Automatic Interleaving for Testing Distributed Systems

Mihal Brumbulli and Emmanuel Gaudin
PragmaDev
{mihal.brumbulli,emmanuel.gaudin}@pragmadev.com

***Abstract*—The constantly ever-growing interest for large-scale distributed systems like the Internet of Things imposes many challenges for developers and researchers from many areas. The development of distributed software applications is by no means trivial, and their inherent complexity becomes apparent during testing. Indeed, testing the operation of single isolated nodes does not suffice, because it may be affected by the distribution and inter-communication between nodes. Re-writing a test case to consider distribution is neither efficient nor simple, because concurrency is never easy to implement. In this paper we present an approach that automatically interleaves execution of test cases to simulate concurrency inherent in distribution. We focus on independent test cases that might exhibit a correlation due to distributed interaction. The approach is applied in the context of standard modeling and testing languages, and enables identification of interaction points during test case execution that depend on distribution. The re-execution of the test case is then interleaved at the identified points to account for distribution.**

***Index Terms*—Distributed systems, testing, modeling, simulation, TTCN-3, SDL**

I. Introduction

The development of distributed systems has gained much attention from industry and researchers with a variety of applications, a trend that is sure to continue in the immediate future due to the ever-growing interest for the Internet of Things [1]. However, the development of software applications for large-scale distributed systems is not a trivial task. This is especially true for testing, an activity which is quite challenging even for simple non-distributed systems. The operation of nodes in a distributed system is not isolated, and as such it requires for test cases to account for the distribution and interaction between nodes. The implication here is that existing test cases and their execution have to be adapted to consider distribution. This adaptation consists of (a) introducing concurrency handling into test cases, and (b) controlled concurrent execution that deals with all relevant interleavings. For the former the tests cases have to be modified, and considering that concurrency handling is never easy to implement, the effort that is required should not be overlooked. The later implies the existence of a scheduler that is able to handle all relevant interleavings.

In this paper we present an approach for automating the interleaved execution of test cases. We apply the approach in the context of standard modeling and testing languages. The system under test is described in SDL [2], TTCN-3 [3] is used for testing, and SDL-RT deployment diagrams [4] describe the distribution of system components. Test cases are executed against the system in a simulated environment extended with an interleaving algorithm.

In Section II we give an overview of related work and position our approach in respect to existing state of the art. We introduce the relevant technologies in Section III and our approach in Section IV. An example is given in Section V to illustrate the use of the presented solution. Finally, we conclude in Section VI with a discussion around the approach, its current status, and future work.

II. Related Work

Testing of distributed communication systems has been approached from different angles. We identify two major groups, i.e., distributed testing and interleaved execution.

Hartman et al. in [5] discuss the execution of abstract tests for distributed software. They underline the importance and benefits of interleaving test case sequences for discovering defects in the system. The approach can be applied to a single test case, it uses either concurrent or sequential synchronization of execution sequences, and there is no automation involved in establishing the synchronization points.

Schieferdecker and Vassiliou-Gioles in [6] discuss the distribution of TTCN-3 test setups. The focus is on the underlying concepts of the language and how they support management and execution of distributed test cases. This approach implies test execution on the target. This is obviously an advantage, however, it also needs a complex synchronization mechanism for controlling execution of each of the distributed test cases.

Bloom et al. in [7] emphasize the fact that, prior to testing in a target environment, the software is usually tested in the host environment. They propose a simulation-based approach and focus on the semantics of time. However, it is not clear how the execution of several test instances is interleaved to account for the inherent concurrency.

Testing of concurrent software is discussed in [8] and [9]. The authors focus on multi-threaded software and propose a solution that interleaves execution in a controlled way. The approach uses neither a standard language (like TTCN-3) nor any kind of abstract notations, instead, it is based on general purpose programming languages.

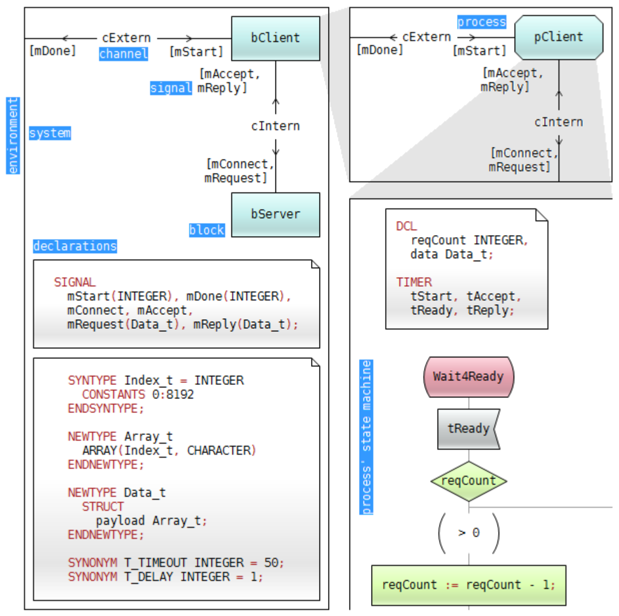
We adopt the idea of controlled interleaved execution in the context of distributed systems. To do so, we consider distributed communication between instances of the system to be similar to thread interleavings, and can potentially impact the behavior induced by the execution of test cases. Furthermore, the approach is based on standard and formal languages with precise semantics. This allows simulation in a controlled environment and, what is of great interest, automatic generation and execution of all relevant interleavings.

III. Technology

A. SDL

The Specification and Description Language (SDL) is a specification language defined by the International Telecommunication Union (ITU-T) in the Z.100 series [2]. SDL is targeted at the unambiguous specification and description of the behavior of reactive and distributed systems.

1) Structure, Behavior, and Communication: In SDL the overall design is called the *system*, and everything outside of it is defined as the *environment*. The system can be composed of *agents* and *communication constructs*. There are two kinds of agents: *blocks* and *processes*. Blocks can be composed of other agents and communication constructs. When the system is decomposed down to the simplest block, the way the block fulfills its functionality is described with processes. A process provides this functionality via extended finite state machines. It has an implicit queue for *signals*. A signal has a name and parameters; they go through *channels* that connect agents and end up in the processes implicit queues. Fig. 1 illustrates these concepts in a simple client-server application example.



```
SIGNAL
  mStart(INTEGER), mDone(INTEGER),
  mConnect, mAccept,
  mRequest(Data_t), mReply(Data_t);
```

```
SYNTYPE Index_t = INTEGER
  CONSTANTS 0:8192
ENDSYNTYPE;

NEWTYPE Array_t
  ARRAY(Index_t, CHARACTER)
ENDNEWTYPE;

NEWTYPE Data_t
  STRUCT
    payload Array_t;
ENDNEWTYPE;

SYNONYM T_TIMEOUT INTEGER = 50;
SYNONYM T_DELAY INTEGER = 1;
```

```
DCL
  reqCount INTEGER,
  data Data_t;

TIMER
  tStart, tAccept,
  tReady, tReply;
```

Fig. 1. Excerpt of the SDL model of a client-server application.

The client is started with *mStart* signal which takes as parameter the number of request the client should send to the server. The client will try to connect to the server, and in case of success it will start sending requests and waiting for replies. When done (or in case of error) it will report back (to the environment) the number of replies received from the server via the *mDone* signal.

2) Deployment: SDL descriptions are platform independent, i.e., they do not capture any information concerning the implementation details. For example, Fig. 1 speaks of a *bClient* and *bServer*, however it does not specify whether these agents are distributed or not. Deployment diagrams as defined in [4] can supplement the models with missing information about distribution. This approach has been used for simulating distributed applications as described in [10] and [11]. We adopt the idea and simplify it by focusing only on *components*, *nodes*, and *connections* as shown in Fig. 2.[1] The semantics are straightforward, i.e., if components (representing SDL agents) are attached to different nodes, then they are considered distributed, otherwise local execution is implied.

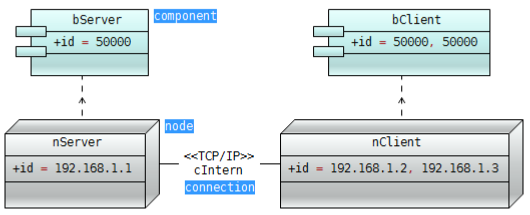


Fig. 2. Deployment model of the client-server application.

B. TTCN-3

The Testing and Test Control Notation Version 3 (TTCN-3) is a standardized testing technology developed and maintained by the European Telecommunication Standards Institute (ETSI). The ETSI TTCN-3 standards [3] have also been adopted by the ITU-T in the Z.160 series [12].

The abstract definition of test cases makes it possible to specify a non-proprietary test systems which are independent of both platform and operating system. The abstract definitions can be either compiled or interpreted for execution.

Fig. 3 shows a TTCN-3 module definition with a single test case that triggers the sending of 10 requests (line 15 and 19) from the client and expects 10 replies (line 16 and 21).

```
 1: module TestClientServer {
 2:   // Types for messages
 3:   type record mStart { integer reqCount };
 4:   type record mDone { integer repCount };
 5:   // Port type for the interface with the SUT
 6:   type port port_cExtern message {
 7:     out mStart;
 8:     in mDone;
 9:   };
10:   // Component type for the MTC and system inteface
11:   type component sClientServer {
12:     port port_cExtern cExtern;
13:   };
14:   // Templates for messages
15:   template mStart startMessage := { reqCount := 10 };
16:   template mDone doneMessage := { repCount := 10 };
17:   // Testcase
18:   testcase tc_start_done() runs on sClientServer {
19:     cExtern.send(startMessage);
20:     alt {
21:       [] cExtern.receive(doneMessage) {
22:         setverdict(pass);
23:       }
24:       [] cExtern.receive {
25:         setverdict(fail);
26:       }
27:     }
28:   }
29: }
```

Fig. 3. TTCN-3 module with a test case for the client-server application.

[1]The number of values in the *id* attribute is the number distributed nodes, e.g., the figure implies two clients and one server.

IV. Approach

A. Problem Statement

We are interested in the effects (if any) induced by distributed execution of test cases. To better understand the problem let us revisit the client-server example, and suppose that the result of the test case presented in Fig. 3 is *pass*, i.e., the system behaves as expected. What will happen if the number of clients is increased; will the system behave the same? There is one effective way to answer this question: test the system with multiple clients. This can be tackled by (a) rewriting the test case so that it accounts for multiple clients, or (b) execute multiple instances of the test case (one for each client) in parallel. The later needs an underlying synchronization mechanism that allows controlled execution of parallel (distributed) test cases, which is never trivial and requires additional expertise (not TTCN-3). That is why the former is more sound from a tester's perspective, because it is confined at the abstract level of testing provided by TTCN-3. However, rewriting the test case to take into account only the number of clients is not enough. Indeed, the difficult part is not in the number of clients but in their parallel (concurrent) execution due to distribution. To model concurrency all possible *interleavings* between test cases should be considered, which implies execution of all permutations of TTCN-3 instructions of different clients. For example, if *startMessage* is sent first to *client_1* and then to *client_2*, it is important to consider also the case where it is first sent to *client_2* and then to *client_1*.

B. Problem Analysis

Supposing concurrent execution of K test cases, with each test case consisting of n_i instructions for $i = 1, 2, \ldots, K$, the number of all interleavings is given by:

$$I = \frac{(\sum_{i=1}^{K} n_i)!}{\prod_{i=1}^{K}(n_i!)} \tag{1}$$

If we consider concurrent execution of K instances of the same test case, then $n_i = N \; \forall i$, where N is the number of instructions in the test case, and (1) can be re-written as:

$$I = \frac{(KN)!}{(N!)^K} \tag{2}$$

This is a typical case of the *state-explosion problem* which makes execution of all interleavings unpractical even in a controlled and automated environment. However, not all interleavings are relevant and their number (in most cases) can be drastically reduced. Indeed, the behavior induced by a test case can be affected by distribution only if there is an interaction between nodes. This means that, if the execution of a test case does not involve any distributed communication, then distribution will not have any impact. For example, if there isn't any communication between the client and the server, then interleaving is pointless because there is no distributed behavior in the first place. That is why it makes sense to interleave execution at critical points, i.e., instructions that trigger interaction between nodes by means of distributed communication.

C. Interleaving Algorithm

We start by grouping the instructions and then interleaving the execution of the groups. The condition is that each group must include at most one instruction which triggers distributed communication. Let m_i^j be an instruction in the test case, where $i = 1, 2, \ldots, N$ is the index (relative order) of the instruction, and $j = 0, 1$ describes whether the given instruction triggers any interaction (interleaving point). A group consists of all subsequent m_i^j for which $\sum j \leq 1$. The following shows an example sequence and corresponding grouping:

$$\underbrace{m_1^0, m_2^1, m_3^0}_{g_1}, \underbrace{m_4^1}_{g_2}, \underbrace{m_5^1, m_6^0}_{g_3}, \underbrace{m_7^1, m_8^0, m_9^0}_{g_4}, \underbrace{m_{10}^1}_{g_5} \tag{3}$$

For two test cases with the above sequence, the algorithm can generate > 700 times less interleavings.[2]

Interleaved execution can be automated using the following algorithm:

```
{$K$ is the number of instances}
{$N$ is the number of groups}
{$I$ is the next interleaving}
for $i := 0$ to $K - 1$ do
  for $j := 0$ to $N - 1$ do
    $I[i * N + j] := i$
  end for
end for
loop
  handleInterleave($I$)
  $i := I.length - 1$
  while $i > 0$ and $I[i-1] \geq I[i]$ do
    $i := i - 1$
  end while
  if $i = 0$ then
    return
  end if
  $j := I.length - 1$
  while $I[j] \leq I[i-1]$ do
    $j := j - 1$
  end while
  $temp := I[i-1]$
  $I[i-1] := I[j]$
  $I[j] := temp$
  $j := I.length - 1$
  while $i < j$ do
    $temp := I[i]$
    $I[i] := I[j]$
    $I[j] := temp$
    $i := i + 1$
    $j := j - 1$
  end while
end loop
```

[2]The result can be obtained by replacing in (2): $K = 2$ and $N = 10$ prior to grouping, and $K = 2$ and $N = 5$ after grouping.

Each group of instructions is represented by a simple integer for ease of calculation. The *handleInterleave* procedure maps the integer to its corresponding sequence of TTCN-3 instructions (permutation) that is to be executed next.

D. Tool Support

PragmaDev Studio[3] is a set of tools that helps specifiers, developers, and testers to manage complexity in the development of today's systems. The tools use the recognized international standards of SDL, TTCN-3, SDL-RT, and UML [13].

A key functionality of the tool-set is provided by the *PragmaDev (Co-) Simulator* as shown in Fig. 4.

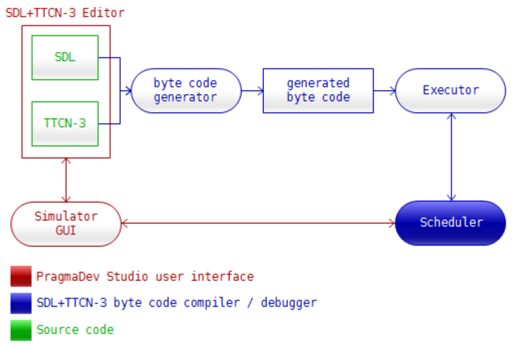


Fig. 4. Architecture of the PragmaDev (Co-) Simulator.

The co-simulator allows execution of TTCN-3 test cases against an SDL system. SDL and TTCN-3 descriptions are translated into an internal representation (byte code) to be interpreted by the *executor*, which in turn forwards the scheduling of events to the *scheduler*. We extend the functionality of the scheduler with the interleaving algorithm.

In the first phase the test case is executed against the system using the scheduler in normal mode (no interleaving involved). All TTCN-3 instructions (*send* statements) that trigger distributed communication between agents of the SDL system are marked during execution. Every communication between agents is checked against a deployment diagram, and if the sender and receiver of a message are attached to different nodes in the diagram, then the last TTCN-3 instruction that triggered such behavior is indeed the one to be marked.

In the second phase the test case is executed against the system in interleaving mode. The scheduler automatically creates K instances[4] of the test case and enters the interleaving algorithm. On each iteration the algorithm creates an interleaved sequence of TTCN-3 instructions based on marking done in the first phase and the number of instances. The sequence is then executed like a normal TTCN-3 test case by the scheduler, and at the end the SDL system is reset to its initial state for the next iteration.

The whole process is completely automated and transparent to the tester. There is no need to rewrite the tests, but just execute them with the interleaving scheduler.

[3]http://www.pragmadev.com

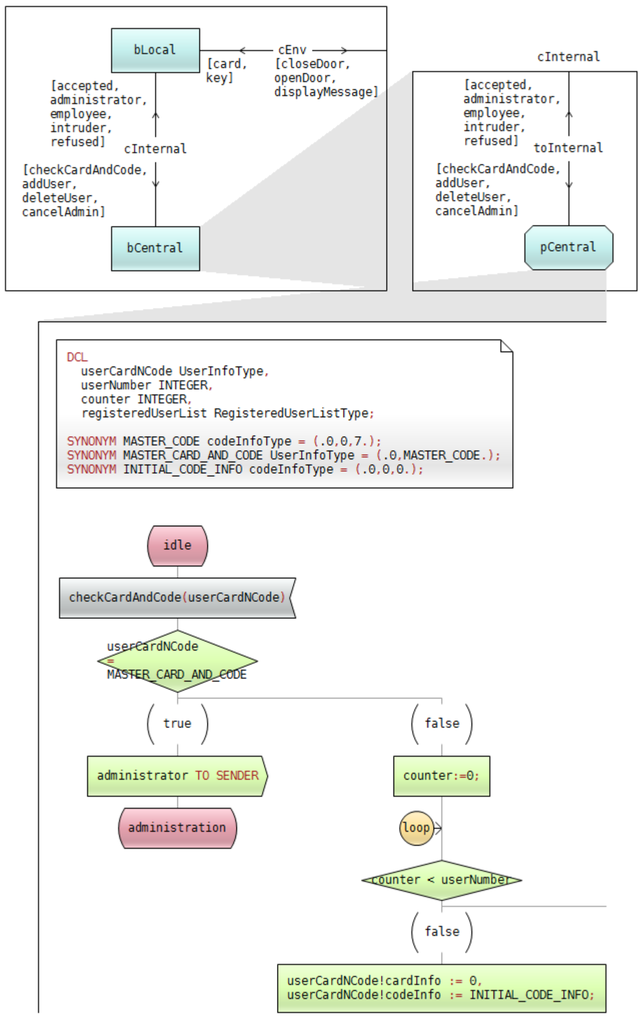
[4]The number of instances is deduced from the deployment diagram.

V. Example

We have used the presented approach in testing an access-control system. The system is composed of several terminals and a central unit. Each terminal has a slot for entering a card and a keypad for entering the key. This information is sent to the central unit which checks whether access should be granted to a user and notifies the terminal from where the request was issued. The user can be either an *administrator* or a *normal* one. The administrator can add or delete normal users and is identified by a special card key.

A. Structure and Behavior

Fig. 5 shows an excerpt of the SDL model of the access-control system. The system is composed of two blocks, namely *bLocal* for terminals and *bCentral* for the central unit. Each block has a single process within which implements the behavior of the system.


bLocal
cEnv
[card, key]
[closeDoor, openDoor, displayMessage]
[accepted, administrator, employee, intruder, refused]
cInternal
[checkCardAndCode, addUser, deleteUser, cancelAdmin]
bCentral
cInternal
[accepted, administrator, employee, intruder, refused]
toInternal
[checkCardAndCode, addUser, deleteUser, cancelAdmin]
pCentral
idle
checkCardAndCode(userCardNCode)
userCardNCode = MASTER_CARD_AND_CODE
true
false
administrator TO SENDER
administration
counter:=0;
loop
counter < userNumber
false
userCardNCode!cardInfo := 0,
userCardNCode!codeInfo := INITIAL_CODE_INFO;

```
DCL
  userCardNCode UserInfoType,
  userNumber INTEGER,
  counter INTEGER,
  registeredUserList RegisteredUserListType;

SYNONYM MASTER_CODE codeInfoType = (.0,0,7.);
SYNONYM MASTER_CARD_AND_CODE UserInfoType = (.0,MASTER_CODE.);
SYNONYM INITIAL_CODE_INFO codeInfoType = (.0,0,0.);
```

Fig. 5. Excerpt of the SDL model of the access-control system.

The user enters the *card* and the *key* which are sent to the central unit via the *checkCardAndCode*. If the card and key are those of the administrator, then the central unit enters in *administration* state, in which normal users can be added or deleted. On the other hand, if the credentials are not those of the administrator, the list of registered normal users is scanned for matching credentials. The *employee* signal is sent back in case of a match, otherwise an *intruder* is signaled.

B. Deployment

A simple deployment scenario for the access-control system is shown in Fig. 6. The figure speaks of two *bLocal* (terminals) connected to a single *bCentral* (central unit). We will use this scenario to automatically generate and execute all relevant interleavings.

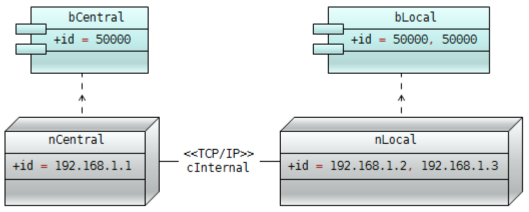


Fig. 6. Deployment model of the access-control system.

C. Test Case

To show the applicability of automatic interleaving we start with the most basic test case for the system, i.e., try to get in and out of (without doing anything else) administrator mode as shown in Fig. 7.

```
 1: module TestAccessControl {
 2:   type record card { integer param1 };
 3:   type record key { integer param1 };
 4:   type record displayMessage { charstring param1 };
 5:   type record openDoor { charstring param1 };
 6:   type record closeDoor { charstring param1 };
 7:   type port cEnv_type message {
 8:     out card;
 9:     out key;
10:     in displayMessage;
11:     in closeDoor;
12:     in openDoor;
13:   };
14:   type component AccessControl {
15:     port cEnv_type cEnv;
16:   };
17:
18:   template displayMessage EnterCardMessage := {param1 := "Enter card"};
19:   template displayMessage EnterCodeMessage := {param1 := "Enter code"};
20:   template displayMessage AddOrDeleteMessage := {param1 := "* add; # delete"};
21:   template displayMessage OneStar := {param1 := "*"};
22:   template displayMessage TwoStar := {param1 := "**"};
23:   template displayMessage CancelledMessage := {param1 := "Cancelled"};
24:   template card UserCard(integer userID) := { param1 := userID };
25:   template key EnterKey(integer keyValue) := { param1 := keyValue };
26:
27:   altstep failOnWrongReceive() runs on AccessControl {
28:     [] cEnv.receive { setverdict(fail); };
29:   }
30:
31:   testcase tc_correctAdministratorAccess() runs on AccessControl {
32:     activate(failOnWrongReceive());
33:     // Enter card
34:     cEnv.receive(EnterCardMessage);
35:     cEnv.send(EnterCard(0));
36:     // Enter administrator code
37:     cEnv.receive(EnterCodeMessage);
38:     cEnv.send(EnterKey(0));
39:     cEnv.receive(OneStar);
40:     cEnv.send(EnterKey(0));
41:     cEnv.receive(TwoStar);
42:     cEnv.send(EnterKey(7));
43:     // Administrator mode
44:     cEnv.receive(AddOrDeleteMessage);
45:     // Get out of administrator mode
46:     cEnv.send(EnterKey(0));
47:     cEnv.receive(CancelledMessage);
48:     // Done
49:     setverdict(pass);
50:   }
51: }
```

Fig. 7. Test case for administrator access on the access-control system.

The user enters the *card* = 0 (administrator) and then the *code* = 007. At this point the information (card + code) is sent to the central unit. Because these are the right credentials, the user is allowed to enter in administrator mode, where he/she can add or delete users. However, none of this actions is taken, and the user just exits from the administrator mode. The request to exit is also processed by the central unit. Executing this scenario against the system using the scheduler in normal mode (no interleaving, one terminal and one central unit) will indeed result in a *pass* for the test case.

When execution in normal mode is finished all the required information will be available for entering the interleaving mode. Based on the description above there should have been identified two interleaving points by now: (1) after entering the last digit of the code and (2) after the request to leave the administrator mode. Indeed, these are the points during execution where a communication between the terminal *bLocal* and the central unit *bCentral* is triggered. This translates into two groups of TTCN-3 statements whose execution shall be interleaved: the first group consists of lines 34-44 and the second of lines 46-47 in Fig. 7. Based on the number of terminals *bLocal* in Fig. 6 and that of groups ($K = 2$, $N = 2$), the total number of interleavings to be executed is 6. It is important to note that, if the grouping algorithm is not applied, the number of interleavings will be 252.[5]

What we found during execution in interleaving mode was in fact quite surprising. We didn't expect to find any problems from such a simple test case, considering that it induces a very basic behavior to the system. The system was modeled so that only one terminal at a time can get administrator access, meaning that an attempt from a second terminal will fail. Indeed, this is the behavior we observed during the interleaved execution, however, what we did not expect is for the second terminal to block indefinitely waiting for a reply from the central unit (i.e., test case execution did block on line 44 in Fig. 6). We were able to immediately jump to the point in the model (using the PragmaDev Studio user interface) causing the problem. The central unit, after granting access to the first terminal, entered the *administration* state so that it could process any requests coming from the terminal for adding or deleting users. However, in this state it was discarding (not handling) all log-in requests, which in turn caused the second terminal to wait indefinitely for a reply. The solution was straightforward: add a new transition in the SDL state machine of the central unit to handle such requests.

We were able to identify 4 similar problems in our model of the access-control system by using the interleaving scheduler on a set of more complex test cases. The set was composed of (a) previously hand written test cases depicting typical usage scenarios and (b) a set of automatically generated test cases using the model-based approach described in [14]. It is important to note that we were able to identify these problems without writing a single line of TTCN-3 code.

[5]In this case the number of groups is equal to that of the TTCN-3 *send* statements ($N = 5$).

VI. Conclusion

Testing of software applications for large-scale distributed systems is not a trivial task, because the test cases and their execution need to be adapted in order to account for the distribution and interaction between nodes.

In this paper we presented an approach for automating the interleaved execution of test cases to mitigate the complexity of concurrent behavior due to distribution. We applied the approach in the context of standard modeling and testing languages where the system under test was described in SDL, the distribution of system components in deployment diagrams, and the test cases in TTCN-3. Test cases were executed against the system in a simulated and controlled environment with an interleaving scheduler.

The presented algorithm allowed us to significantly reduce the number of interleavings without deviating from our target, that is the thorough testing of the system. Instead of using each single instruction of the test case as interleaving point, we focused only on those relevant, i.e., instructions that induce distributed behavior to the system.

Automatic interleaving was used for testing the distributed behavior of an access-control system. Even with a basic test case we were able to identify a problem in the system, a trend that continued also with more complex test cases. Furthermore, we achieved these results by reducing the number of interleavings by a factor of 42 (in the simplest case).

There are however three issues that need further discussion. First, the proposed algorithm may not always produce significantly less interleavings. This depends on the degree of distribution in the behavior, i.e., the number of interleavings grows if more inter-communication between distributed nodes takes place. This can degrade simulation performance especially with the increasing complexity of test cases. However, we believe this type of scenario to be more of an exception than the rule. Indeed, energy consumption is one of the major challenges of distributed systems composed of potentially millions of battery powered devices (e.g., Internet of Things), and reducing inter-communication between devices to an acceptable minimum is always an engineering goal. Second, although we reported a successful application of the approach with a simple example, we strongly believe that its benefits are emphasized when used for testing complex systems, which is still work in progress as the time of writing this paper. Last, the proposed approach is based on simulation and at present cannot be applied for test cases executing on target. The reason is simple: simulation allows fine-grained control over the execution, does not require any complex synchronization mechanism for interleaving, and can exploit the benefits of the proposed algorithm. The last point is very important because, without any means of reducing the number of interleavings, testing every possible combination is unpractical due to the state-explosion problem. In the end, in addition to a mechanism for controlling the interleaving, execution on target would require means to track distributed communication triggered during testing.

References

[1] Gartner Inc., "Gartner says the Internet of Things installed base will grow to 26 billion units by 2020," http://www.gartner.com/newsroom/id/2636073, 2013.

[2] ITU-T, "Specification and Description Language – Overview of SDL-2010," International Telecommunication Union – Telecommunication Standardization Sector, ITU-T Recommendation Z.100, 2011, http://handle.itu.int/11.1002/1000/11387.

[3] ETSI, "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language," European Telecommunications Standards Institute, ETSI Standard ES 201 873-1, 2014, http://www.ttcn-3.org/index.php/downloads/standards.

[4] SDL-RT Consortium, "Specification and Description Language – Real Time," SDL-RT Consortium, SDL-RT Standard V2.3, 2013, http://www.sdl-rt.org/standard/V2.3/html/index.htm.

[5] A. Hartman, A. Kirshin, and K. Nagin, "A Test Execution Environment Running Abstract Tests for Distributed Software," in *Proceedings of Software Engineering and Applications*, ser. SEA '02. Acta Press, 2002.

[6] I. Schieferdecker and T. Vassiliou-Gioles, "Realizing Distributed TTCN-3 Test Systems with TCI," in *Testing of Communicating Systems*, ser. Lecture Notes in Computer Science, D. Hogrefe and A. Wiles, Eds. Springer Berlin Heidelberg, 2003, vol. 2644, pp. 95–109.

[7] S. Blom, T. Deiß, N. Ioustinova, A. Kontio, J. van de Pol, A. Rennoch, and N. Sidorova, "TTCN-3 for Distributed Testing Embedded Software," in *Perspectives of Systems Informatics*, ser. Lecture Notes in Computer Science, I. Virbitskaite and A. Voronkov, Eds. Springer Berlin Heidelberg, 2007, vol. 4378, pp. 98–111.

[8] M. Musuvathi and S. Qadeer, "CHESS: Systematic Stress Testing of Concurrent Software," in *Logic-Based Program Synthesis and Transformation*, ser. Lecture Notes in Computer Science, G. Puebla, Ed. Springer Berlin Heidelberg, 2007, vol. 4407, pp. 15–16.

[9] ——, "Iterative Context Bounding for Systematic Testing of Multithreaded Programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 446–455.

[10] M. Brumbulli and J. Fischer, "Simulation Configuration Modeling of Distributed Communication Systems," in *System Analysis and Modeling: Theory and Practice*, ser. Lecture Notes in Computer Science, Ø. Haugen, R. Reed, and R. Gotzhein, Eds. Springer Berlin Heidelberg, 2013, vol. 7744, pp. 198–211.

[11] M. Brumbulli, "Model-Driven Development and Simulation of Distributed Communication Systems," Ph.D. dissertation, Humboldt Universität zu Berlin, 2015.

[12] ITU-T, "Testing and Test Control Notation version 3: TTCN-3 core language," International Telecommunication Union – Telecommunication Standardization Sector, ITU-T Recommendation Z.160, 2014, http://handle.itu.int/11.1002/1000/12346.

[13] OMG, "OMG Unified Modeling Language (OMG UML). Version 2.5," Object Management Group, OMG Standard, 2015.

[14] J. Deltour, A. Faivre, E. Gaudin, and A. Lapitre, "Model-Based Testing: An Approach with SDL/RTDS and DIVERSITY," in *System Analysis and Modeling: Models and Reusability*, ser. Lecture Notes in Computer Science, D. Amyot, P. Fonseca i Casas, and G. Mussbacher, Eds. Springer International Publishing, 2014, vol. 8769, pp. 198–206.