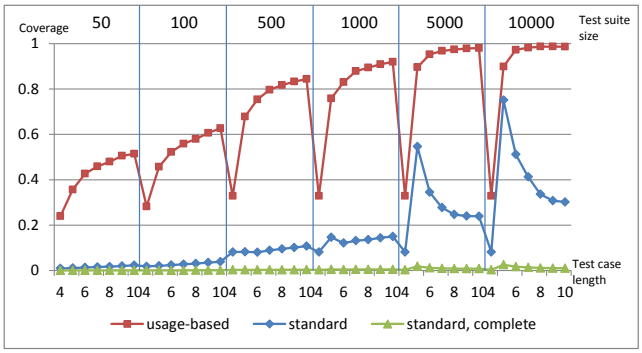
Figure A.27.: Results for the depth three for the hybrid test case generation with the 4th-order MM.




(a) JFC data set.

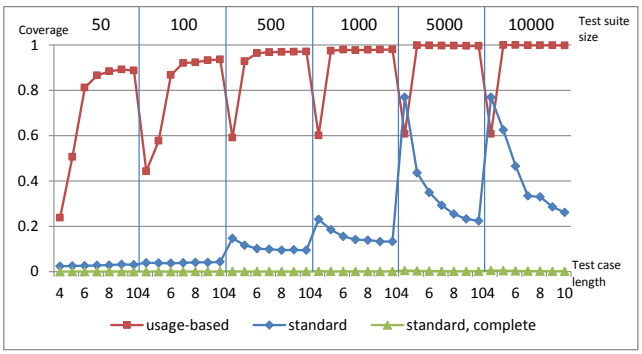



(b) MFC data set.

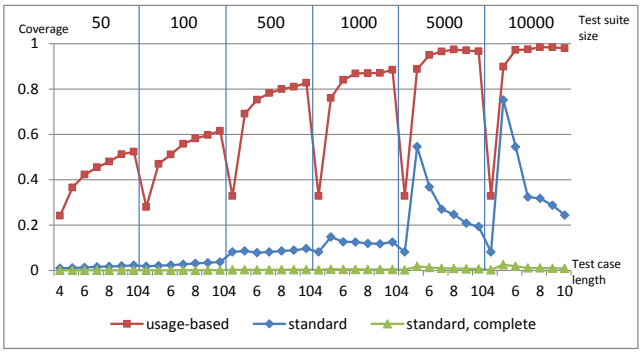



(c) Web application data set.

Figure A.28.: Results for the depth three for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 2$.

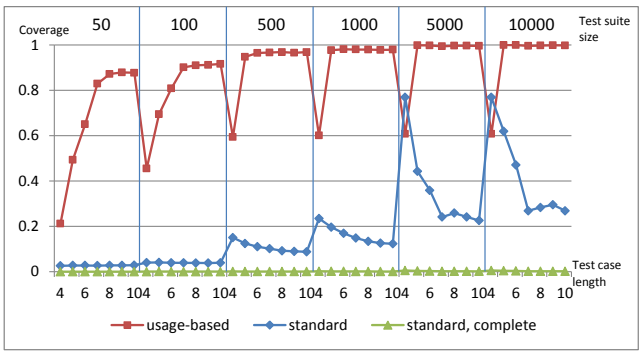



(a) JFC data set.

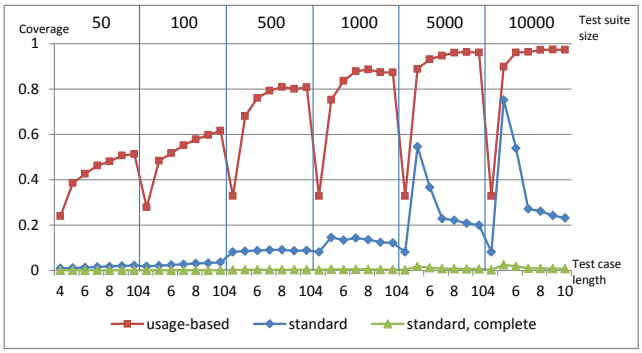



(b) Web application data set.

Figure A.29.: Results for the depth three for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 3$.

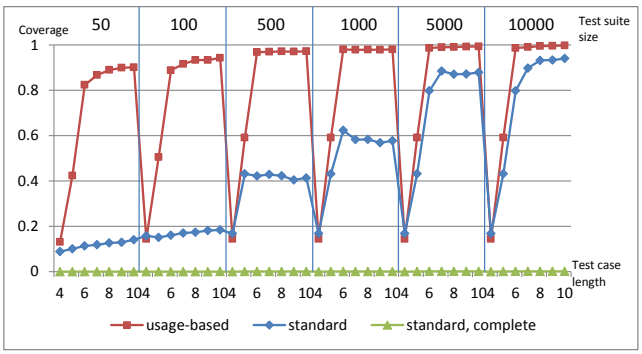



(a) JFC data set.

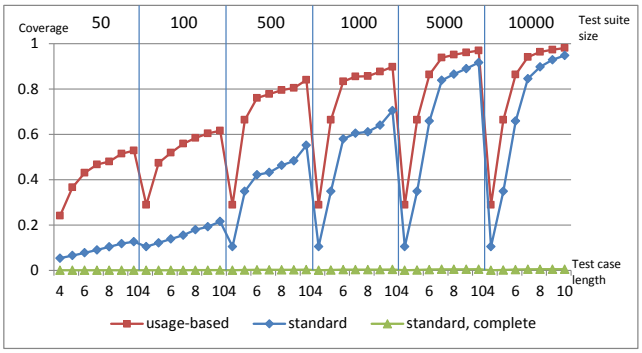



(b) Web application data set.

Figure A.30.: Results for the depth three for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 4$.

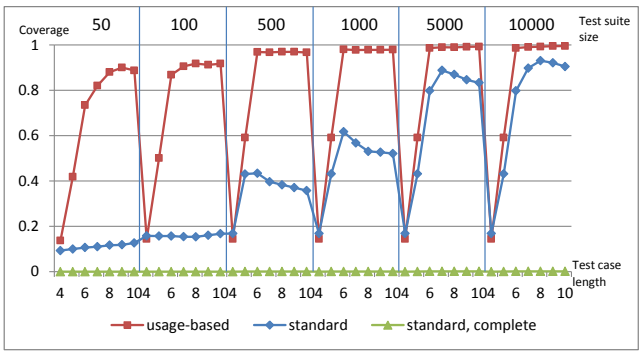



(a) JFC data set.

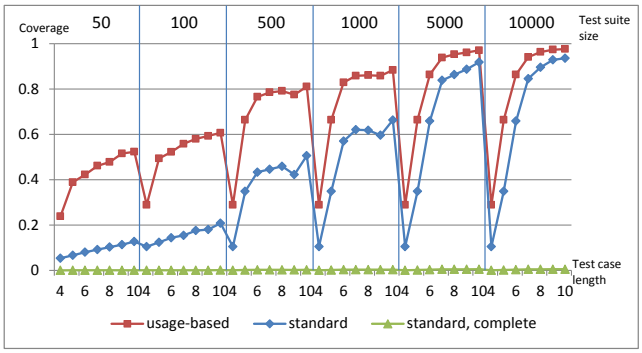



(b) Web application data set.

Figure A.31.: Results for the depth three for the hybrid test case generation with the PPM model with $k_{min} = 2, k_{max} = 3$.

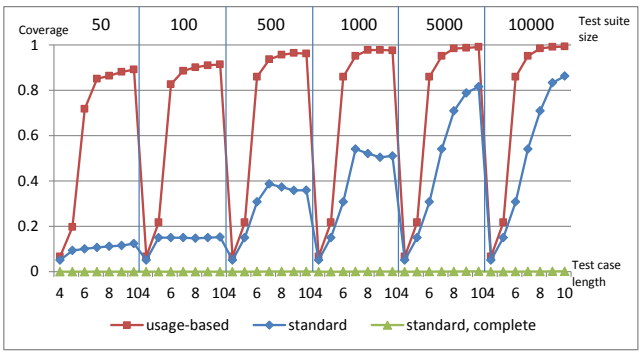



(a) JFC data set.

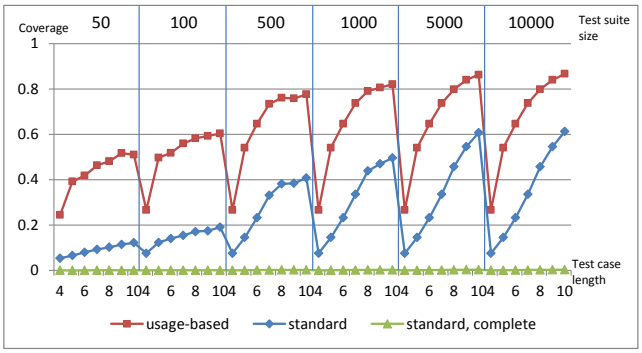



(b) Web application data set.

Figure A.32.: Results for the depth three for the hybrid test case generation with the PPM model with $k_{min} = 2, k_{max} = 4$.



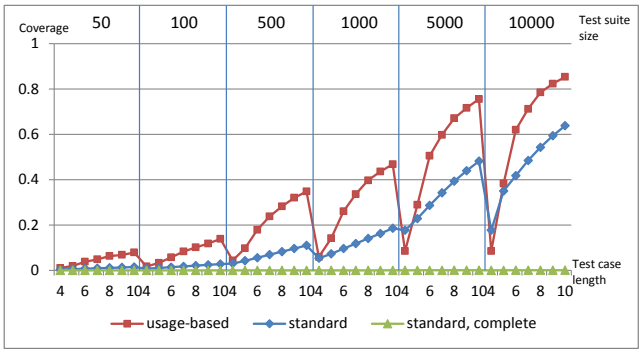

(a) JFC data set.



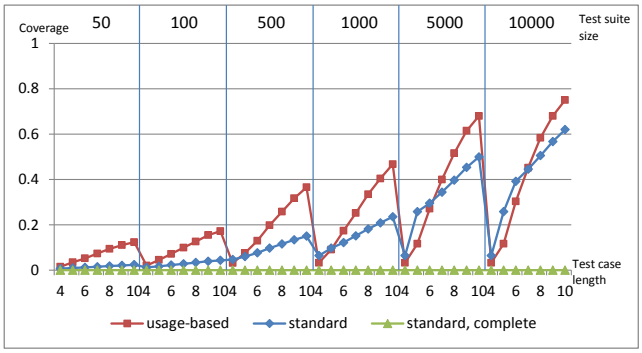

(b) Web application data set.

Figure A.33.: Results for the depth three for the hybrid test case generation with the PPM model with $k_{min} = 3, k_{max} = 4$.

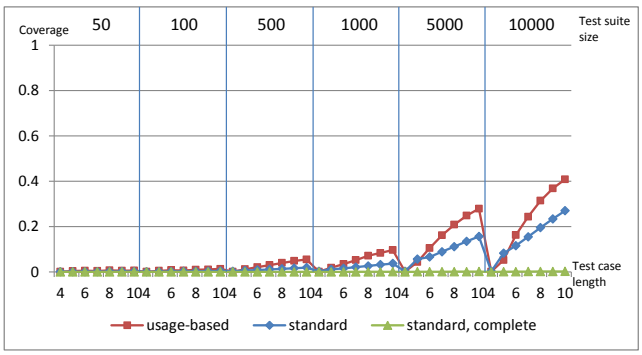
### A.4.4. Depth four for the hybrid.




(a) JFC data set.

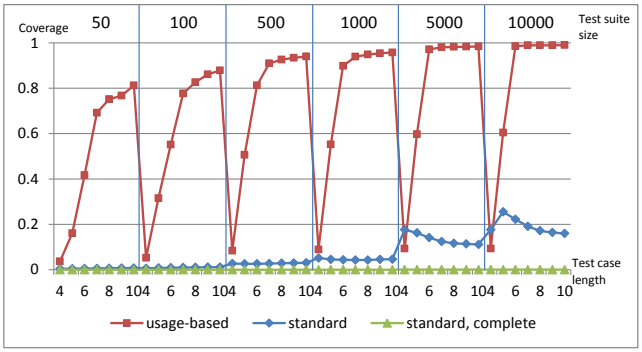



(b) MFC data set.

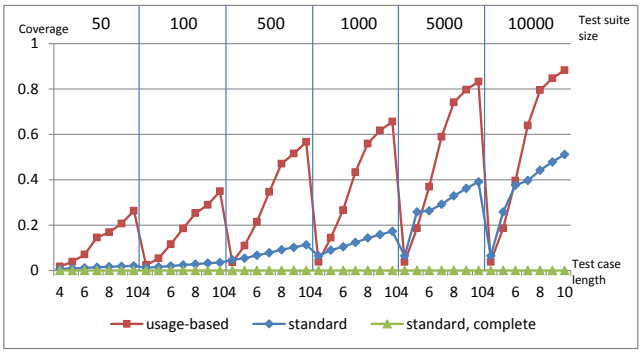



(c) Web application data set.

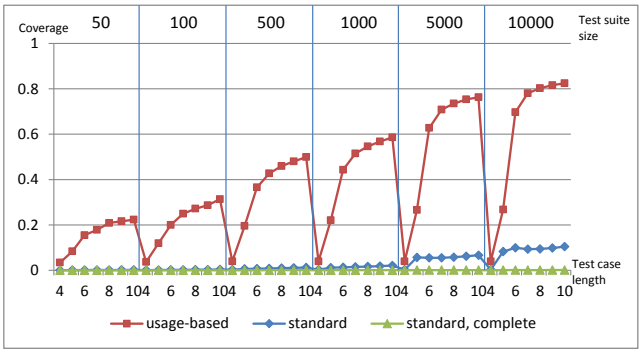
Figure A.34.: Results for the depth four for the hybrid test case generation with the random EFG.




(a) JFC data set.

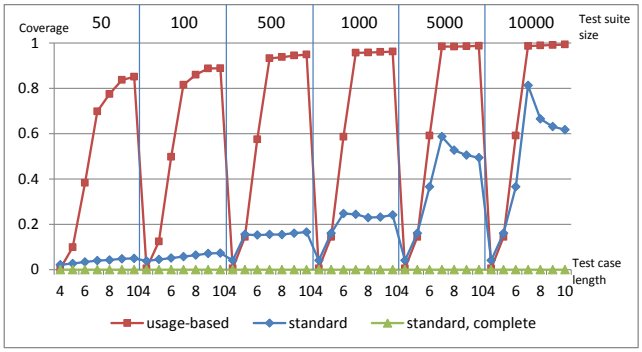



(b) MFC data set.

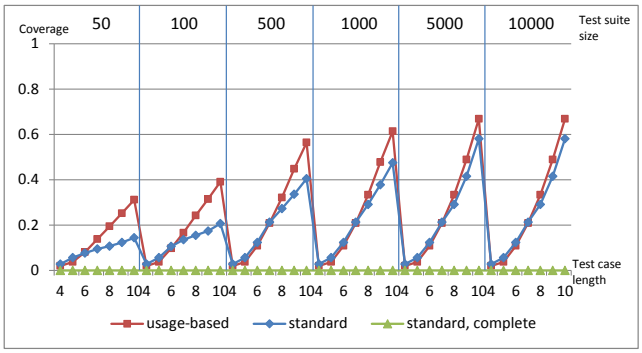



(c) Web application data set.

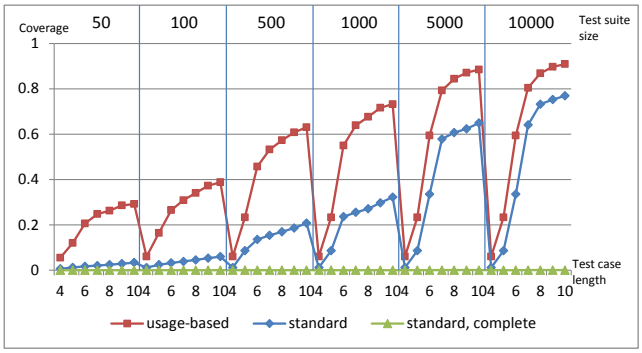
Figure A.35.: Results for the depth four for the hybrid test case generation with the first-order MM.




(a) JFC data set.

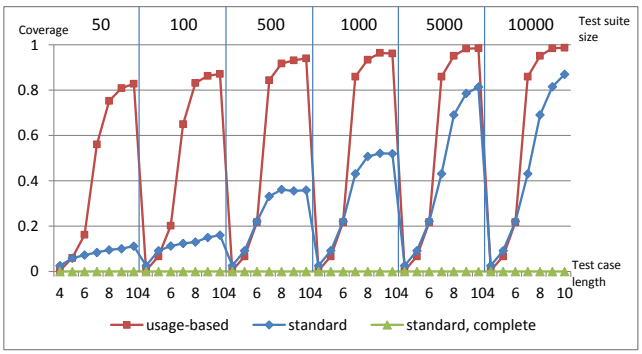



(b) MFC data set.

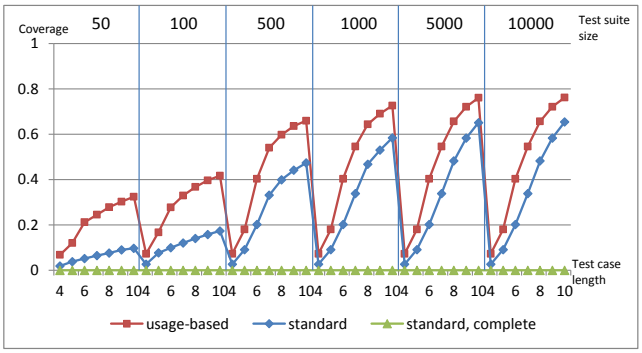



(c) Web application data set.

Figure A.36.: Results for the depth four for the hybrid test case generation with the 2nd-order MM.

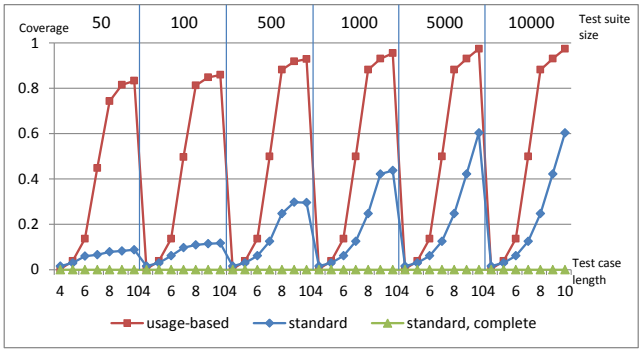



(a) JFC data set.

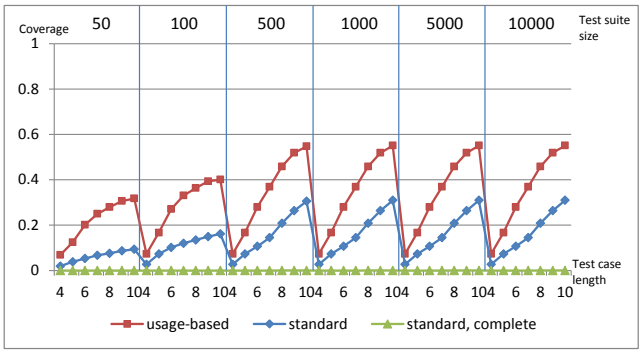



(b) Web application data set.

Figure A.37.: Results for the depth four for the hybrid test case generation with the 3rd-order MM.

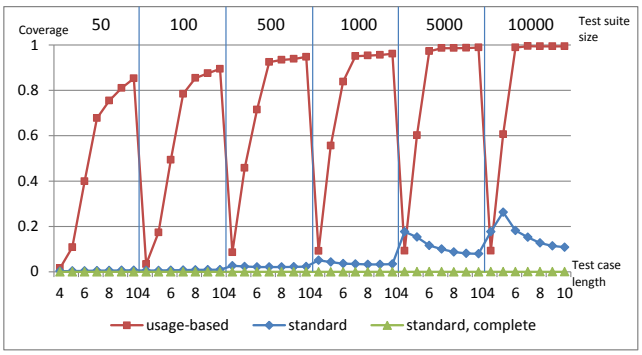



(a) JFC data set.

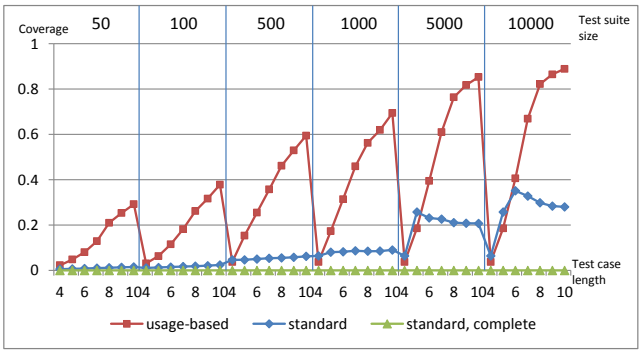



(b) Web application data set.

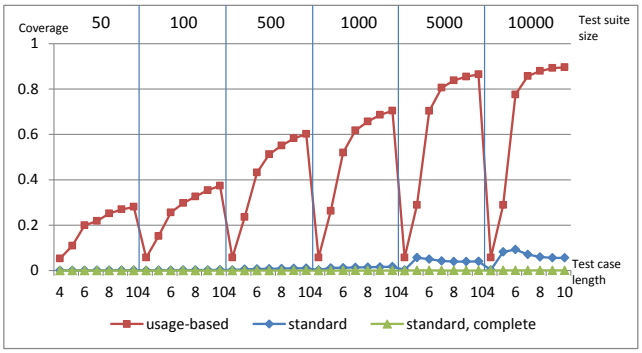
Figure A.38.: Results for the depth four for the hybrid test case generation with the 4th-order MM.




(a) JFC data set.

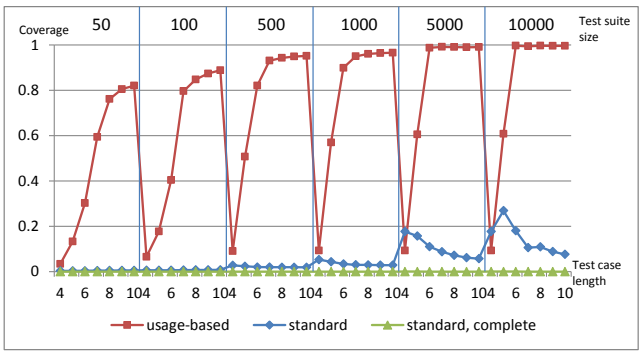



(b) MFC data set.

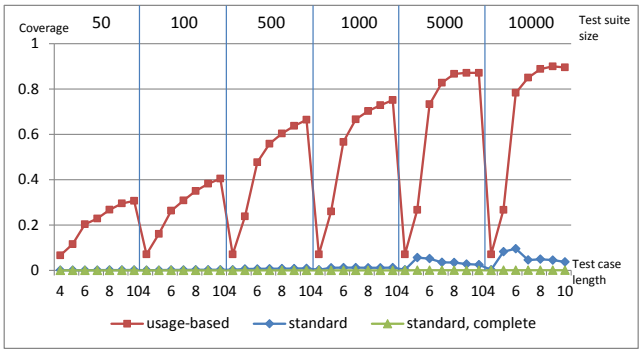



(c) Web application data set.

Figure A.39.: Results for the depth four for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 2$.

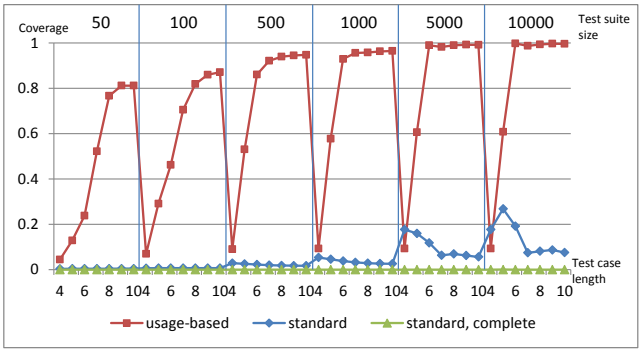



(a) JFC data set.

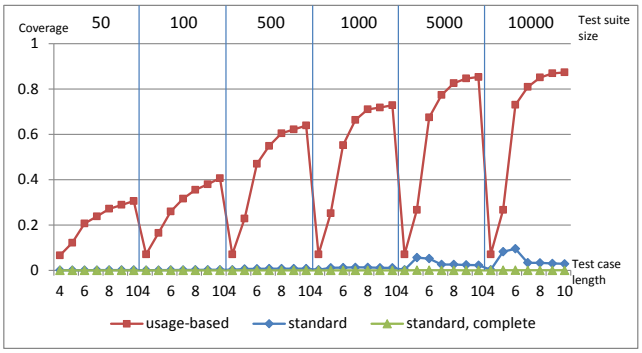



(b) Web application data set.

Figure A.40.: Results for the depth four for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 3$.

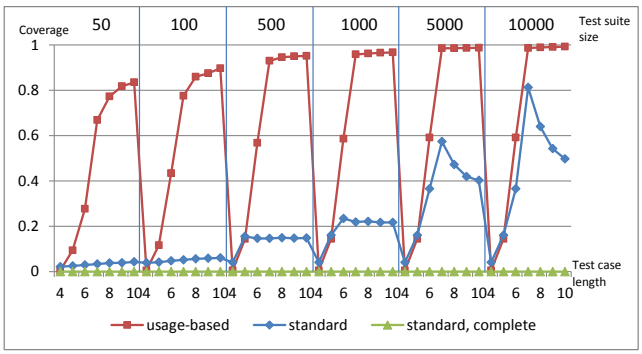



(a) JFC data set.

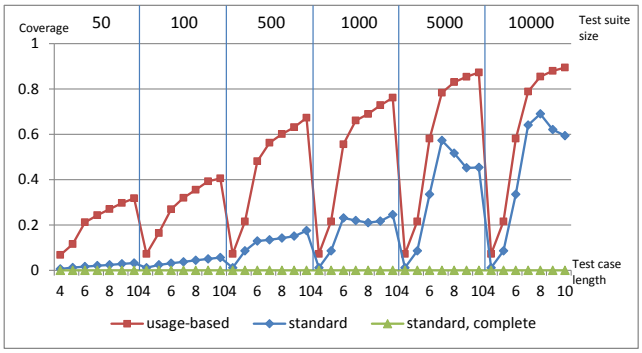



(b) Web application data set.

Figure A.41.: Results for the depth four for the hybrid test case generation with the PPM model with $k_{min} = 1, k_{max} = 4$.

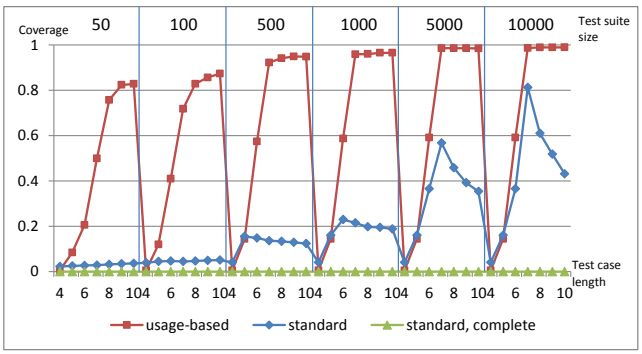



(a) JFC data set.

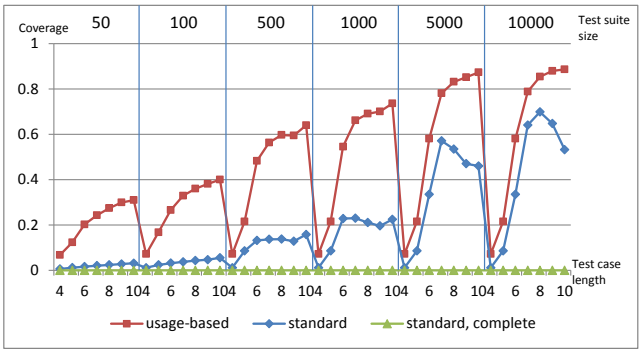



(b) Web application data set.

Figure A.42.: Results for the depth four for the hybrid test case generation with the PPM model with $k_{min} = 2, k_{max} = 3$.

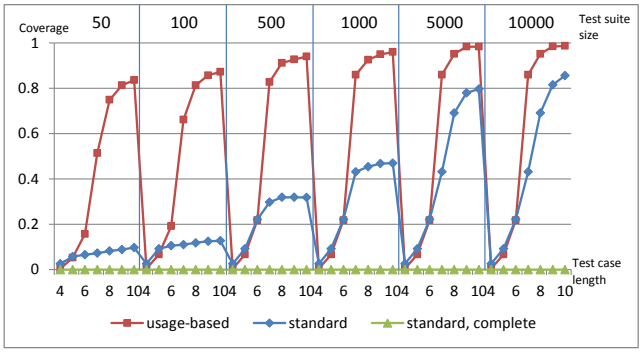



(a) JFC data set.

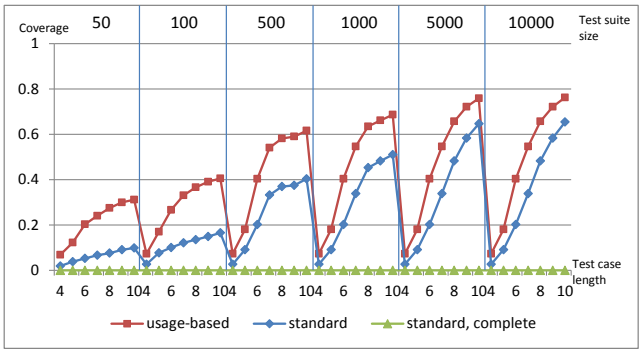



(b) Web application data set.

Figure A.43.: Results for the depth four for the hybrid test case generation with the PPM model with $k_{min} = 2, k_{max} = 4$.



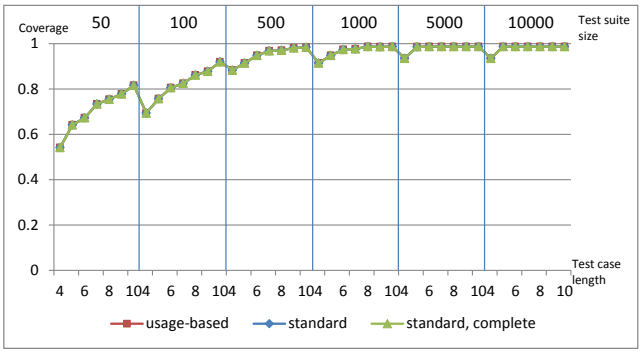

(a) JFC data set.



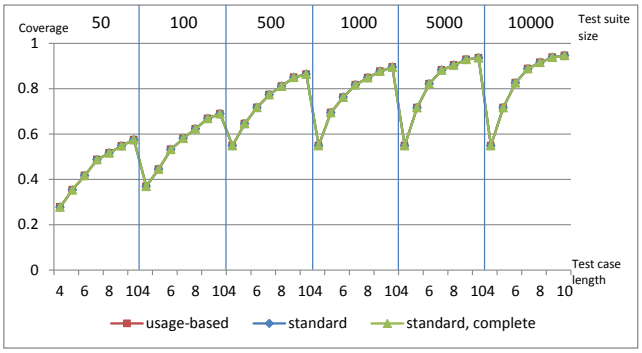

(b) Web application data set.

Figure A.44.: Results for the depth four for the hybrid test case generation with the PPM model with $k_{min} = 3, k_{max} = 4$.

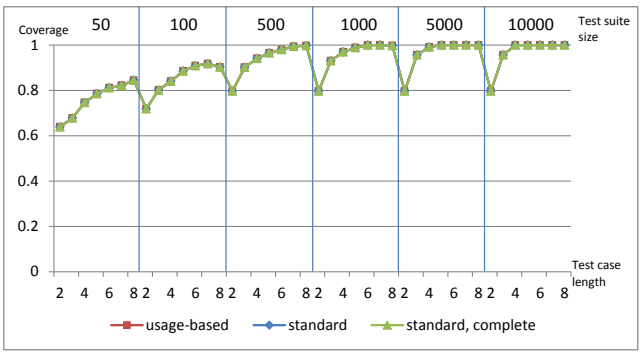
### A.4.5. Depth one for the random walk.




(a) JFC data set.

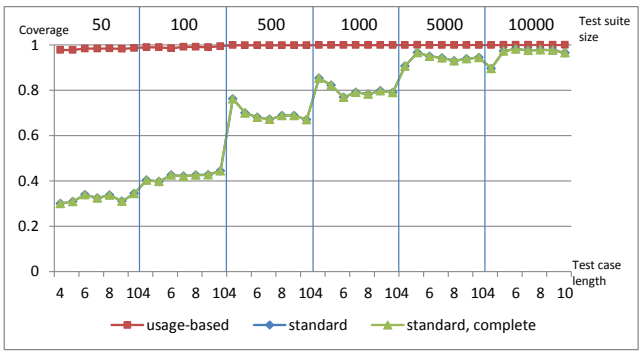



(b) MFC data set.

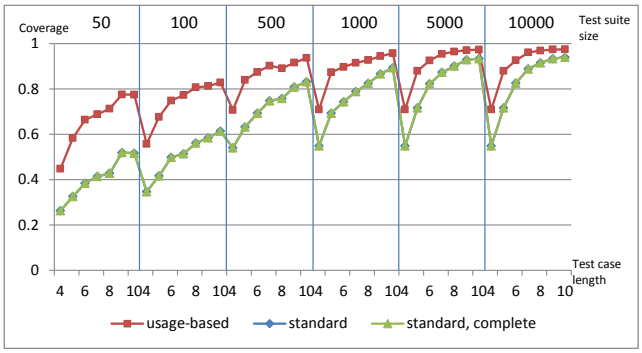



(c) Web application data set.

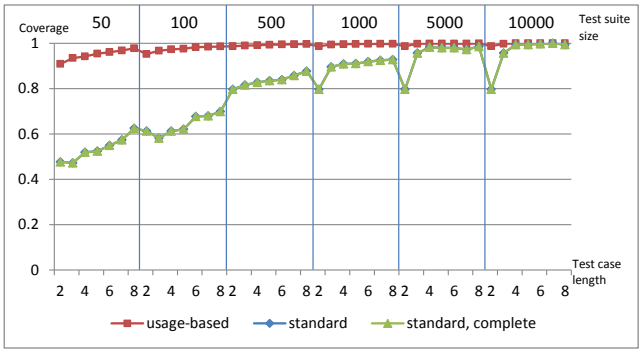
Figure A.45.: Results for the depth one for the random walk test case generation with the random EFG.




(a) JFC data set.

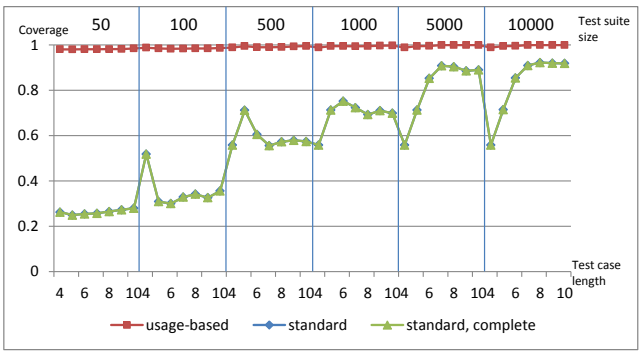



(b) MFC data set.

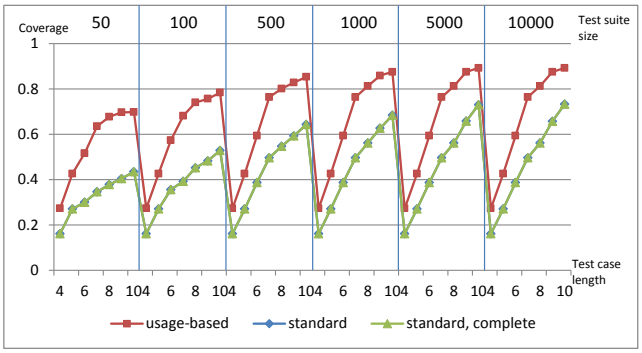



(c) Web application data set.

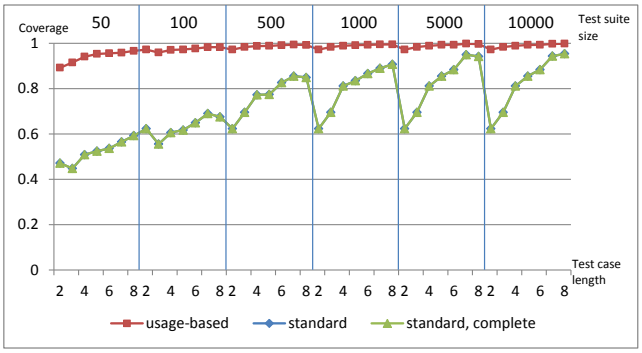
Figure A.46.: Results for the depth one for the random walk test case generation with the first-order MM.




(a) JFC data set.

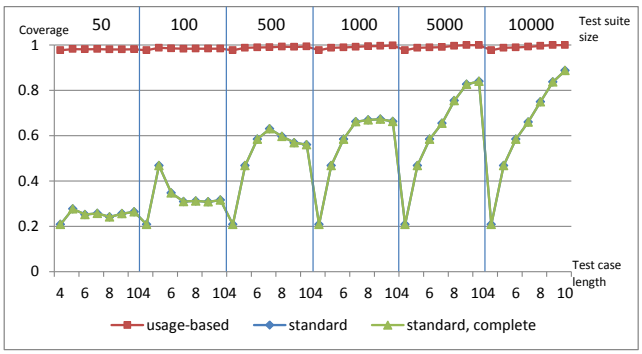



(b) MFC data set.

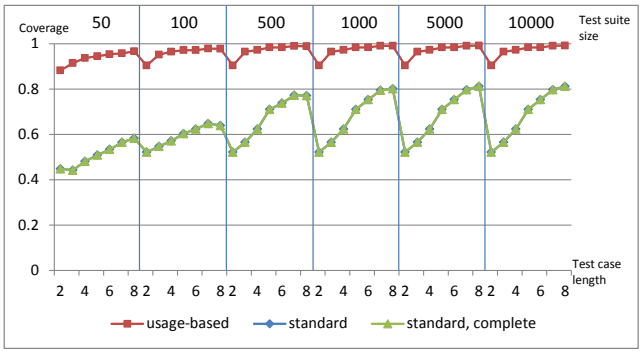



(c) Web application data set.

Figure A.47.: Results for the depth one for the random walk test case generation with the 2nd-order MM.

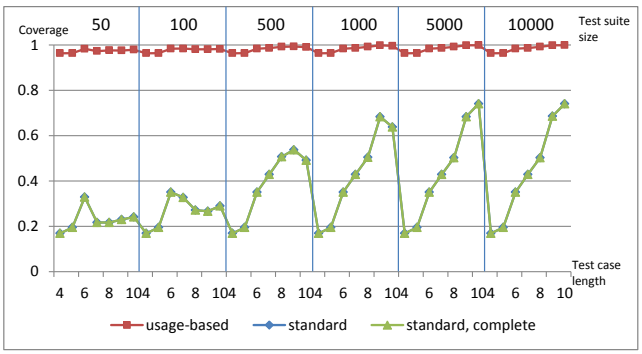



(a) JFC data set.

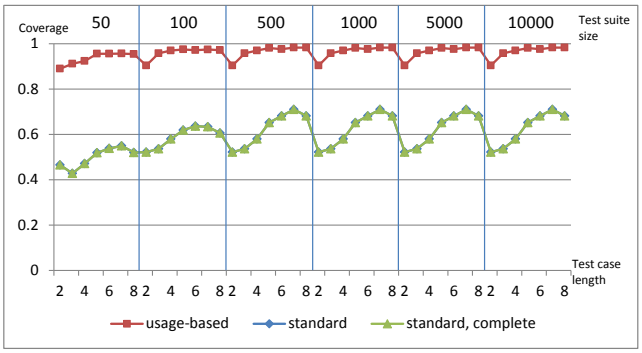



(b) Web application data set.

Figure A.48.: Results for the depth one for the random walk test case generation with the 3rd-order MM.

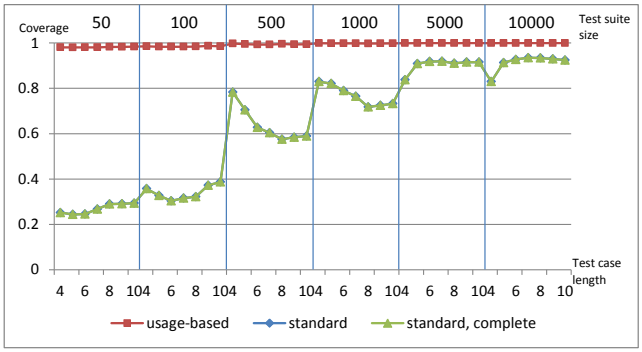



(a) JFC data set.

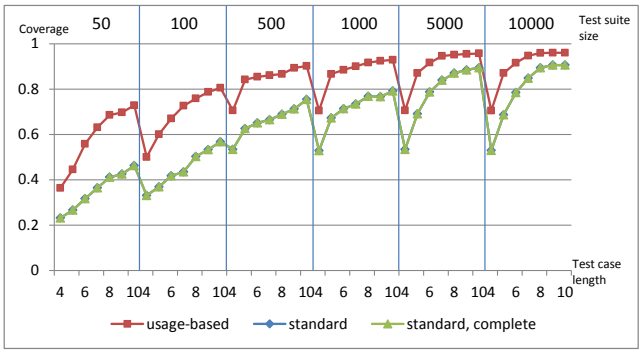



(b) Web application data set.

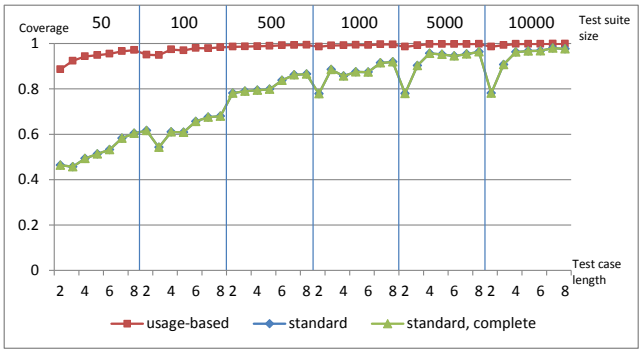
Figure A.49.: Results for the depth one for the random walk test case generation with the 4th-order MM.




(a) JFC data set.

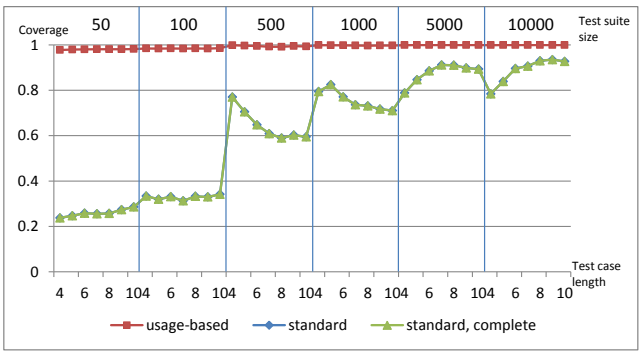



(b) MFC data set.

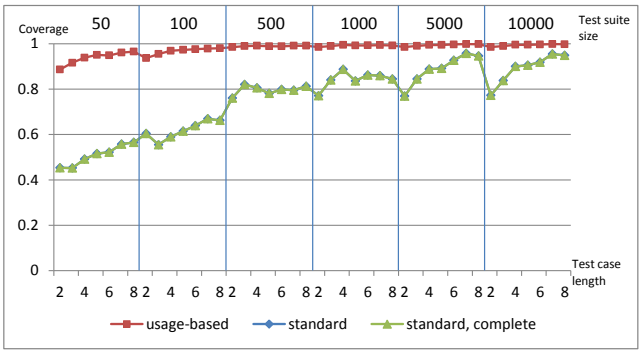



(c) Web application data set.

Figure A.50.: Results for the depth one for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 2$.

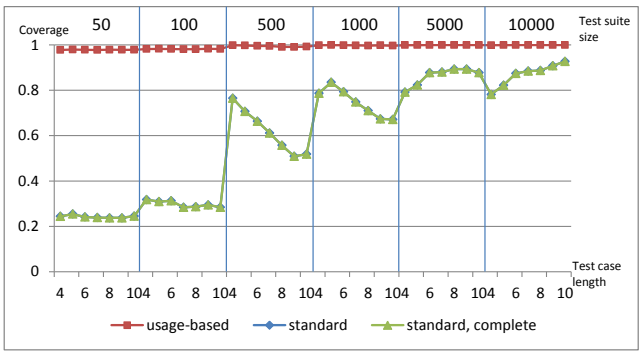



(a) JFC data set.

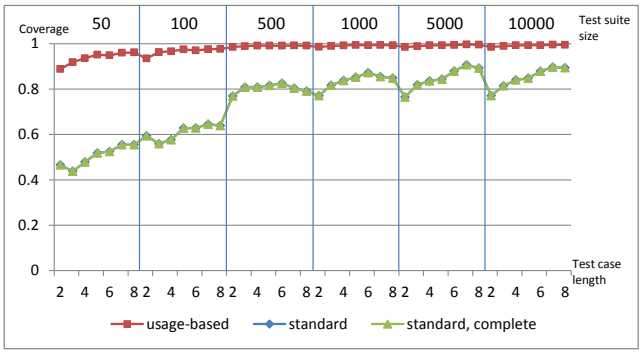



(b) Web application data set.

Figure A.51.: Results for the depth one for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 3$.

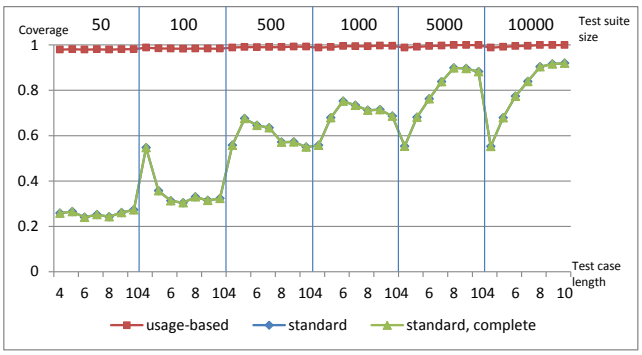



(a) JFC data set.

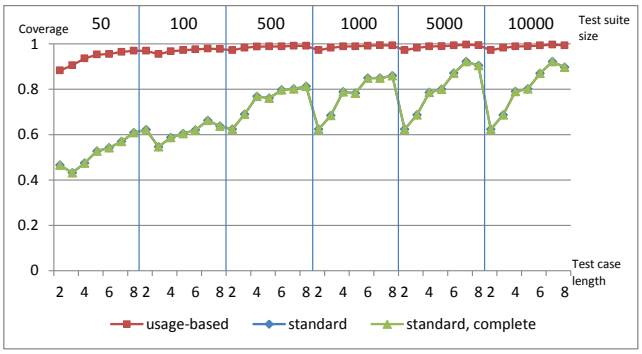



(b) Web application data set.

Figure A.52.: Results for the depth one for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 4$.

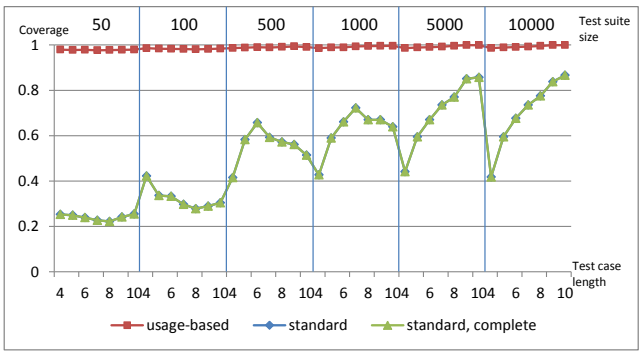



(a) JFC data set.

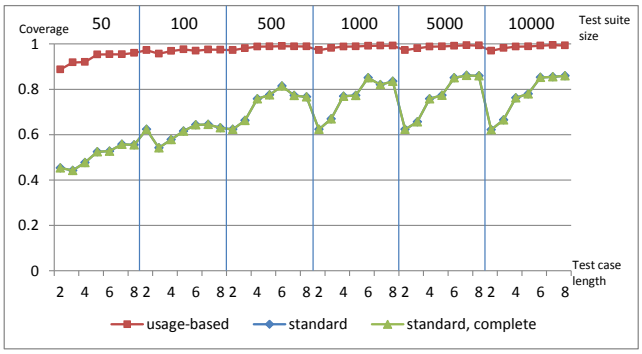



(b) Web application data set.

Figure A.53.: Results for the depth one for the random walk test case generation with the PPM model with $k_{min} = 2, k_{max} = 3$.

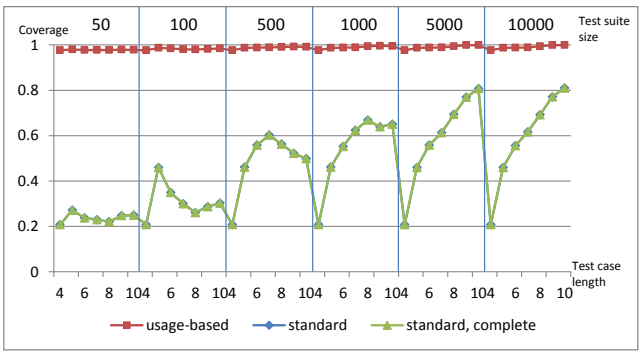



(a) JFC data set.

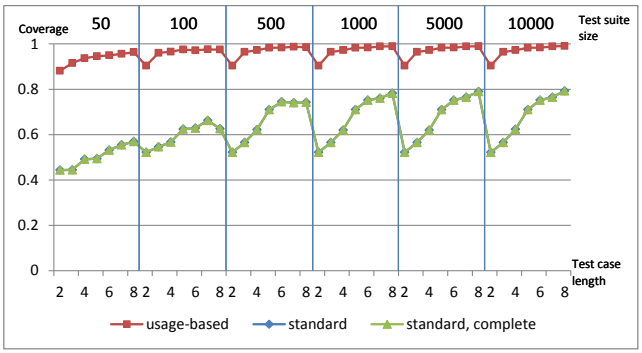



(b) Web application data set.

Figure A.54.: Results for the depth one for the random walk test case generation with the PPM model with $k_{min} = 2, k_{max} = 4$.



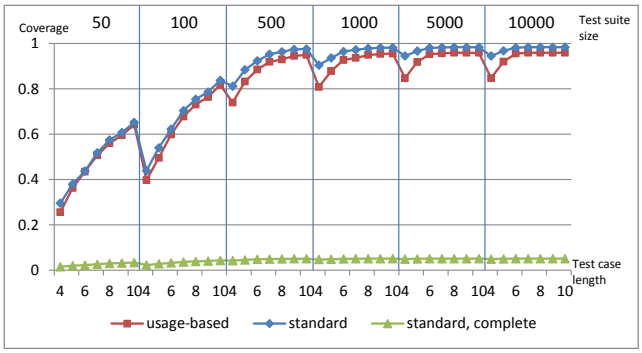

(a) JFC data set.



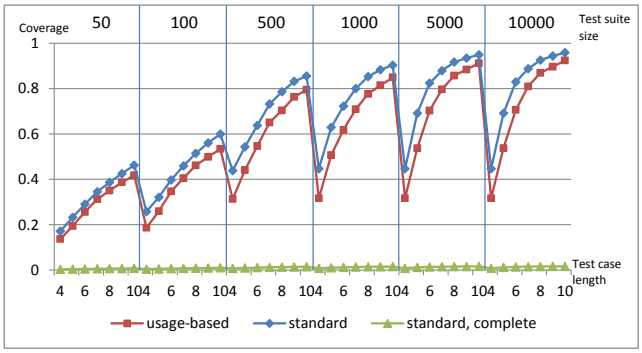

(b) Web application data set.

Figure A.55.: Results for the depth one for the random walk test case generation with the PPM model with $k_{min} = 3, k_{max} = 4$.

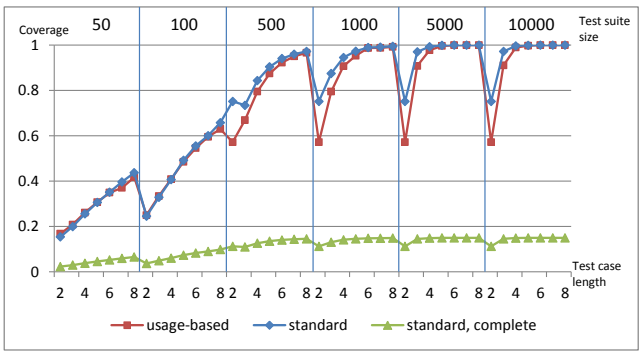
### A.4.6. Depth two for the random walk.




(a) JFC data set.

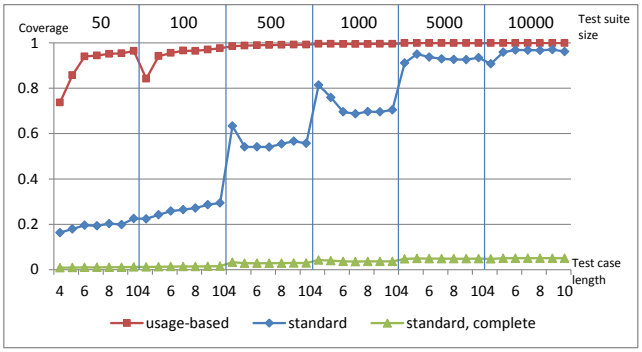



(b) MFC data set.

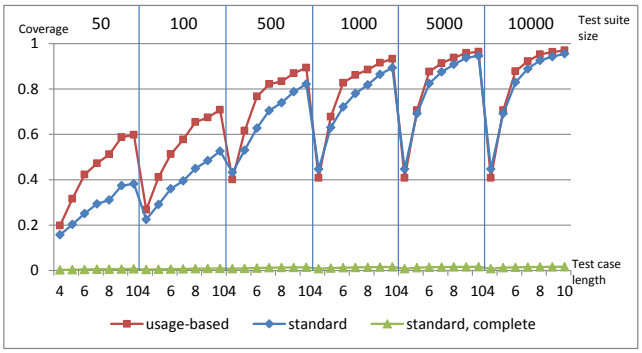



(c) Web application data set.

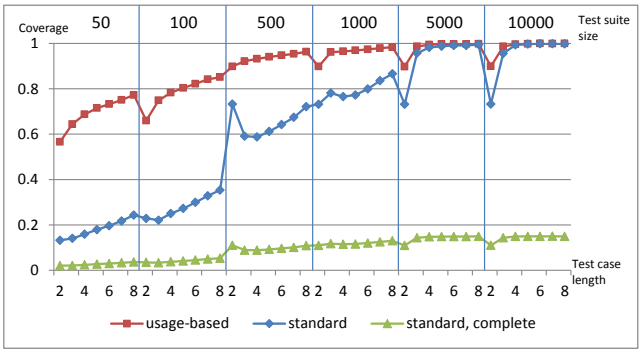
Figure A.56.: Results for the depth two for the random walk test case generation with the random EFG.




(a) JFC data set.

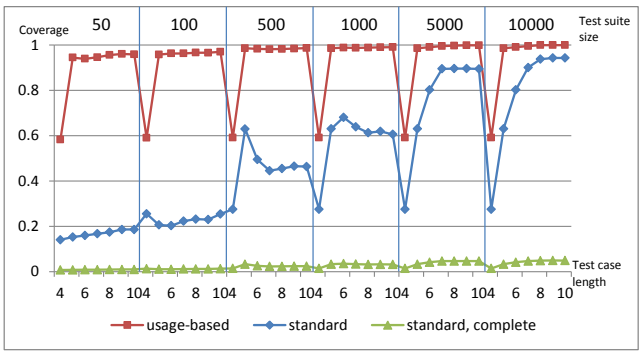



(b) MFC data set.

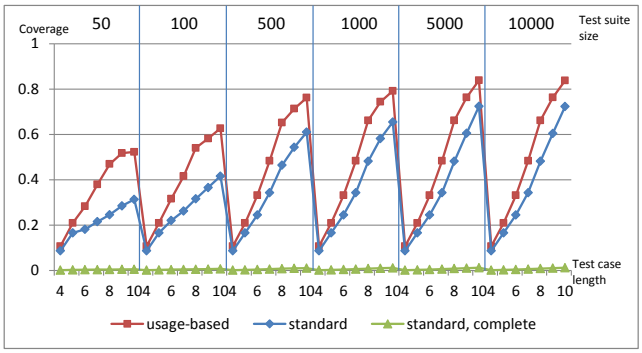



(c) Web application data set.

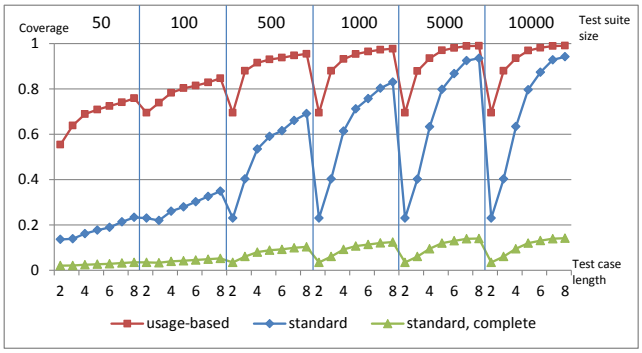
Figure A.57.: Results for the depth two for the random walk test case generation with the first-order MM.




(a) JFC data set.

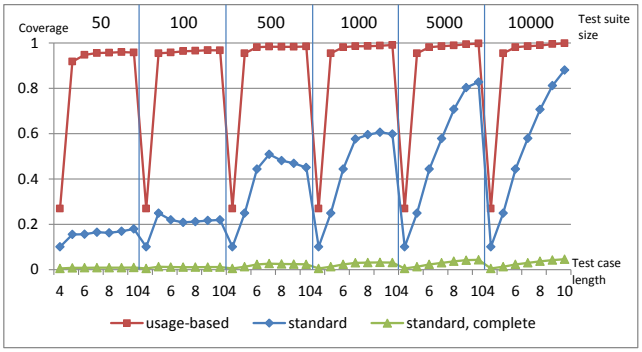



(b) MFC data set.

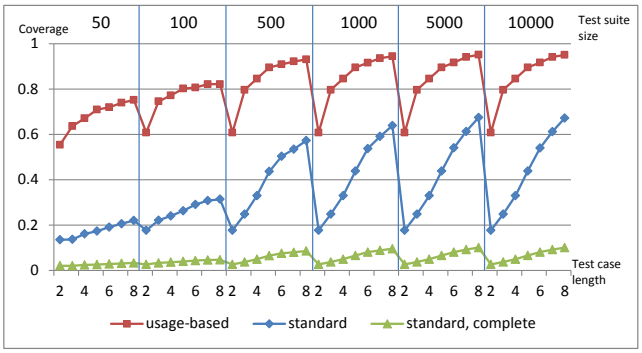



(c) Web application data set.

Figure A.58.: Results for the depth two for the random walk test case generation with the 2nd-order MM.

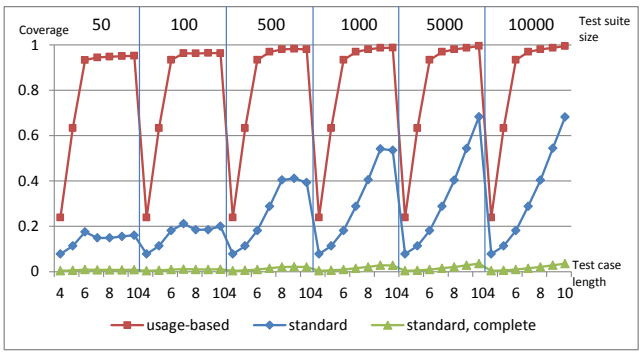



(a) JFC data set.

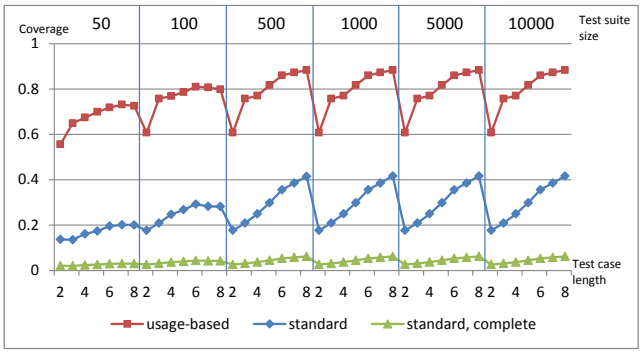



(b) Web application data set.

Figure A.59.: Results for the depth two for the random walk test case generation with the 3rd-order MM.

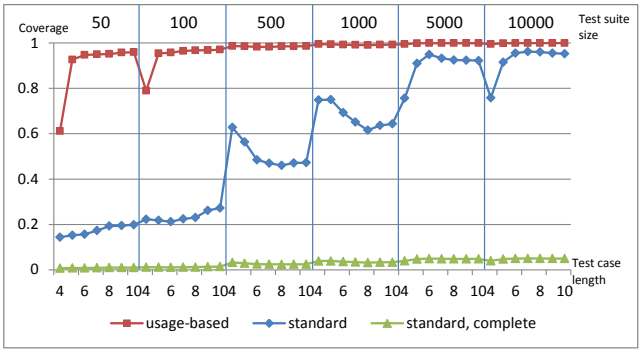



(a) JFC data set.

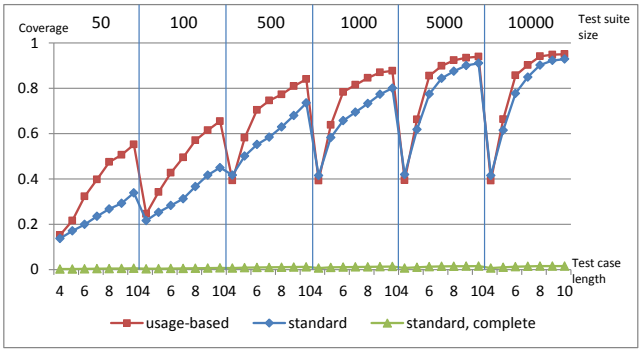



(b) Web application data set.

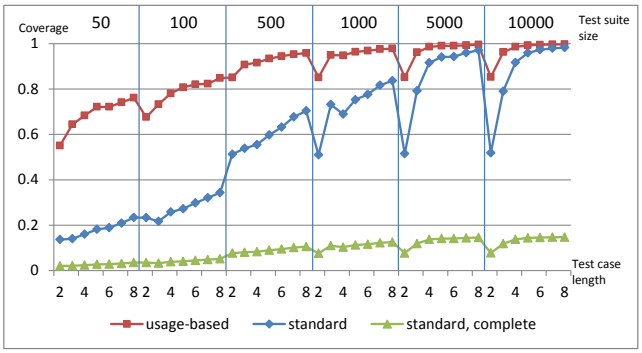
Figure A.60.: Results for the depth two for the random walk test case generation with the 4th-order MM.




(a) JFC data set.

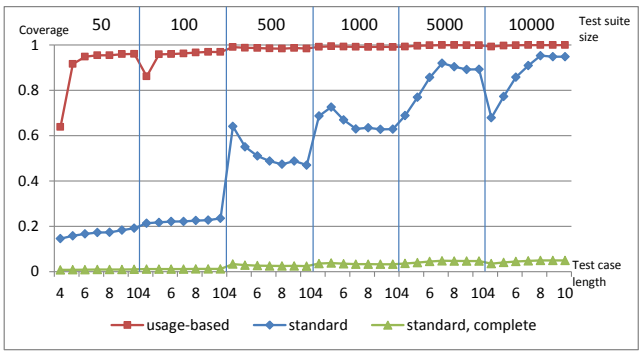



(b) MFC data set.

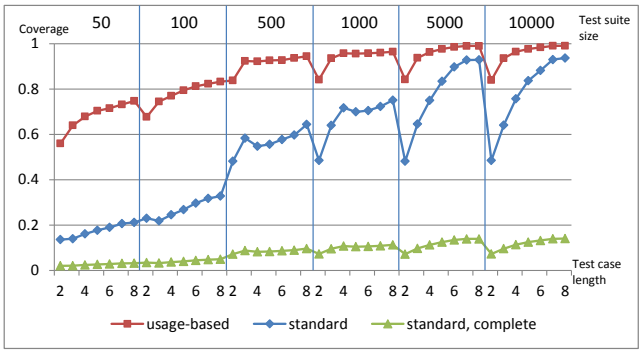



(c) Web application data set.

Figure A.61.: Results for the depth two for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 2$.

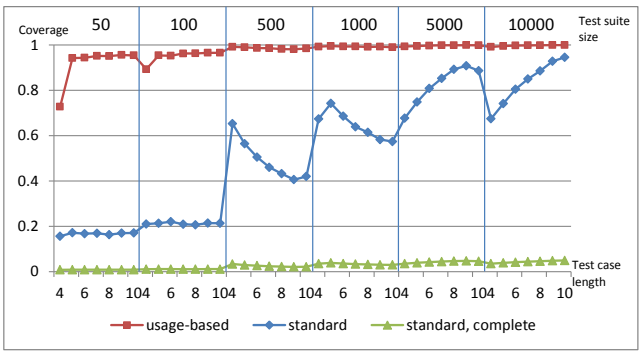



(a) JFC data set.

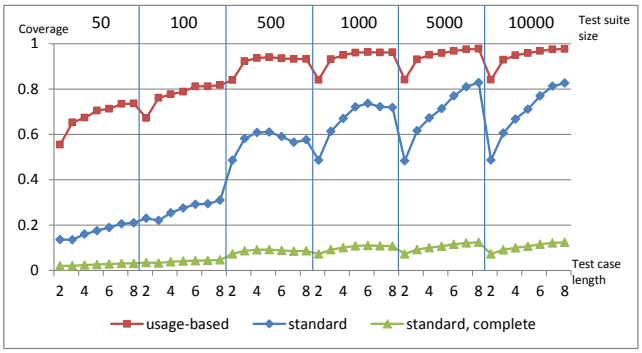



(b) Web application data set.

Figure A.62.: Results for the depth two for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 3$.

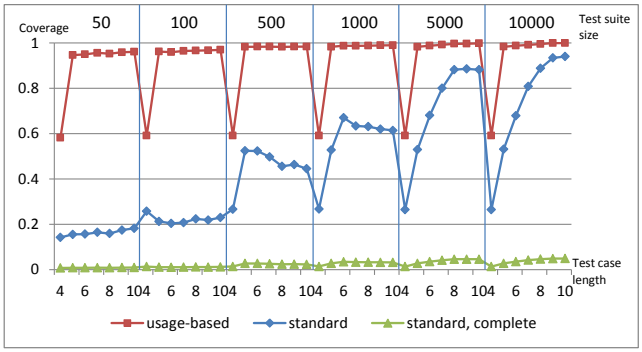



(a) JFC data set.

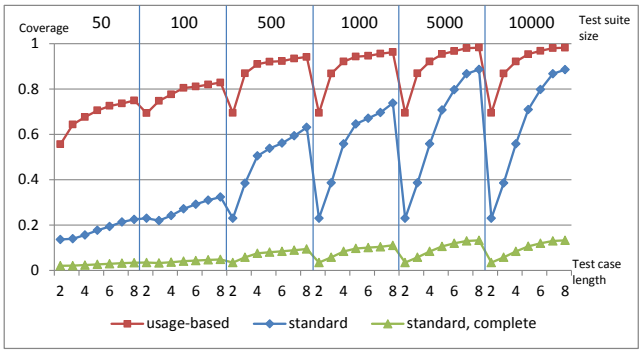



(b) Web application data set.

Figure A.63.: Results for the depth two for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 4$.

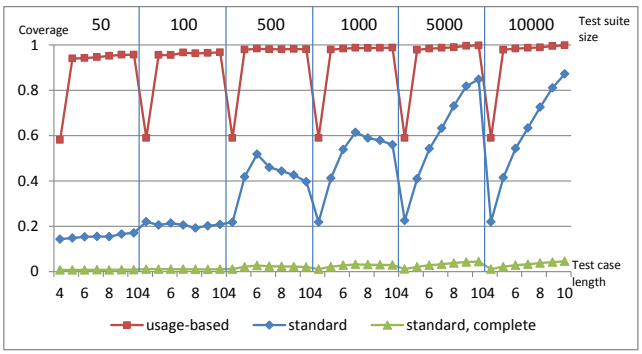



(a) JFC data set.

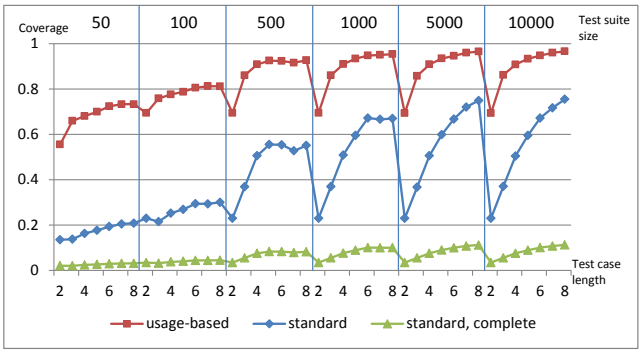



(b) Web application data set.

Figure A.64.: Results for the depth two for the random walk test case generation with the PPM model with $k_{min} = 2, k_{max} = 3$.

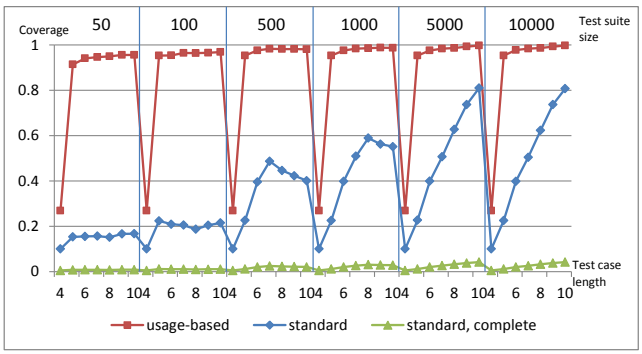



(a) JFC data set.

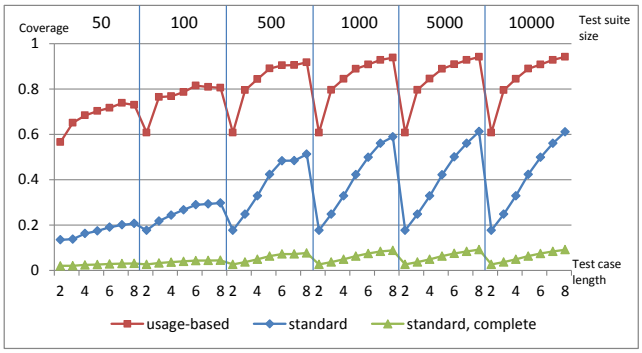



(b) Web application data set.

Figure A.65.: Results for the depth two for the random walk test case generation with the PPM model with $k_{min} = 2, k_{max} = 4$.



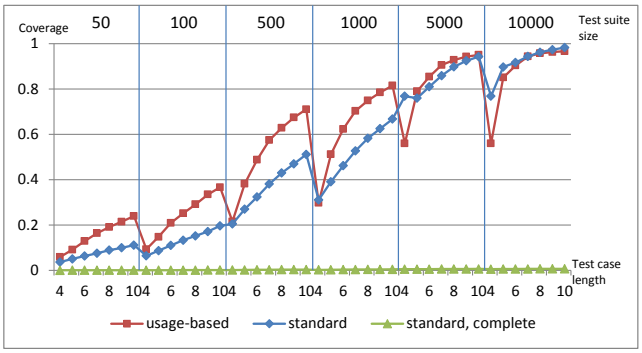

(a) JFC data set.



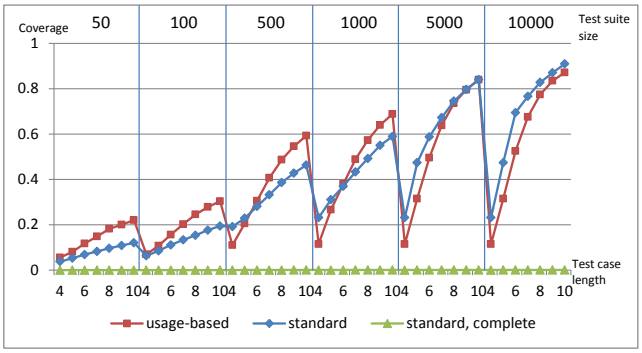

(b) Web application data set.

Figure A.66.: Results for the depth two for the random walk test case generation with the PPM model with $k_{min} = 3, k_{max} = 4$.

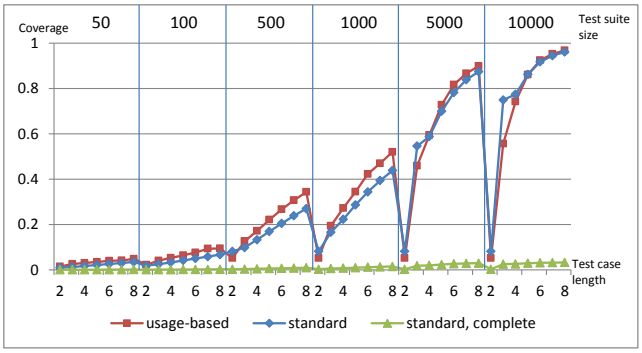
### A.4.7. Depth three for the random walk.




(a) JFC data set.

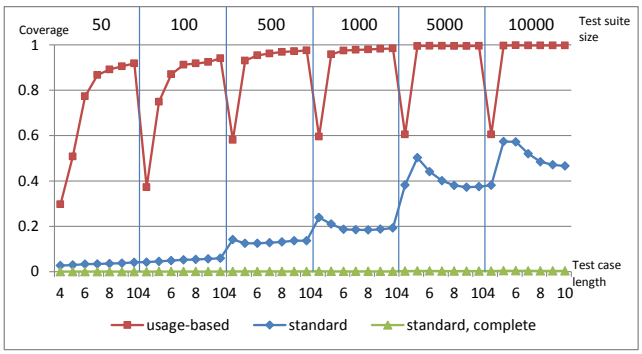



(b) MFC data set.

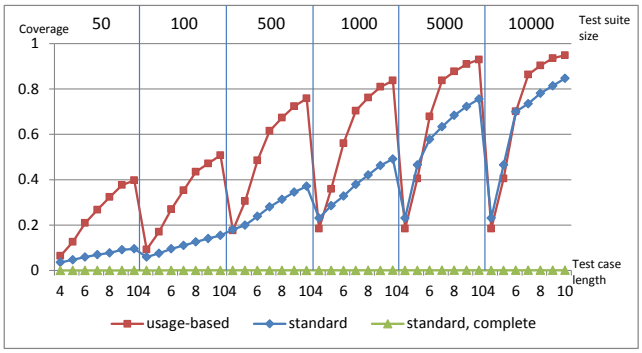



(c) Web application data set.

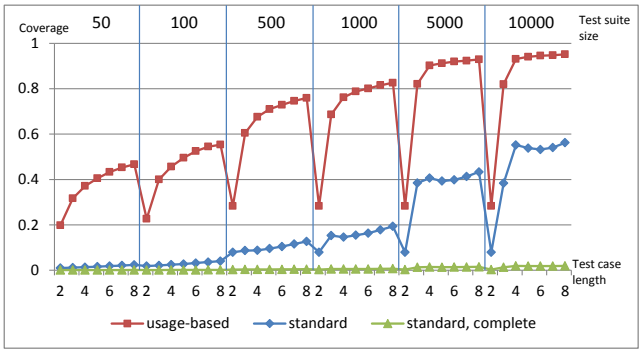
Figure A.67.: Results for the depth three for the random walk test case generation with the random EFG.




(a) JFC data set.

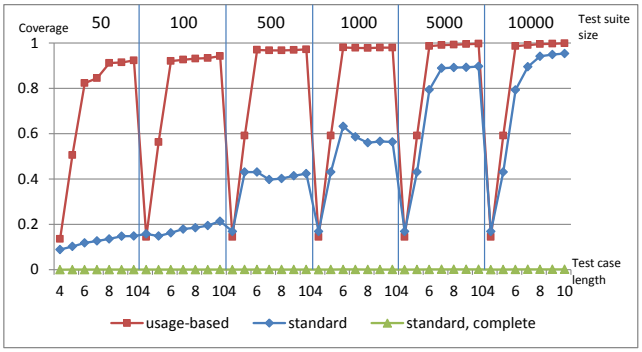



(b) MFC data set.

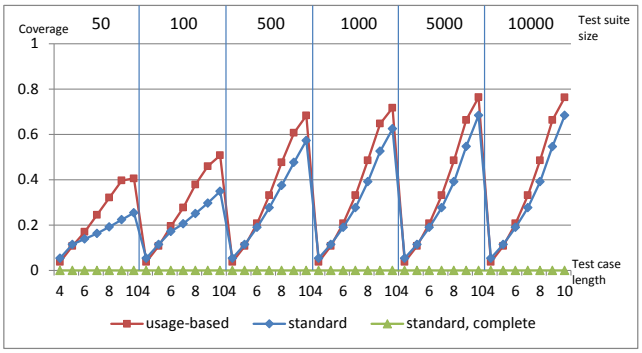



(c) Web application data set.

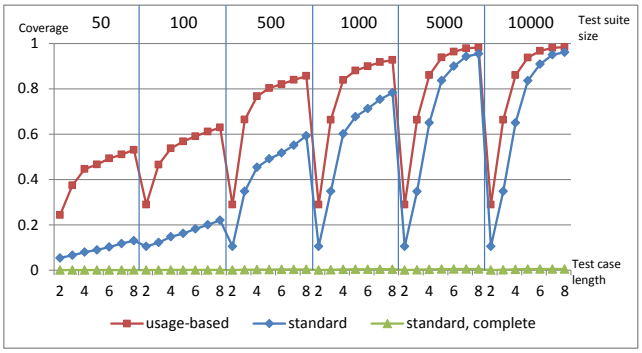
Figure A.68.: Results for the depth three for the random walk test case generation with the first-order MM.




(a) JFC data set.

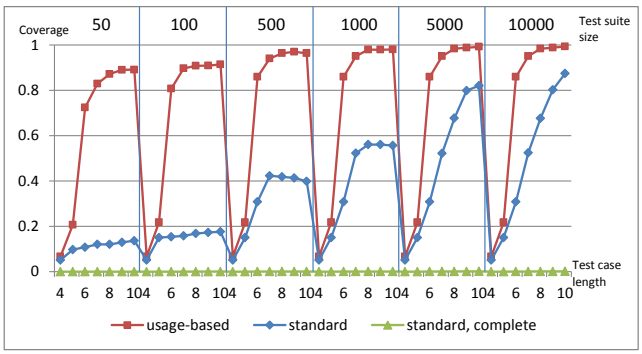



(b) MFC data set.

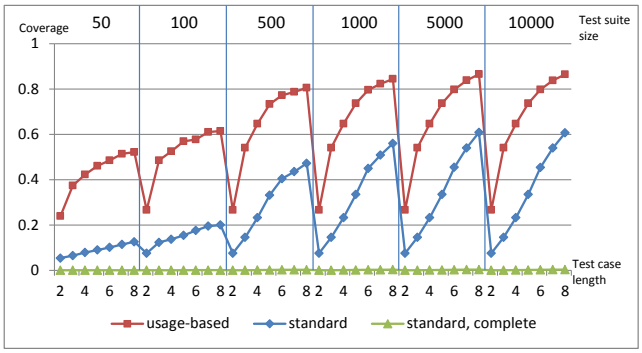



(c) Web application data set.

Figure A.69.: Results for the depth three for the random walk test case generation with the 2nd-order MM.

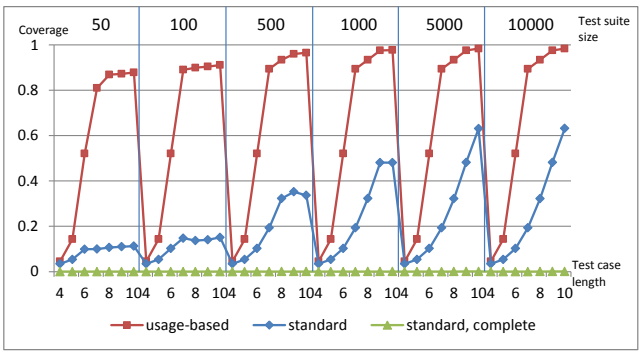



(a) JFC data set.

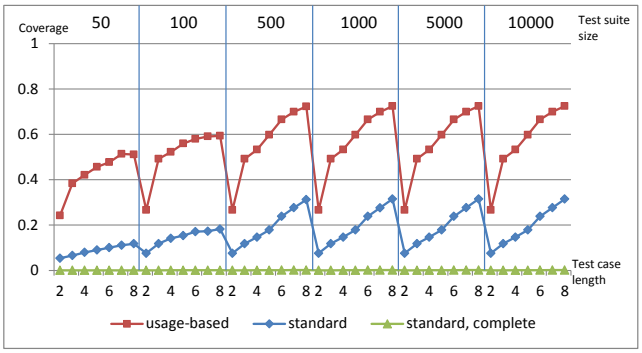



(b) Web application data set.

Figure A.70.: Results for the depth three for the random walk test case generation with the 3rd-order MM.

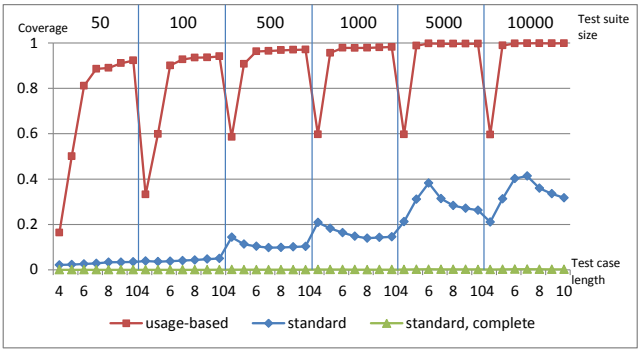



(a) JFC data set.

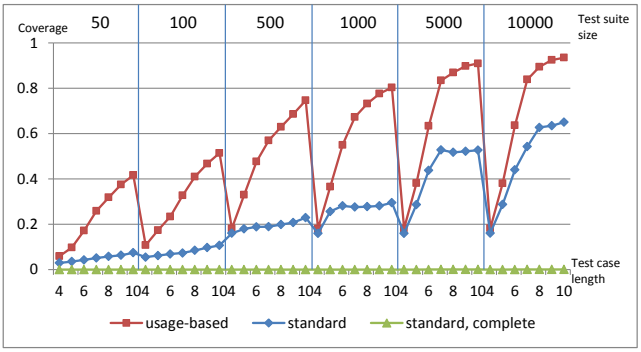



(b) Web application data set.

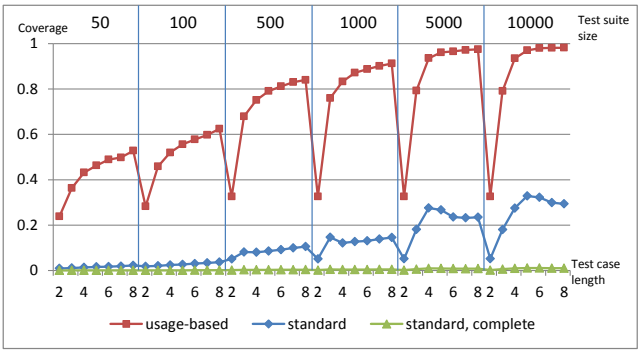
Figure A.71.: Results for the depth three for the random walk test case generation with the 4th-order MM.




(a) JFC data set.

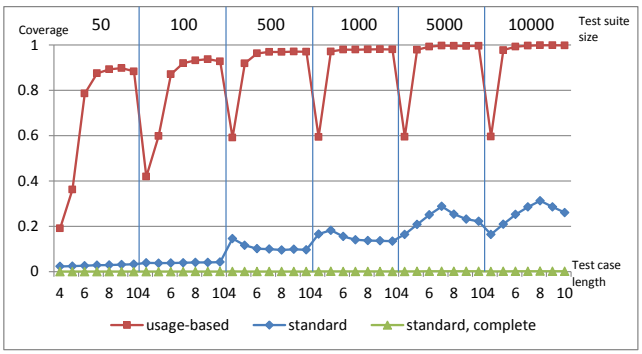



(b) MFC data set.

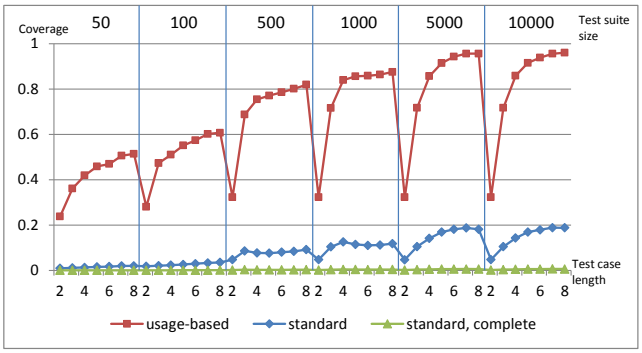



(c) Web application data set.

Figure A.72.: Results for the depth three for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 2$.

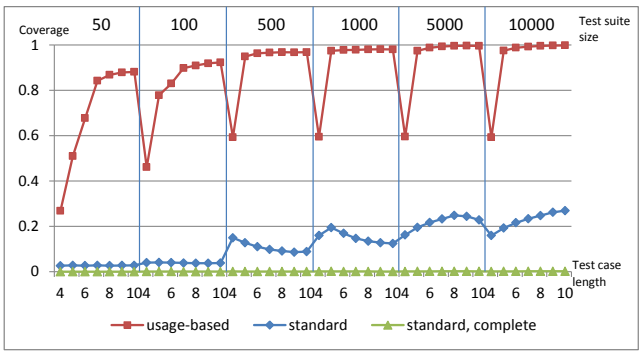



(a) JFC data set.

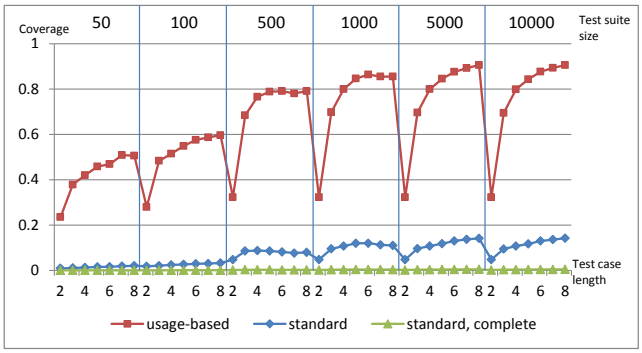



(b) Web application data set.

Figure A.73.: Results for the depth three for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 3$.

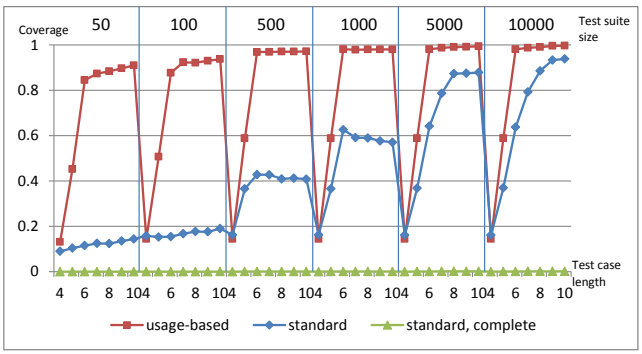



(a) JFC data set.

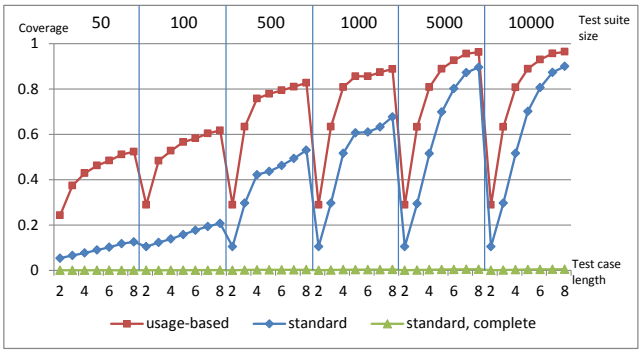



(b) Web application data set.

Figure A.74.: Results for the depth three for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 4$.

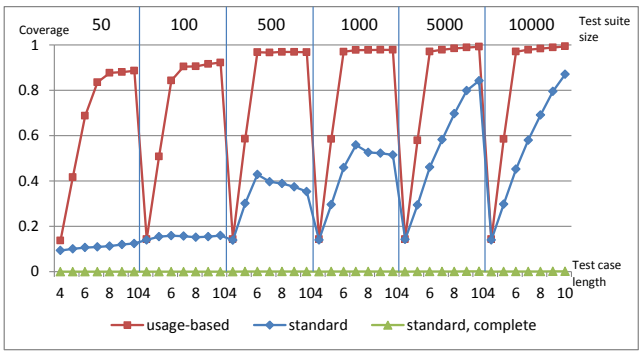



(a) JFC data set.

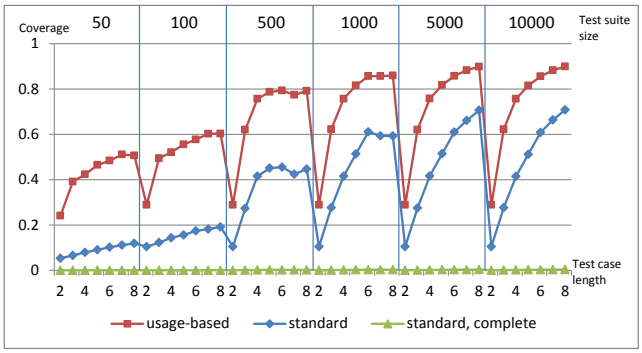



(b) Web application data set.

Figure A.75.: Results for the depth three for the random walk test case generation with the PPM model with $k_{min} = 2, k_{max} = 3$.

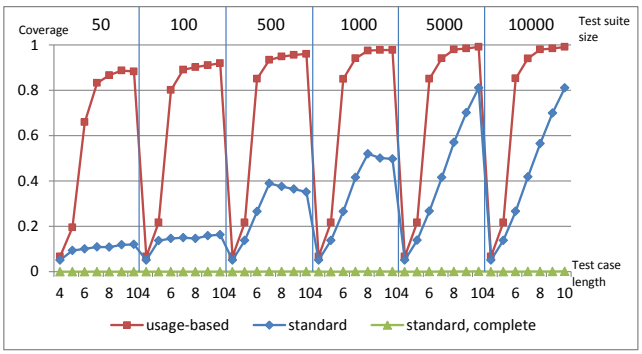



(a) JFC data set.

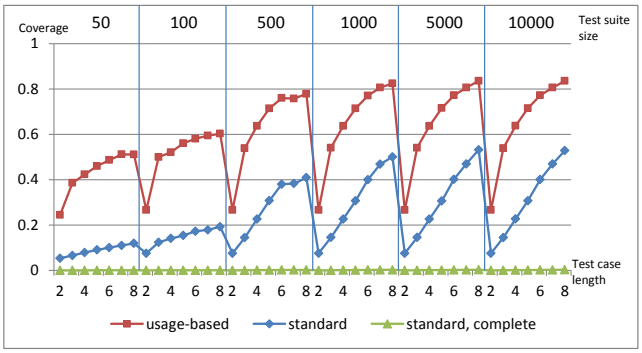



(b) Web application data set.

Figure A.76.: Results for the depth three for the random walk test case generation with the PPM model with $k_{min} = 2, k_{max} = 4$.



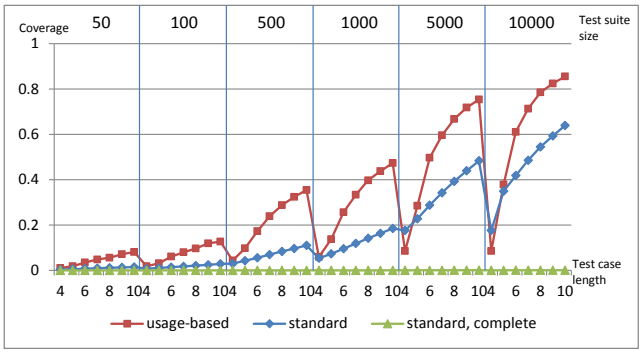

(a) JFC data set.



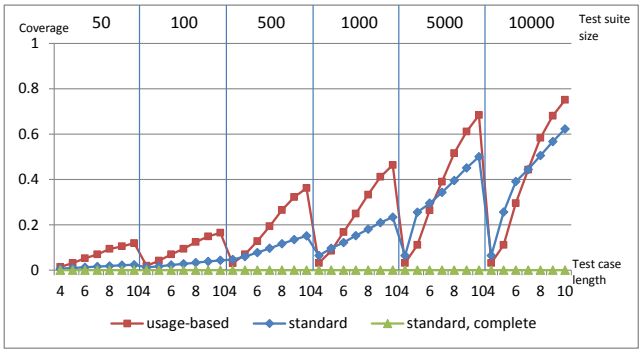

(b) Web application data set.

Figure A.77.: Results for the depth three for the random walk test case generation with the PPM model with $k_{min} = 3, k_{max} = 4$.

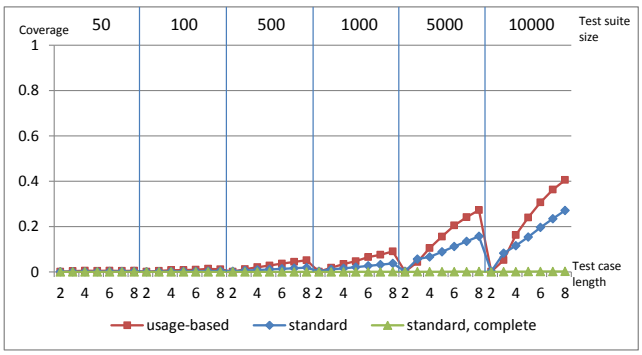
### A.4.8. Depth four for the random walk.




(a) JFC data set.

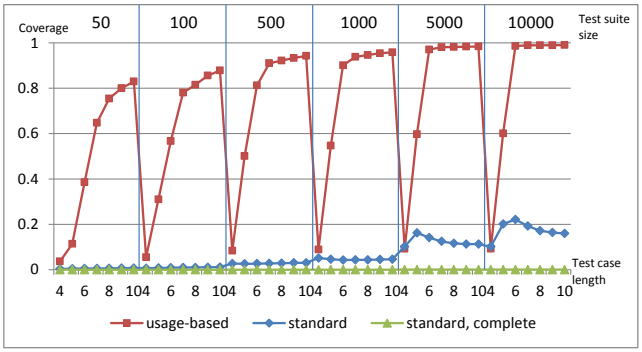



(b) MFC data set.

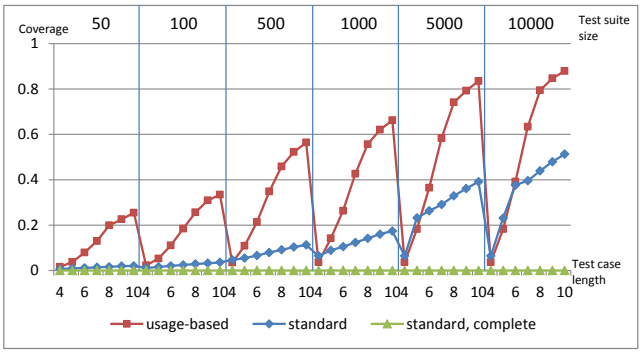



(c) Web application data set.

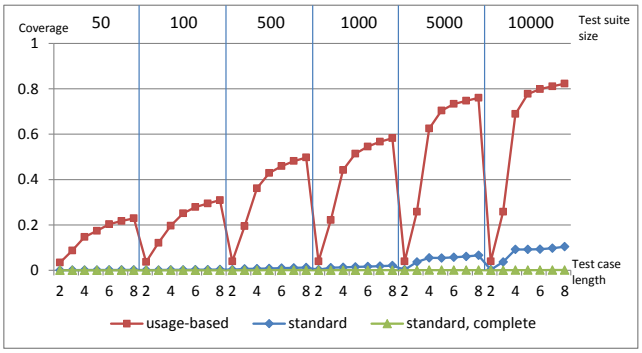
Figure A.78.: Results for the depth four for the random walk test case generation with the random EFG.




(a) JFC data set.

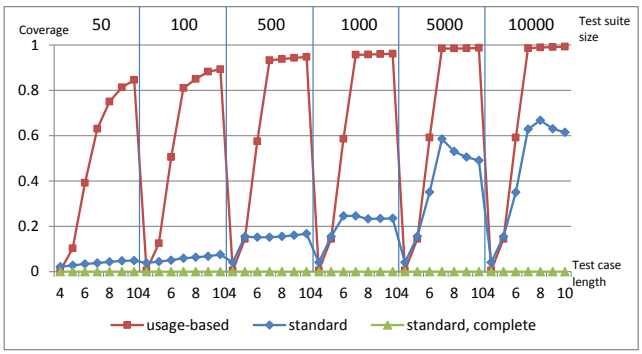



(b) MFC data set.

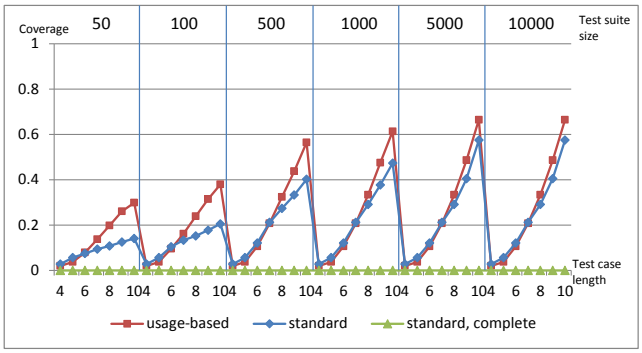



(c) Web application data set.

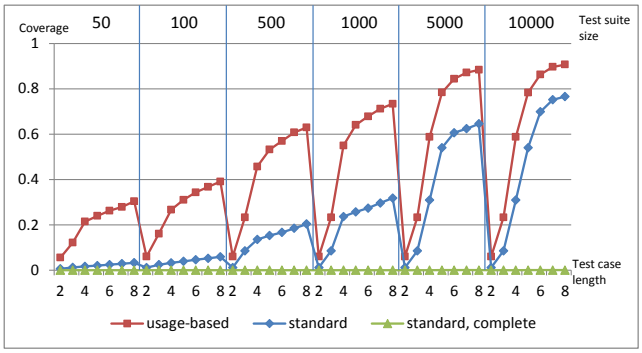
Figure A.79.: Results for the depth four for the random walk test case generation with the first-order MM.




(a) JFC data set.

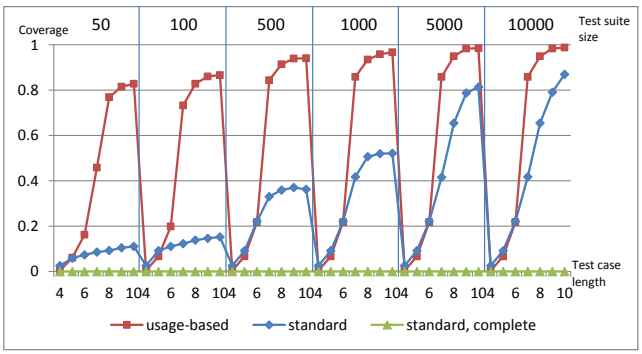



(b) MFC data set.

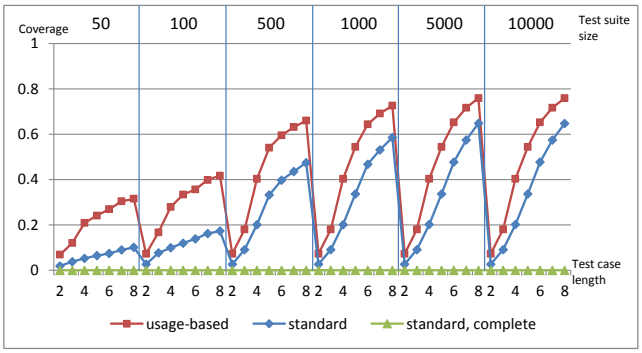



(c) Web application data set.

Figure A.80.: Results for the depth four for the random walk test case generation with the 2nd-order MM.

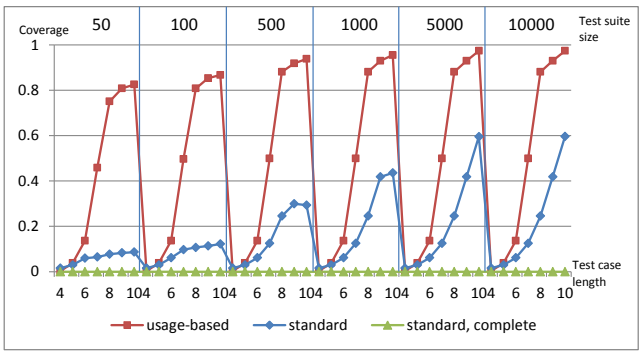



(a) JFC data set.

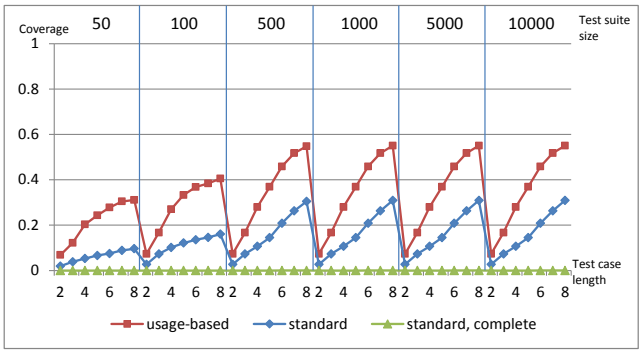



(b) Web application data set.

Figure A.81.: Results for the depth four for the random walk test case generation with the 3rd-order MM.

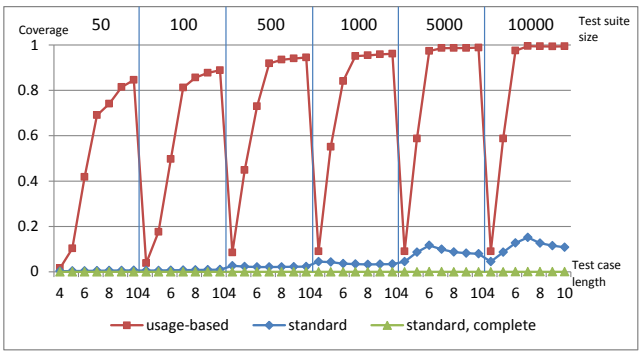



(a) JFC data set.

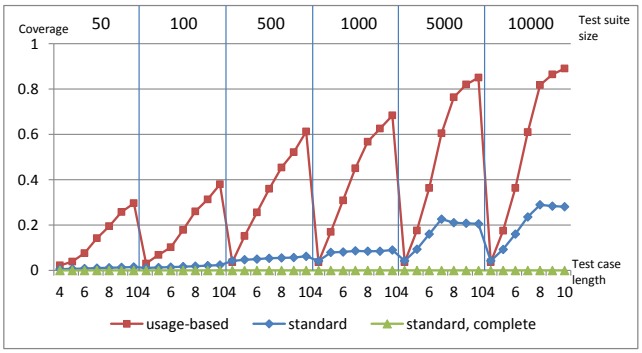



(b) Web application data set.

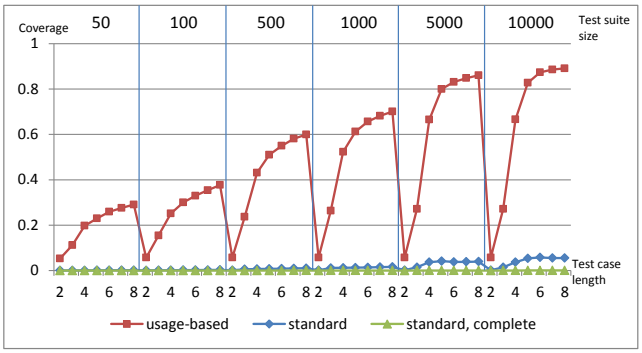
Figure A.82.: Results for the depth four for the random walk test case generation with the 4th-order MM.




(a) JFC data set.

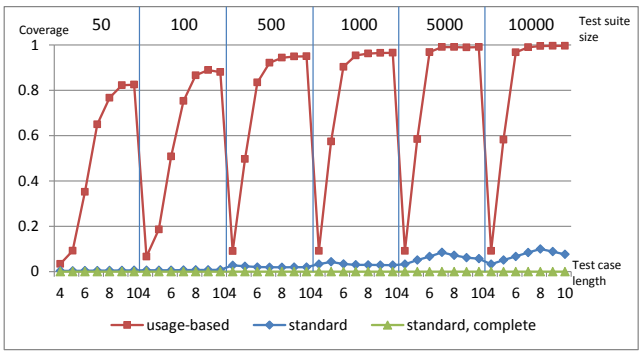



(b) MFC data set.

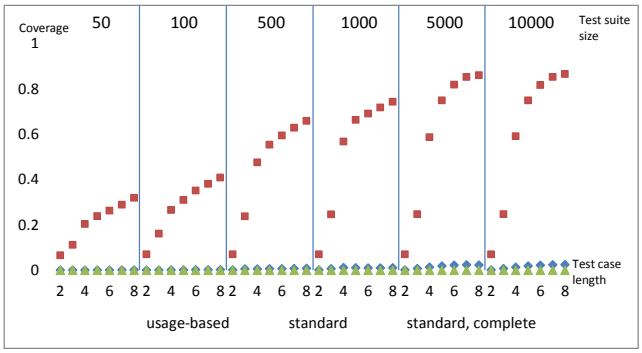



(c) Web application data set.

Figure A.83.: Results for the depth four for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 2$.

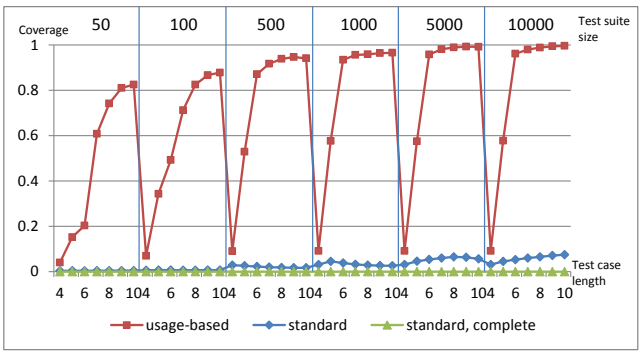



(a) JFC data set.

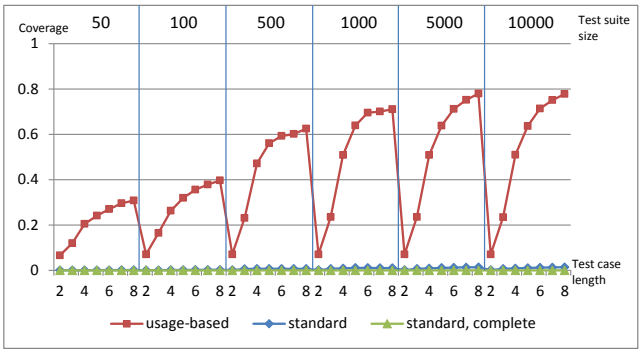



(b) Web application data set.

Figure A.84.: Results for the depth four for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 3$.

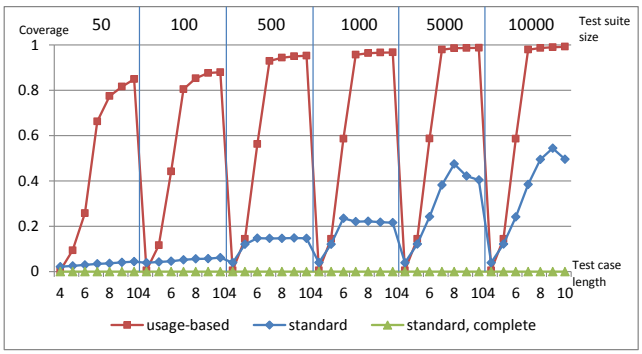



(a) JFC data set.

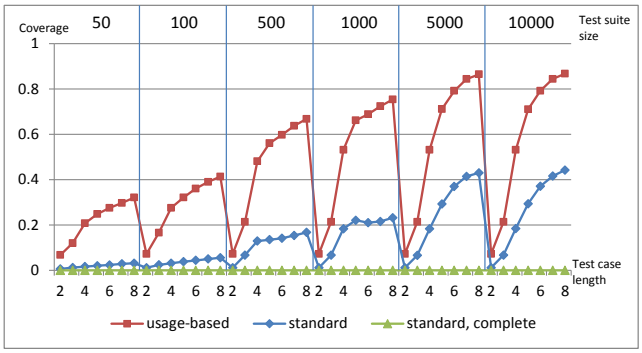



(b) Web application data set.

Figure A.85.: Results for the depth four for the random walk test case generation with the PPM model with $k_{min} = 1, k_{max} = 4$.

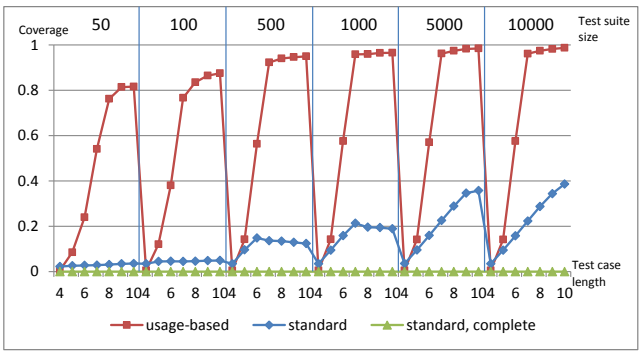



(a) JFC data set.

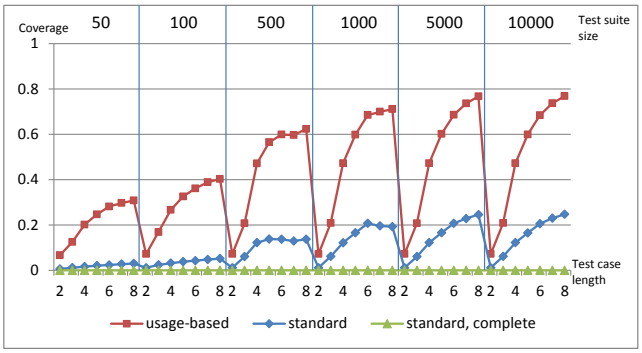



(b) Web application data set.

Figure A.86.: Results for the depth four for the random walk test case generation with the PPM model with $k_{min} = 2, k_{max} = 3$.

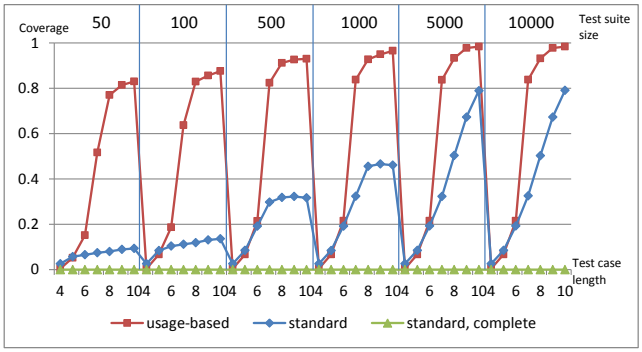



(a) JFC data set.

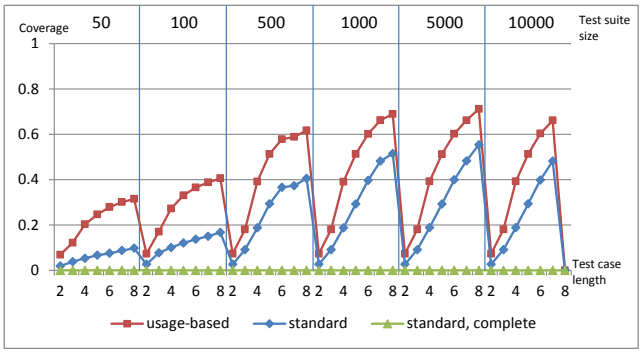



(b) Web application data set.

Figure A.87.: Results for the depth four for the random walk test case generation with the PPM model with $k_{min} = 2, k_{max} = 4$.




(a) JFC data set.




(b) Web application data set.

Figure A.88.: Results for the depth four for the random walk test case generation with the PPM model with $k_{min} = 3, k_{max} = 4$.

## A.5. Results of the heurisitic test case generation

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 2 | 7 | 81.7% | 11 | 91.5% | 15 | 95.5% | 25 | 98% |
| | 3 | 5 | 83.7% | 7 | 90.7% | 10 | 95.6% | 14 | 98% |
| | 4 | 4 | 85.7% | 5 | 96% | 8 | 96% | 11 | 98.2% |
| 2 | 2 | 109 | 80% | 440 | 90% | | 452 | 91.4% | |
| | 3 | 43 | 80% | 97 | 90% | 173 | 95% | 289 | 98% |
| | 4 | 30 | 80% | 65 | 90% | 110 | 95% | 166 | 98% |
| 3 | 2 | | | | 452 | 28.2% | | | |
| | 3 | 1,279 | 80% | | | 7,270 | 87.1% | | |
| | 4 | 414 | 80% | 1,120 | 90% | 2,835 | 95% | 6,416 | 96.2% |
| 4 | 2 | | | | 452 | 3.8% | | | |
| | 2 | | | | 7,270 | 26.7% | | | |
| | 4 | 14,322 | 80% | | | 116616 | 85.4% | | |

Table A.1.: Results of the heuristic test case generation with the first-order MM and the Web data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 2 | 7 | 80.6% | 13 | 91% | 19 | 95.2% | 31 | 97.2% |
| | 3 | 5 | 82.5% | 8 | 90% | 13 | 95% | 24 | 98% |
| | 4 | 4 | 83.8% | 7 | 92.2% | 10 | 95.6% | 16 | 98% |
| | 5 | 3 | 81.9% | 6 | 92.5% | 8 | 95.8% | 12 | 98.2% |
| | 6 | 3 | 85.5% | 5 | 92.8% | 7 | 95.8% | 11 | 98.2% |
| | 7 | 2 | 81.1% | 4 | 93% | 5 | 95% | 9 | 98.2% |
| 2 | 2 | | | | 97 | 69.4% | | | |
| | 3 | 50 | 80.1% | | | 158 | 87.8% | | |
| | 4 | 36 | 80% | 98 | 90% | | 231 | 93.5% | |
| | 5 | 27 | 80.1% | 63 | 90.1% | 144 | 95% | 267 | 96.9% |
| | 6 | 23 | 80.1% | 51 | 90.1% | 101 | 95% | 229 | 98% |
| | 7 | 21 | 80.6% | 44 | 90.1% | 81 | 95% | 148 | 98% |
| 3 | 2 | | | | 97 | 28.9% | | | |
| | 3 | | | | 360 | 66.3% | | | |
| | 4 | 282 | 80% | | | 553 | 86.3% | | |
| | 5 | 175 | 80% | 366 | 90% | | 564 | 94% | |
| | 6 | 130 | 80% | 254 | 90% | 381 | 95% | 484 | 96.8% |
| | 7 | 107 | 80% | 203 | 90% | 286 | 95% | 382 | 98% |
| 4 | 2 | | | | 97 | 5.9% | | | |
| | 3 | | | | 360 | 23.2% | | | |
| | 4 | | | | 1,652 | 59.4% | | | |
| | 5 | 2,185 | 80% | | | 2,944 | 80.4% | | |
| | 6 | 821 | 80% | | | 3,377 | 87.8% | | |
| | 7 | 576 | 80% | 1.571 | 90% | | 3,195 | 91.3% | |

Table A.2.: Results of the heuristic test case generation with the 2nd-order MM and the Web data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 2 | 8 | 81.8% | 19 | 90.3% | 24 | 90.3% | | |
| | 3 | 5 | 81.9% | 10 | 90.7% | 19 | 95.2% | 25 | 96.4% |
| | 4 | 3 | 80.1% | 7 | 90.5% | 13 | 95.2% | 22 | 97.2% |
| | 5 | 3 | 80.8% | 6 | 92.5% | 9 | 95.4% | 18 | 98.1% |
| | 6 | 3 | 83.5% | 5 | 90.8% | 9 | 95.0% | 16 | 98% |
| | 7 | 3 | 84.6% | 5 | 92.4% | 7 | 95.5% | 11 | 98.1% |
| | 8 | 2 | 81.1% | 4 | 92.2% | 6 | 95.4% | 10 | 98% |
| 2 | 2 | 70 | 60.7% | | | | | | |
| | 3 | 90 | 79.6% | | | | | | |
| | 4 | 46 | 80% | 109 | 84.5% | | | | |
| | 5 | 33 | 80.1% | 139 | 89.5% | | | | |
| | 6 | 29 | 80% | 100 | 90% | 163 | 91.6% | | |
| | 7 | 25 | 80% | 70 | 90% | 175 | 94.1% | | |
| | 8 | 22 | 80.1% | 57 | 90% | 162 | 95% | 176 | 95.1% |
| 3 | 2 | 70 | 26.6% | | | | | | |
| | 3 | 119 | 54% | | | | | | |
| | 4 | 181 | 64.6% | | | | | | |
| | 5 | 244 | 73.7% | | | | | | |
| | 6 | 297 | 79.8% | | | | | | |
| | 7 | 176 | 80% | 316 | 84% | | | | |
| | 8 | 138 | 80% | 302 | 86.6% | | | | |
| 4 | 2 | 70 | 7.3% | | | | | | |
| | 3 | 119 | 18% | | | | | | |
| | 4 | 233 | 40.3% | | | | | | |
| | 5 | 337 | 54.5% | | | | | | |
| | 6 | 398 | 65.6% | | | | | | |
| | 7 | 408 | 72.1% | | | | | | |
| | 8 | 389 | 76.2% | | | | | | |

Table A.3.: Results of the heuristic test case generation with the 3rd-order MM and the Web data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 2 | 8 | 81.8% | 19 | 90.1% | 24 | 90.3% | | |
| | 3 | 5 | 81.2% | 10 | 90.3% | 20 | 95.2% | 23 | 95.7% |
| | 4 | 4 | 82.6% | 8 | 90.2% | 15 | 95.1% | 21 | 96.9% |
| | 5 | 4 | 82.3% | 7 | 90.5% | 12 | 95.3% | 21 | 98% |
| | 6 | 3 | 80.8% | 6 | 90.7% | 11 | 95.1% | 22 | 97.5% |
| | 7 | 3 | 85.9% | 4 | 90.1% | 10 | 95.8% | 17 | 98% |
| | 8 | 2 | 81.4% | 4 | 91.1% | 8 | 95.4% | 15 | 98.1% |
| 2 | 2 | 70 | 60.7% | | | | | | |
| | 3 | 76 | 75.8% | | | | | | |
| | 4 | 74 | 77% | | | | | | |
| | 5 | 55 | 80% | | | 85 | 81.7% | | |
| | 6 | 36 | 80% | | | 92 | 86% | | |
| | 7 | 34 | 80% | | | 100 | 87.2% | | |
| | 8 | 30 | 80.1% | | | 105 | 88.4% | | |
| 3 | 2 | 70 | 26.6% | | | | | | |
| | 3 | 95 | 49.2% | | | | | | |
| | 4 | 102 | 53.3% | | | | | | |
| | 5 | 109 | 59.8% | | | | | | |
| | 6 | 141 | 66.5% | | | | | | |
| | 7 | 154 | 69.9% | | | | | | |
| | 8 | 164 | 72.4% | | | | | | |
| 4 | 2 | 70 | 7.3% | | | | | | |
| | 3 | 95 | 16.6% | | | | | | |
| | 4 | 105 | 27.9% | | | | | | |
| | 5 | 122 | 36.8% | | | | | | |
| | 6 | 157 | 45.8% | | | | | | |
| | 7 | 184 | 51.8% | | | | | | |
| | 8 | 206 | 55% | | | | | | |

Table A.4.: Results of the heuristic test case generation with the 4th-order MM and the Web data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 4 | 1 | 89.5% | 2 | 97.1% | | | 3 | 98.1% |
| | 5 | 1 | 92.6% | | | 2 | 97.4% | 3 | 98.6% |
| | 6 | | | 1 | 95.5% | | | 2 | 98% |
| 2 | 4 | 3 | 82% | 6 | 91.9% | 10 | 95.4% | 22 | 98% |
| | 5 | 2 | 82% | 4 | 91.2% | 7 | 95.4% | 15 | 98% |
| | 6 | 2 | 87.2% | 3 | 91.5% | 5 | 95% | 11 | 98% |
| 3 | 4 | | | | 2,040 | 60.7% | | | |
| | 5 | 7 | 81% | 14 | 90.1% | 32 | 95.1% | 96 | 98% |
| | 6 | 4 | 80% | 10 | 90.5% | 22 | 95.1% | 68 | 98% |
| 4 | 4 | | | | 4,027 | 9.2% | | | |
| | 5 | | | | 16,143 | 60.7% | | | |
| | 6 | 14 | 80.8% | 37 | 90% | 109 | 95% | 402 | 98% |

Table A.5.: Results of the heuristic test case generation with the first-order MM and the JFC data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 4 | 1 | 86% | 2 | 92.9% | 3 | 96.1% | 6 | 98% |
| | 5 | 1 | 86.8% | | | 2 | 97.1% | | |
| | 6 | | | | 1 | 92.9% | | | |
| | 7 | | | | 1 | 95.8% | | | |
| | 8 | | | | 1 | 97.1% | | | |
| | 9 | | | | 1 | 97.4% | | | |
| 2 | 4 | | | | 61 | 59% | | | |
| | 5 | 3 | 84.5% | 5 | 91.7% | 8 | 95% | 45 | 98% |
| | 6 | 2 | 82.3% | 4 | 91.7% | 6 | 95.3% | 24 | 98% |
| | 7 | 2 | 87.3% | 3 | 92.3% | 5 | 95.3% | 15 | 98% |
| | 8 | 1 | 81.7% | 3 | 93.9% | 4 | 95.6% | 10 | 98% |
| | 9 | 1 | 82.5% | 2 | 91.4% | 4 | 95.6% | 9 | 98% |
| 3 | 4 | | | | 88 | 14.4% | | | |
| | 5 | | | | 190 | 59.1% | | | |
| | 6 | 6 | 81.7% | 12 | 90.6% | 27 | 95.2% | 155 | 98% |
| | 7 | 4 | 80.2% | 9 | 90.9% | 19 | 95.2% | 88 | 98% |
| | 8 | 3 | 80.1% | 7 | 90.7% | 14 | 95.2% | 54 | 98% |
| | 9 | 3 | 83% | 6 | 91.2% | 12 | 95.2% | 37 | 98% |
| 4 | 4 | | | | 107 | 0.5% | | | |
| | 5 | | | | 392 | 14.4% | | | |
| | 6 | | | | 762 | 59.1% | | | |
| | 7 | 12 | 80.3% | 29 | 90.1% | 96 | 95% | 476 | 98% |
| | 8 | 9 | 81.5% | 20 | 90.1% | 65 | 95% | 277 | 98% |
| | 9 | 7 | 81.3% | 16 | 90.2% | 48 | 95% | 179 | 98% |

Table A.6.: Results of the heuristic test case generation with the 2nd-order MM and the JFC data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 4 | 2 | 89% | 3 | 93.3% | 4 | 97.2% | 8 | 97.6% |
| | 5 | 1 | 86% | 2 | 94.1% | 3 | 97.4% | 6 | 98.1% |
| | 6 | 1 | 89.8% | 2 | | 97.4% | | 4 | 98.1% |
| | 7 | 1 | | 90.4% | | 2 | 97.4% | 3 | 98.1% |
| | 8 | 1 | | 94.1% | | 2 | 97.4% | 3 | 98.6% |
| | 9 | 1 | | 95.8% | | | | 3 | 98.6% |
| | 10 | 1 | | 97.1% | | | | 2 | 98.1% |
| 2 | 4 | 13 | | 26.8% | | | | | |
| | 5 | 3 | 80.9% | 5 | 90.6% | 17 | 95% | 51 | 95.4% |
| | 6 | 3 | 88.1% | 4 | 92% | 7 | 95.2% | 50 | 98% |
| | 7 | 2 | 86.6% | 3 | 90.8% | 5 | 95.1% | 29 | 98% |
| | 8 | 1 | 81.7% | 3 | 92.6% | 5 | 95.6% | 17 | 98% |
| | 9 | 1 | 82.5% | 2 | 90.7% | 4 | 95.2% | 12 | 98.1% |
| | 10 | 1 | 84.4% | 2 | 91.8% | 3 | 95% | 10 | 98% |
| 3 | 4 | 18 | | 6.6% | | | | | |
| | 5 | 69 | | 21.7% | | | | | |
| | 6 | 11 | 80.8% | 117 | | 85.8% | | | |
| | 7 | 5 | 81% | 13 | 90.3% | 158 | 95% | 191 | 95.1% |
| | 8 | 4 | 84.6% | 7 | 90.8% | 18 | 95% | 120 | 98% |
| | 9 | 3 | 82.9% | 6 | 90.5% | 15 | 95% | 69 | 98% |
| | 10 | 3 | 84.6% | 5 | 90.9% | 11 | 95.2% | 43 | 98% |
| 4 | 4 | 21 | | 0.4% | | | | | |
| | 5 | 79 | | 6.6% | | | | | |
| | 6 | 156 | | 21.7% | | | | | |
| | 7 | 277 | 80.1% | 863 | | 85.9% | | | |
| | 8 | 10 | 81% | 36 | 90.1% | 343 | 95% | 387 | 95.1% |
| | 9 | 7 | 81.2% | 17 | 90.2% | 54 | 95% | 238 | 98% |
| | 10 | 5 | 80.8% | 13 | 90.1% | 40 | 95% | 152 | 98% |

Table A.7.: Results of the heuristic test case generation with the 3rd-order MM and the JFC data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 4 | 2 | 83.9% | | 3 | 95.8% | | 6 | 96.3% |
| | 5 | 1 | 81% | 2 | 82.8% | 3 | 95.8% | 7 | 96.3% |
| | 6 | 1 | 86% | 2 | 92.9% | 3 | 95.8% | 9 | 98% |
| | 7 | 1 | 89.8% | | 2 | 95.8% | | 6 | 98.1% |
| | 8 | | 1 | 92.9% | | 2 | 97.1% | 4 | 98% |
| | 9 | | 1 | 94.1% | | 2 | 97.6% | 3 | 98% |
| | 10 | | | 1 | 95.8% | | | 3 | 98.4% |
| 2 | 4 | | | | 8 | 23.8% | | | |
| | 5 | | | | 12 | 63.2% | | | |
| | 6 | 3 | 87.2% | 4 | 90.1% | | 27 | 93.3% | |
| | 7 | 2 | 85.3% | 4 | 91.7% | 7 | 95.2% | 43 | 96.9% |
| | 8 | 2 | 86.5% | 3 | 90.3% | 6 | 95.5% | 53 | 98% |
| | 9 | 1 | 81.8% | 3 | 92.1% | 6 | 95.4% | 22 | 98% |
| | 10 | 1 | 82.7% | 3 | 93.3% | 4 | 95.1% | 13 | 98.8% |
| 3 | 4 | | | | 9 | 4.5% | | | |
| | 5 | | | | 14 | 14.3% | | | |
| | 6 | | | | 35 | 52% | | | |
| | 7 | 6 | 82.1% | | | 67 | 89.4% | | |
| | 8 | 4 | 80.8% | 13 | 90.2% | | 107 | 93.4% | |
| | 9 | 3 | 82.8% | 7 | 90.7% | 20 | 95% | 161 | 97.6% |
| | 10 | 3 | 84% | 6 | 90.5% | 15 | 95.1% | 103 | 98% |
| 4 | 4 | | | | 9 | 0.3% | | | |
| | 5 | | | | 17 | 3.7% | | | |
| | 6 | | | | 38 | 13.6% | | | |
| | 7 | | | | 75 | 49.8% | | | |
| | 8 | 12 | 80.6% | | | 147 | 88.1% | | |
| | 9 | 8 | 80.9% | 40 | 90% | | 238 | 93% | |
| | 10 | 5 | 80.5% | 15 | 90.3% | 63 | 95% | 297 | 97.4% |

Table A.8.: Results of the heuristic test case generation with the 4th-order MM and the JFC data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

| Coverage depth | Test case length | Desired coverage | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 80% | | 90% | | 95% | | 98% | |
| 1 | 4 | | | | 47 | 71.6% | | | |
| | 5 | 19 | 80.5% | | | 52 | 81.2% | | |
| | 6 | 14 | 81.2% | 27 | 90% | | 49 | 93.3% | |
| | 7 | 11 | 80.8% | 19 | 90% | 32 | 95.2% | 50 | 97.2% |
| 2 | 4 | | | | 110 | 41.4% | | | |
| | 5 | | | | 149 | 71.3% | | | |
| | 6 | 72 | 80% | | | 143 | 88.5% | | |
| | 7 | 49 | 80.4% | 89 | 90.1% | | 124 | 93.2% | |
| 3 | 4 | | | | 432 | 18.6% | | | |
| | 5 | | | | 797 | 41% | | | |
| | 6 | | | | 1,233 | 71.2% | | | |
| | 7 | 245 | 80% | | | 1,222 | 88.5% | | |
| 4 | 4 | | | | 726 | 3.7% | | | |
| | 5 | | | | 2,753 | 18.5% | | | |
| | 6 | | | | 4,759 | 40.9% | | | |
| | 7 | | | | 7,254 | 71.1% | | | |

Table A.9.: Results of the heuristic test case generation with the 1st-order MM and the MFC data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.

| Coverage depth | Test case length | Desired coverage 80% | 90% | 95% | 98% |
|---|---|---|---|---|---|
| 1 | 4 | | 14 | 27.2% | |
| | 5 | | 23 | 42.6% | |
| | 6 | | 27 | 60.1% | |
| | 7 | | 32 | 77.7% | |
| | 8 | 23 80% | | 32 81.2% | |
| | 9 | 13 80.6% | | 38 88.3% | |
| | 10 | 12 81% | | 39 90% | |
| 2 | 4 | | 16 | 10.6% | |
| | 5 | | 35 | 20.9% | |
| | 6 | | 46 | 33.9% | |
| | 7 | | 57 | 49% | |
| | 8 | | 78 | 66.1% | |
| | 9 | | 97 | 76.8% | |
| | 10 | 64 80% | | 110 84.5% | |
| 3 | 4 | | 17 | 3.7% | |
| | 5 | | 38 | 10.7% | |
| | 6 | | 54 | 21% | |
| | 7 | | 70 | 33.3% | |
| | 8 | | 95 | 48.5% | |
| | 9 | | 130 | 66.6% | |
| | 10 | | 152 | 76.8% | |
| 4 | 4 | | 24 | 1.9% | |
| | 5 | | 43 | 3.7% | |
| | 6 | | 91 | 10.8% | |
| | 7 | | 151 | 20.8% | |
| | 8 | | 191 | 33.3% | |
| | 9 | | 258 | 48.8% | |
| | 10 | | 351 | 66.8% | |

Table A.10.: Results of the heuristic test case generation with the 2nd-order MM and the MFC data set. The columns of the table depict the number of test cases of the generated test suite and the coverage that is actually achieved.