Automatic Testing of Natural User Interfaces

Chris J. Hunt · Guy J. Brown · Gordon Fraser
University of Sheffield
Sheffield, United Kingdom
Email: mail@cjhunt.co.uk, {guy.brown,gordon.fraser}@sheffield.ac.uk

Abstract—Automated test generation can effectively explore programs through their programmer interfaces and traditional graphical user interfaces, but the recent advent of natural user interfaces (NUI) based on motion and gesture detection, for example the Microsoft Kinect, has outrun software testing research. This leaves a rapidly growing domain of software ranging from entertainment to medical applications without suitable test automation techniques. To address this issue, we propose a technique that automatically tests Kinect-based applications by synthesising realistic sequences of skeletal movement. The novel test cases are generated by a statistical model, which is trained on a corpus of common gestures. Evaluation on a gesture-controlled Kinect web browser application demonstrates that our approach achieves significantly higher code coverage than random test inputs.

Keywords—Test case generation; system testing; natural user interfaces; Kinect

I. INTRODUCTION

Automatic test generation is one of the key factors in improving software quality. On the lowest level, accessing a program’s Application Programmer’s Interface (API) is well understood, and there are many efficient unit test generation tools such as Microsoft’s Pex [22]. However, testing an application through its user interfaces is a crucially important complementary approach to testing [13], allowing one to determine a program’s robustness and whether user requirements are implemented correctly. Successful system test generation relies on specific knowledge of the application interface, for example to identify the widgets provided by a graphical user interface (GUI) and the interactions they offer, such that a test case is a sequence of interactions with these widgets [14]. If the widgets are not known, then the only remaining viable option is to simulate random mouse-movement and key presses, which allows basic robustness testing, but is not efficient at covering “deep” states of program behaviour [13].

The uprise of natural user interfaces (NUI) offers a significant new challenge for test generation, as interactions are no longer based on small sets of widgets with simplistic interactions. For example, the popular Microsoft Kinect input device delivers data from an RGB camera, an infrared laser and CMOS sensor to capture video data in 3D under ambient light conditions, and a multi-array microphone. Inputs from these sensors are interpreted by middleware systems providing motion capture, skeletal tracking, facial recognition and voice recognition. On this array of input possibilities, the fallback approach of test generation by providing random input data is doomed to fail — how likely is it, that random sensor data resembles human movement and gestures?

This leaves the question of how to automatically test programs based on NUIs unsolved. This is particularly worrisome considering the increasingly appearing usage scenarios of NUI programs in the medical domain, where the well-worn Therac-25 incident, which killed several patients in the mid-eighties [15], has ever since cast a shadow on software reliance and driven research on software testing. Incidentally, the fatal accidents with the Therac-25 were triggered by unexpected and untested user interactions, making the case for the necessity to test software through user interfaces.

To enable testing of NUI applications, this paper proposes a technique that synthesises test cases in terms of realistic motion data, as provided by the Kinect and similar 3D depth sensors. The motion data is represented in terms of skeletal positions, where a human body is represented as a number of joints, each given with its 3D coordinates. Each joint represents a body part (arms, shoulders, neck, head). Applications can then access this skeletal information to detect gestures that are used for interaction between the user and a program. In order to increase the chances of reaching “deep” program states, as is desirable in order to reveal faults, one aim is to provide realistic movement that incorporates common gestures used for user interaction (e.g., holding still, waving, wiping, rotating, etc.) To derive these sequences, common gestures are extracted from sample traces of human interaction, and new sequences of skeletal positions are synthesised from these gestures. In detail, the contributions of this paper are:

- An approach to train a Markov chain model with common gestures extracted from recorded sequences of skeletal positions.
- A technique to automatically synthesise realistic new sequences of skeletal movement based on common gestures, and feed them back to Kinect applications to serve as test inputs.
- An evaluation of the effects of the approach and several of its parameters on code coverage on a gesture-controlled web browser application.

Our experiments on the gesture-controlled web browser reveal an increase in achieved code coverage of 19% over a baseline of random test inputs. This is a promising result, yet our experiments only represent initial explorations in the field and lead to identification of several avenues of further research, which are ultimately required to prepare software testing research for the next generation of user interfaces.
II. BACKGROUND

A. Software Testing

In order to find errors in programs, software testing is one of the most frequently applied techniques. A test case represents an input to a system, and the reaction of the system to this input is checked for correctness — if the output is not as expected, then an error has been detected.

The simplest possible technique to produce test cases for a system is to choose inputs randomly. Although random inputs typically do not represent readable or understandable test cases, they represent a cheap and effective method to exercise automated oracles in terms of partial specifications or simply program crashes (e.g., [13]). A well-known application of random testing is fuzz-testing [8], which aims to reveal security issues in programs. Random testing is also applied to test graphical user interfaces in terms of simulated random mouse clicks and key presses, whereas more advanced approaches take advantage of knowledge of the underlying widget set of a user interface [14]. This allows more efficient exploration of an application and has a higher probability of reaching difficult, “deeper” program states.

Beyond simple random techniques, systematic approaches usually exploit external knowledge to explore a wider range of behaviour. For example, given a specification, a systematic approach might aim to cover all parts of that specification. Systematic approaches to test through user interfaces typically assume the existence of a model that represents the possible GUI interactions, or try to learn such a model on-the-fly while testing [13], [17]. Given such a model of the user interface, a systematic approach might aim to cover all the behaviours represented by this model.

Models of common usage have been applied in automated test generation in order to simulate realistic usage. For example, Fraser and Zeller [9] derived Markov chains representing the common temporal ordering of method calls for APIs, and the models are trained with data mined from source code repositories and existing test suites. Whittaker [24] proposed the use of Markov chain models to represent common user behaviour in GUI testing, and the idea of testing based on “usage profiles” has also been explored using different models, e.g., probabilistic event-flow-graphs [5]. In general, the application of usage models in test generation has been shown to lead to test cases that are more realistic and achieve higher coverage and fault detection. Furthermore, statistical testing based on a usage model ensures that the faults that are found are those most relevant from the point of view of the user [23].

With the increasing popularity of mobile phones and their touch-screen based interfaces, software testing research has also started considering this type of applications. Current approaches again mostly rely on known sets of GUI widgets (e.g., [11]). Recent work on record&replay techniques also started including information about advanced multi-touch gestures such as swiping or zoom-and-pinch, and additional sensory input available on most modern smartphones, e.g., data from accelerometer, compass, or proximity sensors [12]. However, such inputs cannot yet be synthesized for automated test generation yet.

B. Natural User Interfaces

Users typically interact with computer programs using established human-machine interfaces such as keyboard and computer mouse. These are artificial devices that users need to learn to control. In contrast, natural user interfaces are those that do not require the use of such artificial devices. Examples are touch screens, speech recognition, or gesture recognition. It is sometimes argued that natural user interfaces (NUI) are the next evolutionary stage in human-machine interfaces, as were graphical user interfaces (GUI) from command-line interfaces (CLI).

Natural user interfaces have been popularised significantly through the use in the computer gaming industry. Probably the most famous example is the Microsoft Kinect, which allows players to control games using their full body. The Kinect has an array of input sensors including an RGB camera, an infrared laser and CMOS sensor to capture video data in 3D under ambient light conditions, and a multi-array microphone. These data are interpreted in terms of skeletal tracking, facial recognition, and voice recognition, and players have to perform gestures and movement to control games. However, the popularity of the Kinect has far exceeded the confines of the gaming industry. NUIs are particularly useful in scenarios where interaction using traditional interfaces such as a mouse or keyboard are not possible. For example, surgeons in a sterile environment must not touch any devices, yet need to interact with computers (e.g., [11]). Further medical applications are in therapy, where NUIs can be used to guide and supervise patients [7], but new application areas are also continuously explored outside medical applications (for example, domestic application in kitchen environments [19]). The popularity of the Kinect in these application areas has led to a number of competing devices on the market today, for example the PrimeSense device[3] ASUS Xtion[3] or the LeapMotion[3] and companies are now explicitly targeting NUI for a range of different application areas.

From a software testing perspective, NUIs are fundamentally different from traditional user interfaces. Keyboard inputs for CLI applications can be produced in a straightforward way. Keyboard and mouse events for GUIs can also be produced easily, although the challenge lies in doing so in a way that results in complex interactions. This is typically achieved by exploiting domain knowledge about a widget-set applied in the application under test, such that testing can directly interact with the components of the GUI, rather than the lower level human-machine interface. However, the many different sensors used on NUI devices makes synthesising raw sensory data for NUI devices is a significant challenge. While randomly sending keyboard and mouse events still has the potential to explore interesting program behaviour (“monkey testing”), there is little hope that random sensory data will produce anything that resembles user interaction and can serve as useful test input.

One potential avenue to address this problem would be to replace the gesture, speech, etc. recognition components with custom testing components that deliver gesture inputs to

2http://www.asus.com/Multimedia/Xtion_PRO, accessed August 2013
3https://www.leapmotion.com, accessed August 2013
programs directly. Unfortunately, this is not easily possible as there is no standardised gesture recognition technique — for example, on the Kinect every application decides on its own what to do with skeletal position data. Therefore, a feasible solution is to synthesise this skeletal position data, and that is what this paper attempts.

C. Motion Synthesis

Synthesis of human motion is of importance in computer graphics research, with its main applications in animation. A typical approach is to learn statistical models for behaviour primitives from captured and classified motion sequences, and then to synthesise new motion sequences based on stochastic exploration of these models [10]. The challenges of such approaches lie in making the synthesised movement look realistic [21], and also physically valid. Furthermore, there is the challenge of joining individual behaviour primitives to result in smooth movement. This can, for example, be achieved by identifying the closest poses at the end of one segment and the beginning of the next, and then smoothly interpolating between them [21].

III. Generating Skeletal Positions as Test Inputs

In this paper we consider the scenario of applications controlled by body motion captured by the Microsoft Kinect or similar devices. The user interacts with an application by performing sequences of different gestures, and the Kinect driver processes raw sensory data to provide the application with the skeletal position data. From the point of view of the application, inputs are therefore not raw sensory data, but such skeletal positions, and the problem of generating test inputs consists of generating sequences of skeletal positions.

There is no standardised set of gestures with which to control an application — each application can define and detect its own gestures. Nevertheless it can be expected that the number of reasonable gestures in practice will be limited, and to increase usability applications will over time converge to a common set of standard gestures. Our approach therefore consists of learning a set of such standard gestures, and then to synthesise new sequences of skeletal positions based on the learned information.

Figure 1 represents our approach at a high level. There are two distinct phases: First, a model of common gestures is learned from recorded data of real human motion. The training phase starts by obtaining a corpus of recorded sequences of skeletal positions. The Kinect makes it easy to record such sequences, and principal component analysis (PCA) can be applied to reduce the data rate and redundancy. Data points related in terms of their gestures are identified using clustering, and temporal information is learned by training a Markov chain model. In the second phase, this Markov model is used to synthesise sequences of skeletal positions. The quantisation performed by the PCA in the training phase needs to be reversed, resulting in skeletal position data.

In the simplest case, this skeletal position data can be provided to the application under test directly, resulting in a random test generation approach. However, in principle the sequences of skeletal data are amenable to more advanced test generation approaches, e.g., search-based testing could be used to learn from interactions with the application which gestures are used by an application at hand, and which sequences of these gestures should be applied, for example to maximize code coverage.

IV. Synthesising Kinect Motion Sequences

After describing the general framework in the previous section, this section now describes how we instantiated it for skeletal motion sequences for the Kinect. Following the general framework, the motion sequence synthesis consists of two parts: First, a corpus of motion sequences (encoded as skeleton positions obtained from the Kinect) were recorded and preprocessed to reduce the dimensionality and remove unreliable data. This corpus was then used to train Markov chains, which captured the temporal behaviour in the motion data. Finally, novel motion sequences were generated from the Markov chains and presented to the application under test.
Fig. 2. Three types of test data. Top: test data generated by choosing skeleton joint locations randomly in 3D space. Middle: skeleton motion generated from cluster centers, after preprocessing and k-means clustering of the data. Transitions between each cluster of the skeleton data are chosen randomly. Bottom: skeleton motion generated by a Markov chain of order 1. The Markov chain captures the temporal dynamics of the motion data in the training corpus, and hence the synthetic test data resembles the smooth sequences of skeleton motion that would be captured directly from the Kinect.

A. Preprocessing

Training data was captured using the OpenNI\footnote{http://www.openni.org, accessed August 2013} and NiTE\footnote{http://www.primesense.com/solutions/nite-middleware, accessed August 2013} software to interface with the Kinect, which delivered a vector of skeleton data consisting of 15 joints at regular intervals (approximately at 30 frames per second). Each joint $k$ in the skeleton is described by a 4-tuple $(x_k, y_k, z_k, \theta_k)$, in which the first three elements give the location of the joint in 3D space and $\theta_k$ represents the confidence associated with the joint location. In the first instance, any joints with a zero confidence were marked as unreliable, in order to identify skeletons that were not correctly tracked (e.g., due to the subject moving into and out of the field of view of the Kinect camera at the start and end of recording). Skeleton vectors with one or more unreliable joints were removed from the corpus, amounting to approximately 35.14% of the recorded data. Subsequently, the confidence values $\theta_k$ were dropped from the data so that each skeleton vector contained $J=45$ elements, corresponding to the $(x_k, y_k, z_k)$ coordinates of each of its 15 joints.

The skeleton data was then subjected to principal components analysis (PCA) in order to reduce the data rate, and remove redundancy in the data whilst preserving significant variations. Given a data matrix $X$ consisting of $N$ skeleton vectors, PCA re-expresses $X$ as a linear combination of a set of basis vectors $P$

$$Y = PX$$

where each row of $P$ is an eigenvector of the sample covariance matrix of the data, $C = XX^T$, and $Y$ is the transformed data. Dimensionality reduction is achieved by retaining the $M$ eigenvectors in $P$ that have the largest eigenvalues, such that $M < J$. Hence, the $N$ skeleton vectors consisting of $J$ elements were approximated by the same number of vectors in a smaller $M$-dimensional space. We chose $M=17$, which considerably reduced the dimensionality of $X$ which still retaining 99.05% of the variance in the skeleton data.

B. Statistical modelling

Following \cite{21}, a statistical model of the skeleton data was obtained by building Markov chains from clustered data. Clustering was achieved by using the $k$-means algorithm to perform a vector quantisation of the skeleton data \cite{16}. More specifically, the $N$ skeleton vectors in our data set were partitioned into $k$ clusters, such that each vector was allocated to the cluster with the nearest mean.

Markov chains were then built to capture the temporal behaviour of the skeleton data. Consider a finite set of states $S$ in which each state corresponds to one of the clusters found by $k$-means clustering above. A Markov chain of order $n$ is a sequence of random variables $\{X_0, X_1, X_2, \ldots\}$, such that each random variable $X_i$ has a value corresponding to one of the states in $S$. The Markov chain then models the temporal behaviour of the clustered data, under the constraint that the probability of a state depends only on the probability of the previous $n$ states, i.e.
An exemplar sequence of skeleton motion generated by the vectors by transitioning to other states. An approximation of the $i$ vector benefit in applying the described approach. The transition probabilities between states were estimated by frequency counting on the clustered training corpus. For example, consider the case of the Markov chain of order $n = 1$. For two quantised skeleton vectors $i$ and $j$, we compute the number of times that $i$ and $j$ occur consecutively in the training corpus, divided by the total number of times that quantised vector $i$ occurs in the corpus. This gives an estimate of the transition probability between states $i$ and $j$ in the Markov chain, i.e. $P(x_i|x_j)$. Similarly, in the case of the Markov chain of order $n = 2$, frequency counts of three consecutive vectors are used to derive the transition probabilities.

C. Synthesis of novel motion sequences

During testing, novel motion sequences are generated from the Markov chain. Starting from an initial random state, the model generates a temporal sequence of quantised skeleton vectors by transitioning to other states. An approximation of the $(x, y, z)$ joint positions for the skeleton is then reconstructed from the PCA coefficients of each vector, giving a moving skeleton that closely resembles the kind of data that would be captured directly from the Kinect. These novel, synthesised motion sequences are presented to the application under test. An exemplar sequence of skeleton motion generated by the system is shown in the bottom row of Fig. 2.

V. Evaluation

In order to study the presented approach, we performed a series of experiments on an example NUI application, a gesture-controlled web browser. In detail, we aimed to investigate the following aspects:

- Experiment 1: How does the order of the Markov chain influence the efficacy of the generated test data?
- Experiment 2: How does the length of the generated sequences influence the efficacy of the generated test data?
- Experiment 3: How does the training data influence the efficacy of the generated test data?
- Experiment 4: How does the synthesised motion data compare to random data?

The first three experiments aim to investigate the properties of our technique in order to make an informed choice about parameter settings for which we cannot rely on established default values from the literature, whereas the fourth experiment serves as a sanity check to validate that there is a significant benefit in applying the described approach.

 experimentation.

$$P(X_i = x_i|X_{i-1} = x_{i-1}, X_{i-2} = x_{i-2}, \ldots, X_0 = x_0) =$$

$$P(X_i = x_i|X_{i-1} = x_{n-1}, \ldots, X_{1-n} = x_{i-n})$$

Fig. 3. Figure showing the screen layout of the browser application. GUI elements were kept to an absolute minimum as they would not be usable through gesture control.

A. Case Study Application

We constructed a sample gesture based application: a gesture controlled web browser built with Eclipse SWT and Mozilla XULRunner. A web browser was judged a suitable sample application due to the wide range of tasks that can be performed while not being dependent on options within a nested menu, whilst also being complex enough to make measuring code coverage worthwhile. OpenNI NITE (see Section IV-A) were selected as the software libraries to interact with the Kinect due to their cross-platform implementations, and for the same reason, the application was implemented in Java. There are two important aspects to this application: The actual web browser behaviour, and the gesture recognition to control the browser.

1) Browser: Initially The Lobo Project was considered as a web browser base to extend to incorporate gestures into, being an open source Java project with full support for user written extensions. However, it is now little used and has not been updated since 2009, while also not supporting HTML5 and facing incompatibilities with Java 7, leading to many websites suffering from poor rendering. Instead, Eclipse SWT’s browser widget\(^6\) and Mozilla XULRunner\(^7\) have been used in conjunction to create a simple web browser with the rendering power of Mozilla Firefox, while also being able to test the code coverage of both the browser widget and of the used components of Firefox itself.

A number of different actions can be performed through gestures in the browser (see Table I), either through calling specific functions of the XULRunner and SWT APIs, or by emulating key presses.

Although a web browser has been chosen as the example program of this paper, there is no reason why other types of application could not be used instead. As the browser merely acts as a listener for gesture events before performing appropriate actions, this could easily be incorporated elsewhere.

\(^6\)http://lobobrowser.org, accessed August 2013
\(^7\)http://www.eclipse.org/swt, accessed August 2013
\(^8\)https://developer.mozilla.org/en/docs/XULRunner
2) Gesture Recognition: In order to facilitate the use of the gesture controlled web browser, a suitable gesture recognition system had to be used within the application. While NiTE supports a small repertoire of gestures it can recognise (click, wave and hand raise), these are primarily used as focus gestures, that is, to be used as a method of detecting a user’s hand location in order to facilitate hand tracking. As the gesture based web browser has significantly more than 3 actions the user can perform, a gesture recognition system capable of recognising more than 3 gestures had to be sought.

HandGKET (Hand Gesture Key Emulation Toolkit) was considered as a possible solution, supporting a wide array of single and double handed gestures, and emulating a specified key press upon detection of a gesture. However, the project is not open source and must be run separately in conjunction with the desired target application. As an integrated gesture recognition system, and the ability to define custom gestures, is desired, HandGKET was ultimately deemed unsuitable.

As no suitable extendible Java based gesture recognition system could be found, it was deemed necessary to create a gesture recognition system that covered the required gestures, as well as being resilient to change in user position and speed of gesture. To achieve this, Dynamic Time Warping (DTW) has been utilised successfully as a basis for gesture recognition in a number recent papers[10], with the chief advantage of allowing for variation in the length of gestures, while preserving the precision of movement throughout. DTW ‘warp’ two sequences which may vary in speed along the time dimension before calculating a measure of similarity. In the case of gesture recognition this is highly valuable as it allows for a gesture to be matched to its accompanying template regardless of the speed it was performed.

Kinect SDK Dynamic Time Warping (DTW) Gesture Recognition (KinectDTW)[9] is an open source C# and Microsoft Kinect SDK implementation of DTW, with an easily extendible list of recognisable gestures while allowing users to record their own gesture templates. The position of joints across the arms are recorded across 32 frames before being normalised, in order to create a template for a gesture, before matching a live stream of these joint locations to the appropriate gesture. As such, it was deemed a suitable basis for this paper’s gesture recognition system, and was ported into Java and OpenNI in order to easily facilitate integration with the other elements of the project.

Although KinectDTW provides a good implementation of the DTW algorithm, it equally weights all tracked joints regardless of the amount of movement they undertake within the gesture, making the positions of non-moving joints as equally important as those performing the gesture. The technique forwarded by Celebi et al. [6] calculates the displacement of joint movement throughout the gesture, weighting the active joints highly, whilst penalising stationary or little moving joints. As a more resilient gesture recognition system is desirable, the technique forward by Celebi et al. was implemented to extend the KinectDTW approach, albeit with single gesture templates rather than a series.

The system has been further extended through the ability to record and play back sequences of user movement in order to permit testing of the system’s performance.

Through the porting of KinectDTW and its extension with the techniques of Celebi et al., a fully extendible gesture recognition system with easily defined gestures, while still being resilient to user position, has been created, fully fitting the required criteria.

B. Experimental Setup

In order to test the effectiveness of the test data generated by the motion synthesis system, a number of experiments were undertaken to vary key parameters in the algorithm. Our initial experiments showed that the system was relative insensitive to the number of clusters that were specified for the $k$-means clustering algorithm, so long as it was set sufficiently large. We chose 100 clusters, which gave a reasonable compromise between quantisation error on the one hand, and the size of the transition probability matrix in the Markov chain on the other. In the following, the number of clusters was fixed and other parameters were varied to evaluate their effect on code coverage of the application under test.

Code coverage was measured using the JaCoCo[11] code coverage library. JaCoCo provides a Java agent that applies bytecode instrumentation to all Java code to collect code coverage information. Coverage values in this paper therefore represent branch coverage values measured at the Java bytecode level.

Across all of our experiments, unless specified otherwise, we used a standardised configuration of the application: the use of order $n = 2$ Markov chains, 2,000 frames per sequence, and 100 clusters used in grouping training data for sequence generation.

When analysed results we followed the guidelines in[4] and used the Wilcoxon-Mann-Whitney U-test for statistical significance, and the Vargha-Delaney effect size to compare test generation approaches.

C. Training Data Acquisition

Training data was acquired through a small application that saved all skeletal data captured by the the Kinect as XML

---

9https://sites.google.com/site/kinecatapps/handgket, accessed August 2013
11http://www.eclemma.org

<table>
<thead>
<tr>
<th>Action</th>
<th>Gesture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back</td>
<td>Left hand swipe left</td>
</tr>
<tr>
<td>Forward</td>
<td>Left hand swipe right</td>
</tr>
<tr>
<td>Navigate left</td>
<td>Right hand swipe left</td>
</tr>
<tr>
<td>Navigate right</td>
<td>Right hand swipe right</td>
</tr>
<tr>
<td>Click</td>
<td>Push right hand forward</td>
</tr>
<tr>
<td>Refresh</td>
<td>Wave right hand</td>
</tr>
<tr>
<td>Stop</td>
<td>Raise both hands</td>
</tr>
<tr>
<td>Zoom in</td>
<td>Bring both arms together</td>
</tr>
<tr>
<td>Zoom out</td>
<td>Move both arms apart</td>
</tr>
</tbody>
</table>

TABLE I. Supported gesture controlled actions of the web browser application
The first order Markov chain captures experiments with more training data. This will need to be verified by further the corpus to accurately estimate the transition probabilities that there might simply not have been enough training data in between first and second order Markov chains is so small is a possible conjecture for why the difference in the results for a 1st order chain and second order Markov chains is 71.5%. The difference is minimal, and a Wilcoxon-Mann-Whitney U-test confirms that the difference of a state depends only on the probability of the previous n states. Intuitively, we would expect that the additional temporal context present in higher order chains result in higher code coverage in the system under test. For this experiment we considered the values n = 1 and n = 2, and generated 100 sequences for each of the two different Markov chain orders. Each generated sequences had a fixed length of 2,000 frames. Table I shows that the average coverage of first order Markov chains is 71.38%, and the average coverage of second order Markov chains is 71.5%. The difference is minimal, and a Wilcoxon-Mann-Whitney U-test confirms that the difference is not statistically significant (p = 0.22, Vargha-Delaney $\hat{A}_{12}$ effect size representing the probability of second order chains being better than first order chains is $\hat{A}_{12} = 0.5434$).

The results of both, first and second order Markov chains, show higher coverage than random sequences (cf. Experiment 4), which suggests that the first order Markov chain captures short-term temporal variation in the motion training data. However, adding more ‘memory’ in the system so that there is more temporal context does not seem to help, at least in terms of the current configuration of the system and when measured in terms of code coverage.

Note that with 100 clusters, the number of transition probabilities that have to be learned from the data is $100^2$ for a 1st order chain and $100^3$ for a 2nd order chain. Therefore, a possible conjecture for why the difference in the results between first and second order Markov chains is so small is that there might simply not have been enough training data in the corpus to accurately estimate the transition probabilities in the longer chain. This will need to be verified by further experiments with more training data.

The first author of this paper interacted with the application, and tried to produce different sequences that all include the majority of the gestures of the application as well as gestures used in different applications. This resulted in a total of 40 sequences. Each sequence spans 158-1,755 frames, and covers a subject performing a range of gestures, which may or may not be an explicitly defined gesture for our application. This led to most sequences covering the majority of the gestures recognised, with some covering all.

These steps were taken to try and reflect the varied usage of a real world user of applications, and to try and maintain a random generation of gestures from a neutral stance.

**D. Experiment 1: Markov Chain Length**

The aim of the first experiment is to determine the influence of the order of the Markov chains used to represent the gesture information. In a Markov chain of order n the temporal behaviour is encoded under the constraint that the probability of a state depends only on the probability of the previous n states. Intuitively, we would expect that the additional temporal context present in higher order chains result in higher code coverage in the system under test. For this experiment we considered the values n = 1 and n = 2, and generated 100 sequences for each of the two different Markov chain orders. Each generated sequences had a fixed length of 2,000 frames.

Table I shows that the average coverage of first order Markov chains is 71.38%, and the average coverage of second order Markov chains is 71.5%. The difference is minimal, and a Wilcoxon-Mann-Whitney U-test confirms that the difference is not statistically significant (p = 0.22, Vargha-Delaney $\hat{A}_{12}$ effect size representing the probability of second order chains being better than first order chains is $\hat{A}_{12} = 0.5434$).

The results of both, first and second order Markov chains, show higher coverage than random sequences (cf. Experiment 4), which suggests that the first order Markov chain captures short-term temporal variation in the motion training data. However, adding more ‘memory’ in the system so that there is more temporal context does not seem to help, at least in terms of the current configuration of the system and when measured in terms of code coverage.

Note that with 100 clusters, the number of transition probabilities that have to be learned from the data is $100^2$ for a 1st order chain and $100^3$ for a 2nd order chain. Therefore, a possible conjecture for why the difference in the results between first and second order Markov chains is so small is that there might simply not have been enough training data in the corpus to accurately estimate the transition probabilities in the longer chain. This will need to be verified by further experiments with more training data.

The aim of the first experiment is to determine the influence of the order of the Markov chains used to represent the gesture information. In a Markov chain of order n the temporal behaviour is encoded under the constraint that the probability of a state depends only on the probability of the previous n states. Intuitively, we would expect that the additional temporal context present in higher order chains result in higher code coverage in the system under test. For this experiment we considered the values n = 1 and n = 2, and generated 100 sequences for each of the two different Markov chain orders. Each generated sequences had a fixed length of 2,000 frames.

Table I shows that the average coverage of first order Markov chains is 71.38%, and the average coverage of second order Markov chains is 71.5%. The difference is minimal, and a Wilcoxon-Mann-Whitney U-test confirms that the difference is not statistically significant (p = 0.22, Vargha-Delaney $\hat{A}_{12}$ effect size representing the probability of second order chains being better than first order chains is $\hat{A}_{12} = 0.5434$).

The results of both, first and second order Markov chains, show higher coverage than random sequences (cf. Experiment 4), which suggests that the first order Markov chain captures short-term temporal variation in the motion training data. However, adding more ‘memory’ in the system so that there is more temporal context does not seem to help, at least in terms of the current configuration of the system and when measured in terms of code coverage.

Note that with 100 clusters, the number of transition probabilities that have to be learned from the data is $100^2$ for a 1st order chain and $100^3$ for a 2nd order chain. Therefore, a possible conjecture for why the difference in the results between first and second order Markov chains is so small is that there might simply not have been enough training data in the corpus to accurately estimate the transition probabilities in the longer chain. This will need to be verified by further experiments with more training data.

The role of the length of test sequences is a thoroughly explored topic in the literature on test generation, and in general longer sequences imply higher code coverage and fault detection ability. In our context, one would expect a short synthesised sequence to only cover a small subsection of the full set of gestures, a longer sequence should detect more of the set, while also reaching more complex branches of the system code.

In order to test this theory we synthesised 100 sequences of 10,000 frames in length, again using 100 clusters and order n = 2 Markov chains, before splitting these into a series of sub-sequences that each get progressively longer in increments of 500 frames. This gives 20 sub-sequences per each 10,000 length sequence, in a range of 500-10,000 frames.

Figure 4 confirms that longer sequences are generally better. The Pearson correlation between sequence length and coverage is strong (0.50). There is a larger increase in coverage in the lower length regions, whereas after that the increase in coverage generally is relatively small. This can be attributed to various reasons, including the quality of the training data, the gesture recogniser, and the general difficulty of the few remaining, uncovered branches in the browser application.

In contrast, increasing the length of sequences of random coordinates has no effect on the coverage (Pearson correlation of -0.0008); this demonstrates how unlikely it is that random coordinates represent valid input. For random frames taken from the training data there is an increase of coverage observable as length increases. However, the coverage is lower, and even though at first look the correlation might look stronger than for the synthesised sequences, the correlation between coverage and length is actually only moderate (Pearson correlation of 0.39); this is likely influenced by the stronger variance (i.e., standard deviation is 0.79 for synthesised sequences, but 2.18 for random frames).

<table>
<thead>
<tr>
<th>Markov chain order</th>
<th>Code coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>71.38</td>
</tr>
<tr>
<td>2</td>
<td>71.50</td>
</tr>
</tbody>
</table>

**TABLE II. Code coverage resulting from use of different order Markov chains**

![Figure 4. Line graph showing the results of the sequence length experiment, plotting means and standard errors.](image)
F. Experiment 3: Influence of Training Data

As indicated in the discussion of the previous experiment, the training data has an influence on the result of the quality of the synthesised motion sequences: The training data used by the gesture synthesis system determines the states that can be selected by the system, and is therefore also responsible for the probability of transitioning to such states.

As such, the amount of training samples used by the system should be seen to have an important effect on the sequences produced, as with too few samples training sequences may just be played back in their entirety in any generated sequences.

In this experiment we generated eight groups of 100 sequences each. For each group we used an increasing number of training samples, starting with 5 sequences, and ranging up to 40 sequences. Each generated sequence was based on a different random sample of the training data. Based on the previous experiment, we used a sequence length of 2,000 frames for the synthesises sequences, and we again used 100 clusters and \( n = 2 \) Markov chains.

Figure 5 summarises the results for this experiment. There is a slight increase of coverage with increasing number of training samples; however, the correlation is very weak (Pearson correlation of 0.11). Using only 5 sequences of training data achieves the lowest coverage, whereas anything from 20 sequences and higher achieves significantly higher coverage. However, between 5 and 20 the plot is quite erratic, with 10 sequences showing significantly higher coverage than with 5 or 15 sequences. Consequently, although the number of sequences has an influence on the resulting coverage, it seems that it is not the dominating factor. In particular, the coverage will strongly depend on the suitability of the training data for the application at hand. For example, if a gesture that is relevant for the application does not happen to be in the training set, then it is unlikely to occur in the synthesised sequence. Similarly, if the training set consists of many irrelevant gestures, then the likelihood of synthesising the important gestures reduces, thus affecting coverage.

G. Experiment 4: Comparison to Random Testing

In order to demonstrate the viability of this paper’s approach compared to a random testing approach, we compared our motion synthesis method to two baseline systems. Firstly, the (x,y,z) skeleton joint locations were selected at random from the valid co-ordinate space accepted by OpenNI. This gave truly random test data (shown in the top panel of Fig. 2) in which each frame of data rarely resembled a human skeleton. In the second baseline condition, each frame was generated by randomly choosing a cluster of skeleton data at each epoch (shown in the middle panel of Fig. 2). Since this approach amounts to randomly jumping between states in the Markov chain (and therefore takes no account of transition probabilities between states), each frame in this test data is a plausible skeleton, but the sequence is not a smooth motion.

As with the other experiments, sequence length was set at 2,000 frames, the number of clusters at 100 and with order \( n = 2 \) Markov chains used for sequences generation.

Figure 6 summarises the achieved branch coverage. The synthesised sequences based on the Markov model exhibit a significantly higher coverage than random skeletal position data (\( p < 0.01, \hat{A}_{12} = 0.99 \)). This is as expected; the only possibly surprising aspect is that the increase in coverage seems to be comparatively small (62.9% for random positions vs 71.5% for synthesised sequences). This is due in part to the nature of the browser application: First, a large number of the branches in the program are trivial to cover. In particular, achieving more than 71.5% coverage seems to be quite difficult.

As a reference point for the achievable coverage in our example application, we measured the coverage achieved when executing the entire training data set of recorded sequences (20,104 frames); this leads to 79.2% branch coverage. Considering that this is a comprehensive corpus of gestures, this demonstrates that around 20% of the branches in the application are very difficult to cover. Besides the usual infeasible branches, to some extent this is due to the implementation of the gesture recogniser, which has difficulty in identifying certain categories
of gestures. In particular, in this case it seems that some gestures were present in the training data, but not commonly detected in the synthesised data. This is an interesting observation, as it means that from a testing point of view NUI applications differ from other types of software: In NUI applications the difficulty of achieving high coverage depends not only on the efficacy of the test generation approach, but also on the quality of the implemented software (i.e., the gesture recogniser).

The highest value observed in all our experiments with synthesised sequences is 73.3%. On one hand, this shows that the achieved coverage of synthesised sequences comes close to that of actual user interactions; on the other hand it shows that further improvement would be possible.

Comparing synthesised sequences to random frames selected from the training data, the increase is again statistically significant ($p < 0.01, \hat{A}_{12} = 0.97$). This result demonstrates the importance of temporal information, rather than just providing valid skeletal data.

### H. Threats to Validity

Threats to **internal validity** might come from how the empirical study was carried out. To reduce the probability of having faults in our framework, it has been carefully tested; however, it is well known that testing alone cannot prove the absence of defects. Furthermore, randomised algorithms are affected by chance. To cope with this problem, we repeated each experiment several times (as indicated in the experiment descriptions, and we followed statistical procedures to evaluate their results.

Our example application uses a custom-developed gesture recognition system which may not be as precise as more advanced implementations. The efficacy of the gesture recogniser may affect our results, in the sense that non-gestures may falsely be recognised as real gestures, and real, properly synthesised gestures may not be recognised by the system. However, we expect that the better the gesture recognition system, the better our approach will work compared to random inputs.

There are threats to **external validity** regarding the generalisation to other types of software and natural user interfaces, which is common for any empirical analysis. Our experiment considers only one type of NUI, and uses only a single application, therefore we cannot claim that our results will generalise to any NUI-based software. However, we see no reason why the principle of training on recorded human interactions and synthesising new gestures does not work on any type of NUI, and we will extend our application areas and empirical studies to other domains in the future.

Threats to **construct validity** are on how we measured performance of our approach. We used code coverage as a proxy measurement for the quality of test data, which is common practice. However, our example application only has a comparatively small number of branches, of which many are easy to cover, and some are very difficult to cover. This means that small improvements in the quality of motion sequences may not be reflected in an increase in code coverage.

A further potential threat to validity was demonstrated by the results to our first experiment on the order of Markov chains: As we did not observe an increase in coverage for higher order chains, it is possible that our training set was not sufficiently large to allow the higher order Markov chain to accurately estimate the transition probabilities in the longer chain. To overcome this problem, further experiments with more training data are necessary.

### VI. Conclusions

Natural user interfaces (NUI) are often described as the next evolutionary step in human-computer interaction, following the step from command-line interfaces to graphical user interfaces. Testing applications based on NUI interactions poses a new challenge to software testing: Automatically generating test inputs is no longer as simple as clicking GUI widgets, it is a matter of simulating real user behaviour. In this paper we have presented a solution to achieve exactly this in the context of gesture-driven Kinect applications.

While our experiments on a gesture-controlled web browser application demonstrate the challenges and the potential of the presented approach, there remains much to be done as future work:

- **Smoothing sequences**: It may be possible to improve the motion sequences synthesised in our approach by interpolating intermediate positions, resulting in smoother sequences, and the possibility to vary the speed of movement.

- **Different NUIs**: Skeletal movement is one out of several different ways to interact with natural user interfaces. Our approach in principle extends to any type of gesture-controlled interface, for example for applications focusing on hand-movement (e.g., the Leap Motion[12]). Other types of approaches include speech-controlled applications, applications controlled through movement of controllers (e.g., Nintendo’s Nunchunk controllers, accelerometer and gyroscope inputs of smartphones, etc.) — although our approach may also extend to such types of inputs, more research is required.

- **Generic gesture corpus**: Our evidence is based on a single application and a single interface. Different applications will use different gestures, and a generic corpus of gesture information from which to synthesise new sequences needs to cover all relevant gestures. Future work therefore needs to include deriving such a generic corpus. However, for any specific application a generic corpus likely contains many irrelevant gestures, therefore there is a need for adaptive approaches that learn from observing interactions with the application under test which gestures are relevant and should be preferred for exploration. Ideally, a test generation approach would even be able to create entirely new gestures not contained in the training data.

- **Robustness testing**: While we aimed to maximise code coverage with the sequences of gestures generated in our approach, other test objectives might be conceivable. An example application would be to test an application for robustness against erroneous...

---

gestures, or test its ability to recognise imprecisely performed gestures by intentionally adding noise to the synthesised inputs.

- **Advanced test generation techniques:** In this paper, we used synthesised motion sequences in a random testing approach. However, it is well known that random testing will tend to cover shallow program states, but may struggle to reach “deeper” program states that require specific interactions. There is potential to use search-based testing techniques to evolve sequences of synthesised gestures in order to arrange the gestures in a way that, for example, maximizes code coverage.

- **Test oracles for NUI applications:** We have so far considered only the problem of deriving test inputs; in order to find software bugs the resulting behaviour needs to be checked for correctness by a test oracle. Existing techniques such as code contracts or assertions are immediately applicable, but checking correctness of a specification may pose additional challenges in the context of natural user interfaces. A common assumption in automated test generation is also that the test oracle will be provided by a user, who manually checks the behaviour of provides the expected output. Whether and how this is possible in the context of natural user interfaces calls for future research.

To foster research in testing of natural user interface applications, we make our prototype implementation and experimental data available at the following URL:

https://bitbucket.org/chrishunt/nuitest

**Acknowledgments.** This work was funded by a Microsoft Software Engineering Innovation Foundation (SEIF) Award on “Fully Automatic Testing of Natural User Interfaces”

**REFERENCES**


