An Empirical Evaluation of Automated Black Box Testing Techniques for Crashing GUIs

Cristiano Bertolini Glaucia Peres Marcelo d'Amorim Alexandre Mota

Center of Informatics, Federal University of Pernambuco
P.O. Box 7851, 50732-970, Recife-PE, Brazil
E-mail: {cbertolini,gbp,damorim,acm}@cin.ufpe.br

Abstract

This paper reports an empirical evaluation of four black box testing techniques for crashing programs through their GUI interface: SH, AF, DH, and BxT. The techniques vary in their level of automation and the results they offer. The experiments we conducted quantify execution time and the capability of finding a crash for each technique on 8 different cellular phone configurations with historical (real) errors. The results show that AF and BxT offered better precision than SH and DH (AF and BxT found crashes in all 8 configurations), and BxT crashes the application the fastest more often (5 out of 8 cases). The experiments reveal that the selection of the random seed to AF and BxT results in a high variance of execution time (i.e., the time the technique takes to either crash the application or timeout in 40h): the median (across 8 phone configurations) of the standard deviation of execution times (for 10 runs per each phone configuration) is 7.79h for AF and 5.21h for BxT. Despite this fact, AF and BxT could crash the application consistently: the median of the precision (fraction of the 10 runs that results in a crash) is 74% for AF and 69% for BxT.

1. Introduction

Despite the technological advances in languages and tools to support systems development, programmers still deliver software with lots of errors. Several techniques have been proposed to this end – to improve software reliability. Testing is one of them. In fact, software testing is the dominant approach in industry to assure software quality. Testing is not cheap though. A study done in 2002 by the National Institute of Standards and Technology (NIST) [12] reports that between 70% and 80% of development costs is due to testing and debugging. *Automation* of software testing then becomes very important as means to reduce this cost.

Black box testing is the activity of testing without knowledge of the program organization [4]. Any part of the system providing a public interface is eligible for black box testing. It consists of exercising the interface of a component (typically the entire system) to find errors. Advantages of black box testing compared to white box testing include independence between the programmer and the tester – which may reduce bias – and no reliance on source code – which may help with work division. Black box testing is particularly important for organizations that outsource testing activities. Often these organizations need to make private the source code they develop, say for confidentiality reasons. In those situations, black box testing becomes the primary choice for independent assurance of system's quality.

One important interface of an interactive system is the graphical user-interface (GUI). This paper focuses on black box testing of GUIs. In this setting, a test consists of (i) a sequence of GUI commands and (ii) a test oracle to check whether the effect of the command is as expected. The oracle is conceptually a boolean function that checks whether the application can still make progress through the GUI. *Automated generation of GUI tests is important* for two main reasons: (a) manual (GUI) tests can become obsolete with the evolution of an application or across the products (applications) of a software product line, and (b) manual tests may fail to capture corner-case scenarios. We specially focus on (b) as requirements documents are often incomplete. For example, manual tests derived from requirements may succeed in covering basic user interactions but fail to cover corner case scenarios that lead to a crash.

We investigate techniques whose *goal* is crashing the system with the *automated generation and execution of GUI tests*. Our *context* is that of applications providing limited or no access to its internal state.

Automation of black-box testing can be challenging: unguided search may be ineffective for too large state spaces [8]. For GUI testing, in particular, the size of the state space reachable (from the GUI) is typically intractable

[5, 16]. To alleviate the state space explosion problem many model-checking techniques need to access the state to perform space reductions [7, 9, 15]. Unfortunately, such optimizations are not possible for black box testing in general.

This paper describes four black box testing techniques for finding program crashes on GUIs. All techniques attempt to *explore the state space* of the application (i.e., to *stress* the application with relevant interactions) with the goal of finding a state that fails the test oracle. The techniques we propose provide distinct tradeoffs between their capability of finding crashes and the level of automation they offer. The main focus of this paper is on the evaluation of these techniques on Motorola cellular phones. The list below highlights the main contributions:

- The proposal of four techniques for GUI testing.
- An empirical evaluation of the techniques using Motorola cellular phones.

We provide next a brief overview about the techniques and the experimental results we obtained. Section 2 discusses the techniques in more detail and Section 3 the experimental evaluation.

1.1. Summary of the techniques

We next summarize (i) Scenario Hunting (SH), (ii) Atoms Framework (AF), (iii) Driven Hopper (DH), and (iv) Behavior eXplorer Tool (BxT), the techniques we propose for crashing applications from the GUI. It is important to note in the following that we do not distinguish between "test" and "test sequence" as the notion of correctness we use is implicit, i.e., the program should not crash. More precisely, the oracle is not physically part of one test.

SH takes a manually written test suite as input, generates a fixed number of random permutations for this suite, and finally executes each test and monitors for a crash. SH is perhaps the simpler technique we discuss. The merit of SH relies mostly on the user selection of the input suite. It serves mainly as a baseline to compare more automated techniques.

The **AF** exploration differs from SH in two important ways: (i) test granularity, and (ii) data generation. As for test granularity, one AF test corresponds to a small fragment of a SH test. For that reason we frequently call one AF test an *atom*. As for data generation, AF enables a test to share data: one test can consume data another test produces. One SH test can only consume data it generates.

Illustrative example. Figure 1 shows a fragment of one test (sequence) that SH uses in the evaluation we conducted on cellular phones (see Section 3). This sequence consists of using one phone to (i) capture an audio message, (ii) playing back that message, and (iii) deleting that same message. This test is written in Java and runs on a regular PC. It uses a library to communicate with the cellular phone. AF divides this test in smaller fragments that makes sense in itself. In this example, *the engineer* may create three atoms, according to the instructions associated to each log instruction. Note that one AF test can have parameters in result of this method extraction. For example, the second atom (for listening the audio message) will require a *multimediaFile* object. This is key to AF as it enables one atom to exercise different inputs.

```
log("Capture an audio message.");
navigationTk.launchApp(PhoneApplication.get("VOICE_RECORDS"));
multimediaTk.captureVoiceNoteFromVoiceRecord(30);
multimediaFile voiceRecord =
multimediaTk.storeMultimediaFileAs(MultimediaItem.get("STORE_ONLY"));
log("Listen to an audio message.");
navigationTk.goTo(PhoneApplication.get("VOICE_NOTES"));
multimediaGoTo.get("ALL_VOICE_NOTES"));
multimediaTk.scrollToAndSelectMultimediaFile(voiceRecord);
log("Delete an audio message.");
phoneTk.returnToPreviousScreen();
multimediaTk.deleteFile(voiceRecord,true);
```

Figure 1. A test sequence for multimedia.

SH and AF build on the user experience to find crashes. The following techniques, in contrast, require less user-input. In particular, they do *not* rely on user-defined tests such as the one Figure 1 shows. **DH**, in particular, repeats the following sequence of steps until finding an error or crashing the application: it drives the application to a particular screen and keeps pressing random keys for some (configured) time. DH requires user-provided tests that conceptually jumps to a screen using, for instance, the instruction `navigationTk.goto()` used in Figure 1.

BxT selects inputs more systematically than DH: it recognizes which controls are available at a screen and selects inputs according to these controls. For instance, BxT can send scroll down and up events when it recognizes a scroll bar control in the current screen. In a screen that contains only two buttons, say "OK" and "Cancel", DH may press several keys before it hits the ones for "OK" and "Cancel". BxT, differently, makes a random selection between one of these two options. Note that BxT has more stringent observability and controllability requirements [4, 11]: it must use a library that provides support for recognizing screen components and sending events to them.

1.2. Summary of results

The experiments we conducted quantify execution time and the capability of finding a crash for each technique on 8 different cellular phone configurations with historical (real) errors. The results show that AF and BxT offered better pre-

Algorithm 1: genList pseudo code

```
1  genList(Set⟨Test⟩ suite, int numRept, int seed): List⟨Test⟩
2  begin genList
3      List⟨Test⟩result = [ ];
4      for i = 1..numRept do  result = result ∪ shuffle(suite, seed);
5      return result;
6  end
```

cision than SH and DH (AF and BxT found crashes in all 8 configurations), and BxT crashes the application the fastest more often (4 out of 8 cases). The experiments reveal that the selection of the random seed to AF and BxT results in a high variance of execution time (i.e., the time the technique takes to either crash the application or timeout in 40h): the median (across 8 phone configurations) of the standard deviation of execution times (for 10 runs per each phone configuration) is 7.79h for AF and 5.21h for BxT. Despite this fact, AF and BxT could crash the application consistently: the median of the precision (fraction of the 10 runs that results in a crash) is 74% for AF and 69% for BxT.

The results indicate that AF and BxT – the more automated techniques – were superior for finding program crashes.

2. Techniques

This section describes four testing techniques this paper proposes for crashing applications through their GUIs.

Note on oracle. To simplify discussion, we assume crashes are unexpected situations which the system can not continue its normal execution. That allows the algorithms to represent the oracle with the external function *isCrash()*. We do not discuss this function here. Note that this paper does not propose test oracles. The user needs to provide the oracle appropriate for detecting crashes.

Note on user-provided test suite. SH, AF, and DH build on existing manually written tests. But DH tests simply perform a jump to one GUI screen.

Note on GUI library. DH and BxT build on operations (provided by some library) that enables some read and write access to the GUI components. Sections 2.3 and 2.4 highlight the operations DH and BxT use to clarify how they can be used in different contexts. It is important to mention that some systems provide rich support for testing, i.e., specific interfaces for reading (i.e., observing behavior) and writing to the state (i.e., controlling the application). For example, the cellular phone platforms Symbian [3] and Linux/Java [2] provide infrastructure to the tester implement monitors that can inspect the memory for safety problems such as buffer overflows and memory leaks.

Algorithm 2: SH pseudo code

```
1  main(Set⟨Test⟩ suite, int numRept, int seed, int timeout): bool
2  begin main
3      List⟨Test⟩ testList = genList(suite, numRept, seed);
4      foreach test in testList do
5          test.run();
6          if isCrash() then return true;
7          if isTimeout(timeout) then return false;
8      endfch
9      return false;
10 end
```

Algorithm 3: AF pseudo code

```
1  main(Set⟨Test⟩ suite, int numRept, int seed1, int seed2, int timeout):
   bool
2  begin main
3      Map⟨String, List⟨Object⟩⟩ dataMap = loadDataMap();
4      Set⟨Atoms⟩ atomSet = ∅;
5      foreach test in suite do atomSet = atomSet ∪ test.atoms();
6      List⟨Test⟩ testList = genList(atomSet, numRept, seed1);
7      foreach test in testList do
8          dataMap = test.run(dataMap, seed2);
9          if isCrash() then return true;
10         if isTimeout(timeout) then return false;
11     endfch
12     return false;
13 end
```

2.1 SH

Algorithm 1 generates a random list of tests from a set of user-provided tests. Each iteration of the loop at line 4 generates one permutation of the input set of tests. Effectively, it provides as result a list that includes *numRept* permutations of *suite*. Section 3.4.2 elaborates on a variation of this algorithm.

Algorithm 2 shows the pseudo code for SH. Function *main* assigns the result of the call to *genList* to variable *testList*. Each iteration of the loop at line 7 executes one test from this list, checks for a crash, and checks for timeout. Execution either terminates reporting a crash (line 6), or reporting a timeout (line 7).

Note on suite selection. The selection of the input test suite (*suite*) is decisive for the final result. Conceptually, the number of tests in the suite affects positively the chances one important part of the application is exercised (i.e., contains the defect) and negatively the exhaustion to which this part is exercised (i.e., may fail to activate the defect).

2.2. AF

Algorithm 3 shows the pseudo code for AF. The map *dataMap* that function *main* declares provides data input for the execution of tests. This map associates a list of objects

Algorithm 4: DH pseudo code

```
1  main(List⟨Test⟩ screens, int seed, int timeout1, int timeout2): bool;
2  begin main
3      while !isTimeout(timeout2) do
4          Test screen = listOfScreens.pickOne(seed);
5          screen.run();  /*goto random screen*/
6          pressRandomKeys(seed, timeout1);                    ⇐
7          if isCrash() then return true;
8      endw
9  end
```

to each input category (the map key). One atom is a parametric test that consists of a user-defined fragment from a user-defined test. The execution of one atom (test) may read from or update the data map.

AF stores in variable *atomSet* a set of atoms derived from the tests in *suite*. The variable *testList* stores the list of atoms resulted from the call to *genList*. Similar to SH each iteration of the loop at line 7 executes one test from *testList*, checks for a crash, and checks for timeout. However, different from AF a test takes as input the data map *dataMap* and the seed *seed2* and produces a new data map, possibly extending the input map with new inputs. The seed allows a test run to randomly choose one input from a list of objects for a specific category.

In summary, AF differs from SH in two important ways: (i) test granularity (atoms are fragments of SH tests), and (ii) data generation (the execution of one test provides inputs to parametric tests).

2.3 DH

Algorithm 4 shows the pseudo code for DH. The algorithm repeats the following sequence of steps until it either timeouts or finds a crash: (i) selects one screen, (ii) drives the application to that screen, (iii) sends random events (key presses) to the GUI for a while, and (iv) checks for a crash. The loop at line 3 repeats this sequence of steps.

The inputs to DH are a sequence of manually written tests – *screens*, a random seed that the event generator uses – *seed*, a bound on execution time for sending random events (key presses) in one iteration – *timeout1*, and a bound on total execution time – *timeout2*.

Note that DH does *not* generate inputs (i.e., GUI events) according to the components active on the current screen. It simply generates control events *randomly* within one important region of the application. Also important to note is that the algorithm uses a library to send events to the GUI. For DH it sends *general* key pressed events – which performs well for the domain of cellular applications (see Section 3.3). Line 6 highlights the use of one library function for sending key pressed events to the application. One needs to provide such a function to enable DH.

Algorithm 5: BxT pseudo code

```
1  main(int seed, int numRept, int timeout): bool;
2  begin main
3      Set⟨Test⟩ screenSet = ∅;
4      if driven() then
           /*random set of goto-screen tests*/
5          screenSet = {tc_1, tc_2, ..., tc_n};
6      else
7          screenSet = {initialScreen()};
8      endif
9      while !isTimeout(timeout) do
10         screenSet.pickOne(seed).run();
11         for i=1 to numRept do
12             Event ev = enabledEvents().pickOne(seed);   ⇐
               /*sends message to the GUI*/
13             ev.genInputs(seed).run();                   ⇐
14             if isCrash() then return true;
15         endfor
16     endw
17     return false;
18 end
```

2.4. BxT

Algorithm 5 shows the pseudo code for BxT. The main difference from DH is the way it generates events. Conceptually, the algorithm contains two main parts. The first decides which screen to focus. The second part stresses the application from a selected screen. This part performs the following steps for a fixed number of times or until it crashes the application: (i) identifies enabled events on the *current* screen (i.e., the events that active components can process), (ii) selects randomly one of such events, (iii) generates data for this event and sends the event to the GUI, and (iv) checks for a crash. The code fragment in the line range 12-14 corresponds to this sequence.

The inputs to BxT are a seed used for generating sequence (i.e., choosing the event) and data (i.e., generating input to the event) – *seed*, an integer denoting the number of iterations on the second part of the algorithm – *numRept*, and a bound on the total time for testing – *timeout*.

Line 10 makes a random choice of which screen in the set *screenSet* execution should stress. Note that the code fragment in the line range 4-8 initializes this variable. The external boolean function *driven()* indicates whether or not execution should perform jumps across different screens (in a similar fashion to DH). We call driven BxT, or simply D-BxT, this variation of BxT. For D-BxT, execution initializes the variable *screenSet* with a fixed set of screens. (This is how we used D-BxT in our experiments. For clarity, we showed the initialization of *screenSet* within the algorithm.)

Lines 12 and 13 highlight the uses of a library to identify which events are enabled and to send the event to the GUI.

3. Evaluation

This section provides details on the empirical evaluation of the techniques using Motorola cellular phones. We conducted 3 sets of experiments. One for comparing the four techniques w.r.t. their capability to crash phones with known bugs (Section 3.3). For this experiment we use SH and DH as baselines. To this date, the Motorola test center in Recife (Brazil) uses among others these techniques for crashing phones. The experimental results indicate that AF and D-BxT outperform SH and DH. Section 3.4 discusses the impact of randomization (for data and sequence generation) on AF and Section 3.5 discusses the impact of randomization on BxT.

3.1. Characterization of subjects

We characterize each subject with (i) the phone model (i.e., a list of external and internal phone features to identify a set of similar phones functions), (ii) the hardware version, (iii) the software version (i.e., the build for the operating system and its applications), and (iv) the flex bit (FB) version. The flex bit configuration allows the user to dynamically configure the phone prior. Example of such configurations includes enabling the phone to send and receive bluetooth signals, and setting the phone to debug mode.

Config.	Model	Hard.	Soft.	FB
A	M1	H3	S1	F1
B	M1	H4	S2	F2
C	M1	H4	S3	F3
D	M2	H2	S4	F4
E	M2	H1	S5	F5
F	M3	H5	S6	F6
G	M3	H6	S7	F7
H	M2	H2	S8	F8

Table 1. Characterization of experimental subjects.

Table 1 shows the subjects we used in our experiments. Column "Config." introduces a unique identifier to distinguish each combination of model, hardware, software and flex bit. The other columns show each of these attributes. The identifiers we use in this table are artificial. Note that some configurations share the same model or hardware, but the software and flex bit vary. The selection of these configurations was driven by the availability of equipment where past errors have been detected.

3.2. Failures

The oracle does not operate on the GUI. It is a general Motorola proprietary program that monitors the phone memory for bad states.

The oracle detects 12 distinct kinds of crashes across all experiments. In the following, we distinguish them using crash identifiers (CIDs) from 1 to 12. Each identifier denotes a different undesirable scenario of the application that the oracle is able to capture. For example, CID=1 is a general report to denote that the system makes no progress but the oracle is unable to ascertain the reason, CID=2 means that an issue with the hardware interface (e.g., it is not possible to allocate memory) prevents the application from making progress, CID=6 denotes a programming error like divide by zero, etc. Important to note is that the oracle reports only the crash event; it does not inform the reason for the crash (as debuggers do). In result, it may happen that distinct techniques report different manifestations (CIDs) of the same defect.

3.3. Comparison of techniques

This section describes the experiment we conducted to compare the techniques.

Setup. The goal of this experiment is to compare the effectiveness of AF and D-BxT with that of SH and DH for crashing cellular phones with historical defects. We used 8 different phone configurations for which SH found 4 crashes and DH found 6 crashes. Neither SH nor DH crashed 2 of the eight configurations. For each configuration we ran once each technique until execution runs out of time (timeout=40h) or finds a crash. The execution of SH and DH confirmed the crashes documented in the bug report database. For AF and D-BxT, we fixed the random seed across different configuration runs.

The *atoms* that AF uses derives from the tests SH used – we did not include any atoms original from a different set of tests. This helps us to compare SH and AF. D-BxT explores the state space similarly to DH – exercise a random sequence of events on the GUI for 30s from one arbitrary GUI screen. To achieve this we align the setting of the paramaters *timeout1* in Algorithm 4 and *stepSize* in Algorithm 5. This similarity helps us to compare DH and D-BxT.

Results. Table 2 shows a summary of the results obtained in the experiments. Column "Config." shows the identifier for one subject configuration, column "CID" shows the identifier of the crash, and column "time" shows the execution time for each experiment. Recall that one experiment either timeouts or finds a crash. Line "avg." reports the aver-

Config.	SH		AF		DH		D-BxT	
-	CID	time	CID	time	CID	time	CID	time
A	-	40.0	6	**6.5**	1	21.2	6	33.8
B	-	40.0	4	33.6	-	40.0	6	**6.2**
C	-	40.0	4	36.5	-	40.0	12	**14.8**
D	2	4.3	7	5.0	8	**3.3**	8	4.4
E	3	3.9	1	3.6	3	7.0	5	**2.7**
F	-	40.0	1	**4.0**	6	5.7	11	**4.0**
G	4	3.1	6	11.2	6	22.7	1	**1.7**
H	5	**2.3**	5	2.8	5	21.7	10	7.9
avg.	50%	21.7	**100%**	12.9	75%	20.2	**100%**	9.4

Table 2. Time (in hours) and Fault revealed by SH, AF, DH and D-BxT per phone configuration.

ages of each column. For column CID, it shows the fraction of experiments that revealed a crash. For column time, it shows the arithmetic mean of the elapsed time.

We list below our key observations:

- Only AF and D-BxT can find a crash for all eight experiments with a timeout of 40h. As such, note that only AF and D-BxT can find the crashes reported in experiments B and C.
- D-BxT can find a crash faster more often than the other techniques. Note (from the highlighted cells) that D--BxT outperforms the other techniques in 5 out of 8 cases.
- SH can find a crash in only 50% of the cases. But when it finds, it outperforms AF, except in Config. E where the difference is of only 0.3h (that is, 18min).
- For each experiment where DH finds a crash, one of the other three techniques can find it and find it faster. In particular, AF + D-BxT find all errors that DH finds and faster, except in Config. D.
- The variation of time that each technique reports is very high. For example, between D-BxT and AF the difference in time for crash for experiment A is +27.3h (i.e., D-BxT takes 27.3h more to find the crash), -27.4h for experiment B, -21.7h for C, -9.5h for G, and +5.1h for experiment H.
- The variation of the CID reported is also high. Note that no experiment reports the same CID for all techniques. For example, in experiment H, SH, AF, and DH report CID=5, while D-BxT reports CID=10.

3.4. Impact of Randomization on AF

This section discusses two experiments we conducted to evaluate the impact of randomization on AF. The first experiment measures the impact of the random data on the effectiveness of AF. For this, we vary the the value of parameter *seed2* used in line 8 of Algorithm 3. The second experiment evaluates the impact of a random selection of the list of *atoms* when compared to the list computed from SH tests used in the experiments Section 3.3 reports.

3.4.1 Random data

This section describes the experiment we conducted to evaluate the impact that the use of different random data has in the effectiveness of AF.

Setup. In this experiment, we run AF for 10 times, on each configuration, varying the value of the parameter *seed2* used in Algorithm 3. We use the same sequence in all executions of a configuration (i.e., we fix the values of *seed1*). Figure 2 shows the distributions of execution time (in hours) for this experiment.

Note on distribution representation. We use box-plot notation to illustrate a data distribution. The lower and upper hinges of one box indicate respectively the upper bounds of the first and third quartiles of the distribution, the line across the box defines the second quartile (i.e., median value). The lines below and above the box limit the first and fourth quartiles. Small circles outside the hinges correspond to outliers. The symbol $\bar{x}$ denotes the mean value, the symbol σ denotes the standard deviation – an average for the dispersion of data points from the mean value, and the symbol $\hat{x}$ denotes the median value.

Results. Table 3 shows detailed data for the 10 runs of AF over each configuration from A to H. The value "-" for column "CID" indicates a missed crash (resp., value "40.0" for column "time" indicates a timeout). For example, for configuration A, AF misses the crash on experiment 4. We list below our key observations:

- *Time.* The dispersion of the data points in AF is high for almost all configurations. The median standard deviation of execution times for all configurations is 7.79h. That means that AF execution time is very sensitive to the selection of the seed.

	Config. A		Config. B		Config. C		Config. D		Config. E		Config. F		Config. G		Config. H	
#	CID	time	CID	time	CID	time	CID	time	CID	time	CID	time	CID	time	CID	time
1	6	6.5	4	33.6	4	36.5	7	5.1	1	3.6	1	4.0	6	11.2	5	2.7
2	6	14.6	-	40.0	-	40.0	7	5.5	5	3.6	-	40.0	6	15.1	5	3.4
3	6	13.9	6	37.8	-	40.0	7	5.3	6	1.9	-	40.0	4	18.3	5	3.3
4	-	40.0	-	40.0	-	40.0	7	5.7	5	3.9	-	40.0	4	16.6	5	3.1
5	4	33.9	6	4.5	12	4.0	7	5.9	6	1.2	-	40.0	-	40.0	5	2.9
6	9	19.0	4	27.2	-	40.0	7	5.1	5	3.7	12	37.8	4	34.7	9	2.9
7	4	30.5	-	40.0	-	40.0	7	7.4	5	1.0	-	40.0	4	17.6	5	2.5
8	6	16.7	6	12.7	-	40.0	7	5.3	9	1.5	-	40.0	4	1.5	5	3.0
9	4	32.4	6	35.3	-	40.0	7	4.9	9	1.1	-	40.0	9	15.2	5	3.2
10	6	13.4	6	2.9	-	40.0	7	4.3	5	3.9	-	40.0	6	1.6	5	1.1
avg.	90%	22.1	70%	27.4	20%	36.1	**100%**	5.5	**100%**	2.5	20%	36.2	90%	17.2	**100%**	2.8

Table 3. Impact of using random seeds in AF.

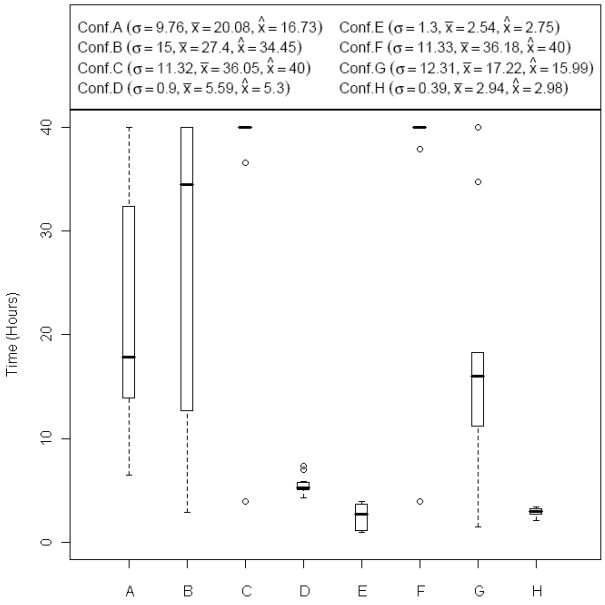

Figure 2. AF time distributions for random data.

- *Kind of crash.* Only configuration D finds the same kind of crash in all executions. All others configurations find more than 2 types of crash (for instance, Config. E finds 4 different types).
- *Precision.* Although some executions do not find a crash, the median of the precision – the value of the avg. CID for each configuration – is high: 74%.

3.4.2 Random sequence

This section describes the experiment we conducted to evaluate the impact of randomizing the list of *atoms* used as input to AF.

Algorithm 6: genList with Allpairs

```
1  genList(Set⟨Test⟩ suite, int nAtoms, long seed): List⟨Test⟩
2  begin genList
3      Map⟨Category, Set⟨Atom⟩⟩ partition = partition(suite);
4      Map⟨Category, Set⟨Atom⟩⟩ selected = ∅;
5      foreach entry in partition do
6          Set⟨Atom⟩ atoms = entry.value();
7          Set⟨Atom⟩ tmp = ∅;
8          for i = 1..nAtoms do  tmp = tmp ∪ atoms.pickOne(seed);
9          selected.put(entry.key(), tmp);
10     endfch
       /*concatenates all sequences of atoms.  each
       sequence includes one atom on each category*/
11     return allpairs(selected);
12 end
```

Setup. We run AF for 10 times on phone configurations E and H. These configurations have the lowest average execution times (see Table 3). We used the same random seed for data and sequence generation in all runs.

Algorithm 6 redefines function *genList* that AF uses. This version associates each atom to one domain category. The categories we define are as follow: applaunch (*atoms* that only go to an application), browser (access the Internet), mms (deal with multimedia messages), multimedia (deal with multimedia files and camera), phonebook (deal with calendar, events and contacts), and sms (deal with text messages). Function *partition* in line 3 takes as input a user-defined test suite and returns a map that associates a set including all atoms of a category with the category it belongs.

The code fragment in the line range 5-10 selects *nAtoms* on each category and assign the resulting map to variable *selected*. We apply pairwise coverage [11] to generate sequences of atoms with the property that each atom of each category is paired to another atom of another category in at least one case. For this, we use the Allpairs [1] tool. Finally, it gives as output sequences of atoms (each sequence includes one atom of each category), and then concatenates these sequences to build one longer sequence AF executes.

	Config. E		Config. H	
#	CID	time	CID	time
1	-	5.5	5	1.6
2	5	2.4	-	5.0
3	5	5.3	-	5.4
4	-	10.4	5	1.3
5	-	6.2	-	6.7
6	-	4.9	5	1.1
7	5	3.1	-	4.3
8	5	6.7	-	5.5
9	-	5.4	-	5.4
10	-	5.5	5	2.2
avg.	40%	5.54	40%	3.8

Table 4. Runs of AF with different execution lists for configurations E and H

Results. Table 4 shows detailed data for the 10 runs of AF over configurations E and H. Our key observations for this experiment are as follows:

- *Precision.* For both configurations, AF found a crash in only 4 out of 10 runs, while AF found a crash for 100% of the cases when using atoms derived from SH tests.

This experiment indicates that the interaction between categories (functionalities of the system) may not be as important as the selection of critical atoms.

3.5. Impact of Randomization on BxT

This section discusses two experiments to evaluate the effect of randomization on BxT. The first experiment evaluates D-BxT compared to BxT. The second experiment evaluates the effect of using different random seeds as input to BxT.

3.5.1 Random data and sequence in BxT

This section shows the impact that the use of different random seeds have in the effectiveness of BxT.

Setup. We ran BxT 10 times for each configuration with different random seeds. The use of different seeds impacts the generation of different sequences of events and data. With this experiment we want to observe the variance of the technique for distinct seed selections. Figure 3 shows the distribution time (in hours) and Table 5 shows the detailed data for all configurations.

Results. We list next key observations:

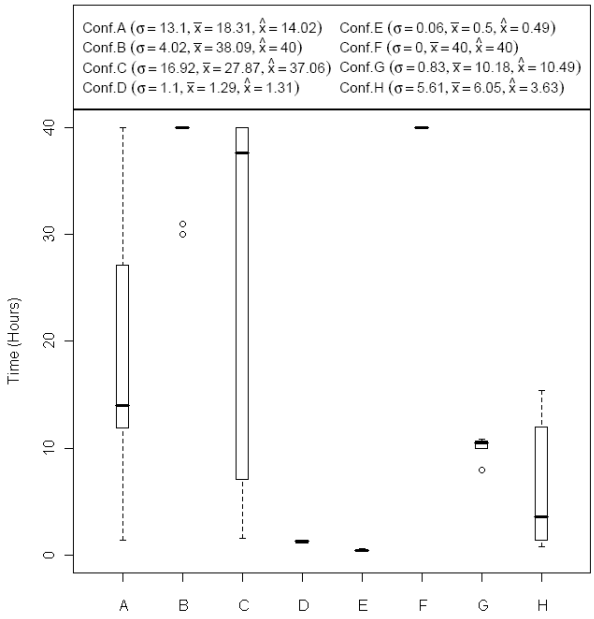

Figure 3. BXT time distributions for random data and sequence.

- *Precision.* BxT can find crashes consistently for Config. A, C, D, G and H. Thus, for all configurations the median of the precision was 69%.
- *Variance.* The standard deviation in BxT is high for configurations A, C and H but low for the other configuration. It is likely that for those case the fault density is low relative to other configurations and the selection plays an important role.

3.5.2 Comparison of BxT and D-BxT

This section compares BxT and D-BxT.

Setup. This experiment configures BxT with a timeout of 40h, and parameter *numRept* set to 1000. BxT runs 1000 events in each iteration, taking approximately 10min. D-BxT explores one screen for some time (less than 10 minutes) and jumps to another screen until it reaches the 40h timeout. The experiment configures D-BxT with the parameter *numRept* set to 50, which results in each iteration taking approximately 30s.

Results. Table 6 summarizes the comparison. Column "diff1" denotes the execution time difference from BxT to D-BxT considering all configuration runs, and "diff2" the time difference considering only those runs that both BxT and D-BxT crash the application. We list below our key observations:

	Config. A		Config. B		Config. C		Config. D		Config. E		Config. F		Config. G		Config. H	
#	CID	time	CID	time	CID	time	CID	time	CID	time	CID	time	CID	time	CID	time
1	6	14.6	-	40.0	-	40.0	7	1.2	5	0.5	-	40.0	6	8.0	4	12.0
2	1	1.4	-	40.0	-	40.0	7	1.4	5	0.5	-	40.0	6	10.0	4	15.4
3	-	40.0	4	39.9	1	7.1	7	1.3	5	0.5	-	40.0	6	10.6	5	1.2
4	11	12.9	-	40.0	12	32.4	7	1.3	5	0.4	-	40.0	6	10.5	5	4.5
5	6	7.8	-	40.0	1	35.2	7	1.3	5	0.5	-	40.0	6	10.7	10	0.8
6	11	11.9	-	40.0	-	40.0	7	1.1	5	0.6	-	40.0	6	9.9	4	5.8
7	11	14.0	-	40.0	12	2.5	7	1.4	5	0.5	-	40.0	6	10.5	6	2.7
8	-	40.0	4	30.0	-	40.0	7	1.4	5	0.6	-	40.0	4	10.9	5	2.7
9	1	27.7	-	40.0	-	40.0	7	1.2	5	0.6	-	40.0	6	10.7	10	1.5
10	11	14.1	-	40.0	12	1.6	7	1.4	5	0.6	-	40.0	6	10.1	5	14.0
avg.	80%	18.4	20%	38.1	50%	27.9	**100%**	1.3	**100%**	0.5	0%	40.0	**100%**	10.2	**100%**	6.1

Table 5. Impact of using random seeds in BxT.

Config.	BxT		D-BxT			
-	CID	time	CID	time	diff1	diff2
A	6	14.6	6	33.8	+19.3	+19.3
B	-	40.0	6	6.2	-33.8	-
C	-	40.0	12	14.8	-25.2	-
D	7	1.2	8	4.4	+3.2	+3.2
E	5	0.5	5	2.7	+2.2	+2.2
F	-	40.0	11	4.0	+35.9	-
G	6	8.0	1	1.7	-6.3	-6.3
H	4	12.0	10	7.9	-4.2	-4.2
avg.	62.5%	19.5	100%	9.44	62.4	22.6

Table 6. BxT versus D-BxT

- *Time.* On average, D-BxT is slower than BxT when they both find a failure. See column "diff2".
- *Precision.* In contrast to D-BxT and AF, BxT could not crash the application in 3 out of 8 configurations. That indicates that the screen jump was effective to improve the exploration. Conceptually, the jumps correspond to a higher weight to the width compared to the depth of the exploration graph.

3.6. Threats to validity

This section describes threats to internal and external validity of our experiments. *Internal validity* determines whether or not (or a degree to which) the experimental observations are effect of the technique. *External validity* determines whether or not one can generalize the experimental observations to other scenarios.

One threat to internal validity is internal randomness. In principle, it is possible that the system does not answer promptly to the commands that the automated test issues. This depends on the operating system's scheduling decisions. Note that the user would operate the system in a much slower pace and such timing issues rarely occur in practice. This effect could impact our observations. Note, however, that the distributions of D, E and H for AF and D, E and G for BxT shows that the dispersion is small, indicating that it performs consistently with different seeds. We believe that the dispersion of data would increase if the effect of internal randomness (from the operating system) was severe in this case.

One threat to external validity is portability of techniques. We implemented all techniques with the goal of testing cellular phones. In principle, there is no reason to believe that they are not applicable to other kinds of application. To generalize the techniques to other platforms or kind of systems we must port the library to access and run GUI components like screens, buttons, check box, etc. In this way, we believe that our techniques are easily generalized to different types of systems (e.g. web and desktops applications).

4 Related and Future Work

Pelánek *et al.* [13] show different techniques to improve the state space efficiency in random state space exploration. The techniques we discuss also use randomization but our focus thus far has not been optimization. BxT, in particular, randomly explores the system considering the context of each screen. We can build on the improvements proposed by Pelánek *et al.*. We plan to investigate how the use of models can speed up the exploration. More precisely, we plan to execute BxT on two rounds. First to extract a model of the GUI. Then, to use the model to drive the actual exploration. We expect the model to speedup the exploration with the reduction of the cost of identifying the current screen and the enabled events on it.

A popular approach in industry to automate test documentation and execution is capture-and-replay. The idea is to record in a test script the actions that the user makes while interacting with the GUI and then to (re-)execute this script when necessary. For instance, Stefen at al. [14] propose the JRapture, a capture-and-replay tool that operates on Java bytecodes. The authors argument that JRapture can capture much more graphical elements (graphical widgets)

than commercial tools. Capture-and-replay is complementary to SH and AF and orthogonal to DH and BxT (as they automate test generation).

Brooks and Memon [5] propose a technique to generate test cases based on usage information, in the form of usage profiles. These profiles describe event sequences captured from the user's experience, i.e., event sequences captured while the user interacts with the GUI. Although the idea of using profiles is appealing, it is not yet practical in the domain we evaluate our techniques (cellular communication). In spite of this limitation, it is important to say that this work is complementary to ours. In particular, SH and AF use test cases drawn from requirements document, history of faults, for example. These do not need to correspond to the actual scenarios of use.

Yuan and Memon [16] propose a technique for test case generation of GUI applications based on the analysis of feedback obtained from the run-time state of GUI widgets. The idea is that it does not make sense to build a test, say with two events if these events do not read or write to the same part of the state. Identifying these data dependencies help to reduce the state space that a test driver needs to explore. We plan to build on these ideas as future work.

Memon at al. [10] design many different generic oracles for GUI that one can use as expected output of a test. The paper shows the importance of oracle definitions to estimate the effectiveness of the testing process. We are using a more specific oracle in the context of phone applications. The oracle itself does not rely on the GUI, but on the underlying phone state.

Dwyer et al. [6] illustrate parallel search for program model checking. As it is not possible to identify the density of faults beforehand, parallel search can improve the chances in general of crashing the application at the expense of using more resources. We plan to apply parallel search for reducing the impact on the seed selection.

5. Conclusions

This paper describes four black-box testing techniques with the goal of crashing GUI. We evaluate these techniques on Motorola cellular phones.

Our empirical results demonstrate that AF and BxT outperformed SH and DH with respect to time and also to the number of crashes reported. In spite of the high variance of execution times for different seed selections, AF and BxT crashed the application consistently. The precision was 74% for AF and 69% for BxT. The results indicate that an automated technique (BxT) performed nearly the same as one using user-provided test suites as input (AF). BxT is more automated but requires a library to implement its main functions (for observing and controlling the GUI). AF, on the other hand, is less automated but does not require a supporting library.

Acknowledgments. This work is in collaboration with the Stress Lab team at Motorola Brazil Test Center. This work was partially supported by the CNPq fellowship 142905/2006-2 and 550466/2005-3.

References

[1] James Bach - Satisfice, Inc webpage. `http://www.satisfice.com/tools/pairs.zip`.

[2] Linux Java webpage. `http://www.motorola.com/motomagx/`.

[3] Symbian webpage. `http://www.symbian.org`.

[4] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[5] P. A. Brooks and A. M. Memon. Automated gui testing guided by usage profiles. In *ASE '07*, pages 333–342, New York, NY, USA, 2007. ACM.

[6] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *ICSE*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[7] A. Gotlieb. Exploiting symmetries to test programs. In *ISSRE*, Denver, Colorado, November 2003.

[8] M. Harman and J. Wegener. Getting results from search-based approaches to software engineering. In *ICSE*, pages 728–729, 2004.

[9] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE*, page 254, Washington, DC, USA, 2001. IEEE Computer Society.

[10] A. M. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *ASE*, pages 164–173, 2003.

[11] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[12] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, May 2002.

[13] R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing Random Walk State Space Exploration. In *FMICS*, pages 98–105, New York, NY, USA, 2005. ACM.

[14] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture/replay tool for observation-based testing. *SIGSOFT Softw. Eng. Notes*, 25(5):158–167, 2000.

[15] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for Java containers using state matching. In *ISSTA*, pages 37–48, 2006.

[16] X. Yuan and A. M. Memon. Using gui run-time state as feedback to generate test cases. In *ICSE*, pages 396–405, Washington, DC, USA, 2007. IEEE Computer Society.