A Systematic Way to LTE Testing

Preprint · December 2018

DOI: 10.13140/RG.2.2.30914.63686

2 authors:

Muhammad Taqi Raza
University of California, Los Angeles
24 PUBLICATIONS **169** CITATIONS
SEE PROFILE

Songwu Lu
University of California, Los Angeles
191 PUBLICATIONS **15,485** CITATIONS
SEE PROFILE

A Systematic Way to LTE Testing

Muhammad Taqi Raza and Songwu Lu
Department of Computer Science, UCLA

Pre-print version to appear at MobiCom 2019

ABSTRACT

LTE test cases are 3GPP standardized. These tests must be executed on every LTE capable device model before they are commercially released. LTE testing practices are unmethodical where the focus of testing is to follow the test cases specified in the standard as they are; resulting incomplete and inefficient testing. We discover that the standardized test cases are incomplete in which a number of test cases related to multiple protocol interactions are missing. Our analysis also show that the individual treatment of test cases and not as a part of a whole system cause repetitive execution of test operations; bringing testing inefficiencies. In this paper, we argue that there is a need for a complete paradigm shift from ad hoc testing to a methodical approach for telecommunication testing. We took a set of guidelines from LTE standard and proposed a systematic and algorithmic approach to testing. In the process, we combat various challenges and provide complete list of test cases without enumerating all possibilities. Our results show that by avoiding repetitive test operations, we reduce up to 70% of LTE testing steps. We also find 87 new valid test cases which are not defined by the LTE testing standards.

1 INTRODUCTION

The service requirements, architecture and protocol functionalities of LTE are standardized by 3GPP. The 3GPP standard provides a number of LTE protocols conformance test cases to ensure that the device and network elements comply to established procedures for their control and user plane functionalities. These test cases validate the device implementation of LTE protocol standards.

Our study on LTE protocol conformance testing reveals that (1) conformance testing mainly focuses on validating/invalidating single protocol test case, mostly ignoring interactions between two protocols. This leads to incomplete LTE testing. (2) Each test case is self contained; its execution information is not shared with other test cases leading to repetitive execution of test operations. These issues arise due to unmethodical testing practices. Instead of taking guidelines from the standard and developing an efficient, optimized and a complete testing system, the focus of current practices is to follow the standard as it is. In this paper, we take a systematic approach to LTE testing based on the following key points: (1) multiple protocol interactions in a system should be defined for testing to get complete set of test cases (achieving test cases completeness), and (2) test cases should facilitate other test cases' execution through common testing goals (improving testing efficiency)

Based on our key point 1 to achieve test cases' completeness, we formulate the problem of finding multiple protocol interaction related test cases. To give a bit of background, a device and network simulator exchange a number of messages to execute a test case. To generate one such message, the device traverses through different states of one or more LTE protocols Finite State Machines (FSMs). Simply looking at device output messages, we can tell that the device has properly transitioned different states of FSM(s). This observation significantly reduces our effort to generate device test cases. A test case is represented as an output messages combination that a device can generate. To provide complete list of test cases, we are required to generate *all* possible combinations of these output messages. For n output messages, there are 2^n possible test cases, which are practically infeasible to analyze. We are interested in finding all those output message combinations that the device will *never* produce, which maps into a problem of finding all don't cares output values for output message combinations. We traverse device protocol FSMs in reverse (from output state towards input state) to find these don't care outputs. This is challenging especially when each protocol FSM has too many states to traverse (consider all states related to configurations, timings and functionalities). This motivates us to reduce the number of states at device protocol FSMs. We refer to FSM reduction and minimization algorithms in finite automaton. We propose two novel algorithms that minimize only deterministic finite automaton (DFA) states through LTE domain knowledge and skip non-deterministic finite automaton (NFA) states minimization (which is an NP complete problem). Once we get compact representation of device protocol FSMs, we can quickly find the don't cares from output message combinations and provide complete set of test cases without enumerating all possible device output message combinations.

For our second key point to improve LTE testing efficiency, we propose an algorithm to remove repetitive steps for a test case. Our algorithm is based on a common graph data structure for all test cases. The graph nodes record the output parameters of a particular step in a test case, thus maintaining

execution history in the graph nodes for all the steps in a test case. The test cases that come later do not repeat the same steps already executed by previous test cases, rather they reuse the stored output parameters and skip the entire execution of those steps. Storing output parameters for all the steps of test cases however is memory intensive and computationally infeasible. We address this issue by maintaining multiple graph data structures mapped to different test scenarios. A graph data structure, however, is shared among test cases belonging to the same scenario.

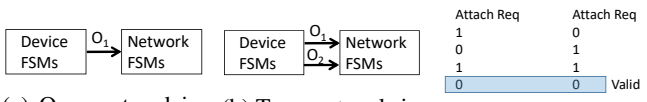
To test our proposed system, we have implemented 3GPP test cases along with our algorithms. We create the device FSMs and their representation as finite automaton. We also generate complete test cases by excluding don't care device outputs. By doing so, we find 87 new test cases where 60 test scenarios were not even mentioned in LTE testing standards. Also, our test case optimization algorithm executes 43%, 11%, 70% and 50% less number of steps for *Attach*, *Detach*, *Tracking Area Update (TAU)*, and *Service Request* functionalities, when compared to 3GPP test cases execution.

2 LTE TESTING

We briefly discuss importance of LTE testing, its limitations, and applications beyond LTE technology.

Importance of LTE testing LTE testing is one of the key factors in making LTE a success. Operators and network vendors not only rely on User Equipments (UEs) (also referred to as 'device' in this paper) to obey the standard properly but they also require the phones on the network to be compliant. 3GPP defines the behavior of the phone and the network in every operational situation. These operational situations are defined in LTE protocol conformance testing specification[1] that defines a number of test cases to validate LTE protocol operations. These test cases ensure that different LTE protocols (such as radio connection control, mobility management, session management, connection management, transport and tunneling protocols) work correctly in every operational situation. These test cases are tested through messages exchange between device and a Network Simulator (NS). Both device and NS implement their Finite State Machines (FSMs) and ensure the correctness of the test case through required states transitions. Figure 1a shows a test case execution scenario, in which device FSM generates an output message (O_1) and NS consumes that message to validate the message correctness.

Current LTE Testing Practices and Limitations Current LTE Testing practices are adhoc; they do not follow a systematic approach to testing and do not guarantee completeness. In our study, we find that although LTE standard discusses LTE testing in every operational situation as well as abnormal device behavior, these test cases do not cover every test scenario.



(a) One protocol interaction (b) Two protocols interaction (c) Output message combinations

Figure 1: LTE test execution scenarios between device and network

This is mainly because current LTE testing practice has focused on testing a single protocol execution. There exist cases where multiple protocols' execution overlap, and currently *all* these protocols' interactions are not tested. Figure 1b) depicts the case when two output messages (O_1 and O_2) are fed to NS, that validates the correctness of their interaction. To elaborate, we take an example of LTE *Attach Request* procedure. In this procedure, two EMM (EPS Mobility Management) protocols interact by using one message (Attach Request message). It generates 4 possible output message combinations (as shown in Figure 1c). (0 , 1) and (1 , 0) represents the absence of message from one protocol making it a test case of single protocol interaction[1]. (1 , 1) represents message from both protocols making it a case of two protocols interaction. (0 , 0) is a test case that corresponds to the absence of messages. The absence of Attach Request message triggers timeout that may impact the behavior of the interacting protocol[2].

We further discover that LTE testing treat all test cases individually based on conformance to LTE test standard; a test case does not transfer its execution information to a later test case in a series of tests. This leads to inefficiencies with repetitive execution of test steps for a new test case to bring device into a certain state (e.g. idle or connected state etc.). For example, almost all LTE *Attach* related test cases require the device to be switched *off* and then *on*, so that the device can initiate an *Attach* test case. However, such repetitive device *power off/on* steps execution can be avoided if the next test case uses the knowledge from the previous test case that has successfully executed these steps.

We also looked into commercially available testing solutions provided by two leading[2] LTE test equipment vendors, namely Anritsu[3] and Anite[4]. Their testing data sheets[5][6] and demos[7] reveal that their test equipments do not bring any changes to testing implementation guide provided by 3GPP[8]. They treat each test case individually and only execute those test cases which are provided by the 3GPP. Their approach to testing is inefficient as they treat test cases individually and do not transfer the previous test case execution knowledge to the following test case. We conclude that LTE chipset manufacturers, device vendors, and network

[1]For example, device protocol generates one EMM message to NS.

[2]Such interaction can be between same protocol (e.g. two EMM (EPS Mobility Management) protocol messages interact) or between two different protocols (e.g. EMM and ESM (EPS Session Management) protocols messages interact.)

operators perform test cases in the same way as test equipment vendors execute them.
The above observations show the current mindset of the testing community that is inclined towards telecommunication standard based textbook testing. We advocate for a shift towards a system design approach that brings computer science way of thinking in telecommunication testing. We aim to design and develop a testing system that treats individual test cases as the building blocks to a testing framework, along with algorithms to optimize test cases' steps as well as improve test cases' efficiency.

Testing beyond 4G LTE Researchers who are developing systems for the next generation wireless networks often face a challenge of verifying their designs. Testing plays a crucial role in making these design practical and deployable in real world. Through testing, researchers ensure that their systems adjust properly and quickly under a number of scenarios which are both common and corner case scenarios. Although, this paper has focused on 4G LTE testing but our methodology is generic and can be applied to testing in general. Other areas of research that benefit from our work include: wireless and mobile computing, e.g. 5G New Radio (NR) testing; application specific testing, e.g. Cellular Internet of Things (CIoT) testing; and many more. Mobilizing mmWave (millimeter Wave) with 5G NR is going to introduce new test challenges when cable access on cellular devices are replaced with transceivers with integrated phased array antennas. These challenges include repeatability, configuration, and coverage, as well as testing accuracy, test time, and ultimately cost. These testing challenges can be solved through our complete and efficient testing methodology. Similarly, 5G cellular IoT solutions that include LTE-M (LTE for Machines) and NB-IoT (Narrow Band IoT) require extensive testing before their deployment. Their test cases are related to network integration, coverage, battery consumption, and more. Our testing methodology can provide complete and efficient testing to these emerging technologies.

3 TEST COMPLETENESS

We discuss current limitations in providing complete list of test cases followed by our approach.

3.1 Limitations and Challenges

To make LTE testing a systematic testing; we view test cases as a sequence of messages exchange rather than testing a particular scenario. We reason that LTE test case procedure involves an exchange of messages between device and network. The sequence of these messages conclude whether the test case has passed or failed. For example, in device *Attach* test case, device and network exchange a number of messages related to Radio Resource Control (RRC), Security Mode, Authentication, and Packet Data Network (PDN or IP) connectivity. We view LTE testing as the interaction between device and network. If *all* such interactions between device and network (*in any order*) are successful then we have tested *all* test cases; otherwise not. To provide complete list of test cases, we are required to generate a list of test cases that include *all* possible combinations of these messages. For n device output messages, there are 2^n possible test cases, which are practically infeasible to analyze. The real challenge is to provide complete list of test cases without considering *all* possible message combinations.

3.2 Our approach

We aim to validate LTE protocols interaction to ensure that all different combinations of protocol messages are tested for correctness.

Testing as protocols interaction The purpose of device state transitions and their interactions is to generate a message for the network simulator. Hence, LTE testing looks into message exchange between the device and the network. We can significantly reduce device protocol FSMs testing by simply looking at the output message (O_1) that device FSM has produced, as shown in Figure 1a. If the message produced by the device FSM is the one that the NS is expecting, then device internal FSMs state transitions and interactions were correct. If the NS has received an unexpected message from device then we only debug those states which have produced that output message, rather than traversing all the states of device FSM(s). Therefore, we can say that by reverse-engineering device output values, the device states transition can be found, through which a device test case can be defined.

Don't care outputs We provide complete set of test cases which explore *all* possible output message combinations produced by device FSMs when two protocols are interacting between device and NS. For n possible output messages produced by two protocols running at the device, one is required to test 2^n message combinations.

We consider optimizing the choices of output message combinations from two protocols interactions. We do not want to generate test cases for those message combinations which were never produced by device FSMs. We annotate these message combinations as don't care outputs. To find don't care outputs, we have to traverse device FSMs in reverse and find out if the corresponding output is valid or not. Finding don't care outputs however is practically infeasible when device FSMs have too many states.

Compressing device FSMs We can quickly find the don't care outputs if we skip few states in device protocol FSM. We can even skip visiting a complete portion of FSM which might have some constraints on message combinations. This

motivates us to compress device FSMs by merging certain states from FSMs.

Finite Automaton To compress FSMs, we model these FSMs as a finite automaton. Much like FSMs in finite automaton, we have a start state, finite set of states, a final state, set of transitions, and a transition function. Some states are deterministic while others are non-deterministic. If for each pair of states and possible transition conditions, there is a unique next state (as specified by the transition function), then the finite automaton is deterministic, i.e. Deterministic Finite Automaton (DFA); otherwise, the finite automaton is non-deterministic, i.e. Non-deterministic Finite Automaton (NFA).

Converting NFA to DFA We restate that our aim is to reduce number of states and to keep FSM running in linear time. We find that reducing number of NFA states (by removing unnecessary states) is proved to be NP-complete[9]. Moreover, running time for an NFA is $O(n^2m)$ compared to $O(m)$ in case of DFA, where n is number of states, and m is the number of *identical* transition conditions[10][11]. This is because NFA has n possible next states compared to DFA that has only 1 path to next state for a given transition condition. This motivates us to convert NFA into DFA.

3.3 LTE Testing as Finite Automata

Overview of our solution We model LTE testing as a problem of finding don't care output combination values in Finite Automata. All possible output combinations *minus* don't care outputs are complete list of test cases. To find don't care outputs efficiently (i.e. running time of FSM interaction remains linear), we first convert NFA into DFA. We propose a novel algorithm that reduces FSMs states by converting selected NFA states to DFA. We further minimize FSMs states through DFA minimization procedure. We can do so because two or more DFA states can be equivalent, where these states transition to same next states for same transition condition. We merge these equivalent states and get compact representation of LTE protocols FSMs. To find as many equivalent states as possible to merge, we provide a new definition of states equivalence. We uses LTE domain specific knowledge and argue that a number of FSM states have LTE timing and protocol constraints. These states are said to be equivalent and merged because in reality they never occur together. In this regard, we propose a novel DFA minimization algorithm that converts non-equivalent states to equivalent states and then merges these states.

3.3.1 Reducing FSM States

We first convert NFA states to DFA states to reduce the total number of states in FSM.

Inefficiencies in NFA to DFA conversion Robins and Scott algorithm[12] is the best known algorithm to convert NFA to DFA[13][14]. Such conversion is made through power set construction. The DFA is obtained through a state set 2^n, the power set of n, containing all subsets of original NFA state set n. The exponential number of DFA states are due to the degree of non-determinism of current state. The current state can transition to a number of next states for n possible transition conditions. Through power set construction – which is practically inefficient – all possible states are recorded.
In literature[15][16], it has been shown that number of states in DFA dramatically increase when more than one transition conditions are considered. It has been proved that maximum number of states in DFA reach $2^{\frac{n}{2}}$, $2^{\frac{2n}{3}}$, $2^{\frac{3n}{4}}$, and 2^n when number of transition conditions are 2, 4, 8, and n, respectively.

Algorithm 1 Selected NFA states to DFA states conversion

```
 1: input: = {nstates, trans_cond, trans_func, curr_state, next_state}
 2: Call procedure Reverse-Edges()
 3: procedure NFA-TO-DFA-PROCEDURE
 4:     while nstates are not visited do
 5:         if curr_state transitions to two or more next_state then
 6:             if trans_func takes only one trans_cond then
 7:                 for all next_states do
 8:                     dfa_state ∪ next_state
 9:                 divert edge arrows from next_states to dfa_state
10:                 add edge from dfa_state to curr_state
11:                 add trans_cond to the edges
12: Call procedure Reverse-Edges()
13: procedure REVERSE-EDGES
14:     if curr_state transitions next_state(s) on trans_cond then
15:         for all next_state(s) do
16:             swap (curr_state, next_state(s))
17:             reverse edges arrows
18:             keep trans_cond and trans_func
```

Converting selected NFA states to DFA We proposed NFA to DFA conversion (Algorithm 1) that converts only selected NFA states to DFA and does not generate power set of NFA states. In our algorithm, we mainly focus on two points: (1) visiting the states from output (in reverse), and (2) converting those NFA states to DFA which have *only one transition condition* (which is very common in LTE, as we show later).
For (1), we swap current state with the next state (algo step 16) and then reverse the edges while keeping transition condition and transition function unmodified (algo steps 17-18). Thereafter, the FSM can be traversed in reverse and the initial FSM state can be reached that has generated final output value.
For (2), our algorithm processes only those states which have exactly one transition condition to the next state. Our algorithm first checks whether current state is indeed an NFA; that is, it has more than one next states on a given transition condition (algo step 5). Note that, when current state has only one next state then that current state is already deterministic and algorithm moves to next state (algo step 4). When first *if* condition (algo step 5) yields true, then the algorithm checks whether current state to next states transitions are carried

through one transition condition or not (algo step 6). If this condition is satisfied then NFA to DFA conversion begins. In such conversion, our algorithm merges all next states of current state and creates one DFA state, which is a set of merged states (algo step 8). Thereafter, it changes the edges such that all incoming and outgoing edges of all merged next states are diverted to newly created DFA state, and a transition edge is inserted between DFA state and current state (steps 9-11). Lastly, we again reverse the edges from current state to next state(s) before we perform further action on DFA states. Note that this is an important step before we further reduce DFA states (section 3.3.2). If we do not reverse the edges then process of reducing DFA will give us NFA back again. This has been shown in Brzozowski's algorithm[17] that DFA minimization converts the input DFA into an NFA by reversing all its arrows and exchanging the roles of current and next states.

States with one transition condition are common in LTE We elaborate that in fact one transition condition states are common in LTE which are related to LTE timers and functionalities. The most common examples we see are timers handling *reject* conditions. The standard specifies, if subscriber device request is *rejected* for certain number of times then current state should transition to a particular next state (with *reject* transition condition); otherwise current state should transition (with same *reject* transition condition) to some different next state. Similarly, most of LTE functionalities also have NFA states with exactly one-transition condition. The transition condition remains same for different types of actions, such as cell search, camping on cell, multiple or single bearer request, and priority related features etc. To take an example, if the device serving cell state meets certain threshold value, it should transition to *intra-freq measurement state*, otherwise it transitions to *inter-frequency measurement state*. The transition condition for both future states is *measurement*.

Discussion Our algorithm processes those states which have only one transition condition to next state and does not construct the power set of states. The complexity of our algorithm is linear where *while()* loop (algo step 4) iterates over constant number of states (finite set of states is provided as an input to algorithm). The *for()* loop (algo step 7) also iterates over constant number of next states because our algorithm executes this step only when current state transitions to next states over single transition condition. Note that, number of next states can be found by looking at number of arrows coming to current state (we are looking at arrows not tails as we are processing in reverse)

Example We now provide an example in which our algorithm converts an NFA to a DFA, as shown in Figure 2. Two different test cases (test case numbers 10.5.1 and 10.5.1b in[1]) transition from current state of *PDN_req_init* into two different states *Procedure_Transaction_Pending* and

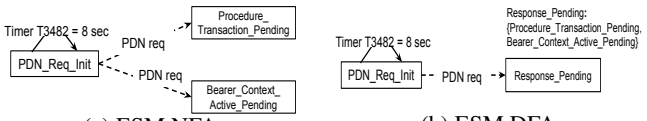


Figure 2: One-transition NFA states are converted into DFA

Bearer_Context_Active_Pending states respectively, using one transition condition *"PDN req"*. Both these test cases are requesting PDN from the network. The first test case is requesting an additional PDN for its uplink (UL) data, whereas the second test case also requests an additional PDN but using NAS signalling low priority. In 10.5.1b, UE establishes a dedicated radio bearer associated with the default EPS bearer context, before sending an additional PDN connectivity request. That's why UE moves to *Bearer_Context_Active_Pending* state.

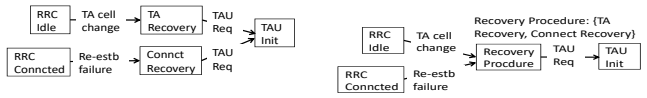
3.3.2 Minimizing FSM States

In DFA minimization, two or more equivalent DFA states are merged and represented as one state. Two or more different states are said to be equivalent, if these states transition to same next states for same transition condition.

DFA minimization overview Hopcroft algorithm for DFA minimization[18] is the best known algorithm for minimizing a DFA[19][20] and has complexity of *O(nlogn)*. The key idea of the algorithm is to partition the states when two states are not equivalent. At first all states are placed into one partition and thereafter the partition in refined. The states which are not equivalent are removed from the partition, whereas the equivalent states are merged. The key ingredient of Hopcroft algorithm lies in the way the partitioning is done. The idea is to not partition on already visited transition condition until the partition is further split. In that case, the algorithm only checks one of the two new partitions.

DFA minimization of mixed NFA-DFA FSM We propose a new algorithm for DFA minimization but use the partitioning procedure from Hopcroft algorithm. Unlike Hopcroft algorithm that works on only DFA FSM, our algorithm reduces FSM which is a mix of NFA-DFA. Indeed, like many other FSMs, LTE protocol FSMs are mix of NFA and DFA states, where we have also converted few NFA states into DFA states (algorithm 1). From Hopcroft algorithm, we find that as the number of equivalent states increase, the overall number of states in FSM decrease (as equivalent sates are merged together). Although, we cannot increase the states equivalence in FSM, we can achieve same result by merging two or more states which have constraints on each other. We use our LTE domain knowledge and introduce protocol states and timing constraints.

Protocol states constraints In LTE protocol FSMs, some protocol states have constraints on other states. Few of such constraints have been shown in LTE NAS standard (Figure



(a) DFA states before minimizing

(b) DFA states after minimizing

Figure 3: Two equivalent DFA states are combined

5.1.3.2.2.7.1: EMM main states in the UE)[21]. For example, when the device is in idle state and wants to send/receive voice/data packets, it initiates the *Service Request (SR)* procedure and moves to *Service_Request_Init* state. While the *SR* procedure is ongoing, assume the device changes its location and wants to perform *Tracking Area Update (TAU)* procedure, however it cannot do so. Because the *TAU* procedure can only be initiated from device state *EMM_Registered*. Thus we can say that all *TAU* and *SR* related states have constraints on each other, where device cannot be in both states at the same time, and we can merge these states. As a result, our modified FSMs will never produce those output values that capture *TAU* and *SR* interactions (which are don't care outputs). In short, we list all such protocol constraints and merge them.

Timing constraints: Similar to protocol states constraints, there are timing constraints between different states. For example, to initiate the *normal TAU* request message, the device must have moved to a different LTE base station cell (that is, its location must have been changed). Further, LTE base station cell can only corresponds to one tracking area, because *System Information Block-1 (SIB1)* broadcasts only one tracking area code for a cell. Therefore, one cell can't be part of two different tracking areas. This example shows timing constraint situation where *normal TAU* procedure can only be initiated after UE has moved to different cell belonging to a different tracking area.

Algorithm We now explain our algorithm of minimizing DFA states, as described in Algorithm 2. At the start of the algorithm, there is only one partition which contains sets of all states in an FSM (algo line 6). Like Hopcroft DFA minimization algorithm, our algorithm iteratively reduces partition by removing non-identical states. However, on each such iteration, it only performs further action when the state is deterministic (algo line 10), otherwise it splits the partition and removes the non-deterministic state (algo lines 8-9). Before it performs further actions on deterministic states, it makes sure that (1) for state p, there is no equivalent state, i.e. state p does not belong to current partition ($p \rightarrow$ other partitions); and (2) there does not exist any state in current partition which has constraint with state p (i.e. $p \notin constraint$). If both (1) and (2) are false, that means current state has equivalent state or constraint state in the partition and *for()* loop (algo line 7) moves to next iteration. If either (1) or (2) is true, then state p is moved out of the partition (algo line 11) and becomes a different partition (algo line 13). Once the algorithm moves p out of the partition then it checks whether p belongs to current state or the next state and updates accordingly (algo line 14-17). Note that it is possible that merged states may not be fully connected to other states. Therefore, on each iteration, we make sure that all incoming and outgoing transition arrows of merged states are updated accordingly (algo line 18).

Algorithm 2 Minimizing DFA states

```
1:  input: = {nstates, trans_cond, trans_func, curr_state, next_state}
2:  procedure MINIMIZE-DFA
3:      p1 is sub-partition 1; p2 is sub-partition 2
4:      constraint, constraint vector
5:      while no further partitions can be done do
6:          parition_1, all set of states in FSM; parition_2, ∅
7:          for each element p of parition_1 do
8:              if p is non-deterministic then
9:                  split(p, parition_1)
10:             else if (p → other partitions) OR (p ∉ constraint) then
11:                 {p1,p2} = split(p, parition_1)
12:                 if p belongs to {p1,p2} and p ≠ ∅ then
13:                     parition_2 = p
14:                     if curr_state == p then
15:                         curr_state_min = p
16:                     else if next_state == p then
17:                         next_state_min = p
18:                     update trans_cond for p
```

Discussion Our algorithm does not introduce any further complexity compared to Hopcroft algorithm, where it makes the decision of state being deterministic or non-deterministic in one step (algo line 8). It simply checks that the current state must not transition to more than one next states for a given transition condition. That means the state is NFA without even checking the next states. The step of splitting on non-deterministic state can be viewed as like that state does not have equivalence behavior. Therefore, the splitting procedure (algo line 9) does not add extra complexity either. The step of checking protocol state constraint and timing constraints require to first know whether the current state being partitioned is part of a set of constraints or not. If it is part of a set of constraints, then next algorithm checks whether current partition contains those states or not. We make these two steps efficient by first logging all such constraints as a constraint vector. Because constraints vector is of fixed length and the number of constraints are not too many (because these constraints are related to overall LTE functionalities, and not specific to states or timers etc.). Therefore, checking this constraint can always be done in linear time. Further, we avoid creating extra steps by merging constraint condition with partition condition.

Example We now provide an example from three LTE test cases in which our algorithm minimizes equivalent DFA states. Three test cases (test case number 9.2.3.1, 9.2.3.1.9a, and 8.5.1.4 in[1]) share part of common procedure. As shown in Figure 3, these test cases want to perform *TAU* procedure but under different triggering conditions. In test case 9.2.3.1, when UE moves to a different cell belonging to a different

Table 1: Summary of test cases – our procedure finds new legitimate test cases

Protocol Interaction	Procedure defined by 3GPP	Tests cases defined by 3GPP	Procedure *not* defined by 3GPP	Total *missing* test cases by 3GPP
ECM[3]and ECM	141	118	18	14
ECM and EMM	14	10	7	11
EMM and EMM	161	142	29	54
ESM and ESM	25	22	6	8
Total	**341**	**292**	**60**	**87**

tracking area, it initiates *TAU* procedure. However, in test cases 9.2.3.1.9a and 8.5.1.4, UE fails to recover from radio link failure and needs to perform connection recovery. Once the UE connection is recovered, *TAU* procedure is performed. Hence, we can minimize these DFAs by merging *TA Recovery* and *Connect Recovery* states (shown in Figure 3b).

3.4 Proof of Completeness

We provide the completeness proof by contradiction. The intuition of our proof is based on contradicting a number of possibilities and reason that our test cases are complete. We can say that our test cases are complete if we have transitioned all FSM states, ignoring some output values do not impact the completeness, and testing those two protocols interactions which are valid and supported by standard.

Proof by contradiction We put forward two possible cases and contradict them to prove the completeness of our approach.

Case 1: We assume that a practical input value was not tested: It's true that we do not test for don't care output values, but these output values never happen in reality. We ignore all those input values I which the FSM will never produce that we call don't care outputs X. Skipping test cases which require don't care values will not lead to missing test cases (i.e. incompleteness scenario) because in reality there exists no such FSM transition. We test all *practical* combinations of output values of a completely defined FSM by the 3GPP standard documents. C is a specification domain that includes all FSM states S that work with finite set of inputs, I. That is $C = S \times I$. Further, we traverse FSM in reverse for a given output value O (Algorithm 1), we always trace back to the given input value I. In other words, the output leads us to the deterministic FSM and all states can be traversed. It means that there is not any practical value which was not tested on a completely defined FSM. Hence, our argument contradicts the assumption that we can miss any practical input value that drive the FSM state(s).

Case 2: We assume that certain protocols interactions are missing: Missing those protocol interactions which never occur in reality means not testing the case which will never occur. Recall that we consider two protocols interactions. It means for every one output value, there are two possibilities for the other output (where the other protocol output exists or not). In other words, we have four possible combinations (these are not four values). Because all such possibilities are within the 3GPP standardized device behavior, we can test all valid combinations (that an FSM can generate). Hence, it is not true that we could miss certain protocols interactions.

By contradicting case 1 and case 2 we show that the test cases we generate are complete.

3.5 Analysis

Once we have reduced the device side FSMs and identify don't care output values, we can generate *complete* test cases for LTE protocol interactions. We defer to Section 5 for discussion on our implementation, and provide analysis on test case completeness first.

Our procedure generates 30% more test cases compared to the number of test cases currently defined by 3GPP. The summary of our result is shown in Table 1. We also find that 60 protocol interaction behavior is not defined even by 3GPP protocols standards. Refer to Table 2 that provides list of 10 new test cases and identifies new vulnerabilities in the 3GPP standard. The rest of 74 test cases expose relatively less serious issues, such as delay in service access, procedure repetition, downgrade to lower priority cell, temporary loop between device states (i.e. idle and connected states), two procedures temporarily blocking each others, and more. Now we provide brief analysis on three novel vulnerabilities discovered by our test cases (other than those discussed in Table 2).

Integrity and ciphering is not enforced We found that device can skip applying RRC ciphering and integrity protection even if they are enabled at the network. Such a scenario has been shown in Figure 4, where *Attach Request* message is forwarded to MME (Mobility Management Entity) (step 4) before RRC security procedure starts (step 6). If the UE sends *Security Mode Failure* message to eNodeB (Evolved NodeB is a term used for LTE base station), the 3GPP standard allows device to communicate with the network without any protection, whereas the *Attach* procedure is allowed to complete. We verify this from LTE RRC standard (Section 5.3.4 Initial security activation procedure in 3GPP TS 36.331[22]). The standard states that after sending *Security Mode Failure* message, the UE shall "continue using the configuration used *prior* to the reception of the *Security Mode Command* message, i.e. neither applies integrity protection nor ciphering". To address this issue, we create a test case that makes 5 retries of *Security Mode Command* message when it receives *Security Mode Failure* message from UE. On receiving 5^{th} *Security Mode Failure* message, the eNodeB bars the device to camp on its cell for small amount of time (60 seconds in our test case).

Sending data without RRC security success We find that the device can send uplink (UL) data even if it has failed

[3]EPS Connection Management (ECM) involve signalling connection that is made up of two parts: an RRC connection and an S1_MME connection. In this paper, RRC test cases are part of ECM procedure.

Table 2: Summary of novel findings. These use cases are not defined in 3GPP testing standard and potential vulnerabilities remain untested.

Issue	Protocols	Problem	Root Cause	Impact	3GPP test case exists?	Standard discusses issue?
Detaching victims	ECM – EMM	Device can send non-integrity protected Detach with cause *power off*	The standard allows certain types of messages can be sent as non-integrity protected	Adversary can let victim device detach.	No	No
Service provisioning	ECM – EMM	Local EPS bearer context is deactivated without ESM signalling	The device fails to establish user plane radio bearers when ECM process at network is delayed	EPS bearer context deactivated.	No	No
Skipping integrity	ECM – EMM	TAU message without integrity protection is accepted	TAU due to an inter-system change in idle mode is accepted by the MME even without integrity protection	The device reports wrong location to network.	No	No
Privacy leakage	EMM – ECM	After 4 retries from MME, the GUTI reallocation procedure stops	MME does not mark the device which has failed to perform GUTI reallocation procedure as vulnerable.	Using old GUTI compromises user location.	No	No
Null integrity	EMM – EMM	2^{nd} attach is processed by MME whose IE differs from 1^{st} attach	The device capability related information element (IE) in the Attach Req differs from the ones received earlier	Device attaches as non-integrity protected.	No	No
Barring to Attach	EMM – EMM	Processing Attach request without receiving Identity Response	The MME processes the Attach Request while waiting on Identity Response message from UE	Sending Attach instead of Identity Req bars UE.	No	No
Inconsistent states	EMM – EMM	Device proceeds Detach procedure whereas MME proceeds TAU	Before the detach request is received at UE, UE initiates TAU procedure. MME aborts detach and proceeds TAU	MME and Device states are inconsistent.	No	Yes, TS 24.301, 5.5.2.3.5 e
TAU is blocked	ESM – ECM	PDN procedure is blocked by RRC reconfiguration (doing TAU)	Bearer Modification and RRC-Reconfiguration with TAI change collide. TAU is blocked by earlier procedure	UE will end up keeping invalid tracking area.	No	No
Unauthorized connection	ECM – ECM	UE keeps radio connection for rejected RRC request	When the user identities are not found at EPC, the RRC req is rejected but UE remains camped on eNodeB cell	Connecting eNodeB with expired USIM cards.	No	Yes, TS 36.413, 8.3.3
Deadlock	ESM – ESM	UE and network both initiates Dedicated Bearer Procedures	Both UE and network has received/sent Activate Dedicated Bearer Request and enter into undefined behavior	Network and UE wait on each other request.	No	No

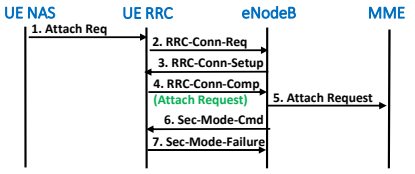


Figure 4: RRC integrity and ciphering is not applied

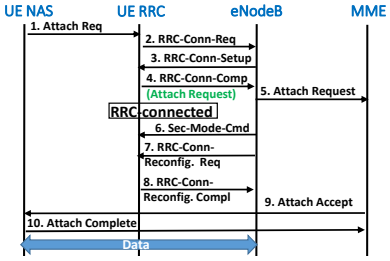


Figure 5: Data transmission starts without activating RRC security

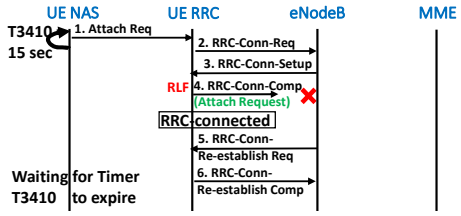


Figure 6: Re-Attach Request is delayed due to Radio Link Failure

to complete *RRC* security procedure, as shown in Figure 5. The device sends *Attach Request* message as piggybacked with *RRC Connection Complete* message and moves to *RRC-Connected*s state. In *RRC-Connected* state, the eNodeB sends *Security Mode Command* to UE (step 6). However, before it receives *Security Mode Response* from device, eNodeB establishes Signalling Radio Bearer 2 (SRB) for device uplink/downlink data by performing *RRC Connection Reconfiguration* procedure (steps 7-8). Meanwhile, the *Attach* procedure completes (step 9-10); whereas the device does not generate a security mode response at all. We find that the device is able to send UL data even if the RRC security procedure did not conclude. We discover that this is indeed a loophole in the standard (Section 5.3.5.3 Reception of an RRC Connection Reconfiguration procedure in 3GPP TS 36.331[22] and Note 3). It has been stated, "if the *RRC Connection Reconfiguration* message includes the establishment of radio bearers other than SRB1, the UE may start using these radio bearers immediately, i.e. there is no need to wait for an outstanding acknowledgment of the *Security Mode Complete* message." Note that eNodeB does not have any timer linked to security mode command and cannot resend *Security Mode Command* message, if the response to previous request is not made. To address this vulnerability, we add a test case that makes sure that the device has sent security mode response (either complete or reject).

Re-Attach Request is delayed In this issue, the device registration procedure is delayed upto 15 seconds (timer T3410's default value). Figure 6 shows that although the eNodeB has failed to receive *RRC Connection Complete* message piggybacking *Attach Request* from device, the device enters into *RRC Connected* state. Such failure of message comes because of Radio Link Failure (RLF). On RLF, the device recovers the radio connection by performing *RRC Connection Reestablishment* procedure (step 5-6), but does not re-send the *Attach Request* message. Therefore, UE NAS layer times out for Attach Request message and re-send the request. One can argue that *RRC Connection Complete* message which is sent over SRB1 will be recovered by Radio Link Control Acknowledgement procedure (RLC ACK). However, RLC procedure recovers the bit errors/retransmisison failures over the wireless link, and does not recover the failure because of UE getting out of sync with eNodeB cell (RLF scenario). Moreover, RLC ACK mode has very short timer value (default value of 45msec[22]) and cannot recover the failure when radio recovery procedure takes long.

To address this issue we create a test case in which EMM layer requests RRC layer for the notification of its piggybacked request. If RRC is not able to deliver the piggbacked message within a couple of seconds, the EMM layer will re-send the packet.

4 LTE TESTING EFFICIENCY

We first discuss inefficiencies on running LTE test cases and later put forward graph based test case execution methodology to address these inefficiencies.

4.1 Limitations and Challenges

Testing limitations and challenges arise during test cases scheduling, and their execution.

Test cases scheduling Protocol conformance testing validates whether a device complies with LTE protocols specifications or not. Each protocol supports a number of functions that logically separates one protocol from the other. For example, the main functions of *Radio Resource Layer (RRC)* protocol are establishing, configuring, maintaining and terminating the device wireless connectivity with LTE base station. Similarly, LTE *EPS Mobility Management (EMM)* protocol supports functions related to mobility of UE, that include device registration, authentication and security, location update, and deregistration with the LTE network. Each protocol function is further divided into multiple test cases, which supports multiple operations that further executes a number of steps. Figure 8 shows an example of EMM protocol *Attach* function that has several test cases. Each test case performs several operations, which finally execute test steps. In current testing practice, all test cases from a particular function must finish before test cases from other function could start. However, we find that these functions are independent from each other, their pre-conditions are different and do not overlap with each other. We take an example of test cases that validate two different functions (i.e. device *Attach* and *Tracking Area Update (TAU)* functions) of same protocol (i.e. EMM protocol). Device *Attach* function is responsible of device registration with the network, whereas the *TAU* function provides new device location to network when its location has changed. To initiate EMM protocol's *Attach* related test cases, the device must be *deregistered*; whereas to initiate EMM protocol's *TAU* related test cases, the device must be *registered and its location should have been changed*. We argue that two independent functions can be executed in parallel (such as EMM protocol's *Attach* and *TAU* functions), which is not currently supported by 3GPP (**inefficiency 1**).

Test case execution Protocol conformance testing certifies that each protocol function is executed for all possible usable conditions. To test each such condition, 3GPP defines a new test case that ensures correctness in terms of the signaling flow and content of each message. However, all such test cases execute repetitive operations, whereas a test operation pertaining to test condition is only sufficient to execute. For example, *Attach* function is validated for two different types of device identities (i.e. GUTI, and IMSI). These are test conditions for which two separate *Attach* test cases are defined (Test case 1 and Test case 2 in Figure 8). Both test cases validate the *Attach* operation when "Valid GUTI" is passed as a test condition, or "IMSI/GUTI reallocation" is set as a test condition. But we see that both these test cases perform a number of repetitive operations (such as *Power*, *RRC*, *Security*, and *ESM*) that are not related to test condition. As a result, test case 2 (Figure 8) executes (i.e. *exec()*) 63% redundant steps which are already performed by test case 1 (Figure 8) (**inefficiency 2**).

Operations	Steps	Time (sec.msec)
Power	Power off	00.00
	Power on	34.002
RRC	RRC Connection Req	34.009
	RRC Connection setup	34.104
	RRC Connection complete	34.147
Security	Authentication Request	34.150
	Authentication Response	34.285
	Security Mode Command	34.288
	Security Mode Complete	34.428
ESM	ESM Information Request	34.459
	ESM Information Response	34.770
Attach	Attach Accept	35.455
	Attach Complete	35.486

Table 3: Time taken for each step in one of LTE Attach test case

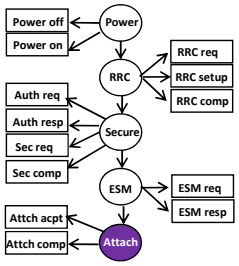


Figure 7: Graph data structure of test case execution

4.2 LTE Testing as a Graph Data Structure

We make LTE testing efficient through graph data structure approach.

Overview of our solution We view test case execution as a graph data structure, where graph non-leaf vertices represent protocol operations and leaf vertices represent the steps that protocol operations take, as shown in Figure 7. We aim to reduce the execution of those test steps which are common among different test cases of a protocol function (**mitigating inefficiency 2**). We let all test cases of a particular protocol function to execute one-by-one on single graph data structure. As an operation step executes, we record its complete execution information in memory, such as its output parameters. By doing so, we can find whether non-leaf vertex was previously visited or not. If it was previously visited then we retrieve the output of that operation steps from memory and move to next operation without executing already executed steps. Note that, we are not aiming to skip any operation step, rather to skip operation steps execution by retrieving their last execution output parameters. Furthermore, we change current test cases execution practice by allowing test cases from different protocol functions to co-execute (**alleviating inefficiency 1**).

Savings To quantify efficiency of our approach compared to contemporary testing approach, we execute LTE *Attach*

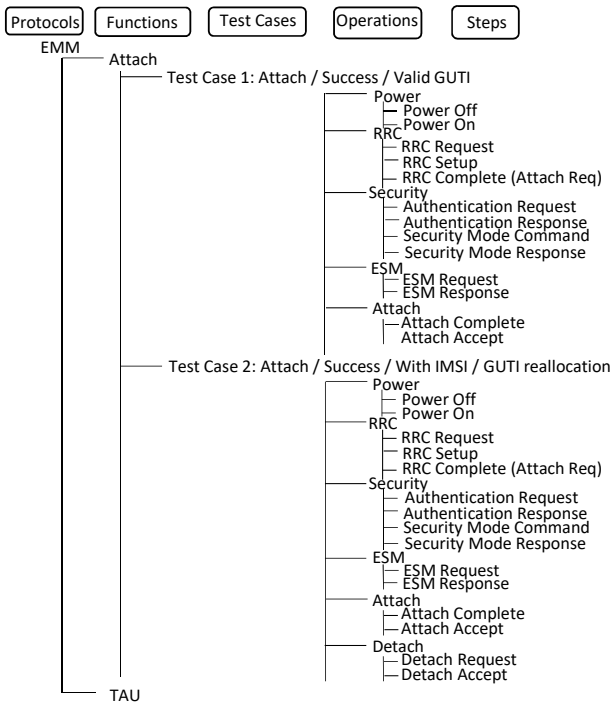


Figure 8: Most of the test case steps are repetitive among test functions. Each LTE protocol (EMM protocol in the Figure) tests a number of test cases (test case 1 and test case 2) that belong to particular LTE function (LTE Attach function). These test cases perform a number of operations (power cycle, RRC, security, etc.). These operations further execute a number of steps (RRC Reques, RRC Setup, RRC Complete etc.)

related test case over Anritsu test simulator[23]. Table 3 captures time taken by each step during LTE *Attach* test case. Overall LTE attach test case takes 35 seconds and 486 milliseconds to execute. If we avoid repeating only device power off/on steps for following 56 Attach related test cases then we can save roughly *31 minutes*. If we further avoid repeating execution of RRC, Security and ESM related steps then our testing time decreases from 1.486 seconds to merely about 0.031 seconds for each of the following *Attach* test case. Hence, in total, our approach has potential to decrease Attach related test cases (for example) from 33 minutes to 37.222 seconds, which is 53.66X (5366%) steps execution time savings. In this paper, we discuss test case efficiency in terms of number of steps; the lower number of steps a test case execute, the faster it runs.

Contributions We make two contributions. First, to achieve test case efficiency, we reduce test steps execution by reusing the results from those finished test cases. Second, we enable concurrency between different test cases and execute concurrent test cases in parallel to speed up the execution.

4.2.1 Optimizing test cases

In test cases optimization, we first initialize a graph that represents all the test cases of a protocol function. Then we perform graph traversal and avoid re-executing common operation steps.

Graph initialization Graph initialization is performed at the beginning of testing. It is a procedure that maps the test case to a graph representation. There are two major things done in graph initialization.

- Create the graph non-leaf vertices. Each non-leaf vertex represents an operation.
- Add leaf vertices to a non-leaf vertex. Each leaf vertex represents a step of an operation.

Since the purpose of the graph representation is to represent all test cases of a protocol function, repetition of common graph vertices is avoided as early as in the initialization phase. If a particular test operation performs an extra step (for example *RRC Connection Release*, other than performing three RRC steps – *RRC Connection Request*, *RRC Connection Setup*, and *RRC Connection Complete*), this step must be added with original operation (*RRC*) rather than creating a new *RRC* vertex with just one step. Such an approach provides a compact representation of graph data structure, reducing memory and space utilization and computation.

Graph traversal Graph traversal starts as soon as first test case of a protocol function executes. For the first most test case, all leaf and non-leaf vertices are executed. During such execution, each leaf vertex reports its execution parameters to associated non-leaf vertex. The non-leaf vertex *mark* itself visited and store reported parameters.

For each non-leaf vertex i we store an array of the vertices (its leaf vertices) adjacent to it. Each array element represents the reference to a character array that contains all parameters produced during test operation step execution. After the execution of first most test case, the second test case of same protocol function starts executing. The second test case is most probably executing the same set of operations as executed by first test case. However, one or two protocol operations are required to be re-executed because the test condition has changed. Therefore, simply looking at the *mark* field to know whether the operation was previously executed or not is misleading. For example, *Attach with valid GUTI*, and *Attach with IMSI* are two different test cases but perform mostly same operations (*RRC*, *Security*, *ESM*, and *Attach* operations, as shown in Figure 8), where they only differ at *Attach* operation execution. To address this issue we leverage domain knowledge and identify the operation(s) which are pertaining to the test case.We can do so by looking test case conformance requirements that describe why test is being executed and which protocol operations will be impacted. For example, both *Attach with valid GUTI*, and *Attach with IMSI* require the *Attach* operation must be validated under two different conditions (i.e. GUTI and IMSI). Therefore, even though *Attach* operation is *marked* by first test case as executed, we are required to re-execute the operation. To achieve this, we modify our implementation by first creating an hash

value of the test condition, and then associating all those leaf vertices of the operation that corresponds to the test condition with that hash value. Therefore, for each non-leaf vertex *i* the array of the vertices (its leaf vertices) adjacent to it are represented by hash value of the test condition.

Let's say that *Attach with valid GUTI* was the first most test case that has finished its execution. Now the second test case, say, *Attach with IMSI* starts its execution. We know that the hash value of the condition (IMSI) corresponds to *Attach* operation. Therefore, the running test case retrieves the *RRC*, *Security*, and *ESM* operations steps parameters from memory instead of executing these operations' steps and move to *Attach* operation. At *Attach* operation, the test case searches whether hash value of the condition (IMSI) exists or not. As it does not exist, the test case will execute all the steps of the *Attach* operation and proceeds to the next operation (if any).

Note that storing the leaf vertices and their parameters with different hash conditions may grow the memory size. To address this, we limit the number of test cases that can be traversed over a single graph. We can do this because there are a number of independent test cases within a same protocol function. For example, the *Attach* function supports attach related test cases under three different scenarios, i.e. *attach/success*, *attach/failure*, and *attach/abnormal* having 17, 23 and 11 test cases, respectively[1]. Therefore, the graph data structure testing *Attach* function only needs to store operation steps parameters associated with 23 different conditions.

Parallelizing test cases We parallelize mutually exclusive test cases to speed-up the overall testing time. The mutually exclusive test cases are those which are testing different test scenarios and do not share the testing condition, variables, and results with each other. We adopt two step procedure (1) enabling test case concurrency, and (2) running concurrent test cases in parallel. The first step is relatively simple and taken before test cases are executed. We feed complete list of test cases to our program which uses *switch()* statement to differentiate test cases based on their three different scenarios (i.e. success, failure, and abnormal). Test cases of same scenario are grouped together. In second step, we execute test cases belonging from three different scenario in parallel. We adopt process-oriented programming approach, based on ideas derived from CSP (Communicating Sequential Processes)[24], to achieve this. Each processes is a separate pieces of code which is independent from other process(es). Such programming approach does not create race hazards involving shared data or scheduling corner-cases, and deadlocks.

5 IMPLEMENTATION

Our implementation includes 3GPP test cases implementation, implementation of our proposed algorithms, and creation of FSMs and their representation as finite automaton, generating complete sets of test cases by excluding don't care output device outputs. Our implementation is highly modular with the focus of code re-use.

Test cases and their execution For 3GPP test case implementation, we implement RRC, EMM and ESM test cases as described in 3GPP specification 36.523-1[1] section series 8, 9 and 10. We carefully implemented each test cases by following the test case procedure described in the specification and executed the test as a message sequence between device and NS. The device and NS are two processes running in Linux machine. The device generates a message for NS, where NS generates the response (as described by 3GPP specification) by following pre-defined behavior. In case, the device does not receive a response from NS (which is usually the case in our protocol interaction testing) then we mark it as vulnerable/missing test case and manually confirm it with 3GPP test standard, as well 3GPP protocol specifications. Before, each test starts running, our program takes a set of configuration defined from a number of *config* files as required by test case. We create different *config* files representing different purposes, such as *preamble.config*, *cells.config*, *timers.config*, *ie.config*, representing test start states, cells related config, timers and their values, and information elements including device capabilities, respectively.

Finite state machine The test case is executed such that the execution is captured as a transition between different states. Such an implementation choice is important to first represent test cases as a finite automaton, and then generating new test cases for protocol interaction. The current states, next states, and transition conditions are *enum* type values. For each state, the set of valid state transitions are stored as a *multimap*. The transition function is an action that current state performs and transitions to next state. The action is basically a callback function that tells what steps the UE should perform and for which next state (*ActionCallbackFunc callback = stateTransition[std::pair(current_State, next_State)]*). Using this logic, we can easily represent our FSM as a finite automaton. In finite automaton, the current state is allowed to have more than one transition (DFA or NFA).
We modify the Hopfcroft algorithm code provided by Antti Valmari[25] and defined contradictory states as like equivalent states.

Protocol Interaction Protocol interaction is represented as test cases collision and their interaction (with/without delay), where each test case represents either same or two different protocols. Such interaction has to be tested on each valid output value (in any sequence). To implement this, the messages that a test case produce during testing, as well as their corresponding responses are put in the queue. The execution of this test case is basically what the 3GPP testing standard mostly tests (single protocol interaction). To find protocol interaction vulnerabilities, we let messages from two protocols interact

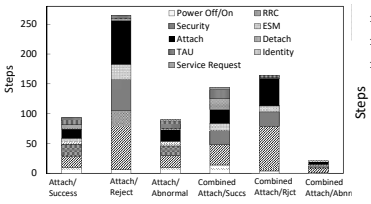


Figure 9: No. of steps in Attach func.

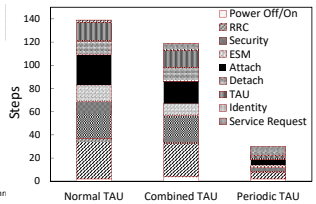


Figure 10: No. of steps in TAU func.

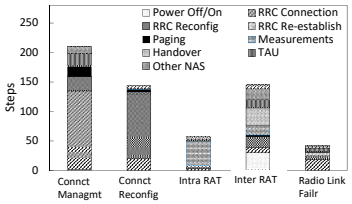


Figure 11: No. of steps in RRC func.

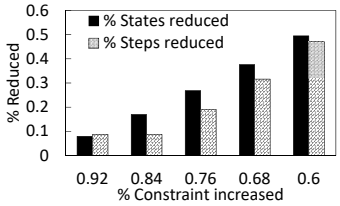


Figure 12: No. of constraint vs No. of steps

with the NS and then observe the behavior. Each vulnerability that we find was manually verified in 3GPP protocol specifications. For each vulnerability, we propose a new test case that describes the procedure (by detailing the test steps) and the fix (the expected behavior).

6 EVALUATION

6.1 Completeness

To evaluate test cases completeness, we are mainly interested to find that how many number of states our algorism reduces when compared with best known finite automaton algorithms. We also want to know how many new test cases we can find without enumerating all possible output sequences (results are discussed in Section 3.5, Table 1 . Section 3.5 also provides detail evaluation analysis). For algorithms comparison, we vary total number of NFA states from 100 states to 500 states and assuming half of these states have 1 transition condition. In Section 3.3.1, we discuss that in LTE most of NFA states have only 1 transition condition. Table 4 compares our algorithm with Robins and Scott algorithm on NFA to DFA conversation. Robins and Scott algorithm converts all NFA states to DFA, therefore, the number of steps and states are always exponential to total number of NFA states (which is being converted). After powerset conversion, the algorithm converges by removing unreachable states and optimal number of states can be obtained. However, the power set conversion at the first step is the major bottleneck. On the other hand, our algorithm only converts those NFA states which have 1 transition condition. We can reduce 1/3 of these NFA states on average. Our algorithm makes NFA to DFA conversion linear through selective state conversion at the cost of just 10% more states.

Table 5 compares Hopcroft DFA minimization algorithm with our proposed algorithm. The Hopcroft algorithm does not work on NFA-DFA mixed FSM, therefore, we only show DFA states in Table 5. The Hopcroft algorithm cannot get benefit from constraints between different states, whereas, our algorithm reduces more states by considering states constraints. We find that we can reduce more number of states by adding fewer constraints in the FSMs (as depicted by Figure 12). Table 5 shows that we can reduce 26% states by adding just 20% constraints. Further, our algorithm only performs 10% more steps than Hopcroft algorithm, where it skips NFA minimization (which is an NP complete problem), and merges those states which have constraints (treating them as equivalent states).

Table 4: Our algorithm comparison with Robin and Scott algorithm

States			Robin and Scott Algorithm			Proposed Algorithm		
NFA (Total)	1 Trans NFA	DFA (Total)	Steps	States	Optimal States	Steps	Reduced DFA	Total States
100	50	100	2^{200}	2^{200}	167	600	34	184
200	100	200	2^{400}	2^{400}	334	1381	67	367
300	150	300	2^{600}	2^{600}	500	2230	100	550
400	200	400	2^{800}	2^{800}	667	3123	134	734
500	250	500	2^{1000}	2^{1000}	834	4049	167	917

Table 5: Our algorithm comparison with Hopcroft algorithm

States			Hopcroft Algorithm		Proposed Algorithm	
DFA (Total)	Equivalent	Constraints	Steps	States	Steps	States
100	50	20	400	75	440	56
200	100	40	921	150	1013	111
300	150	60	1487	225	1635	166
400	200	80	2082	300	2290	221
500	250	100	2699	375	2969	276

6.2 Efficiency

We measure the efficiency of a test case as the number of steps it executes. The rationale is that every test case consists of a number of test operations where every operation is executing a number of steps. Because these test steps execute sequentially (one after the other) in a test case, their execution time reflects into testing time. We show that we can efficiently execute LTE test cases by executing them in a systematic way. Table 6 shows the test case execution comparison between adhoc way of testing (as used by LTE standard) and systematic way of testing (through our solution). First, we see that our approach executes 43%, 11%, 70% and 50% less number of steps for *Attach*, *Detach*, *TAU*, and *Service Request* functions, respectively. This is because, graph data structure shares a test case execution knowledge with all other test cases. We did not save much in *Detach* function, because these tests either require the device to reboot or USIM to be removed. Therefore, our graph data structure cannot hold the information about previous test cases and all the steps in the new test case have to be re-executed. The saving from *Attach* function comes because our algorithm does not always execute the reboot device steps, unless explicitly mentioned by the test

case. In *TAU* function our algorithm executes 34 number of *TAU* related steps compared to 375 steps execution by 3GPP standard. We find that most of TAU steps (including same tracking area code, and other configurations) are repetitive and their execution can be avoided by simply retrieving the execution outcome from the memory.

Figure 9, Figure 10, and Figure 11 show the number of steps taken at different test cases belonging to *Attach*, *TAU* and *RRC* functions, respectively. We can see that *Attach* function can be split into 6 independent test case scenarios, making their parallel execution possible. Similarly, *TAU* and *RRC* functions can be split into 3 and 5 test case scenarios, respectively, which can also execute in parallel. The *Detach* and *Service Request* functions (not shown in the Figure) do not offer any parallelism. Note that in *Detach* function, "UE initiated" and "network initiated" detach test cases cannot be separated because UE needs to be physically rebooted. However, there is no much gain even if we are able to parallelize these functions. This is because *Detach* and *service request* have only 12 and 10 test cases, respectively, compared to *51*, *56* and *114* test cases for *Attach*, *TAU* and *RRC* functions.

Table 6: Comparison of number of steps without (original in test case standard) and with our optimization approach (based on graph data structure)

	Steps Before Optimization				**Steps After Optimization**			
Operation	**Attach**	**Detach**	**TAU**	**SR**	**Attach**	**Detach**	**TAU**	**SR**
Power Off/On	118	13	20	10	46	12	8	10
RRC	465	54	186	94	254	46	70	35
Security	216	40	164	28	140	32	60	16
ESM	100	14	54	12	68	14	26	8
Attach Req	195	13	68	16	65	11	15	7
Attach Success	130	18	70	28	68	16	34	8
Attach Reject	50	1	1	NA	47	1	1	NA
Detach	42	36	24	16	36	36	24	16
TAU	52	15	375	10	47	12	34	6
Identity Req/Resp	2	2	NA	NA	2	2	0	NA
Service Req/Resp	8	2	20	26	4	2	16	14
Total	**1378**	**208**	**982**	**240**	**777**	**184**	**288**	**120**

7 RELATED WORK

We are unaware of earlier work that has either discussed efficient execution of LTE test cases or provided a methodology to generate complete set of test cases. Closest to our work are those that discuss cellular protocol interactions, such as [26], [27] and [28]. Both [26] and [27] disclose performance issues in operational LTE network. They discuss 3G and 4G interaction issues and do not provide LTE protocol interaction analysis in general. [27] discusses LTE performance related issues due to ARQ-HARQ protocols interactions. Other works [29][30] discuss practical attacks in LTE network. In contrast, our work argues that problems reported by LTEInspector[29] and [30] will not occur if LTE testing is complete. [31][32] discuss importance of performance related test cases and provide LTE performance results. [33] performs fuzzing tests to identify LTE RLC vulnerabilities, whereas [34] discusses energy and delay analysis on RACH performance. All these works have either provided new ways of LTE testing or discussed LTE performance issues. However, in our work, we discuss making LTE testing more time efficient by reducing number of execution steps, and provide a methodology that can generate complete set of LTE test cases.

Other relevant works are related to wireless and network protocol testing[35][36][37], testing using model checkers[38][39][40], and test cases generation by learning queries[41] and finite state machines[42][43]. [35] solves runtime wireless protocol validation by sniffing wireless transmission first and later adding nondeterministic transitions to incorporate uncertainty. Their technique cannot solve the NP-completeness coming from searching and uses heuristics to limit the search. [36] and [37] discuss model based approach for NFV testing and network fault detection, respectively. Both approaches model the network nodes as FSMs and generates test traffic for FSM execution. However, [36] and [37] fail to provide complete list of test cases and do not discuss the efficiency of their approaches. [38] and [39] verify the state-space exploration. They either require constraint metrics as input or require to search all system states. In this paper, we discuss that finding all possible inputs are practically not feasible for LTE testing. [40] uses model checking to find TCP implementation bugs, whereas, our approach does not aim to find implementation bugs. Angluin seminal work[41] learns test cases through learner and teacher interaction. Contrary to [41], we do not require the device to learn by interacting with the network, rather, our approach finds valid input values through device FSM transitions. [42][43] solve the nondeterminism of FSM by keeping the input alphabet small, whereas, our approach does not constraint the input alphabet.

8 CONCLUSION

We present the first methodical approach to LTE testing. We provide complete list of test cases by considering multiple protocols interaction, and exclude those test cases whose corresponding output message combinations are not generated in protocols interaction. We also optimize LTE test cases by avoiding re-execution of repetitive steps among different test cases.

Future work In future, we will consider other test case scenarios which are being developed for 5G networks, such as LTE-WiFi co-existence, test cases with micro base stations, and others.

ACKNOWLEDGEMENT

We thank our shepherd Dr. Sunghyun Choi and the anonymous reviewers for their insightful comments. This work was partly funded by NSF grants 1423576 and 1526985.

REFERENCES

[1] 3GPP TS 36.523-1: Protocol conformance specification, 2018.

[2] Anite maintains LTE conformance testing lead. http://www.anite.com/businesses/handset-testing/news/anite-maintains-lte-conformance-testing-lead.

[3] Anritsu conformance test systems. https://www.anritsu.com/en-US/test-measurement/mobile-wireless-communications/conformance-test-systems.

[4] Anite conformance test systems. http://www.anite.com/businesses/handset-testing/our-products.

[5] Anritsu: How conformance tests are carried out. http://dl.cdn-anritsu.com/en-en/test-measurement/files/Product-Introductions/Product-Introduction/me7873la-el1100.pdf.

[6] Anite test documents. http://www.keysight.com/en/pd-2372474-pn-E7515A/uxm-wireless-test-set?pm=PL&nid=-33762.1078013&cc=US&lc=eng.

[7] UE demonstration of conformance testing – Anite. http://www.anite.com/businesses/handset-testing/our-products.

[8] 3GPP implementation conformance statement. http://www.etsi.org/deliver/etsi_ts/136500_136599/13652302/11.03.00_60/ts_136523 02v110300p.pdf.

[9] H. Gruber and M. Holzer. Computational Complexity of NFA Minimization for Finite and Unary Languages. *LATA*, 8:261–272, 2007.

[10] Y.-H. E. Yang and V. K. Prasanna. Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In *IEEE Infocom 2011*.

[11] M. Sipser. Chapter 1: Regular languages. *Introduction to the Theory of Computation*, pages 31–90, 1998.

[12] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[13] M. Sipser. *Theorem 1.19 in Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[14] F. R. Moore. On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. *IEEE Transactions on computers*, 100(10):1211–1214, 1971.

[15] K. Salomaa and S. Yu. NFA to DFA transformation for finite languages. In *International Workshop on Implementing Automata*, 1996.

[16] R. Mandl. Precise bounds associated with the subset construction on various classes of nondeterministic finite automata. In *Inf & Sys Sciences*, 1973.

[17] J.-M. Champarnaud, A. Khorsi, and T. Paranthoën. Split and join for minimizing: Brzozowski's algorithm. *Stringology*, 2002:96–104, 2002.

[18] Hopcroft. An/n log n algo for minimiz. states in kf in ite automaton. 1971.

[19] M. Almeida, N. Moreira, and R. Reis. On the performance of automata minimization algorithms. *Logic and Theory of Algorithms*, page 3, 2007.

[20] Gómez and et al. DFA minimization: from Brzozowski to Hopcroft. 2013.

[21] 3GPP. TS24.301: Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3, Jun. 2018.

[22] 3GPP. TS36.331: Radio Resource Control (RRC), 2018.

[23] Anritsu: verification and test for deployment of LTE. http://www.chinacom.tw/ngn2010/pdf/ngn/Anritsu.pdf.

[24] Easy Concurrency for C++. https://www.cs.kent.ac.uk/projects/ofa/c++csp/.

[25] DFA minimization using Hopcroft alogirithm: C++ implementation. http://www.cs.tut.fi/~ava/DFA_minimizer.cc.

[26] G.-H. Tu, Y. Li, C. Peng, C.-Y. Li, H. Wang, and S. Lu. Control-Plane Protocol Interactions in Cellular Networks. In *ACM SIGCOMM*, August 2014.

[27] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An in-depth study of LTE: effect of network protocol and application behavior on performance. In *ACM SIGCOMM*, 2013.

[28] C.-H. Chuang and et al. Performance study for HARQ–ARQ interaction of LTE. *Wireless Communications and Mobile Comp.*, 10(11):1459–1469, 2010.

[29] Hussain, Syed Rafiul and Chowdhury, Omar and Mehnaz, Shagufta and Bertino, Elisa. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[30] A. Shaik and et al. Practical attacks against privacy and availability in 4G/LTE mobile communication systems. 2015.

[31] R. Subramanian, K. Sandrasegaran, and X. Kong. Benchmarking of real-time LTE network in dynamic environment. In *IEEE APCC*, 2016.

[32] H. Zhao and H. Jiang. LTE-M system performance of integrated services based on field test results. In *IEEE IMCEC*, 2016.

[33] B. Cui, S. Feng, Q. Xiao, and M. Li. Detection of LTE Protocol Based on Format Fuzz. In *IEEE BWCCA*, 2015.

[34] Gerasimenko and et al. Energy and delay analysis of LTE-advanced RACH performance under MTC overload. In *IEEE Globecom Workshops*, 2012.

[35] Shi, Jinghao and Lahiri, Shuvendu K and Chandra, Ranveer and Challen, Geoffrey. Wireless protocol validation under uncertainty. In *International Conference on Runtime Verification*. Springer, 2016.

[36] Fayaz, Seyed Kaveh and Yu, Tianlong and Tobioka, Yoshiaki and Chaki, Sagar and Sekar, Vyas. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *NSDI*, 2016.

[37] Lee, David and Netravali, Arun N and Sabnani, Krishan K and Sugla, Binay and John, Ajita. Passive testing and applications to network management. In *International Conference on Network Protocols*. IEEE, 1997.

[38] Godefroid, and Patrice. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997.

[39] Khurshid, Sarfraz and Păsăreanu, Corina S and Visser, Willem. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003.

[40] Fiterău-Broştean, Paul and Janssen, Ramon and Vaandrager, Frits. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification*. Springer, 2016.

[41] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. In *IEEE Information and Computation*, 1987.

[42] T.S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 1978.

[43] Petrenko, and Alexandre. Toward testing from finite state machines with symbolic inputs and outputs. *Software & Systems Modeling*, 2017.