View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Open Repository and Bibliography - Luxembourg

N-Gram Based Test Sequence Generation from Finite State Models

Paolo Tonella, Roberto Tiella, and Cu D. Nguyen

Software Engineering Research Unit
Fondazione Bruno Kessler, Trento, Italy
{tonella, tiella, cunduy}@fbk.eu
http://se.fbk.eu

Abstract. Model based testing offers a powerful mechanism to test applications that change dynamically and continuously, for which only some limited black-box knowledge is available (this is typically the case of future internet applications). Models can be inferred from observations of real executions and test cases can be derived from models, according to various strategies (e.g., graph or random visits). The problem is that a relatively large proportion of the test cases obtained in this way might result to be non executable, because they involve infeasible paths.

In this paper, we propose a novel test case derivation strategy, based on the computation of the N-gram statistics. Event sequences are generated for which the subsequences of size N respect the distribution of the N-tuples observed in the execution traces. In this way, generated and observed sequences share the same context (up to length N), hence increasing the likelihood for the generated ones of being actually executable. A consequence of the increased proportion of feasible test cases is that model coverage is also expected to increase.

1 Introduction

In model based testing, test cases are derived from models, which are often presented by mean of finite state machines, of the application under test [1]. The underlying idea is that the model encodes all relevant application behaviours and abstracts away the irrelevant implementation details, so that testing can be focused on covering all critical application behaviours, without wasting time on non-critical features of the application. Model based testing does not require white box knowledge of the application under test. Moreover, incremental model update algorithms [2] can be used with continuously and autonomously changing applications, as future internet applications.

Models can be defined upfront, at design time, but such practice is not very common and is costly due to labour work. Alternatively, models can be inferred from observations of actual executions, recorded as log traces. The two main techniques for model inference from execution traces are state abstraction and event sequence abstraction. In state abstraction, abstraction functions are defined to map concrete states into abstract states, so as to control the size of the inferred model and to allow for generalisation from the actually observed states [3]. Event sequence abstraction takes advantage of regular language inference algorithms, such as k-tail [4], or its variants [5, 6]. A finite state machine is obtained which recognises the language of the event sequences observed in execution logs. Such finite state machines are actually a generalisation of the observed sequences (not just their union).

The generalisation performed by model inference is usually *unsound*, which means the inferred models might introduce infeasible behaviours (paths allowed in the model that are impossible in the real application), hence over-generalising, and they might exclude some possible behaviours (paths allowed in the real application that do not exist in the model), hence under-generalising [7]. While the latter problem can be tackled by increasing the number and the representativeness of the execution traces used during model inference, so as to make sure the model includes as many behaviours as possible, the former problem is difficult to overcome and is particularly troublesome for testers. In fact, during test sequence generation an over-generalising model might produce test cases that traverse infeasible event sequences. For the tester, proving that a given test case derived from the model is associated with an infeasible path is a quite difficult task. In the general case, the problem is undecidable [8]. While particularly severe with inferred models, the presence of infeasible paths that cannot be tested is also a major problem with manually defined models.

In this work, we investigate a test sequence derivation strategy based on the notion of N-grams [9], which aims at mitigating the problem of the generation of infeasible paths. The idea is that, during the generation of a test sequence, the next event to add to the sequence should be selected according to the probability of the N-grams, as observed in the corpus of training traces. By constructing test sequences as concatenations of N-grams, inserted according to the observed frequency of occurrence, we expect to reduce dramatically the generation of infeasible test sequences. Our preliminary results on two applications confirm this speculation. Moreover, in comparison with other strategies (e.g., graph visit or random generation), N-gram based test sequence generation exploits some context information (the previous $N - 1$ events in the sequence) to determine the probability of occurrence of the next event, according to the actual sequences recorded in the execution logs. By reducing the number of infeasible sequences, N-gram based test sequence derivation delivers several benefits to testers: (1) the manual effort to confirm that some test sequences are infeasible is reduced; (2) coverage is increased, since a higher number of test sequences can be enacted and executed during testing; (3) the test case concretisation effort (necessary to supply concrete input data and the surrounding test harness) is also reduced, since less effort is wasted on infeasible cases.

The paper is organised as follows: Section 2 provides some background on model based testing and on the most widely used test sequence derivation strategies (namely, graph visit and random generation). Our novel technique for N-gram based test case derivation is presented in Section 3. Empirical data comparing our approach with graph visit and random generation are provided in

Section 4. Related works (Section 5) are followed by conclusions and future work (Section 6).

2 Baseline Sequence Generation Strategies in Model Based Testing

Model-based testing is an approach to generate test cases using a model of the application under test [10]. Dias Neto et al. [1] and Shafique [11] surveyed the state of the art in model-based testing. A model that describes structure and behaviour of the SUT is useful to acquire the knowledge needed to generate effective test cases. The model, in fact, provides an abstract and concise view of the application by focusing on specific aspects, i.e., classes of application behaviours associated with different application states. One of the most frequently used kinds of model is the Finite State Machine (FSM) model, even though some alternatives do exist (e.g., Briand et al. [12] use UML class and sequence diagrams). A node in the FSM represents a state of the application and it can be determined by, e.g., the values of class attributes (in case of object-oriented applications [13, 14]) or the values of graphical objects (in case of GUI-based applications [15–17]). FSMs are named *concrete* if each FSM state represents an actual application state or *abstract* if each FSM state represents a set (i.e., an equivalence class) of concrete states (e.g., [16, 18]). A transition in the FSM represents an application event/action (e.g., a method call, an event handler invocation) that can change the application state, if executed. Additionally, guards and conditions can enrich the model to capture the context in which events and actions are executed.

FSM models are exploited to generate test cases. By traversing the application FSM, sequences of application events can be extracted, so as to satisfy given coverage criteria [13]. For example, state or transition coverage (every FSM state or transition needs to be exercised by at least one test case) are often used, even if domain-specific (e.g., semantically interacting events [15]) criteria are sometimes preferred. As examples of output test sequences, based on the model presented in Figure 1 we can extract the following test sequences: $\langle e_1, e_2 \rangle$, $\langle e_1, e_2, e_3 \rangle$, $\langle e_4, e_5, e_3 \rangle$.

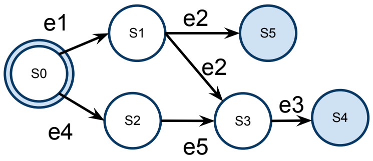

Fig. 1. An example of a model in FSM format; S_0 is the initial node, S_4 and S_5 are final nodes.

Different test case derivation strategies can be used to produce a set of test cases that satisfies the adequacy criterion of choice (e.g., transition coverage). Graph visit and random model traversal are among the most widely used strategies. We describe them in the next two subsections.

2.1 Graph Visit

```
proc depthFirstVisit($n : Node, p : Path$) $\equiv$
  do
     $next := \{m^e \in$ succ$[n] \mid \neg$visited$[\langle n, m\rangle_e]\}$;
     if ($\neg$empty($next$))
       then
          while ($\neg$empty($next$)) do
             $s^e :=$ randChoose($next$);
             if ($\neg$visited$[\langle n, s\rangle_e]$)
               then
                  visited$[\langle n, s\rangle_e] := true$;
                  depthFirstVisit($s^e, p + s^e$);
             fi
             $next :=$ remove($next, s^e$);
          od
       else
          if ($p$ increases adequacy) then addToTestSuite($p$); fi
     fi
  od.
while adequacy criterion not satisfied do
     visited := $\emptyset$
     $p :=$ depthFirstVisit($startNode, \langle\rangle$);
od
```

Fig. 2. Test sequence derivation by depth first visit

Depth-first and breadth-first model traversal can be used to derive test sequences from the model, interpreted as a directed graph. Figure 2 shows the pseudocode of the depth first test sequence generation strategy (DFV: Depth First Visit). Procedure *depthFirstVisit* determines the set of successor nodes that have not been visited yet (set *next*, where notation m^e indicates that m is a successor node if event e is triggered). Each of them is randomly chosen and the graph visit is recursively activated on such node, if the associated transition has remained yet to be visited (in fact, a previous recursive call of *depthFirstVisit* might have changed its visited state). This node is concatenated to the path traversed so far during the visit ($p + s^e$). When no unexplored successor node is found (*next* is empty), the visit terminates and the traversed path is added to the test suite being generated. The depth first visit procedure is called multiple times from the start node, until some adequacy criterion is satisfied (by construction, when the adequacy criterion is transition coverage, *depthFirstVisit* will be called just once). The global hash table *visited* records whether each transition has been visited or not during the sequence generation (it is reset inside the main loop, before calling *depthFirstVisit*). The result of this procedure is non deterministic, since it depends on the successor node s^e chosen for the continuation of the visit. Different choices may result in transitions being covered at different times during the visit, which in turn might give raise to different test sequences being added to the final test suite.

```
proc breadthFirstVisit($n : Node, p : Path$) $\equiv$
  do
      $end := true$;
      $next := \{m^e \in$ succ$[n]$ | $\neg$visited$[\langle n, m\rangle_e]\}$;
      while ($\neg$empty($next$)) do
            $end := false$;
            $s^e :=$ randChoose($next$);
            visited$[\langle n, s\rangle_e] := true$;
            addToFifoQueue($Q, p + s^e$);
            $next :=$ remove($next, s^e$);
      od
      if ($end \wedge p$ increases adequacy) then addToTestSuite($p$); fi
      if ($\neg$emptyFifoQueue($Q$))
         then
               $p =$ getFromFifoQueue($Q$);
               breadthFirstVisit(last($p$), $p$);
      fi
  od.
while adequacy criterion not satisfied do
      visited := $\emptyset$
      $p :=$ breadthFirstVisit($startNode, \langle\rangle$);
od
```

Fig. 3. Test sequence derivation by breadth first visit

Figure 3 shows the pseudocode of the breadth first test sequence generation strategy (BFV: Breadth First Visit). Procedure *breadthFirstVisit* uses a global queue data structure (Q) to store the partially explored paths to be considered later during the visit. All unexplored successor nodes (set *next*) are concatenated to the current traversal path p and are added to Q for future exploration. When no unexplored successor node exists, the visit terminates and the traversed path p is added to the test suite. If queue Q is not empty, there are some pending paths that need to be traversed through recursive invocation of *breadthFirstVisit*. This procedure involves some degree of non-determinism, related to the order in which the paths to be explored are added to Q. In fact, different orders will change the *visited* state of transitions at different times, possibly resulting in different test sequences being added to the final test suite. This is accounted for by the random selection of the node s^e from set *next*, when $p + s^e$ is added to Q. Transition coverage is granted by BFV by construction. An adequacy criterion different from transition coverage (e.g., maximum test budget) may require multiple invocations of procedure *breadthFirstVisit*.

2.2 Random visit

```
proc randomVisit($n : Node, p : Path$) $\equiv$
  do
      if (randProb() $\leq RECURSE\_PROB \wedge$ succ[$n$] $\neq \emptyset$)
        then
            $s^e$ := randChoose(succ[$n$]);
            $p$ := randomVisit($s, p + s^e$);
      fi
      if ($p$ increases adequacy) then addToTestSuite($p$); fi
  od.
while adequacy criterion not satisfied do
      $p$ := randomVisit($startNode$, $\langle\rangle$);
od
```

Fig. 4. Test sequence derivation by random visit

Figure 4 shows the pseudocode of the random test sequence generation strategy (RAND: Random visit). Procedure *randomVisit* decides whether to add another event to the current test sequence or not in a stochastic way. With probability *RECURSE_PROB*, a randomly selected successor of the current node is added to the current event sequence and *randomVisit* is invoked recursively. When recursion is not activated, the generated test sequence is added to the test suite, if it increases the test suite adequacy level. Multiple random visits are performed, until the adequacy criterion (e.g., transition coverage) is satisfied.

The algorithm is clearly non deterministic. The algorithm parameter *RECURSE_PROB* determines the length of the generated event sequences. To produce event sequences with an average length equal to that observed in the execution traces, *RECURSE_PROB* can be set to *AVG_TRC_SZ/(1+AVG_TRC_SZ)*, where *AVG_TRC_SZ* is the average length of the logged event sequences.

3 N-Gram Based Test Sequence Derivation

N-gram language models are widely used in Natural Language Processing (NLP) [9]. A N-gram language model is a probabilistic language model where the probability that a word (an event in our case) e is preceded by a sequence of words (events) depends only on the last $N - 1$ words.

For example, given a training corpus of English sentences and N equals to 2 (bi-gram), the probability that the word “the” appears after the sentence “she is so beautiful that” is approximated with the probability that “the” follows “that”.

Using probabilistic models, knowledge about the N-gram statistics supports *word prediction*. In turn, word prediction is a key component used to address several NLP tasks, such as speech recognition, handwriting recognition, machine translation, spell correction, natural language generation, etc. In fact, NLP algorithms admit usually multiple sentence derivations and N-gram statistics can be used to select the most likely among the possible derivations.

```
proc ngramVisit($n : Node, p : Path$) $\equiv$
  do
     if (randProb() $\leq RECURSE\_PROB$)
       then
           $s^e :=$ ngramChoose(succ[$n$], suffix($p, N-1$));
           $p :=$ ngramVisit($s, p + s^e$);
     fi
     if ($p$ increases adequacy) then addToTestSuite($p$); fi
  od.
while adequacy criterion not satisfied do
     $p :=$ ngramVisit($startNode, \langle\rangle$);
od
```

Fig. 5. Test sequence derivation by N-gram probability

The problem with model based test sequence generation is somewhat similar. Among all possible event sequences that satisfy some adequacy criterion (e.g., transition coverage), only a subset represent *feasible* event sequences, i.e., event sequences that can be actually executed against the application under test. Infeasible sequences involve execution steps whose order is forbidden by the application under test or events whose valid inputs prevents the execution of a later subsequence. Avoiding the generation of infeasible event sequences is very similar to avoiding the derivation of unlikely sentences and N-gram statistics can be used in a similar way as in NLP to achieve such purpose. In fact, by generating event sequences that contain N-grams previously observed in real executions we increase the likelihood that such sequences will in turn be executable.

Figure 5 shows the pseudocode of the N-gram test sequence generation strategy (by NGRAM2, NGRAM3, etc., we indicate that N, the size of the tuples considered in the N-gram statistics, is respectively 2, 3, etc.). The procedure is similar to the random visit shown in Figure 4, the key difference being the way in which the next event to add to the event sequence is chosen. Instead of choosing which successor node to add randomly (i.e., according to a uniform probability distribution), the successor to add is chosen in accordance with the conditioned probabilities of the next events given the last $N-1$ events in the current path p:

$$ngramChoose(S, \langle e_1, \ldots, e_{N-1}\rangle) := s^e \in S \;\; with\ prob.\ P(e \mid e_1, \ldots, e_{N-1})$$

The conditioned probabilities $P(e \mid e_1, \ldots, e_{N-1})$ are estimated from the frequency of occurrence of the N-tuples $\langle e^{(1)}, e_1, \ldots, e_{N-1}\rangle, \ldots, \langle e^{(k)}, e_1, \ldots, e_{N-1}\rangle$ in the execution logs. Specifically, the choice (among $e^{(1)}, \ldots, e^{(k)}$) of the next event to add to the event sequence has a probability which is proportional to the frequency of occurrence of the respective tuples ($\langle e^{(1)}, e_1, \ldots, e_{N-1}\rangle, \ldots, \langle e^{(k)}, e_1, \ldots, e_{N-1}\rangle$).

When no N-tuple $\langle e, e_1, \ldots, e_{N-1}\rangle$ appears in the available execution traces, the next event e is selected randomly among the possible transitions outgoing from the current node n. In such cases the NGRAM strategy degenerates to the random strategy. This is expected to occur at increased frequency when N takes higher and higher values, since only a small fraction of the possible N-tuples will be represented in the observed traces. We can thus predict that the performance of NGRAM will converge to the performance of RAND as the value of the parameter N increases.

4 Case Studies

We have conducted two case studies to answer the following research questions:

- **RQ1 (Feasibility):** How many feasible test sequences are generated by the N-gram strategy, as compared to the graph visit and the random strategies?
- **RQ2 (Coverage):** What level of transition coverage is achieved by the N-gram strategy, as compared to the graph visit and the random strategies?
- **RQ3 (Test suite size):** How many test sequences are generated by the N-gram strategy, as compared to the graph visit and the random strategies?
- **RQ4 (Test case length):** What is the length of the test sequences generated by the N-gram strategy, as compared to the graph visit and the random strategies?

The first two research questions are key to validate the proposed approach. We conjecture that N-gram based test sequence generation will produce less infeasible test sequences (hence, higher coverage) than graph visit or random model traversal. With these two research questions we want to empirically assess whether our conjecture is confirmed or not by the experimental data.

The last two research questions deal with some interesting properties of the automatically generated test suites, namely, the number of test sequences they contain and their length. To make the comparison fair, we adopt the same adequacy criterion with all alternative test sequence generation strategies: transition coverage. All test suites produced by the various strategies will be transition coverage adequate and will contain only test cases that contribute to increasing such adequacy level.

4.1 Metrics

To address the four research questions listed above, we have collected the following metrics:

- **FEAS (RQ1):** Ratio between feasible test sequences and total number of test sequences generated by each test strategy.
- **COV (RQ2):** Ratio between covered transitions and total number of transitions in the model.
- **SZ (RQ3):** Number of test sequences in the test suite.
- **LEN (RQ4):** Average number of events in each test sequence of the test suite.

While metrics COV, SZ and LEN can be easily measured automatically, using tools, metrics FEAS requires human judgment, since it is not possible to automatically decide if a test sequence is feasible (i.e., whether it can be executed by providing proper input data) or not (the problem is undecidable in the general case). We manually defined a set of constraints for the subject applications to characterise the event sequences that can be legally submitted to and executed by the system under test.

4.2 Subjects

State	Abstraction
$n1$	[initial] *false*
$n2$	$s1 = 0 \wedge s2 = 0 \wedge s3 > 0$
$n3$	$s1 = 0 \wedge s2 = 0 \wedge s3 = 0$
$n4$	$s1 > 0 \wedge s2 = 0 \wedge s3 > 0$
$n5$	$s1 > 0 \wedge s2 = 0 \wedge s3 = 0$
$n6$	$s1 > 0 \wedge s2 > 0 \wedge s3 > 0$
$n7$	$s1 > 0 \wedge s2 > 0 \wedge s3 = 0$
$n8$	$s1 = 0 \wedge s2 > 0 \wedge s3 > 0$
$n9$	$s1 = 0 \wedge s2 > 0 \wedge s3 = 0$
$s1$	numInShopCart
$s2$	numInCompareCart
$s3$	numOfSelectedItems

Table 1. Abstraction functions used to infer the model of Flexstore

The applications under test are Flexstore and Cyclos. Flexstore[1] is an on-line shopping application developed by Adobe and made available from the companys web site to demonstrate the capabilities of their testing framework. It is a client-side application developed in Flex and run by the Flash plug-in. The application

[1] http://www.adobe.com/devnet/flex/samples/flex_store_v2.html

State	Abstraction
$n1$	[initial] *false*
$n2$	$s1 = true \wedge s2 = true$
$n3$	$s3 = null \wedge s4 = null \wedge s5 \neq null \wedge s6 = null \wedge s7 = null \wedge s8 = 0$
$n4$	$s3 = null \wedge s4 = null \wedge s5 \neq null \wedge s6 = null \wedge s7 \neq null \wedge s8 = 0$
$n5$	$s9 = true$
$n6$	$s3 = null \wedge s4 = null \wedge s5 \neq null \wedge s6 = null \wedge s7 \neq null \wedge s8 > 0$
$n7$	$s3 = null \wedge s4 = null \wedge s5 \neq null \wedge s6 \neq null \wedge s7 = null \wedge s8 = 0$
$s1$	LoggedIn
$s2$	PaymentReady
$s3$	typeRow
$s4$	customValuesRow
$s5$	trSchedulingType
$s6$	scheduling_singlePayment
$s7$	scheduling_multiplePayments
$s8$	paymentCounts
$s9$	End

Table 2. Abstraction functions used to infer the model of Cyclos

allows the user to browse a catalog of mobile phones and to focus on a subset of models by means of filters, such as price range, camera, tri-band, and video availability. The customer can select one or more models to perform comparisons among features. Eventually, the customer can put one or more phones in their shopping cart.

Cyclos[2] is a popular open source Java/Servlet Web Application, supporting e-commerce and banking. Its main features include: banking (e.g., payments, loans, brokering), e-commerce (e.g., advertising, member payments) and many others (e.g. access control, management). Cyclos is a quite large system. In our experiment we focused only on the payment feature of Cyclos.

We have obtained execution traces for both applications by navigating and exercising the various application features, according to a high level functional coverage criterion: all functionalities provided in the application menus have been executed with input data that a user would be typically expected to provide. The two applications have been instrumented so as to support trace collection. We collected 100 traces for Flexstore and 999 traces for Cyclos. In total, 3,000 events have been executed in Flexstore during trace production, while 5,965 events have been executed in Cyclos.

We have obtained a FSM model of each application by applying state-based abstraction to the execution traces. Specifically, we have used the model inference component [16] of the FITTEST[3] Integrated Testing Environment (ITE). The state abstraction functions used for model inference with Flexstore and Cyclos are shown respectively in Tables 1 and 2 (state variables are mapped to

[2] http://project.cyclos.org

[3] EU FP7 project n. 257574

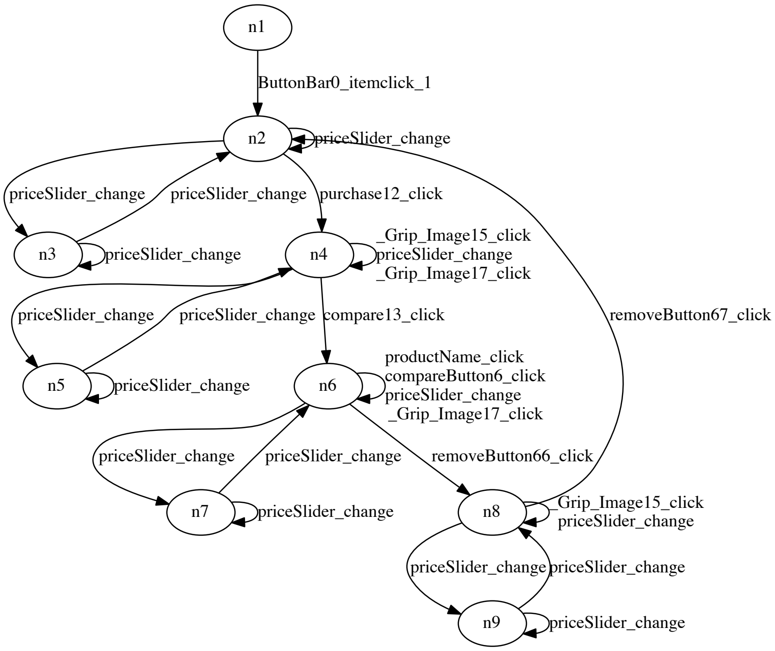

Fig. 6. Finite state model of Flexstore

application variables at the bottom of each table). The resulting FSM models are shown in Figure 6 and 7.

4.3 Procedure

The test sequence generation algorithms DFV, BFV, RAND and NGRAM, whose pseudocode is shown in Figures 2, 3, 4, 5, have been applied to the models inferred for Flexstore and Cyclos (shown in Figures 6 and 7). NGRAM has been applied with N = 2, 3, 4, using the same traces used for model inference to obtain the N-gram statistics needed by the algorithm. Since all these algorithms are non deterministic, each of them has been run 100 times. Results are averaged over the 100 runs. In all runs, each algorithm was executed with transition coverage set as the adequacy criterion to satisfy.

Metrics SZ and LEN are obtained by measuring the number of test cases and the number of events per test case in each test suite. Metrics COV has been obtained by means of a tool that visits the FSM models of the two applications based on the feasible input sequences in each test suite, keeping track of the covered transitions. Metrics FEAS has been measured based on two sets of constraints that define the legal sequences of actions that can be executed on

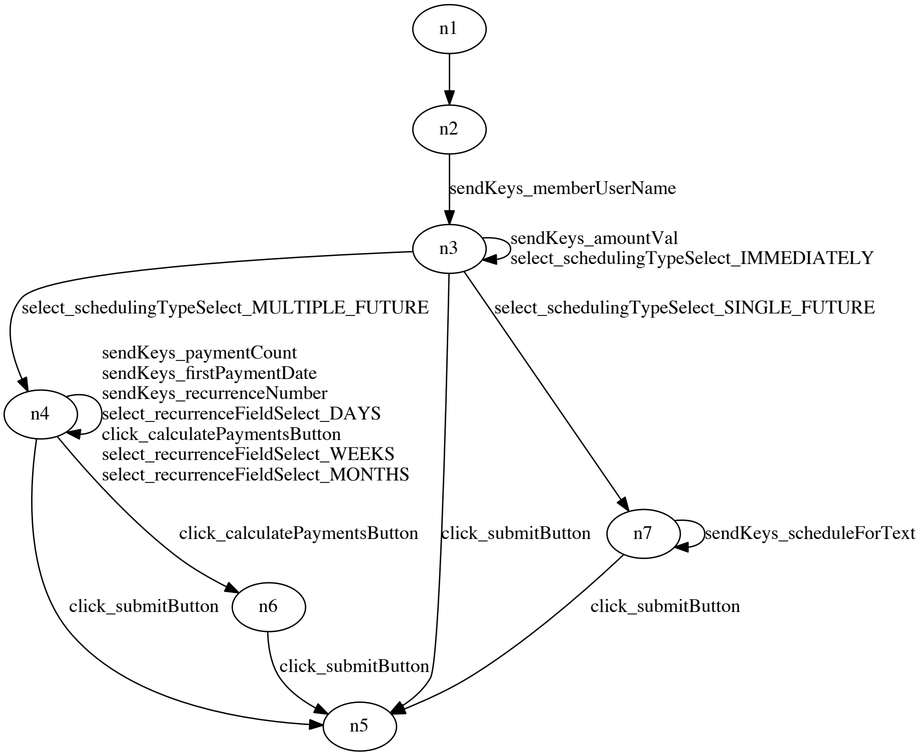
n1
n2
sendKeys_memberUserName
n3
sendKeys_amountVal
select_schedulingTypeSelect_IMMEDIATELY
select_schedulingTypeSelect_MULTIPLE_FUTURE
select_schedulingTypeSelect_SINGLE_FUTURE
sendKeys_paymentCount
sendKeys_firstPaymentDate
sendKeys_recurrenceNumber
select_recurrenceFieldSelect_DAYS
click_calculatePaymentsButton
select_recurrenceFieldSelect_WEEKS
select_recurrenceFieldSelect_MONTHS
n4
click_calculatePaymentsButton
click_submitButton
n7
sendKeys_scheduleForText
click_submitButton
n6
click_submitButton
click_submitButton
click_submitButton
n5

Fig. 7. Finite state model of Cyclos

the two applications. Test cases which contain event sequences violating such constraints have been marked as infeasible and have been subtracted from the count of the feasible sequences. They also do not contribute to metrics COV.

Feasibility constraints have the form: $[g]e$, indicating that event e can be triggered only if the guard condition g is true; otherwise, the event sequence being executed is deemed infeasible. An example of a constraint manually defined for Flexstore is the following:

$$[numOfSelectedItems > 0]compare13\_click$$

indicating that event *compare13_click* can be triggered only in a state where *numOfSelectedItems* is greater than zero. For Cyclos, an example of constraint is:

$$[amount > 0 \wedge (immediate \vee single \vee (multiple \wedge count > 0)]click\_submitButton$$

It is possible to submit a payment if the amount is greater than zero and the payment is immediate or single. Multiple payments require in addition that the recurrence count is greater than zero.

4.4 Results

	FEAS	COV	SZ	LEN
DFV	0.10	0.19	5.33	13.41
BFV	**0.53**	**0.37**	19.00	4.11
RAND	0.07	0.21	3.05	51.68
NGRAM2	0.49	0.48	3.41	52.83
NGRAM3	**0.72**	**0.51**	3.83	49.86
NGRAM4	0.39	0.44	3.93	51.58

p-value(NGRAM3, BFV, FEAS) = 1.793e-09
p-value(NGRAM3, BFV, COV) < 2.2e-16

Table 3. Results obtained for the Flexstore case study

	FEAS	COV	SZ	LEN
DFV	0.53	0.64	5.61	6.22
BFV	**0.64**	0.61	14.00	2.86
RAND	0.32	**0.63**	7.36	5.65
NGRAM2	**1.00**	**0.83**	5.61	6.98
NGRAM3	0.36	0.69	7.11	5.39
NGRAM4	0.36	0.67	7.22	5.64

p-value(NGRAM2, BFV, FEAS) < 2.2e-16
p-value(NGRAM2, RAND, COV) < 2.2e-16

Table 4. Results obtained for the Cyclos case study

The results of the experiment are shown in Tables 3 and 4. Let us consider the graph visit and random strategies first. On Flexstore, BFV achieved a relatively high feasibility and coverage score, while the other strategies performed substantially worse. On Cyclos, BFV achieved comparatively high feasibility, while on coverage DFV, BFV and RAND performed similarly, with RAND slightly higher than the others.

On both applications, the NGRAM based approach was superior to the other strategies in terms of both feasibility and coverage. On Flexstore the best performance was achieved by NGRAM3, while on Cyclos it was achieved by NGRAM2.

On Flexstore, the absolute difference of the best (NGRAM3) FEAS/COV values with respect to BFV is substantial (0.19 and 0.14) and statistically significant (very low p-values), according to the Wilcoxon test. Similar results hold for Cyclos. The difference between NGRAM2 and BFV/RAND on FEAS/COV is substantial (0.36 and 0.20) and statistically significant.

On Cyclos, NGRAM2 gave the best performance in terms of feasibility. NGRAM2 achieved the maximum possible FEAS score (1.0), as in this application bi-grams subsume the feasibility constraints. Correspondingly, coverage reached also a very high value (0.83). With tri-grams (NGRAM3) several bi-gram constraints are still met, but there are cases where no tri-gram is applicable (because there is no such triple in the collected execution traces), which results in a random selection of the next event. Coverage is correspondingly highest with NGRAM2, but it is still quite high with NGRAM3. It can be noticed that in both applications as N increases (from 2 to 3 and 4) the performance of the N-gram based methods tends to converge to that of random, both in terms of FEAS and COV, as expected.

In summary, we can positively answer RQ1 and RQ2: *N-gram based test sequence generation produces a higher proportion of feasible event sequences and achieves higher coverage than graph visit or random approaches.*

In terms of test suite size and test case length (RQ3, RQ4), we can notice that BFV produced test suites with a lot of very short test cases (19/14 test cases for Flexstore/Cyclos, each containing on average 4.11/2.86 events). The other methods are not very different from each other. NGRAM converges to the size and length of RAND as N increases, as expected. DFV produced more/shorter test cases than NGRAM (and RAND) on Flexstore, while on Cyclos the opposite holds: NGRAM produced a bit more test cases per test suite, each with slightly more events, than DFV, but in both cases the difference is just marginal. Overall, test suite size and test case length associated with NGRAM seem reasonable for the applications under test and in line with the values obtained from the other considered methods.

5 Related Works

FSM models have been exploited to generate test cases in several existing works [13, 15, 19–22, 16]. In case of a design-time model, abstract test cases can be used to check how the expected behaviours (coming from the model) have been implemented in the application [19, 20]. Hence, the role of behaviour specifications is to provide abstract test cases, test oracles, and a measure of the test adequacy [21]. Instead, the role of an inferred model is mainly to provide abstract test cases (e.g., [22, 19, 16]), to be instantiated in executable test cases by adding inputs and, eventually, test oracles (e.g., [16]). This instantiation is an expensive activity and the presence of infeasible test cases demand for substantial effort and knowledge from the tester's side, who is required to recognise them and filter them out. To the best of our knowledge, this is the first work which tries to address the problem of test case infeasibility by adopting a test case derivation strategy (based on the N-gram statistics) that aims at increasing the likelihood of feasibility and correspondingly the level of coverage actually achieved.

6 Conclusions and Future Work

In the context of model based testing, we have proposed a test sequence derivation strategy based on the N-gram statistics. To avoid (or limit) the generation of infeasible event sequences, the next event to add to a sequence is selected based on the frequency of occurrence of the N-tuple of events that ends with the event considered for addition. Experimental results show that the proposed approach generates a higher proportion of feasible test sequences than graph visit and random strategies. In turn, the higher proportion of feasible sequences increases the level of model coverage reached by the test suite.

In our future work we plan to extend the empirical study to additional subject applications. We intend to study empirically the role of the training material (i.e., the execution traces) used to learn the N-gram statistics, which is key for a successful application of the approach. We also plan to investigate the possibility of further increasing the coverage achieved by hybridising and extending the N-gram based method (e.g., through smoothing [9]).

Acknowledgments

This work has been funded by the European Union FP7 project FITTEST (grant agreement n. 257574).

References

1. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: Proc. of the International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (co-located with ASE'07). WEASELTech '07, New York, NY, USA, ACM (2007) 31–36
2. Mariani, L., Marchetto, A., Nguyen, C.D., Tonella, P., Baars, A.I.: Revolution: Automatic evolution of mined specifications. In: Proc of the 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE). (2012) 241–250
3. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: Proceedings of the 2006 international workshop on Dynamic systems analysis. WODA '06, New York, NY, USA, ACM (2006) 17–24
4. Biermann, A., Feldman, J.: On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. on Computers **21**(6) (1972)
5. Krka, I., Brun, Y., Popescu, D., Garcia, J., Medvidovic, N.: Using dynamic execution traces and program invariants to enhance behavioral model inference. In: ICSE (2). (2010) 179–182
6. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: 30th International Conference on Software Engineering (ICSE), IEEE Computer Society (2008)

7. Tonella, P., Marchetto, A., Nguyen, C.D., Jia, Y., Lakhotia, K., Harman, M.: Finding the optimal balance between over and under approximation of models inferred from execution logs. In: Proc of the Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST). (2012) 21–30
8. Hedley, D., Hennell, M.A.: The causes and effects of infeasible paths in computer programs. In: Proceedings of the 8th international conference on Software engineering. ICSE '85, Los Alamitos, CA, USA, IEEE Computer Society Press (1985) 259–266
9. Jurafsky, D., Martin, J.H.: Speech and Language Processing: An introduction to speech recognition, computational linguistics and natural language processing. Pearson Prentice Hall (2007)
10. Pezzè, M., Young, M.: Software Testing and Analysis: Process, Principles and Techniques. John Wiley and Sons, USA (2007)
11. Shafique, M., Labiche, Y.: A systematic review of model based testing tool support. Technical Report Technical Report SCE-10-04, Carleton University, Canada (2010)
12. Briand, L.C., Labiche, Y.: A UML-based approach to system testing. In: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, London, UK, UK, Springer-Verlag (2001) 194–208
13. Kim, Y., Hong, H., Bae, D., Cha, S.: Test cases generation from UML state diagrams. Software, IEE Proceedings - **146**(4) (1999) 187 –192
14. Turner, C.D., Robson, D.J.: The state-based testing of object-oriented programs. In: Proc. of the Conference on Software Maintenance (ICSM), Montreal, Canada, IEEE Computer Society (1993) 302–310
15. Yuan, X., Memon, A.M.: Using GUI run-time state as feedback to generate test cases. In: Proc. the International Conference on Software Engineering (ICSE), Washington, DC, USA, IEEE Computer Society (2007) 396–405
16. Marchetto, A., Tonella, P., Ricca, F.: State-based testing of ajax web applications. In: Proc. of IEEE International Conference on Software Testing (ICST), Lillehammer, Norway (2008) 121–131
17. Andrews, A., Offutt, J., Alexander, R.: Testing Web Applications by Modeling with FSMs. Software and System Modeling, Vol 4, n. 3 (2005) 326–345
18. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: Proc. of the International Workshop on Dynamic Analysis (WODA), Shangai, China (2006) 17–24
19. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: Proceedings of the 19th international symposium on Software testing and analysis. ISSTA '10, New York, NY, USA, ACM (2010) 85–96
20. Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In France, R., Rumpe, B., eds.: UML 99 : The Unified Modeling Language. Volume 1723 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1999) 76–76
21. Stocks, P., Carrington, D.: A framework for specification-based testing. IEEE Trans. Softw. Eng. **22** (1996) 777–793
22. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: Extracting finite-state models from java source code. In: Proceedings of the International Conference on Software Engineering. (2000) 439–448
23. Nguyen, C.D., Marchetto, A., Tonella, P.: Combining model-based and combinatorial testing for effective test case generation. In: Proc of the ACM International Symposium on Software Testing and Analysis (ISSTA). (2012) 100–110