An Automata-Based Generation Method for Combinatorial Sequence Testing of Finite State Machines

Andrea Bombarda
Department of Engineering
University of Bergamo
Bergamo, Italy
andrea.bombarda@unibg.it

Angelo Gargantini
Department of Engineering
University of Bergamo
Bergamo, Italy
angelo.gargantini@unibg.it

***Abstract*—Combinatorial Interaction Testing has been applied to event-driven software systems by using as test suite a set of sequences of inputs in desired combinations. This is generally called *combinatorial sequence testing* (CST). CST requires possibly new system models from which tests are generated and new test generation methods (or an adaptation of the classical ones). Finite State Machines (FSMs) can easily represent event-based systems where certain inputs are valid only in some states and such constraints can be represented by the incompleteness of the FSM. In this paper, we propose an approach to CST where tests are generated from FSMs which are represented by automata together with test requirements. First, automata can be used to check if test sequences contain invalid inputs. We propose three methods to repair tests with invalid inputs. Moreover, we can directly embed into automata the system constraints over the inputs during generations, to generate only valid test sequences. We compare our automata-based method with the standard approach of Sequences Covering Arrays (SCAs) that produces a set of sequences, all with the same length, composed by the permutation of all the events supported by the system. We found that generating only valid tests from automata provides several advantages w.r.t. repairing tests and SCAs.**

***Index Terms*—Test Sequence Generation; Sequencing Constraint; T-way Sequence Coverage; Sequence Testing; Event-based Testing; Combinatorial Testing; Constrained Combinatorial Testing**

I. Introduction

Combinatorial interaction testing (CIT) has been an active area of research for many years, since it has proven to be very effective to test complex systems with multiple input parameters. In [23] the authors count several research groups that actively work on CIT area and many other recent groups and tools are not considered in that paper, while in [16] a lot of algorithms and tools available for CIT are analyzed. Recently, the CIT approach has proven to be effective not only to test a system by varying the values of its input parameters, but also to test combinations of events in event-driven software. In this case, the extension of CIT to sequences of events is also referred in the literature as *combinatorial sequence testing* (CST) [10], [13], [18], [27]. CST can be successfully applied to test event-driven software or systems ([3]–[5], [15], [21], [28], [35]). A common technique for CST consists in exploiting *sequence covering arrays* (SCA) for testing. Given a set of events E, a SCA of strength t is a set of permutations of all the events in E such that each sequence of t distinct elements of E is a subsequence of at least one of the permutations.

Among system inputs there are typically several constraints, and the failure revealing ability of CIT methods might be significantly reduced if the system has to comply with constraints and the test suite generator does not take them into account. For this reason, several approaches are extended to Constrained Combinatorial Testing [17], but all of them are focused on input testing. For event-driven software there are constraints over the events that can be used as inputs during testing and, also in this case, considering the constraints over the events during generation can increase the efficiency of testing. For instance, a SUT may require that a given event $read$ must appear after another event $open$ and if a test does not meet this constraint, the test is invalid, and it cannot be applied.

In this paper, we present a method that can be used to generate test sequences for the events of incomplete Finite State Machines (FSMs), by taking into account also their constraints. Our method exploits the representation of FSMs using the automata notation. We have decided to focus on incomplete FSMs because they represent the situation in which the constraints of events are stricter, since a particular event may not be defined in some state. Moreover, we supported FSMs in the form of Mealy machines (Fig. 1), that are a rather general implementation of FSMs. We introduced also three different approaches to repair invalid test sequences, generated without taking into account the constraints imposed by the FSM of the system.

We have discovered that using our method to generate sequences, complying the constraints imposed by the FSM of the system, can lead to a greater number of valid sequences and to a better coverage of both of states, event tuples and transitions.

The paper is structured as follows. In Sect. II we provide some necessary background about the combinatorial testing,

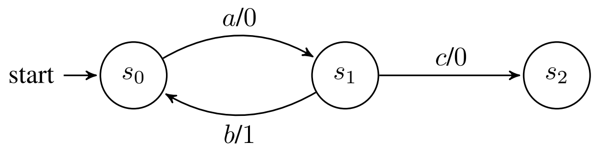

$$\begin{tikzcd} \text{start} \arrow[r] & s_0 \arrow[r, bend left, "a/0"] & s_1 \arrow[l, bend left, "b/1"] \arrow[r, "c/0"] & s_2 \end{tikzcd}$$

Fig. 1. Example of Mealy machine. $a/0$ means that when the input is a, the FSM produces the output 0 and it moves to the target state shown by the arrow.

the FSMs and the constraints that we have to satisfy in our System Under Test (SUT). Our automata-based sequence generation method is explained in Sect. III and its application is evaluated in Sect. IV. Sect. V reviews some works related to application of combinatorial sequence testing and constrained combinatorial testing, and Sect. VI concludes the paper.

II. Background

In this paper we generate sequences of events to test event-driven software [30]. This kind of software can be well described by using Finite State Machines and, in particular, Mealy machines since they allow to manage not only the states and the input events, but also output events.

Definition 1 (Mealy machine)**.** *A* Mealy machine *F is a 6-tuple $(S, s_0, \Sigma, \Lambda, T, G)$ in which:*

- *S is a finite set of states.*
- *$s_0 \in S$ is the initial state of the machine F.*
- *Σ is a finite set that represents the input alphabet.*
- *Λ is a finite set that represents the output alphabet.*
- *$T : S \times \Sigma \rightarrow S$ is the transition function that maps pairs of a state and an input symbol to the corresponding next state.*
- *$G : S \times \Sigma \rightarrow \Lambda$ is the output function that maps pairs of a state and an input symbol to the corresponding output symbol.*

Since we aim to deal with real systems, we have to define our combinatorial sequence generation method for *incomplete* FSMs.

Definition 2 (Complete and incomplete FSM)**.** *Given a FSM $F(S, s_0, \Sigma, \Lambda, T, G)$ we say that F is a* complete machine *iff for all $s \in S$ and for all $e \in \Sigma$ the transition function $T(s, e)$ and the output function $G(s, e)$ are defined. Contrariwise, we say that F is a* incomplete machine *iff there exist a state $s \in S$ and an event $e \in \Sigma$ for which the transition function $T(s, e)$ is not defined (neither is $G(s, e)$).*

Example 1. The FSM in Fig. 1 is an incomplete machine, because T is not defined for the input symbols b and c in the state s_0, for the input symbol a in the state s_1, and for the symbols a, b, and c in the state s_2.

In the following pages we will refer to input symbols as *events* to use the same nomenclature as the one used in event-driven software.

A. Combinatorial sequence testing of FSMs

While in classical combinatorial testing we are interested in covering the interaction among a fixed set of inputs [19], each with a given set of possible values, in *combinatorial sequence testing* (CST) [18] of FSMs we focus on covering the interaction of inputs taken from a unique set (the input alphabet) but provided to the machine in different orders. This requires the redefinition of *test* as a sequence of inputs of variable length (in other approaches also for CST tests are still organized in Sequence Covering Arrays and they have all the same length). In our approach, a test is a finite sequence of events $(e_1, e_2, \ldots, e_n)$ all belonging to Σ.

Definition 3 (Combinatorial sequence coverage)**.** *We say that a test suite achieves the t-way combinatorial sequence coverage iff for any tuple of t inputs there exists a test sequence in which these t inputs occur in any possible order (allowing interleaving extra inputs among the elements of the tuple).*

With the standard pairwise CIT, a test suite covers for each pair of input parameters all the possible combinations of values. Pairwise CIT can be extended to t-wise CIT when tuples of length t are considered instead of simple pairs. In our case, we want to generate sequences of events covering each tuple of t events.

Most of the event-driven software can be represented using an incomplete FSM, since in some states, some events cannot be fired. This representation implicitly defines some *constraints* on the FSM, meaning that only some test sequences are valid, while others are not.

Definition 4 (Valid test sequence)**.** *Given a FSM $F(S, S_0, \Sigma, \Lambda, T, G)$ as per Definition 1, let $ts = (e_1, e_2, \ldots, e_n)$ be a test sequence composed of a sequence of n events. Assume that ts^i is the list of the events in ts starting from e_1 to e_i and $s(ts^i)$ is the state reached starting from the initial state s_0 by applying all the events in ts^i. We call ts a* valid test sequence *iff, for all $e_i \in ts$, e_i can be fired starting from the state $s(ts^{i-1})$, i.e., $T(s(ts^{i-1}), e_i)$ and $G(s(ts^{i-1}), e_i)$ are both defined.*

Example 2. Let's suppose to be in the initial state s_0 of the example in Fig. 1 in which only the event a can be fired. A test sequence (b, b, a) is a *invalid test sequence*, because the event b is not defined in the initial state. Contrariwise, for the same example, the test sequence (a, b, a) is a *valid test sequence*.

For this reason, for an incomplete FSM, we may be unable to cover all the tuples of events because some of them can be covered only by sequences that are invalid.

Example 3. In the example of Fig. 1, the pair of c followed (also non immediately) by a cannot be covered by any valid test sequence.

III. Combinatorial Sequences Generation

Having introduced what we mean with *combinatorial sequence testing* and *coverage*, we can now introduce our sequence generation method. One could extend classical combinatorial testing algorithms in order to generate SCAs, and this has been done for example in [18]. However, in our approach we are not bound to have all the tests of the same length (usually the number of the events), so classical methods that build covering arrays may be not well suited, and we decided to devise an automata-based approach.

First, we introduce an automaton representing a t-wise permutation of t events.

Definition 5 (T-wise automaton)**.** *Given a permutation p of t events $(e_1, e_2, \ldots, e_t)$ the automaton built as in Fig. 2 is called t-wise automaton. We call* `automaton(p)` *the function that builds the t-wise automaton that represents the tuple p.*

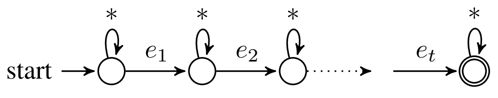


Fig. 2. Example of automaton for the recognition of the sequence $(e_1, e_2, \ldots, e_t)$

A t-wise automaton $\mathcal{A}$ can be used to check if a sequence s covers the tuple it represents: if s is accepted by $\mathcal{A}$, than the tuple is covered.

If there are n events, there exist $\frac{n!}{(n-t)!}$ permutations and hence t-wise automata.

Exploiting the operations among automata we can build a test suite for CST as shown in Alg. 1.

The algorithm is a typical one-test-at-the-time test generator. At the beginning it builds an empty automaton A and then it tries to add many t-wise automata to it in a random order. We say that it *collects* multiple t-wise automata in a unique automaton. At the end any string that can be derived form A is a test that covers all the permutations from which the t-wise automata are built. In particular we use the function `string(A)` that returns the shortest string accepted by the automaton A. We allow the user to set a limit N of automata to be collected together: in this way the user can favor few long sequences (high N) or many short sequences (low N). The effects of the variation of the parameter N will be analysed in Sect. IV.

This approach is similar to the one presented in [6] where a logical context and an SMT solver is used to collect tuples in order to generate tests for classical constrained combinatorial interaction testing.

This standard algorithm, however, could generate invalid test sequences. So, we have devised three different approaches to repair invalid tests:

- **Reject_not_valid (REJ)**: if a sequence contains an event that is invalid at the time in which it is applied, the whole sequence is rejected.
- **Stop_at_error (STP)**: if a sequence contains an event that is invalid at the time in which it is applied, the sequence is executed only until the error is reached. The following events are not tested.

Algorithm 1 Algorithm for test generation

```
Require: $I$ the set of events
Require: $t$ the strength of the tests
Require: $N$ the max number of tuples for each test sequence
Ensure: $TS$ the test suite for CST
  $T \leftarrow$ t-permutations of $I$
  $TS \leftarrow \emptyset$
  $i \leftarrow 0$
  $A \leftarrow$ empty automaton
  while $T \neq \emptyset$ do
      $p \leftarrow$ a random element in $T$
      $a \leftarrow automaton(p)$
      if $a \cap A \neq \emptyset$ then
          $A \leftarrow a \cap A$
          $T \leftarrow T - \{p\}$
          $i \leftarrow i + 1$
          if $i \geq N$ then
              ADDTEST($TS$,$A$)
              $i \leftarrow 0$
          end if
      end if
  end while
  ADDTEST($TS$,$A$)

  procedure ADDTEST($TS$,$A$)
      $TS \leftarrow TS + string(A)$
      $A \leftarrow$ empty automaton
  end procedure
```

- **Skip_error (SKP)**: if a sequence contains an event that is invalid at the time in which it is applied, the single event is skipped, and the following events are executed.

A. Generation of only valid tests

In order to avoid the generation of invalid tests, we modify the algorithm as presented in Alg. 2. In this new version of the generation algorithm, called CNST, we collect the t-wise automata not starting from an empty automaton but from the automaton that accepts only valid sequences of inputs for the FSM under test (lines 4 and 24). Moreover, the FSM may never accept a given permutation of t events, and in this case this tuple is said *infeasible*.

Example 4. For the Mealy machine in Fig. 1, the tuple $a{-}c{-}b$ is infeasible, because the input symbol b cannot be accepted after the first two symbols.

Since there is no valid test that covers infeasible tuples, it is important to detect and discard them form the requirements. This is done in the algorithm at line 16 where it checks if a tuple p that cannot be collected with the current automaton, can instead be collected with the automaton containing only the constraints of the FSM ($automaton(F)$). If p cannot be collected even with the $automaton(F)$, then it means that the tuple is infeasible.

Algorithm 2 Algorithm for test generation

```
Require: $I$ the set of events
Require: $F$ the finite state machine
Require: $t$ the strength of the tests
Require: $N$ the max number of tuples for each test sequence
Ensure: $TS$ the test suite for CST
 1: $T \leftarrow$ t-permutations of $I$
 2: $TS \leftarrow \emptyset$
 3: $i \leftarrow 0$
 4: $A \leftarrow automaton(F)$    ▷ init A with the FSM automaton
 5: while $T \neq \emptyset$ do
 6:     $p \leftarrow$ a random element in $T$
 7:     $a \leftarrow automaton(p)$
 8:     if $a \cap A \neq \emptyset$ then
 9:         $A \leftarrow a \cap A$
10:         $T \leftarrow T - \{p\}$
11:         $i \leftarrow i + 1$
12:         if $i \geq N$ then
13:             AddTest($TS$,$A$)
14:             $i \leftarrow 0$
15:         end if
16:     else if $automaton(F) \cap a = \emptyset$ then ▷ $p$ is infeasible
17:         $T \leftarrow T - \{p\}$
18:     end if
19: end while
20: AddTest($TS$,$A$)
21:
22: procedure AddTest($TS$,$A$)
23:     $TS \leftarrow TS + string(A)$
24:     $A \leftarrow automaton(F)$    ▷ init A with the FSM automaton
25: end procedure
```

Example 5. Fig. 3 shows the collecting operation between an automaton representing the SUT and the pair $1 - 0$. As the figure shows, the resulting automaton can contain much more states and transitions than the original one.

B. Monitoring

To further optimize the generation, we can perform *monitoring* which consists in checking if a test generated for a set of tuples accidentally covers other tuples as well. Algorithm 3 implements monitoring works: once a test is generated, all the tuples that are still not covered are checked against the test. If a tuple is covered, then its is discarded. To check if a tuple is covered by a test, we can check if the automaton representing that tuple accepts the test sequence. Note that while the collecting of Alg. 2 can be expensive, since it requires the operation of intersection among automata, monitoring is generally much faster since acceptance is easily computed.

Algorithm 3 Monitoring

```
 1: procedure AddTest($TS$,$A$,$T$)
 2:     $test \leftarrow string(A)$
 3:     for all $t \in T$ do
 4:         if $isAccepted(test, automaton(t))$ then
 5:             $T \leftarrow T - \{t\}$
 6:         end if
 7:     end for
 8:     $TS \leftarrow TS + test$
 9:     $A \leftarrow automaton(F)$
10: end procedure
```

IV. Method evaluation

We use the `dk.brics.automaton` [22] Java package to build automata representing the FSM of the whole system and each tuple of events. The code we have used to execute the method evaluation can be found in the following public repository: https://github.com/fmselab/FiniteStateMachineCombinatorial.

To evaluate our automata-based generation method for sequence combinatorial testing of Finite State Machines, we have tested and analysed the coverage of the pair-wise test sequences over four different systems described using FSMs (see Table I): the IEEE 11073 PHD's communication model (already analysed and tested with different approaches in [4] and [33]), a pattern matching system (for the recognition of the regular expression 01[0*]1), a simple elevator and a vault that can be unlocked only by the combination "12345". As shown in Table I, we use different values of N among the models since the PHD communication model is more complex than the others, and the intersection operation times out with N greater than 10 for it.

We have represented all the benchmark systems using the SMC (State Machine Compiler) standard language [26] that allows to express the behavior and generate the classes in a lot of different languages, by using the included compiler. Listing 1 shows an example of the SMC description of the *vault* benchmark, where char1...char5 are the events fired by the FSM when a number is pressed.

The results of the evaluation of our methods, by executing the test generation process 10 times for every combination of options, are reported in Table II.

In particular, we are interested in answering the following research questions:

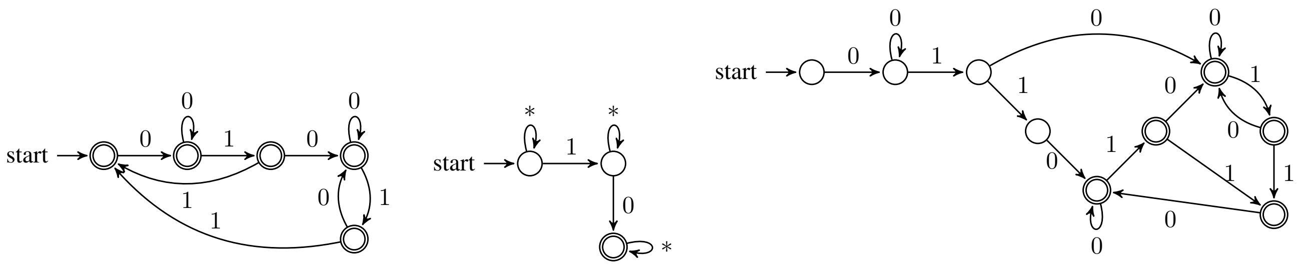
RQ1 How does the sequence generation time correlate with the size of the system, depending on the method?

RQ2 How the CNST method impacts the number of valid sequences and coverage w.r.t. the other methods (REJ, SKP, and STP)?

RQ3 How does the monitoring optimization influence the coverage of the sequences?

RQ4 How does the number of pairs covered by the sequences correlates with the value chosen for the parameter N?

RQ5 How does the sequence generation time correlate with the value chosen for the parameter N?



(a) Automaton of the system (b) Automaton of the pair $1-0$ (c) Intersection among the two previous automatons

Fig. 3. Intersection process among automata for the pattern recognition system

Listing 1. Example of SMC description of the vault benchmark

```
%class Vault
%package examples

%start MainMap::Idle
%map MainMap

%%
Idle
{
   char1   First_number { no_response(); }
}
First_number
{
   char2   Second_number         { no_response(); }
}
Second_number
{
   char3   Third_number { no_response(); }
}
Third_number
{
   char4   Fourth_number         { no_response(); }
}
Fourth_number
{
   char5   unlocked       { no_response(); }
}
%%
```

TABLE I
BENCHMARKS

	PHD Communication Model	Pattern Recognition	Elevator	Vault
# Automata per test sequence (N)	10	20	20	20
# Transitions	65	9	6	5
# States	5	5	4	6
# Events	23	2	8	5
# Event pairs	529	4	64	25
# Valid event pairs	484	4	25	10
# Event triples	12,167	8	512	125
# Valid event triples	10,648	8	125	10

RQ6 How does the total length of sequences correlate with

TABLE II
METHOD EVALUATION (PAIRWISE TESTING)

Benchmark	Monitoring	Method	# Seq.	Max. Len.	Min. Len.	Avg. Len.	Tot. Len.	# Valid Seq.	# Cov. pairs	# Cov. states	# Cov. trans.	Gen. t [s]
PHD	NO	CNST	41	20	11	17	708	41	484	5	51	428.40
PHD	NO	SKP	45	20	2	15	693	0	270	5	39	135.60
PHD	NO	REJ	45	19	2	15	701	0	0	0	0	144.24
PHD	NO	STP	45	20	2	15	692	0	49	2	12	150.80
PHD	YES	CNST	41	21	15	17	723	41	484	5	51	474.92
PHD	YES	SKP	45	20	2	15	686	0	271	5	39	185.08
PHD	YES	REJ	45	18	2	15	695	1	1	1	2	131.00
PHD	YES	STP	45	18	2	15	701	0	55	3	16	168.05
Pattern rec.	NO	CNST	1	4	4	4	4	1	4	5	4	0.07
Pattern rec.	NO	SKP	1	4	4	4	4	1	4	5	4	0.06
Pattern rec.	NO	REJ	1	4	4	4	4	1	4	5	4	0.06
Pattern rec.	NO	STP	1	4	4	4	4	1	4	4	3	0.07
Pattern rec.	YES	CNST	1	4	4	4	4	1	4	5	4	0.07
Pattern rec.	YES	SKP	1	4	4	4	4	1	4	5	4	0.06
Pattern rec.	YES	REJ	1	4	4	4	4	1	4	5	4	0.07
Pattern rec.	YES	STP	1	4	4	4	4	1	4	4	3	0.07
Elevator	NO	CNST	1	12	12	12	12	1	25	4	6	0.55
Elevator	NO	SKP	3	13	11	12	36	0	14	4	6	74.33
Elevator	NO	REJ	3	13	12	12	37	0	0	0	0	65.50
Elevator	NO	STP	3	13	11	12	36	0	0	2	1	79.08
Elevator	YES	CNST	1	12	12	12	12	1	25	4	6	0.58
Elevator	YES	SKP	3	14	11	12	36	0	8	4	5	62.25
Elevator	YES	REJ	3	13	11	12	37	0	0	0	0	99.37
Elevator	YES	STP	3	12	10	11	34	0	3	4	4	87.16
Vault	NO	CNST	1	5	5	5	5	1	10	6	5	0.21
Vault	NO	SKP	2	10	5	7	15	1	10	6	5	2.43
Vault	NO	REJ	2	10	4	7	14	0	0	0	0	2.10
Vault	NO	STP	2	10	5	7	15	0	6	5	4	1.87
Vault	YES	CNST	1	5	5	5	5	1	10	6	5	0.21
Vault	YES	SKP	2	9	5	7	14	0	10	6	5	1.40
Vault	YES	REJ	2	9	5	7	14	0	0	0	0	1.80
Vault	YES	STP	2	10	4	7	14	0	10	6	5	1.50

the value chosen for the parameter N?

RQ7 Is our method better than the standard sequence generation method based on SCAs?

A. RQ1: Sequence generation time and system size

By observing the *generation time*[1] of the sequences in Table II, we can see that for small systems (such as the elevator) the

[1]Experiments have been run on a computer with 14GB of RAM and a Intel® Core™ i5-750 CPU

TABLE III
EVALUATION OF THE RESULTS OBTAINED WITH DIFFERENT GENERATION METHODS

	% **Valid Seq.**	% **Pairs Cov.**	% **States Cov.**	% **Transitions Cov.**
CNST	100.00	100.00	100.00	77.65
SKP	3.92	55.50	100.00	62.94
REJ	2.94	0.86	27.50	5.88
STP	1.96	12.52	75.00	28.24

TABLE IV
COVERAGE WITH AND WITHOUT MONITORING DEPENDING ON THE GENERATION METHODS

	CNST	**SKP**	**REJ**	**STP**
No monitoring	92.55%	73.50%	10.16%	33.27%
Monitoring	92.55%	72.79%	12.67%	43.90%

time required by CNST is much smaller than the time required by the others. The reason is that repairing the sequences significantly takes more time than the sole generation time. Contrariwise, for systems that have many events, CNST is the slowest since the generation of the sequences by complying the constraints of the FSM requires more time than the repairing of the sequences. In this case, building the intersection among automata is time consuming, since they must contain the system constraints from the beginning. However, CNST leads to better results in terms of coverage as discussed below.

B. *RQ2: Coverage and valid sequences with* CNST

As can be seen from the results in Table III, our method CNST, that generates test sequences following the constraints imposed by the FSM of the system, leads to better (or equal) results than the other approaches:

- The percentage of valid sequences is higher. In many cases, other methods do not produce valid sequences. In those cases, we must repair the sequences (with one of the three proposed approaches) to still perform testing.
- The overall coverage (event pairs, states and transitions) is higher or the same for CNST compared to the other methods, because we can execute all the sequences since they contain only valid events.

Note that the pairs coverage is computed only over the number of feasible pairs because some of them cannot be covered due to the constraints imposed by the system.

C. *RQ3: Monitoring*

By comparing the results obtained without using the monitoring optimization and the ones obtained with the monitoring optimization (Table IV) we can see that the methods that involve the repairment of the sequences generally have a better or equal coverage when the monitoring is executed. This is reasonable because without monitoring we have more sequences that can fail and, in some cases, when the test sequence is invalid we have to stop its execution before its termination. Also considering all the methods together (Table V), the monitoring optimization always produces better results.

Moreover, in these experiments, the monitoring optimization has shown to be not time consuming, so we expect that it is a good choice to apply it.

TABLE V
COVERAGE WITH AND WITHOUT MONITORING - AVERAGE AMONG BENCHMARKS AND GENERATION METHODS

	% **Pairs Cov.**	% **States Cov.**	% **Transitions Cov.**
No monitoring	42.25	72.5	42.35
Monitoring	42.69	78.75	45.00

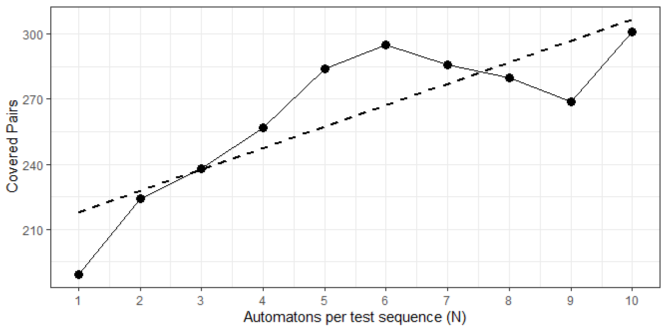


Fig. 4. Number of pairs covered with different values for the parameter N when the SKP method is used

D. *RQ4: Correlation between the number of covered pairs and N*

When the sequence repairing process is used, the number of the pairs covered is influenced also by the value chosen for the parameter N (number of automata per batch). Figure 4 (in the case of the PHD communication model) shows that for the SKP repairment method, the number of covered pairs has a growing trend with increasing N. This happens because if we have long sequences, we can include into them more pairs and, since we skip the events that are invalid, we can cover more pairs. On the other hand, Figure 5 (in the case of the PHD communication model) shows that for the STP repairment method, the number of covered pairs has a decreasing trend with the growth of N because having long sequences means that, when an invalid event is reached, we stop the execution of the whole sequence, so we do not execute a lot of events. A similar behavior can be observed by using REJ. Contrariwise, if the CNST method is used, the number of covered pairs remains constant when N varies, because all the pairs that are added in a test sequence satisfy all the constraints. This means that, for SKP method, a big N can improve the coverage while for STP and REJ it is better to have many short tests.

E. *RQ5: Correlation between generation time and N*

The tester can arbitrarily choose the value of N depending on the generation and repairment method chosen but nevertheless, it is important to consider that the sequence generation

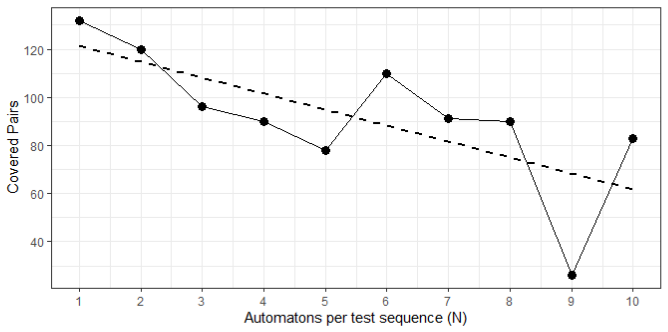


Fig. 5. Number of pairs covered with different values for the parameter N when the STP method is used

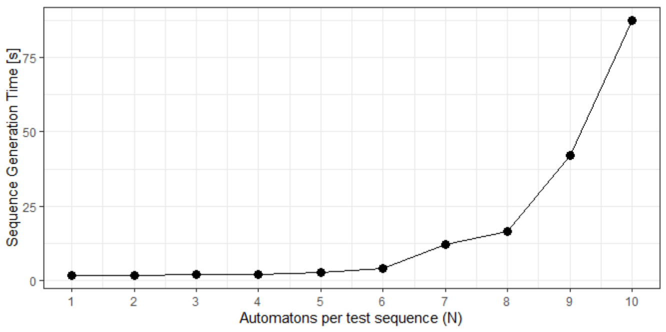


Fig. 7. Sequence generation time $[s]$ with different values for the parameter N when the CNST and pairwise testing are used

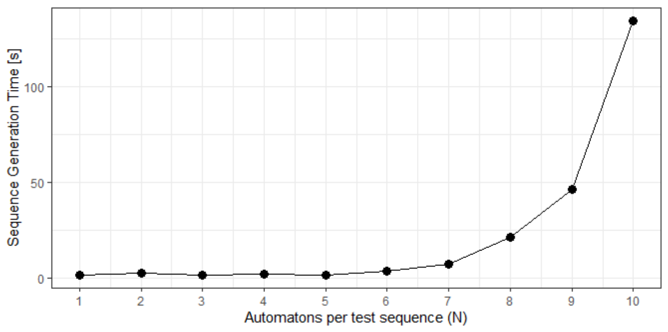


Fig. 6. Sequence generation time $[s]$ with different values for the parameter N when the STP and pairwise testing are used

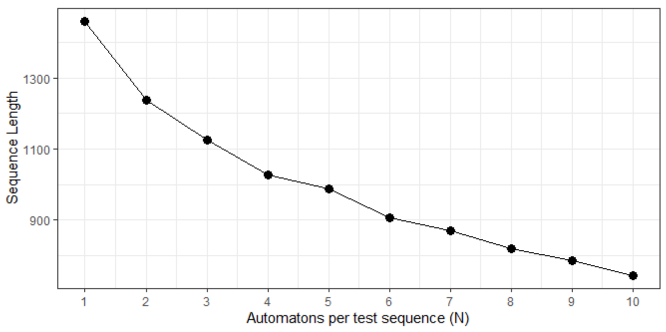


Fig. 8. Total sequences length with different values for the parameter N when the CNST method and pairwise testing are used

time increases exponentially with the increment of N, because the intersection of N automata usually requires much more time than the intersection of $N-1$ automata, especially for rather high value of N. Fig. 6 shows the correlation between generation time and N when the constraints are not considered in the generation phase (in the example the STP method is used on the PHD benchmark). Even when considering the constraints during the generation phase (CNST method on the PHD benchmark) the correlation between N and the generation time is exponential (see Fig. 7). However, increasing N leads to smaller test suites, as shown in the following RQ.

F. RQ6: Correlation between the length of the sequences and N

Increasing the value of the number of automata per test (N) obviously leads to a decrease of the number of sequences. Moreover, increasing N the total number of events in the test suite decreases too. Fig. 8 reports that the sum of the lengths of each single sequence decreases when N increases. This is reasonable because the more is the length of the sequence, more possible is that we can avoid repeating the first event of the pair we want to test. Even if Fig. 8 shows the plot for the CNST method, we have verified that the same trend is respected also by methods that repair the sequences.

G. RQ7: Comparison between automata-based generation method and SCAs

To compare the results obtained by our automata-based method with the ones coming from the use of SCAs we have computed also the coverage information for the 3-wise testing (Table VI)[2]. We have generated the SCAs for each our benchmark by using the tool provided by [24].

SCAs are based only on permutations of n events, thus they don't consider the constraints of the system. For this reason, we need to repair even the sequences produced by the SCAs generator. As the standard approach with SCAs produces sequences all with the same length (equal to the number of the events considered), the total length of the test sequences is shorter than those obtained with our automata-based method. Also the sequences generation time is shorter because compute all the permutations is less complex than compute the intersection among automata.

Table VII shows the summary of the comparison between the coverage obtained by our method and the one by SCAs (with different repairing procedures). The results confirm that our automata-based method performs better than SCAs in every analyzed aspects. Even comparing the coverage of the

[2]For the PHD benchmark we had to reduce the value of N due to the high complexity of the system: with $N > 6$ 3-wise automata the generation times out.

TABLE VI
METHOD EVALUATION (3-WISE TESTING)

Benchmark	Monitor.	Method	N	# Seq.	Max. Len.	Min. Len.	Avg. Len.	Tot. Len.	# Valid Seq.	# Cov. triads	# Cov. states	# Cov. trans.	Gen. t [s]
PHD	NO	Cnst	6	1,775	22	10	15	28,076	1,775	10,648	5	65	538.37
PHD	NO	Skp	6	1,521	22	12	16	25,198	0	7,075	5	65	981.22
PHD	NO	Rej	6	1,521	21	12	16	25,178	0	0	0	0	827.73
PHD	NO	Stp	6	1,521	22	12	16	25,277	0	1,298	5	42	881.79
PHD	YES	Cnst	6	1,775	23	11	15	28,032	1,775	10,648	5	65	7475.46
PHD	YES	Skp	6	1,521	21	12	16	25,159	0	7170	5	65	7754.58
PHD	YES	Rej	6	1,521	21	12	16	25,165	0	0	0	0	7329.89
PHD	YES	Stp	6	1,521	21	11	16	25,194	0	1,263	5	39	7109.64
Pattern rec.	NO	Cnst	10	1	6	6	6	6	1	8	5	5	0.09
Pattern rec.	NO	Skp	10	1	6	6	6	6	1	8	5	5	0.10
Pattern rec.	NO	Rej	10	1	6	6	6	6	1	8	5	5	0.08
Pattern rec.	NO	Stp	10	1	6	6	6	6	1	8	5	5	0.09
Pattern rec.	YES	Cnst	10	1	6	6	6	6	1	8	5	5	0.10
Pattern rec.	YES	Skp	10	1	6	6	6	6	1	8	5	5	0.09
Pattern rec.	YES	Rej	10	1	6	6	6	6	1	8	5	5	0.08
Pattern rec.	YES	Stp	10	1	6	6	6	6	1	8	5	5	0.09
Elevator	NO	Cnst	10	12	15	10	12	147	12	125	4	6	3.39
Elevator	NO	Skp	10	43	16	11	12	554	0	65	4	6	162.74
Elevator	NO	Rej	10	43	16	11	13	561	0	0	0	0	146.56
Elevator	NO	Stp	10	43	15	11	13	565	0	0	3	2	154.50
Elevator	YES	Cnst	10	13	14	10	12	158	13	125	4	6	4.67
Elevator	YES	Skp	10	43	16	11	13	565	0	65	4	6	161.99
Elevator	YES	Rej	10	43	15	10	13	561	0	0	0	0	190.43
Elevator	YES	Stp	10	43	15	11	13	561	0	1	4	5	157.33
Vault	NO	Cnst	10	1	5	5	5	5	1	10	6	5	0.97
Vault	NO	Skp	10	11	12	5	9	108	0	10	6	5	5.47
Vault	NO	Rej	10	11	12	7	9	107	0	0	0	0	4.13
Vault	NO	Stp	10	11	11	6	9	109	0	10	6	5	4.28
Vault	YES	Cnst	10	1	5	5	5	5	1	10	6	5	1.26
Vault	YES	Skp	10	11	11	8	9	107	0	10	6	5	4.08
Vault	YES	Rej	10	11	12	8	10	111	0	0	0	0	4.58
Vault	YES	Stp	10	11	12	9	10	112	0	1	4	3	4.26

TABLE VII
COMPARISON BETWEEN SCAS AND AUTOMATA-BASED METHOD (3-WISE TESTING)

		Valid Seq.	Triads Cov.	States Cov.	Trans. Cov.	Avg. Cov
Automata-based	**Cnst**	100.00%	100.00%	100.00%	95.29%	98.43%
Automata-based	**Skp**	0.06%	66.77%	100.00%	95.29%	51.10%
	Rej	0.06%	0.07%	25.00%	5.88%	
	Stp	0.06%	12.00%	92.50%	62.35%	
SCAs	**Skp**	2.17%	26.44%	85.00%	57.65%	29.03%
	Rej	2.17%	0.00%	15.00%	2.35%	
	Stp	2.17%	0.15%	50.00%	24.17%	

SCAs with the one got by repairing the sequences obtained by using automata-based approach, the results are the same (51.10% of overall coverage for automata-based method vs 29.03% for SCA standard approach) excepts for the percentage of valid sequences, since SCAs generate fewer test sequences. The results tell that the automata-based method is overall better than SCAs method, and it is confirmed by the *paired t-test* [32] with:

- H_0: the two methods perform in the same way, in terms of coverage
- 8 degrees of freedom
- $t = 3.8507$
- $p_{value} = 0.004873$
- $\alpha = 0.05$

We can see that $p_{value} < \alpha$, so we can reject the hypothesis H_0 and claim that our method performs better.

V. Related work

Combinatorial interaction testing (CIT) has been shown to be an effective approach to manage the complexity of the test of event-based software. In the last years, because of the growth of the number of software based on the interaction with the user, the combinatorial sequence testing (CST) has been used in many fields.

The classical approach for CST requires the use of Sequence Covering Arrays (SCAs) [8], [29], [36], that provide a set of permutations of the events supported by the system. Many techniques have been proposed to generate these kind of data.

In [1], the authors apply combinatorial-based event sequence methods to test Android applications, aiming to minimize the execution of events and maximize the coverage of event combinations. However, they only use a greedy algorithm and they don't consider the constraints about the order of events imposed by the SUT. CST has been also used for browser fingerprinting in [14], where the authors show that combinatorial properties have an impact on browsers' behavior during the TLS handshake with a server, and in [31] where an interaction-based test sequences generation method for testing Web Apps is proposed. The same TLS protocol has been tested also in [13], where weighted t-way sequences are used to derive sequence test cases for its testing. The methods described in [18] can be used for testing mission critical systems that accept multiple inputs and generate outputs to several communication links, where it is important to test the order in which events occur.

A different technique is used in [25], where the authors present a feasible test suite generation technique using a meta heuristic search called Simulated Annealing (SA) for T-way EDISTC-SA generator.

The main problem of the application of combinatorial testing in actual event-based systems is that, in many cases, some event can be fired only when a certain event has already been fired. This is why it is essential to consider also the constraints of the system while generating test sequences. The authors of [2] describe an approach to test suite generation for Constrained Combinatorial Testing (CCT) based on Answer Set Programming. In [7], the authors propose a solution for CCT in which the constraints on the input parameter values are expressed as logical predicates that can be solved by a formal logic tool.

Another approach that aims to deal with constraints in CIT is the algorithm IPOG-C [34] which includes optimizations to improve the performance of constraints handling. In [20], two novel algorithms to deal with constraints in CIT are presented: CCS (Construct Constraint Set) and CTWC (Combinatorial Testing With Constraint). The former computes implied constraints, while the latter uses the results of CCS to facilitate the test generation process.

Latest software systems permit a high configurability, in terms of parameter. In this research field, CIT has become widely used. For example, the authors of [11] describe how CIT can be extended, by adding some new testing policies able to check if the model correctly identifies constraints among the various software parameters.

In the methods described into this paper we have used FSMs to represent the constraints of the system. Other notations have been proposed in other papers. For example, in [9], the authors develop a notation for specifying sequencing constraints and present a t-way test sequence generation method that handles the constraints specified in this notation. The authors of [12] discuss automatic test sequence generation and coverage criteria for testing abstract state machine (ASMs).

VI. Conclusions

Testing event-based systems can be very challenging because most of them are described using incomplete Finite State Machines. For this reason, it is possible that a specific input cannot be applied when the system is in a particular state, or also that the response of the system for it is not defined.

Classical approaches used to test event-based systems use Sequence Covering Arrays (SCA) but they do not consider the constraints imposed by the system, so some of the test sequences can be useless or in need of repair.

In this paper, we have proposed a novel solution for test-sequences generation that exploits the FSMs and their representation through automata. The approach consists in using the SUT FSM as a description of constrains, turned into an automaton, and representing each tuple of events to be tested with a t-wise automaton. The intersection between the two kinds of automata, if it exists, produces another automaton that comply with the constraints of the system and covers the considered tuple. We have also devised three repairing methods that allow the execution of invalid test sequences, by rejecting the whole sequence, skipping the wrong event or stopping at the first wrong event.

Our method has shown better performance (in terms of coverage and valid sequences) than the standard SCAs approach, even when the repairment of the test sequences is applied. Moreover, with the automata-based method we can also generate test sequences with multiple repetitions of the same event (for example by testing the pair $e_i - e_i$), while with SCA, a single occurrence for each event is allowed. This is an important aspect because some systems can show malfunctions only when an event is repeated multiple times. Using automata to generate tests can be very time consuming and, for this reason, we have introduced a limit on the number of t-wise automata that can be collected together.

As future work, we will try to apply this method in systems that do not have a well know FSM structure, by applying a preprocessing procedure to automatically learn the behavior of the SUT and representing it using the FSM formalism or an automata-based representation.

References

[1] D. Adamo, D. Nurmuradov, S. Piparia, and R. Bryce. Combinatorial-based event sequence testing of android applications. *Information and Software Technology*, 99:98–117, jul 2018.

[2] M. Banbara, K. Inoue, H. Kaneyuki, T. Okimoto, T. Schaub, T. Soh, and N. Tamura. catnap: Generating test suites of constrained combinatorial testing with answer set programming. In *Logic Programming and Nonmonotonic Reasoning*, pages 265–278. Springer International Publishing, 2017.

[3] G. Becci, G. Dhadyalla, A. Mouzakitis, J. Marco, and A. D. Moore. Robustness testing of real-time automotive systems using sequence covering arrays. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, 6(1):287–293, apr 2013.

[4] A. Bombarda, S. Bonfanti, A. Gargantini, M. Radavelli, F. Duan, and Y. Lei. Combining model refinement and test generation for conformance testing of the IEEE PHD protocol using abstract state machines. In *Testing Software and Systems*, pages 67–85. Springer International Publishing, 2019.

[5] R. C. Bryce, S. Sampath, and A. M. Memon. Developing a single model and test prioritization strategies for event-driven software. *IEEE Transactions on Software Engineering*, 37(1):48–64, jan 2011.

[6] A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In B. Beckert and R. Hähnle, editors, *Tests and Proofs*, pages 66–83. Springer Berlin Heidelberg, 2008.

[7] A. Calvagna and A. Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, apr 2010.

[8] Y. M. Chee, C. J. Colbourn, D. Horsley, and J. Zhou. Sequence covering arrays. *SIAM Journal on Discrete Mathematics*, 27(4):1844–1861, jan 2013.

[9] F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn. An approach to t-way test sequence generation with constraints. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, apr 2019.

[10] J. Ferrer, P. M. Kruse, F. Chicano, and E. Alba. Search based algorithms for test sequence generation in functional testing. *Information and Software Technology*, 58:419–432, feb 2015.

[11] A. Gargantini, J. Petke, M. Radavelli, and P. Vavassori. Validation of constraints among configuration parameters using search-based combinatorial interaction testing. In *Search Based Software Engineering*, pages 49–63. Springer International Publishing, 2016.

[12] A. Gargantini and E. Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *Journal of Universal Computer Science*, 7, 02 2003.

[13] B. Garn, D. E. Simos, F. Duan, Y. Lei, J. Bozic, and F. Wotawa. Weighted combinatorial sequence testing for the TLS protocol. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, apr 2019.

[14] B. Garn, D. E. Simos, S. Zauner, R. Kuhn, and R. Kacker. Browser fingerprinting using combinatorial sequence testing. In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security - HotSoS '19*. ACM Press, 2019.

[15] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*. ACM Press, 2013.

[16] S. K. Khalsa and Y. Labiche. An orchestrated survey of available algorithms and tools for combinatorial testing. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, nov 2014.

[17] T. Kitamura, A. Yamada, G. Hatayama, C. Artho, E.-H. Choi, N. T. B. Do, Y. Oiwa, and S. Sakuragi. Combinatorial testing for tree-structured test models with constraints. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, aug 2015.

[18] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei. Combinatorial methods for event sequence testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, apr 2012.

[19] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. IEEE, mar 2007.

[20] L. Li, Y. Cui, and Y. Yang. Combinatorial test cases with constraints in software systems. In *Proceedings of the 2012 IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, may 2012.

[21] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing combinatorics in GUI testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, 2016.

[22] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2017. `http://www.brics.dk/automaton/`.

[23] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):1–29, jan 2011.

[24] NIST. NIST sequence covering array generator. https://csrc.nist.gov/Projects/automated-combinatorial-testing-for-software/event-sequence-testing/unders, 2016.

[25] M. M. Rahman, R. R. Othman, R. Ahmad, and M. Rahman. A meta heuristic search based t-way event driven input sequence test case generator. *International Journal of Simulation Systems, Science & Technology (IJSSST)*, 15:70–77, 01 2015.

[26] C. W. Rapp. SMC – the state machine compiler, 2019. `http://smc.sourceforge.net/`.

[27] Y. Sheng, C. Sun, S. Jiang, and C. Wei. Extended covering arrays for sequence coverage. *Symmetry*, 10(5):146, may 2018.

[28] D. E. Simos, L. Kampel, and M. Ozcan. Combinatorial methods for testing communication protocols in smart cities. In R. Battiti, M. Brunato, I. Kotsireas, and P. M. Pardalos, editors, *Learning and Intelligent Optimization*, pages 437–440, Cham, 2019. Springer International Publishing.

[29] G. Tzanakis. Covering arrays from maximal sequences over finite fields.

[30] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme. *Modeling software with finite state machines: a practical approach*. Auerbach Publications, 2006.

[31] W. Wang, S. Sampath, Y. Lei, and R. Kacker. An interaction-based test sequence generation approach for testing web applications. In *2008 11th IEEE High Assurance Systems Engineering Symposium*. IEEE, dec 2008.

[32] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012.

[33] L. Yu, Y. Lei, R. N. Kacker, D. R. Kuhn, R. D. Sriram, and K. Brady. A general conformance testing framework for IEEE 11073 PHD's communication model. In *Proceedings of the 6th International Conference on PErvasive Technologies Related to Assistive Environments - PETRA '13*. ACM Press, 2013.

[34] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn. An efficient algorithm for constraint handling in combinatorial test generation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, mar 2013.

[35] X. Yuan, M. Cohen, and A. M. Memon. Covering array sampling of input event sequences for automated gui testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*. ACM Press, 2007.

[36] R. Yuster. Perfect sequence covering arrays. *Designs, Codes and Cryptography*, nov 2019.