Effectively Testing of Timed Composite Systems using Test Case Prioritization*

Huu Nghia Nguyen
Montimage EURL
Paris, France
huunghia.nguyen@montimage.com

Fatiha Zaïdi
LRI-CNRS, Univ. Paris-Saclay,
91405, Orsay, France
fatiha.zaidi@lri.fr

Ana R. Cavalli
SAMOVAR-CNRS, Télécom SudParis
Univ. Paris-Saclay, Évry, France
ana.cavalli@telecom-sudparis.eu

***Abstract*—A composite system consists of several components which can be developed separately and deployed in distributed environments. Executing test cases on such kind of systems requires more effort due to their size and their distributed environments. A critical issue is to prioritize efficient test cases to be firstly executed. We present in this paper a framework to generate test cases and to select the efficient ones to test the composite systems with taking into account time properties. Particularly, the framework generates a set of test cases based on a model of the system, which cover a given test objective. The test cases are then prioritized in an execution order to detect quickly faults, thus reducing the efforts of test execution and increasing the effectiveness of the testing process. The framework is complemented with an open-source toolchain for automating test case generation. It has been experimentally evaluated on the European Train Control System case study. The initial results show that the approach can save 40% of test execution effort.**

I. Introduction

Even if formal methods can be used to avoid faults in the design and implementation processes, such as by generating code skeleton of system from its specification, testing remains the only means to gain some confidence in a final product [1]. The testing process consists of test case generation and test case execution. The model-based testing approach generates test cases based on formal models that represent the expected behaviors of System Under Tests (SUTs) rather than on source codes of the SUTs. Such testing process is known as black-box testing. An advantage of black-box test case generation is that it is possible to perform it as soon as a specification model is available, that is, before the production code is written.

The testing process is highly dependent on the faults detection ability of test cases. Test case generation is a main area of research in the field of software testing. Efficient test cases decrease the chances of failure of the system and ensure the quality of the system. They are very important in distributed testing of a composite system in which we need to execute them on the overall system. This is usually hard because tests must be deployed on a high number of components which can be autonomous, developed independently, and deployed in a (real) distributed environment. Metrics, *e.g.*, [2], used to evaluate test cases efficiency are usually based on their execution results such as, number of real faults detected, execution time, code coverage. This means that a test case can be known as efficient only after it is executed. The question is how we can predict a test case is better than another one in order to execute it firstly. Furthermore, as the set of faults in a SUT is usually unknown, the definition of an efficient test case based on the number of real faults, which are detected by the test case, is not useful to practitioners who are creating test cases, nor to researchers who are creating and evaluating tools that generate test cases

The authors in [3] deal with distributed testing of systems that interact with their environment at physically distributed interfaces, called ports. The authors present several distributed conformance relations based on ioco, *e.g.*, dioco, c-dioco, p-dioco. An algorithm of test generation for each relation is also presented. The approach is based on a passive testing approach, *i.e.*, the testers do not send stimulus to the SUT but only observe it. The generation of test cases is not tackled in their framework. In [4], the authors use Timed Input Output Transition System (TIOTS) to theoretically reason about conformance, then they propose to use Timed Input Output Symbolic Transition System to describe test models. It gives an algorithm applying to offline testing to check observed logs of SUT against traces of the specification.

Optimization of test cases has been studied since long time [5]. Besides test cases prioritization approach for optimizing, there exists another approach, such as [6], [7], that tries to minimize the number of test cases in a test suite. This technique uses information about the program and the test suite to remove test cases which became redundant with time. An advantage of the prioritization with respect to this approach is that it does not discard or permanently remove some test cases from the test suite. Our framework can be used by both approaches as it gives the fault detection abilities of test cases. Indeed, one is able, based on these abilities, to decide, depending on the selected test strategy, either to execute firstly a test case, *e.g.*, when its ability is high; or to remove a test case, *e.g.*, when it can kill mutants that can be killed by another test case having higher ability.

We propose an effective test cases generation method applied to timed composite systems to maximize the ability of faults detection, consequently minimizing the effort, time and cost of tests execution. Our contributions are mainly on: (i) a formal model of timed distributed composite systems described by means of cooperating TIOTSs, (ii) a generation of distributed test cases, (iii) an evaluation of generated test cases to predict their abilities of fault detection, and (iv) the availability of an open-source toolchain[1] to support the automatic generation and evaluation of test cases.

The rest of the paper is structured as follows. Section II contains the basis of the proposed approach. Section III introduces an experimental evaluation of the proposed approach on the European Train Control System (ETCS). We also present

The work described in this paper has been partially financed by the ITEA3 MEASURE project nº 14009

DOI reference number: 10.18293/SEKE2016-153

[1] The tools are freely available under GPL 2.0 licence at http://github.com/nhnghia/testgen-ifx, and https://github.com/nhnghia/if2dot/tree/2java

in this section our tools' implementation for automatically generating and evaluating test cases and an evaluation of the scalability of the tools. We give the conclusion and future work in Section IV.

II. Proposed Approach

The SUT in our approach is defined by n components. Its testing system has n testers, each tester is attached to one component in order to test it. There is no need of communication between the testers. Each tester has a clock. These clocks progress at the same rate. A tester acts as the environment of its component, *e.g.*, it can send messages to the component and receive the responses from the component. It can also observe all inputs and outputs of its component with other components. A tester is put nearly enough with its component such that the communications between them have no delay, although the communications between components may have delays.

Let us take a simple example of a composite system having two components p^1 and p^2. The component p^1 can output either a or c, denoted as $!a{+}!c$, while p^2 can receive either a or b, denoted as $?a{+}?b$. It is easy to see that their composition can do $!a$, then $?a$ and that p^1 (resp. p^2) cannot do $!c$ (resp. $?b$) in the composition. A test case tc of the system should be the sequence $!a; ?a$. It is projected to get local test cases: a local test case of p^1 is $!a$ while the one of p^2 is $?a$. In the test case execution process, let say the tester of p^1 sees that p^1 emits a while the one of p^2 sees that p^2 receives b, *e.g.*, it is sent by the network. In that case, we will have two local verdicts: *pass* for the first tester, and *fail* for the second one. Hence the verdict of tc will be *fail*, thus a fault is detected. Let us note that the local test of p^2 cannot detect this fault if it is tested separately since $?b$ is allowed by the local model of p^2.

Modeling. We use TIOTSs to model components of SUTs. Basically, a TIOTS is a Labeled Transition System (LTS) over the set of (real) events with inputs $\mathcal{I}$ and outputs $\mathcal{O}$, and the set of time distances $\mathcal{D}$ between events. Time distances between events are measured by duration variables $d \in \mathcal{D}$, *e.g.*, they are positive real numbers. A transition is *untimed* if it is labeled by an event α with $\alpha \in \mathcal{I} \cup \mathcal{O}$; or *timed* if it is labeled by a duration d with $d \in \mathcal{D}$. A *trace* of a TIOTS is defined as a sequence of labels $tc = \langle l_1, l_2, \ldots, l_n \rangle$ with $l_i \in \mathcal{L}$ if there exist a sequence of transitions labeled by $l_1, ..., l_n$ respectively.

A SUT is a system consisting of several components. A component can interact with other components in the SUT but also with the environment that can be considered as a special participant of the SUT. Each component is identified by a unique name in its composite system. It has a model which describes the expected behaviors realized by the component. We model a component by a TIOTS. The basic events of a component are defined by a set of inputs or outputs of the component with the other participants in the composite system or with its environment.

Let $p^1, p^2, \ldots p^n, e$ be a finite set of identifiers of components participating in the composite system to be tested, and $a, b, \ldots \in \mathcal{M}$ be a finite set of messages, an *output* realized by the component p^1 to p^2 is denoted by $!a^{[1,2]}$ while $?b^{[2,1]}$ denotes an *input* b of p^1 from p^2. In this definition, we use a special identity e to denote the *environment*, *e.g.*, $?a^{[e,1]}$ denotes an input a of the component p^1 from the environment.

We also model *timed composite systems* by TIOTSs. The legal behaviors of a composition depends on the communication model used for the description of the message exchanges between its components. A communication model is characterized by a set of queues being put among components. A queue Δ is characterized by its size and its kind of message ordering, *e.g.*, ordered or unordered. We proposed 5 rules and their symmetrical rules [8] to construct a composition model from n local models.

Generating Test Cases. A *(timed) test case* is a sequence of inputs, outputs and durations. It represents a trace of a TIOTS and it covers some test objectives. We distinguish two kinds of test cases: global test cases and local ones. A *global test case* tc is a test case generated from a model of a composition system. On the contrary, a *local test case* tc^i of a component p^i participating in the system contains only events concerning the component. Once having a global test case, we need to project it on each participant component to get local test cases.

A *local test case* of tester t^k that tests the component p^k of the SUT is a sequence $tc^k = \langle d_1, \alpha_1, d_2, \alpha_2, \ldots, \alpha_m \rangle$, where $d_i \in \mathcal{D}$ and each α_i is one of the following, where e denotes the environment:

- $!b^{[i,e]}$: an output of message b to the tester
- $?a^{[e,i]}$: an input of message a sent by the tester
- $!b^{[i,j]}$: an observation of a send message b of p^i to its component partner p^j, with $i \neq j$
- $?a^{[j,i]}$: an observation of a reception message a of p^i from its component partner p^j, with $i \neq j$

This definition requires that a test case always ends by a real event that could be an input or an output. Indeed, if we had a test case $\langle d_1, \alpha_1, \ldots \alpha_n, d_n \rangle$ with $d_n \neq 0$, then a tester would need to wait d_n time units before giving its final verdict. Thus this delay is not necessary as it has no effect to the verdict.

The *projection* process inputs a global test case tc and outputs a set of local test cases $\{tc^1, \ldots, tc^n\}$. The local test case tc^k is obtained from tc by retaining only the events such that the component p^k participates in, *e.g.*, $!a^{[i,j]}$ or $?b^{[i,j]}$ with $k \in \{i, j\}$. The durations are preserved in all local test cases. Since an event, that does not concern p^k, will be removed in tc^k, there may be two durations that will be put successively in tc^k. We finally need to sum up all consecutive durations of local test cases.

A *verdict* of a local test case is given by comparing the expected result in the test case with the result given by the test execution. That is either an output, or a duration after that an output should happen. Let us take an example, a local tester needs to execute the local test case: $!a^{[e,1]}; 1; !b^{[1,2]}$. After the tester sends a to p^1, there are the following possibilities:

- if the tester receives an output from p^1 then the verdict is *fail*,
- if the tester sees a message from p^1 to p^2:
  - if the message is not b then the verdict is *fail*,
  - otherwise:
    - if the duration between the two messages equals 1 then the verdict is *pass*,
    - otherwise the verdict is *fail*

The execution of a global test case is done through executing its local test cases. A global test case tc gives a *pass* verdict only if all of its local test cases give *pass* verdicts. Basically, the test execution of a global test case will pass the following steps:

1) Generate local test cases $\{tc^1, \ldots, tc^n\}$ from tc
2) Set the local test case tc^i to the local tester t^i
3) Launch each local tester
4) Wait until all testers finish
5) Emit a verdict *pass* if all testers emit *pass*, otherwise *fail*

Prioritizing Test Cases. An important factor to evaluate a good test case is the number of real faults detected by the test case. This means that the evaluation can be performed only after the execution of test cases. We propose to evaluate a test case by executing it against a simulator rather than a real implementation of the SUT. Thus it is evaluated even the SUT has not been yet completely implemented. This also permits to use fault injection testing [9] to evaluate the fault detection ability of the test case.

We use the mutation analysis technique to evaluate the prioritization of the generated test cases. Mutation analysis is a well-known approach to assess the quality of test cases or testing techniques [10]. We have built a simulator programming in Java for each component of a SUT. A simulator acts with respect to the model of its component. A SUT has n components that will be simulated by n simulators executing together. Artificial faults are then injected into the simulators. Each mutated version of a simulator, called *mutant*, is tested against test cases to detect deviations in behaviors of the original version that differ from the mutant, *i.e.*, the verdicts should be *fail*. We consider a mutant being killed by a test case if the verdict of executing the test case on the mutant is *fail*, *i.e.*, a fault is detected by the test case. A test case which does not kill (or detect) any mutants (or faults) is considered defective. The mutation score of a test case is measured by the percentage of mutants being killed by itself. Generally, a test case that has a higher mutation score is assumed to detect more real faults than the one that has a lower mutation score [11]. The evaluation of a local test case passes the following steps:

1) Generate a set of simulators $\{s^1, \ldots, s^n\}$ from models of n components of the SUT,
2) Inject faults in each simulator s^i to obtain its mutants,
3) For each global test case tc:
   a) Project to local test cases $\{tc^1, \ldots, tc^n\}$,
   b) Perform local test case tc^i on each mutant of s^i,
   c) Calculate the number of mutants being killed,
   d) Set mutation score of the test case tc to the percent of mutants being killed.

We prioritize the generated test cases based on their mutation scores for the goal of fault detection rate, that is a measure of how quickly faults are detected during the testing process:

1) The first test case being selected is the one having the highest mutation score
2) The next test case is the one that kills the most mutants that are not killed by the previous test cases. This means that the second test case has the ability of detecting faults that cannot be detected by the first one
3) If two test cases tc_1 and tc_2 have the same mutation score, and the number of events of tc_1 is greater than the ones of tc_2, then tc_2 has a higher priority than tc_1 as its tester will have fewer interactions with the SUT but the same mutation score, *i.e.*, same ability of fault detection.

III. Experimental Validation

Tool Implementation. We have built two open-source tools to support the framework, the TestGen-IFx to generate test cases, and IF2Java to generate Java simulator from Intermediate Format (IF) specification. IF is a formal language used to describe models of real-time systems. We use IF language to describe the models of each component of the SUT rather than using directly TIOTSs which are more intuitive to reason about the framework such as timed model, composition, trace, test cases. It allows to describe quickly a model due to its expressiveness: the IF models of the case study presented in this paper have 8 states and 14 transitions but their TIOTSs contain totally 49488 states and 80598 transitions. The readers are invited to refer to [12] for further details of IF language.

The TestGen-IFx tool has been developed to generate distributed test cases for distributed testing. It inputs an IF specification file, and a configuration file. The IF specification file contains all models of components of the SUT, each model is represented by an IF process. The selected exploration strategy is recorded into a configuration file. Depending on the kind of the strategy, some other parameters may be required, *e.g.*, search depth number, and test objectives. The main output of the tool is a set of local test cases. For each global test case found, the tool projects it in n files, each one contains a local test. The tool also displays statistics about the test generation process such as execution time, number of generated test cases.

The IF2Java tool has been developed for generating simulators that is used for the evaluation of the test cases generated by TestGen-IFx. It inputs an IF file, and outputs a Java file. The IF file contains models of components of a timed distributed system. The Java file contains several Java classes which each one describes the behavior of a component. We obtain a Java program, called simulator, after compiling the Java file and our simulator library. The simulator acts, *e.g.*, receiving inputs and sending outputs, *wrt.* its model, which is specified in the IF file. When executing, the simulator inputs a local test case, then gives a verdict.

Case Study Description. The ETCS is an automatic train control system designed to replace progressively the incompatible safety systems currently used by European railways. In the ETCS level 3 [13], a train is equipped with an Onboard Unit (OBU) and it is controlled by a Radio Block Center (RBC). We consider the ETCS as a composite system consisting of two distributed components, OBU and RBC, running in parallel, see Figure 1, and communicating by message exchanges via Global System for Mobile Communications - Railway.

In the ETCS, a train moves in virtual *moving blocks* defined by its RBC via Movement Authorities (MAs). A MA defines a location, called End Of Authority (EOA), to which the train is authorized to move. Beyond this location, the location is a danger point such as the entry point of an occupied block section or the position of the safe rear end of a precedent train. To ensure the train is able to stop at the given EOA, a MA contains also a *release speed* that is a speed limit under which the train is allowed to run. We do not present its formal models due to

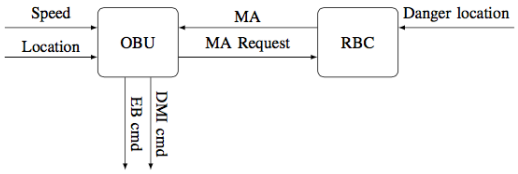


Fig. 1. Interaction between ETCS Components

lack of space[2]. Particularly, the OBU receives its information about its current estimated location, *e.g.*, `ELocation`, then sends this location to RBC via MA requests. Based on received information: location of the train, and location of danger point, *e.g.*, `DLocation`, the RBC determines MAs of the train and sends it to the OBU. The release speeds are calculated by the procedure `getReleaseSpeed`. The calculation is based on the distance from the train to the EOA and on the capacity of the brake system of the train such that the train is able to stop the train at the EOA. By comparing the current estimated speed of the train, *e.g.*, `ESpeed`, to the release speed encapsulated in MA, the OBU may generate an emergency brake command, *e.g.*, `EBcmd(1)`, and Driver Machine Interface (DMI) commands to display relevant information to the driver, *e.g.*, `DMIcmd(speed)`. The brake is applied until the train stopped [13].

To focus on release speed monitoring, we consider only release speed and distance to EOA in MAs. Since the MAs are needed to update periodically, the default period is 60s [13], we set one clock in OBU to issue a `MARequest`; and two clocks in RBC to send `MA` to OBU and to get `DLocation`. These clocks are reset when their events have occurred. For the purpose of experimental evaluation of the approach, we consider `speed` from 0 to 240 (km/h), `location` and `distance` from 0 to 300 (km), `EBcmd` from 0 (no brake) to 1 (brake) and `clock` from 0 to 60 (second).

Test Generation. A critical property of the ETCS to be tested is the capability of the system of taking over control if the driver appears to be going too fast. We need to consider a lot of scenarios to test this property such as when a break command is issued, when the current speed of the train passes over the limited speed, the break is released only if the speed is less than the limit speed, and so on. We present in the detail the test case generation for a specific scenario that represents the situation which caused the Spain train accident on 24 July 2013[3]. Considering that the train is in the indication state and is travelling in an area of track at 190 km/h meanwhile the speed limit is 80 km/h. At appropriate time-units, which are usually very close to each other, the RBC controls the train, checking the position, the speed and the acceleration. In this case of too high-speed, the OBU has to generate a brake command to reduce the speed, thus the accident would not happened because the driver cannot accelerate the train at 190 km/h. The test objective is formulated as the following, in which `{OBU}0` and `{RBC}0` are used to identify respectively the first instance of the OBU and RBC processes in the IF description:

[2] The complete IF description of the case study is available at https://github.com/openETCS/validation/tree/master/VnVUserStories/VnVUserStoryMinesTelecom/05-Work/IF%20models

[3] El Pais Journal, Saturday 27th of July 2013.

TABLE I. SCALABILITY OF TESTGEN-IFX

Depth	#Global Test Cases	Time(s)	Coverage of Variables		
			v	l	x
1	301	0.47		✓	
2	90,601	172.12		✓	✓
3	90,601	183.74		✓	✓
4	162,058	262.83		✓	✓
5	1,273,563	3803.81		✓	✓
6	65,913,084	90698.15	✓	✓	✓

tp_1 := "$process : instance = \{\texttt{OBU}\}0$" $\wedge$"$variable : \texttt{m.speed} = 80$" $\wedge$"$state : source = \texttt{INDICATION}$" $\wedge$"$variable : \texttt{v} = 190$"

A global test case tc_1 being delivered by the TestGen-IFx tool is as the following:

```
?;  e;       ELocation{100};    {OBU}0
!;  {OBU}0;  MARequest{100};    {RBC}0
?;  e;       DLocation{103};    {RBC}0
delay 2
?;  {RBC}0;  MA{{30,80}};       {OBU}0
!;  {OBU}0;  DMIcmd{80};        e
?;  e;       ESpeed{190};       {OBU}0
!;  {OBU}0;  EBcmd{1};          e
```

The global test case is projected to obtain local test cases:

```
-- Test case for OBU
!;  e;       ELocation{100};    {OBU}0
!;  {OBU}0;  MARequest{100};    {RBC}0
delay 2
?;  {RBC}0;  MA{{30,80}};       {OBU}0
?;  {OBU}0;  DMIcmd{80};        e
!;  e;       ESpeed{190};       {OBU}0
?;  {OBU}0;  EBcmd{1};          e
-- Test case for RBC
!;  e;       DLocation{103};    {RBC}0
delay 2
?;  {OBU}0;  MARequest{100};    {RBC}0
!;  {RBC}0;  MA{{3,80}};        {OBU}0
```

Table I shows some metrics of test generations using Depth-First Search (DFS) exhaustive strategy to evaluate the scalability of TestGen-IFx. The experiments have been performed on a laptop with 2.2GHz Intel Core i7 processor and 16GB of RAM. The first column presents the depths explored. The second one relates to numbers of generated global test cases. The next one relates to the processing times in seconds. The three last columns represent the coverages of the variables in the models: train speed (v), train location (l), and danger point location (x). The rows corresponds to the results of different exploration depths. When the depth is 1, only one transition is fired, thus its events are executed: `?ELocation(`l`)`, `!MARequest(`l`)`. Since we consider data domain of variable l from 0 to 300, there are 301 possibilities of executions of the transition corresponding to 301 global test cases. These test cases cover (✓) the data domain of the variable location l using in the IF model of OBU. The number of test cases grows very quickly when the depth is 6. We obtain at this level a set of test cases that cover all possible values of the variables used in the models. Intuitively, we cannot execute all of these test cases against the SUT. We need to select the best test cases to be firstly executed.

Test Prioritization. After generating Java simulators representing the IF model of the ETCS system using IF2Java, we generated mutants by injecting faults that violate the safety properties of the ETCS. We use the Major [14] framework to generate the mutants. It allows us to create Java source code of mutants by injecting faults into some specific methods of a class. We inserted the 6 following types of faults:

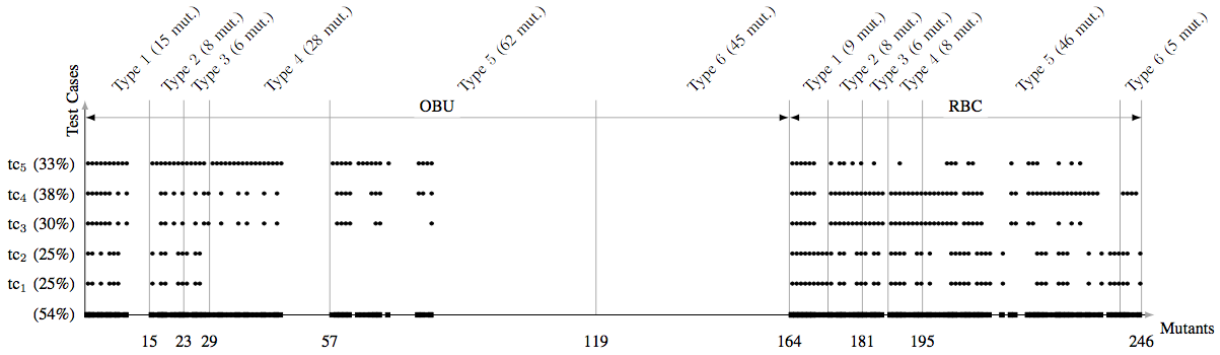


Fig. 2. Mutation Score and Mutant Coverage by Generated Test Cases

1) incorrectly implemented destinations of transitions,
2) incorrectly got inputs,
3) incorrectly sent outputs,
4) incorrectly evaluated guards of transitions,
5) incorrectly updated internal variables,
6) incorrectly updated clocks.

We obtain totally 246 mutants and it takes about 17 seconds. The fault *Type 5* generates the highest number of mutants (108: 62 for OBU and 46 for RBC). We then evaluate the generated mutants on 5 test cases, $tc_1, \ldots, tc_5$, which are generated from 5 scenarios to test the monitoring process of release speed[4]:

1) the current speed is 190 while the limit speed is 80,
2) the limit speed is 80 and the current speed of the train is 81, this just overs the limit speed,
3) the current speed is equal to the limit speed, 80,
4) when the break is hold,
5) when the break is released,

Figure 2 presents a cloud of mutants being killed by the test cases. Ox axis represents the mutants which are grouped by fault types, from 1 to 6, and by components which are either OBU or RBC. Oy axis represents the mutants killed by the 5 test cases. Average time of an execution process (compiling, executing and parsing results) of a test case on the 246 mutants takes about 240 seconds. Each mutant being killed by a test case is represented by a dot on the figure. For example, test case tc_1 kills 6 of 15 mutants, which are numbered 1,2,4,6,7,8 of fault *Type 1* of OBU. It kills in total 62 of the 246 mutants, thus its mutation score is 25%. Test cases tc_1 and tc_2 kill the same set of mutants, thus they have the same score. Test case tc_4 has the highest mutant score, 38%, then tc_5 with 33%.

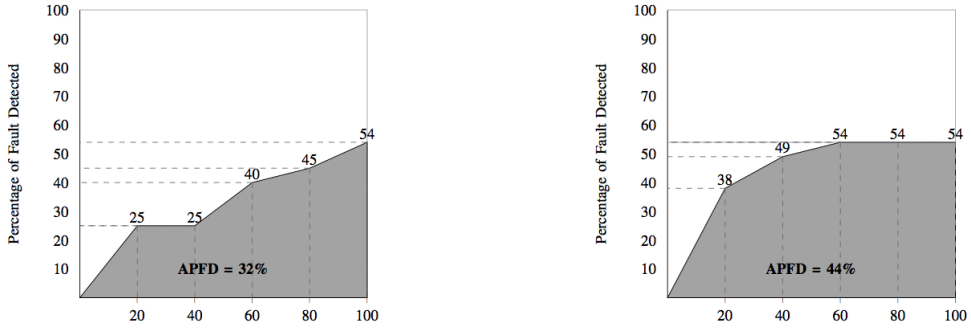
After having the mutation scores, we prioritize the generated test cases with the goal of fast fault detection. Figure 3 represents the evolution of the Average Percentage of Faults Detected (APFD) [15] over the life of the execution of 2 test suites consisting of the 5 test cases on two prioritization ways. The values of APFD range from 0 to 100. A higher APFD number means faster (better) fault detection. Although the figure does not directly measure the fault detection ability of the two test suites, *e.g.*, they are always 54%, it allows us to compare the different prioritization orders to create faster detecting through the ordering of test cases. In particular, suppose we place the test cases in order tc_1, tc_2, tc_3, tc_4, tc_5 to form a prioritized test suite T_1. Figure 3(a) shows the percentage of detected faults versus the percentage of executed test cases of T_1. After executing tc_1, 62 of 246 faults are detected; thus 25% of the faults have been detected after executing 1 of the 5 test cases, hence 20%, in T_1. No more new faults are detected after running test case tc_2, thus 25% of the faults have been detected after 40% of T_1 has been used. The curve in the figure represents the cumulative percentage of faults detected. The gray area under the curve represents the APFD of the test suite T_1, with 32%.

Figure 3(b) represents what happens when the order of test cases is changed to tc_4, tc_5, tc_1, tc_3, tc_2. It detects faults more quickly with APFD raising to 44%. It achieves the fault detection ability of T_1, 54%, by using only 3 of 5 test cases, *i.e.*, the two other test cases detect the faults that were detected by the 3 previous test cases. Testers may not need to execute the two last test cases. Thus it can save 40% of test execution effort.

Discussion. Mutation scores of the generated test cases are low. Generally, this testing approach considers the SUT as a black-box. It does not know how the SUT works in back end. It focuses on the user perspective, *i.e.*, the inputs and outputs of the SUT. Consequently, it is not possible to guarantee that all mutations of the code in the SUT are covered as the test cases are generated without knowledge of the implementation of the SUT. This is known as a NP-hard problem [16]. Particularly, the test cases generated above tend to test a particular functionality of the ETCS system. Consequently, they are not able to detect violations of the other behaviors of the system.

The generation of mutants gives an overview of a distribution of fault types in a SUT as the number of generated mutants depends highly on the fault types being injected. Figure 4(a) presents a distribution of the 6 fault Types on the ETCS. The fault *Type 5* has the highest possibility of occurrence, with 44%. This is understandable because almost computation of the system is internal calculation. The fault *Type 6* has the second highest possibility of occurrence, that is 20%. It shows an important impact of correctly implementing the operations of clocks, such as update, reset, etc. in a timed composite system. Figure 4(b) presents the fault detection abilities of the 5 generated test cases on the 6 fault Types. It is clear that the fault Types 1, 2, 3, and 4 can be detected more easily than the others because they influence directly the outputs of the system.

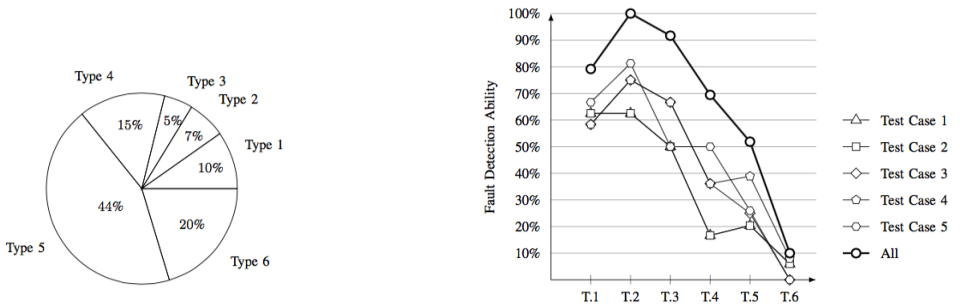
[4]The complete formal description of the test objectives and their data can be found at https://github.com/nhnghia/if2dot/tree/2java/example



(a) Test Case Order: tc_1, tc_2, tc_3, tc_4, tc_5

(b) Test Case Order: tc_4, tc_5, tc_1, tc_3, tc_2

Fig. 3. Average Percentage of Faults Detected for Non-Prioritized (a) and Prioritized Test Cases (b)



(a) (b)

Fig. 4. Fault Types Distribution of the ETCS (a) and Detection Ability of the Generated Test Cases (b)

IV. Conclusion & Future Work

We have presented a framework for prioritization distributed testing of timed composite systems. The model of a SUT is composed of models of components described as TIOTSs. The testing system is established by several testers. Each tester tests one component of the SUT and there is no need of communications among them. Test cases are generated from the model of the SUT to cover some behaviors to be tested. We have performed an experimental validation of the approach on the ETCS case study. We also illustrated an example to how prioritize the generated test cases for faster detecting faults in the ETCS. The initial result shows that we can save 40% of test execution effort of the generated test cases. An open-source tool chain[1] has been implemented to automate the generation and evaluation processes of the test cases.

In future work, we firstly intend to improve the expressiveness of the test objective description to be able to specify more complex test scenarios. We also plan to consider the framework in case of unobservable communications among components of the SUT and in the case of clocks drifting.

References

[1] L. Cacciari and O. Rafiq, "Controllability and Observability in Distributed Testing," *Information and Software technology*, vol. 41, pp. 767–780, 1999.

[2] R. Singh, "Test Case Generation for Object-Oriented Systems: A Review," in *Proc. of CSNT*, 2014, pp. 981–989.

[3] R. M. Hierons, M. G. Merayo, and M. Núñez, "Implementation Relations and Test Generation for Systems with Distributed Interfaces," *Distributed Computing*, vol. 25, no. 1, pp. 35–62, Nov. 2011.

[4] C. Gaston, R. M. Hierons, and P. L. Gall, "An Implementation Relation and Test Framework for Timed Distributed Systems," in *Proc. of ICTSS*, 2013, pp. 82–97.

[5] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," in *Proc. of ICSE*, 1995, pp. 41–50.

[6] N. Yevtushenko, A. Cavalli, and J. Lima, Luiz, "Test Suite Minimization for Testing in Context," in *Proc. of IWTCS*, 1998, pp. 127–145.

[7] N. Asoudeh and Y. Labiche, "Multi-Objective Construction of an Entire Adequate Test Suite for an EFSM," in *Proc. of ISSRE*, 2014, pp. 288–299.

[8] H. N. Nguyen, F. Zaidi, and A. Cavalli, "A Framework for Distributed Testing of Timed Composite Systems," in *Proc. of APSEC*, 2014, pp. 47–54.

[9] S. Ghosh, "Fault Injection Testing for Distributed Object Systems," in *Proc. of TOOLS*, 2001, pp. 276–285.

[10] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[11] R. Just, D. Jalali, L. Inozemtseva, M. Ernst, R. Holmes, and G. Fraser, "Are Mutants a Valid Substitute for Real Faults in Software Testing?" in *Proc. of FSE*, 2014, pp. 654–665.

[12] M. Bozga, S. Graf, I. Ober, and J. Sifakis, "The IF toolset," in *Formal Methods for the Design of Real-Time Systems*, 2004, pp. 237–267.

[13] UNISIG, "SUBSET-026 – System Requirements Specification," ERA, SRS 3.3.0, Mar. 2012.

[14] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using Non-redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis," in *Proc. of ISSRE*, 2012, pp. 11–20.

[15] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test Case Prioritization: An Empirical Study," in *Proc. of ICSM*, 1999, pp. 1–10.

[16] K. Chatterjee, Luca Alfaro, and R. Majumdar, "The Complexity of Covrage," in *Proc. of APLAS*, 2008, pp. 91 – 106.