A model-based approach for interoperability test generation

Nanxing Chen

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1982 – août 2011

A Model-based Approach for Interoperability Test Generation

Nanxing CHEN*
nanxing.chen@irisa.fr

Abstract: The purpose of interoperability testing is to ensure that interconnected protocol implementations communicate correctly while providing the expected services. However, interoperability test generation is known to be complex. On one hand, most current interoperability test cases are still generated in a manual way, which is time-consuming and error-prone. On the other hand, existing automatic test case generation methods often encounter the state explosion problem due to the multi-component nature of interoperability testing. In this report, we propose an approach for automatic interoperability test case generation. This method, based on formal interoperability definitions, intends to obtain a composite product of the specifications of each protocol implementation with respect to a preselected property to be verified. To relieve the state explosion problem, only the necessary global behavior of specifications are calculated. The obtained composite product intends to keep only the events relevant to the property to be verified. We show that an interoperability test case derived by using this approach is unbiased and nonpermissive.

Key-words: interoperability testing, time stamps, passive testing

---

Une Approche de la Génération de Tests d'interopérabilité basée sur les Modèles Formels

Résumé : *L'objectif du test d'interopérabilité est de s'assurer à la fois que les implémentations interagissent correctement et qu'elles rendent les services prévus dans leur spécification pendant leur interaction. Cependant, la génération de test d'interopérabilité est problématique. D'une part, la plupart des cas de test d'interopérabilité sont toujours décrits manuellement à l'aide de spécifications complexes et non nécessairement formelles, il y a donc un risque d'erreurs important lors de l'écriture. D'autre part, les méthodes de génération automatiques de tests d'interopérabilité existantes rencontrent souvent le problème d'explosion combinatoire du nombre d'états : La manipulation des entités lors de la génération de test peut conduire à un nombre d'états trop important. Ce rapport propose une approche pour la génération automatique de tests d'interopérabilité. Cette méthode, basée sur les définitions formelles d'interopérabilité, prend en entrée un objectif de test et deux spécifications. Le cas de test correspondant est généré avec un algorithme de recherche en profondeur, considérant l'interaction asynchrone partielle entre les spécifications par rapport à l'objectif de test. Nous montrons qu'un cas de test obtenue en utilisant cette approche n'est ni biaisé ni permissif.*

Mots clés : *le test d'interoperabilité, model IOLTS, cas de test*

Travail pour la nouvelle collection de publications internes de l'Irisa version électronique compatible avec les RR Inria
* Projet Dionysos

1 Introduction

Today's networks are becoming heterogeneous and complex. As they are composed of equipments based on new technologies from a variety of vendors, it happens that errors are introduced by differences in protocol implementations. Correct coordination and communication of different protocol implementations is essential to guarantee customer expectations.

To increase the confidence that network products conform to international standards, conformance testing methodologies have been standardized and deployed. Conformance testing [1] verifies whether a network component conforms to its specification. A specification is usually a standard defined by standards organizations such as ISO (International Standards Organization)[1], IETF (Internet Engineering Task Force)[2], ITU (International Telecommunication Union)[3], etc. Although ISO-9646 standard [1] mentions that conformance testing increases the probability of interoperability, it cannot guarantee interoperability. Conforming products may not interoperate due to several reasons [2, 3]: ambiguity in protocol standards, poorly specified protocol options, incompleteness of conformance testing, etc. Therefore, interoperability testing is required to ensure the correct cooperation of network products from different product lines and provide expected services.

Due to the multi-component nature, interoperability test generation is complex. Currently, most interoperability testing cases are generated manually. Interoperability testing events such as Plugtests[4] , Testfests etc., are held regularly, where vendors may test interoperability with fellow industry participants in a pair wise way. However, manual interoperability testing is complicated: Test suites derivation is performed on the basis of complex specifications which are not necessarily formal. Consequently, test cases may exhibit errors. Besides, manual test cases should be derived for each test purpose: A test suite need therefore much time to be developed. Moreover, analyzing massive traces by visual inspection is error-prone and requires an expert.

Unlike conformance testing which uses a piece of test equipment to generate all the test scenarios and check for received message syntax, timing, etc, automation and repeatability is difficult to achieve in interoperability testing. Over the last few years, efforts have been made on automatic test generation for the sake of accuracy and economical reasons. One straight mean is to select test cases from the composed finite state machine (FSM), which is constructed from IUT specifications by reachability analysis [7, 3, 12, 9]. However, this approach may produce large or even infinite numbers of test cases. Indeed, the state number of the specification asynchronous interaction is in the order of $O((n.m^f)^2)$ where n is the state number of the specifications, f the size of the input FIFO queue on lower interfaces and m the number of messages in the alphabet of possible inputs on lower interfaces. This result can be infinite if the size of the input FIFO queues is unbounded. Other works are generally based on fault models or on the search of some particular kinds of error [11, 8]. In [13], a method was proposed for automatic interoperability test generation by making use of a single entity (through the system interface) that interoperates with the rest of the integrated communication systems. But these works are not based on a rigorous definition of interoperability.

This report, overcoming the limitations of the work mentioned above, presents a formal specification-based approach to the automatic interoperability test cases generation. The method involves in carrying out a partial asynchronous interaction calculation of the specifications of each implementation under test with respect to a preselected test objective. To relieve state exploration, several techniques are applied: The partial asynchronous calculation is carried out in a depth-first manner to minimize the memory requirements. Each test objective is assigned with attributes to indicate the important states to be explored as well as to inhibit unnecessary states exploration. We also apply rules to avoid generating redundant transitions. The obtained graph intends to keep only the events which are relevant to the test objective. We further apply verdict assignment and minimization rules to build an executable minimized interoperability test case. The derived test case must be unbiased, i.e., applied on implementations to be tested, the test case declares *Fail* only if the interoperability of implementations is not satisfied. We also require that implementations that do not interoperate must be detected by repeating the application of a test case, under a fairness assumption on the implementations.

This report is organized as follows: in Section 2, we introduce interoperability testing activities and formal interoperability definitions. Section 3 describes the interoperability test generation approach, including the test case derivation and minimization. Section 4 shows the results. We conclude the report and suggest further research directions in Section 5.

2 Preliminaries

2.1 Interoperability Testing Architecture

The purpose of interoperability testing (*iop* for short in the sequel) is to verify n ($n \geqslant 2$) products from different product lines interoperate correctly and provide expected services. General interoperability testing architecture involves a *Test System*

[1] http://www.iso.org/iso
[2] http://www.ietf.org/
[3] http://www.itu.int/en/pages/default.aspx
[4] http://www.etsi.org/Website/OurServices/Plugtests/home.aspx

(TS) and a *System Under Test (SUT)* composed of n ($n \geqslant 2$) interconnected *Implementations Under Test (IUT)* from different product lines. According to the quantity of the IUTs, interoperability testing can be in either of the following context: *(i) Multi-Component* context where SUT has n ($n \geqslant 2$) IUTs. *(ii) One-to-One* context where SUT is composed of two IUTs. In practice, *One-to-One* iop is the most common context, either by testing the interoperability of IUTs in a pairwise way, or by testing the interoperability between one protocol implementation and a system already in operation (which may be composed of n IUTs).

In interoperability testing, testing entities communicate with each other through a variety of interfaces:

- *Upper Implementation Access Point* ($UIAP_i$, $i = \{1, 2, ...n\}$) is the interface through which IUT_i communicates with its upper layer. UI_i is observable and controllable. Test system connected to $UIAP_i$ via *Upper Point of Control and Observation* ($UPCO_i$) interface can send test messages to the corresponding IUI_i and observe its output responses.
- *Lower Implementation Access Point* ($LIAP_i$) is the interface through which IUT_i communicates with its peer IUT. Contrary to $UIAP_i$, $LIAP_i$ is only observable. Test system connected to $LIAP_i$ via *Lower Point of Observation* (LPO_i) interface can only observe the interaction of IUT_i with its peer IUT but must not send any message to it.

Like most of the functional tests in the field of networks, interoperability testing is a kind of *"black-box"* test [15]. Test system has no knowledge of the IUT inner structure. Only the external behavior of IUTs can be verified from the interfaces during test execution.

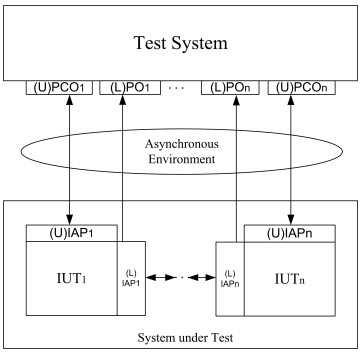

Figure 1: Interoperability testing architecture

The accessibility to interfaces IAP could be different from one testing architecture to another. In practice, the most common architecture is the *global total architecture* [10]. i.e., both upper and lower IAP of all IUTs are available. But in reality, it is possible that TS has only access to the upper (or lower) IAP of all IUTs or only some of them. According to the different IAP accessibility, the ability of the test system to control/observe the IUTs varies. In this report, we consider the *global total one-to-one architecture* without loss of generality.

2.2 Formal Model

For interoperability testing, a distinction must be made between events of the SUT that are *controllable* by the test system, from those that are only *observable*. Therefore, in this report we use the IOLTS (Input-Output Labeled Transition System) model[6] , which allows differentiating input, output and internal events while precisely indicating the interfaces specified for each event.

Definition 1 An IOLTS is a tuple $M = (Q^M, \Sigma^M, T^M, q_0^M)$ where Q^M is the set of states of the system M with q_0^M its initial state. Σ^M is the set of observable events at the interfaces of M. In IOLTS model, input and output actions are differentiated: We note $p?a$ (resp. $p!a$) for an input (resp. output) of message a at interface p. $T^M \subseteq Q^M \times (\Sigma^M \cup \tau) \times Q^M$ is the transition relation, where $\tau \notin \Sigma^M$ stands for an internal action. Transition $(q, \alpha, q') \in T^M$ can be equally noted $q \xrightarrow{\alpha}_M q'$.

Other notations Let us consider an IOLTS M, and let an observable event $\alpha \in \Sigma^M$ with $\alpha = p \cdot \{?, !\} \cdot m$, an executable event of the system $\mu_i \in \Sigma^M \cup \tau$, a succession of events $\sigma \in (\Sigma^M)^*$, and q, $q' \in Q^M$. We define the following notations:

- Σ^M can be partitioned: $\Sigma^M = \Sigma_U^M \cup \Sigma_L^M$, where Σ_U^M (resp. Σ_L^M) is the set of messages exchanged on the upper (resp. lower) interfaces. Σ^M can also be partitioned to distinguish input (Σ^{M^I}) and output messages (Σ^{M^O}). In addition, in interoperability testing, controllable events and observable events should be differentiated. Indeed, $\Sigma_U^{M^I}$ is the set of the input events on the upper interfaces, i.e., they represent the message coming from the upper layer. In practice, they are often considered as controllable stimuli sent by the test system. All other types of events are observable events.

- $Out(q) =_{def} \{\alpha \in \Sigma^{M^O} | \exists q', q \xrightarrow{\alpha}_M q'\}$ is the set of possible outputs at state q. Similarly, $In(q) =_{def} \{\alpha \in \Sigma^{M^I} | \exists q', q \xrightarrow{\alpha}_M q'\}$ is the set of possible inputs and $\Gamma(q) =_{def} \{\alpha \in \Sigma^M | \exists q', q \xrightarrow{\alpha}_M q'\}$ is the set of all possible events at the state q.
- q after $\sigma =_{def} \{q' \in Q^M | q \stackrel{\sigma}{\Rightarrow}_M q'\}$ is the set of states which can be reached from q by the sequence of actions σ.

Suspension IOLTS In interoperability testing, the test system does not only observe traces produced by a system, but also quiescence by timers. Therefore, we consider quiescence an observable output. Three situations may lead to quiescence of a system and are illustrated in the following figure. A *deadlock* state (cf. Fig.2-a) is a state where the system cannot evolve anymore. i.e. $\Gamma(q) = \emptyset$. An *outputlock* state (cf. Fig.2-b) is a state where the system is waiting only for input(s), i.e. $\Gamma(q) \subseteq \Sigma^{M^I}$. A *livelock* state (cf. Fig.2-c) is a state from which the system diverges by an infinite sequence of internal actions. i.e. $q \in livelock(M) =_{def} \exists \tau_1, ..., \tau_n, q \xrightarrow{\tau_1 ... \tau_n}_M q$.

We denote the set of deadlocked states, outputlock states and livelock states of the *IOLTS M* by *deadlock(M), outputlock(M),* and *livelock(M)* respectively. A deadlock is in fact a special case of outputlock, i.e., $deadlock(M) \subseteq outputlock(M)$. Therefore, the set of all quiescent states is denoted by: $quiescent(M) = outputlock(M) \cup livelock(M)$.

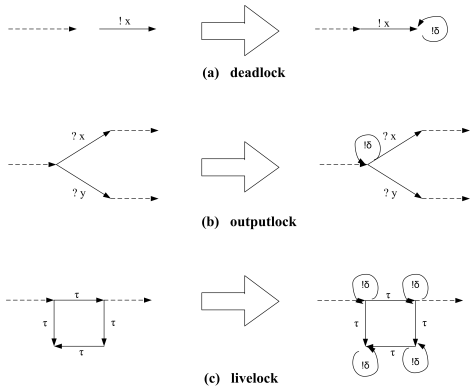

Figure 2: Quiescent states

An IOLTS in which quiescence is modeled is called a suspension IOLTS and is defined bellow:

Definition 2 The suspension automaton of an IOLTS $M = (Q^M, \Sigma^M, T^M, q_0^M)$ is an IOLTS $\triangle(M) = (Q^{\Delta(M)}, \Sigma^{\Delta(M)}, T^{\Delta(M)}, q_0^{\Delta(M)})$ where $Q^{\Delta(M)} = Q^M$; $q_0^{\Delta(M)} = q_0^M$; $\Sigma^{\Delta(M)} = \Sigma^M \cup \{\delta\}$ with $\delta \in \Sigma^{\Delta(M)^O}$. i.e., δ is used to represent quiescence, which is regarded as an observable output of the system. The transition relation $T^{\Delta(M)}$ is obtained from T^M by adding loops for each quiescent state q. i.e., $T^{\Delta(M)} = T^M \cup \{(q, \delta, q) \mid q \in quiescent(M)\}$

For a specification S, the traces of $\triangle(S)$ are called the suspension traces of S and are denoted by *STraces(S). STraces (S)* represents all the behavior of S that can be observed, including quiescence. Similarly, the visible behavior of an IUT is also characterized by its suspension traces *STraces(IUT).*

Asynchronous Interaction In interoperability testing, the interaction between two IUTs is asynchronous [10, 6] as messages may be actually exchanged by traversing several protocol layers. According to [6], asynchronous interaction is usually modeled as two unidirectional FIFO queues. And, as usually done in the field of testing, we consider that the testing environment is error-free. Formally, asynchronous interaction is noted $S_1 \|_A S_2$ for two communicating IOLTS S_1 and S_2. A state in $S_1 \|_A S_2$ is a four-tuple (q_1, q_2, f_1, f_2) where $q_1 \in Q^{S_1}, q_2 \in Q^{S_2}$, f_1 (resp. f_2) stands for the messages in the input channel of S_1(resp.S_2). Informally, a state (q_1, q_2, f_1, f_2) means that the execution of S_1 and S_2 have reached state q_1 and q_2 respectively, while the input channels of S_1 and S_2 store the list of messages f_1 and f_2 respectively.

Iop Criteria In this report, we deal with the *global total iop testing architecture*, i.e, both lower and upper IAP of two IUTs are accessible. The corresponding *global total iop criterion* says that: Two IUTs are considered interoperable iff after a suspension trace of the specifications interaction, all outputs, including quiescence observed during the interaction of the implementations must be foreseen in the interaction of their specifications.
$I_1 iop_G I_2 = \forall \sigma \in STraces\ (S_1 \|_A S_2),\ Out(I_1 \|_A I_2, \sigma) \subseteq Out(\triangle(S_1 \|_A S_2), \sigma)$

3 Interoperability Test Generation

3.1 Outlines

In this section, we investigate an approach to global total iop test generation. The objective of the method is to generate executable Interoperability Test Cases (ITC) to verify the interoperability of two IUTs. The inputs of the algorithm are the suspension specifications on which the IUTs are based, and an Interoperability Test Purpose (ITP) which represents a particular objective to be verified. The execution of an ITC on SUT returns a verdict : $verdict(ITC, SUT) \in \{Pass, Fail, Inconclusive\}$. Verdict *Pass* means the test purpose is reached without any error detected, *Fail* means at least one error is detected, and *Inconclusive* means the behavior of IUTs is correct w.r.t the interaction of their specifications, however does not correspond to the test purpose.

The method begins with the construction of a composite product of the suspension specifications $\triangle(S_1)$, $\triangle(S_2)$ and ITP, which corresponds to partially calculates the specifications $\triangle(S_1 \|_A S_2)$ w.r.t the ITP. The product intends to calculate the different ways of specifications' interaction to reach the ITP. To obtain the product, as well as to minimize the memory requirement, a depth-first algorithm is applied. To reduce state exploration, the trap states of each ITP are assigned with attributes *Accept* (mandatory) and *Refuse* (optional), where *Accept* (resp. *Refuse*) represents the desired behavior (resp. undesired behavior) w.r.t ITP. The attributes also serve to inhibit the generation of redundant transitions. i.e., asynchronous interaction calculation is cut as soon as an attribute is reached. During the calculation, it may happen that some states contain conflict (controllable events and an observable events), we apply conflict management rules to avoid generating unnecessary conflict transitions. Finaly, we further apply verdicts assignment and test case minimization rules to obtain a minimized executable iop test case.

3.2 Formalizing Interoperability Test Purpose and Test Case

3.2.1 InteroperabilityTest Purpose

The goal of testing is to find errors in SUT. However proving correctness is elusive as it is generally impossible to validate all possible behavior described in specifications [5]. From the methodological point of view, most tests are carried on by setting test purposes. A test purpose is in general informal, in the form of an incomplete sequence of actions representing a critical property to be verified. It can be designed by experts or provided by standards guidelines for test selection [4].

In this report, we argue on the use of interoperability test purpose to select test cases. Informal interoperability test purposes (ITP) are formalized by IOLTS model. To select the target behavior, some states in the ITP are associated with attributes *Accept* to define *Pass* verdict, while *Refuse* to represent the possible options in specifications but not the objectives of the test.

Definition 3 An Interoperability Test Purpose is a deterministic acyclic controllable IOLTS: ITP = (Q^{ITP}, Σ^{ITP}, T^{ITP}, q_0^{ITP}) where:

- $\Sigma^{ITP} \subseteq \Sigma^{S_1} \cup \Sigma^{S_2}$. The set of behavior described by the ITP must be included in the set of behavior of the specifications. i.e, an ITP must be consistent w.r.t the specifications.
- Q^{ITP} is the set of states. An ITP has two sets of trap states $Accept^{ITP}$ and $Refuse^{ITP}$. $Accept^{ITP}$ stand for *the desired behavior* to be observed and are associated with attribute *Accept*. $Refuse^{ITP}$ stand for possible options in specifications rather than the desired behavior to be verified, and are associated with attribute *Refuse*. States in $Accept^{ITP}$ and $Refuse^{ITP}$ are only directly reachable by the observation of outputs produced by the IUTs. This is because in black box testing, the verification of outputs is more reliable. In fact, the test system can only know if a message has been sent to an IUT, but not whether this IUT has actually processed the message. Moreover, $Accept^{ITP} \neq \emptyset \wedge Accept^{ITP} \cap Refuse^{ITP} = \emptyset$.
- ITP satisfies the *controlability condition*: for each state in ITP, no choice is allowed between several stimuli sent by TS (controllable events) or between a stimulus and an output of IUT (observable events).
- ITP is complete, which means that each state allows all actions. This is done by inserting "$*$" lable at each state of the ITP, where "$*$ " is an abbrevation for the complement set of all other events leaving q. Indeed, "$*$ " allows describing test purposes at an abstract level without considering detailed interaction.

3.3 Interoperability Test Purpose - Specifications Product

The interoperability test generation method begins with the construction of a composite product of $\triangle(S_1)$, $\triangle(S_2)$ and ITP, which corresponds to the asynchronous interaction calculation $\triangle(S_1 \|_A S_2)$ w.r.t the ITP. The composite product intends to calculate the different ways to observe/execute the ITP on the SUT.

Definition 4 The product of $\triangle(S_1)$, $\triangle(S_2)$ and ITP is a controllable suspension IOLTS $\triangle(STP)$ =($Q^{\triangle(STP)}$, $\Sigma^{\triangle(STP)}$, $T^{\triangle(STP)}$, $q_0^{\triangle(STP)}$) where:

- $\Sigma^{\triangle(STP)} \subseteq \Sigma^{\triangle(S_1)} \cup \Sigma^{\triangle(S_2)}$ is the alphabet.
- $Q^{\triangle(STP)}$is the set of states. Each state $(q^{\triangle(S_1)}, q^{\triangle(S_2)}, q^{ITP}, f_1, f_2) \in Q^{\triangle(STP)}$ is such that $q^{\triangle(S_1)} \in Q^{\triangle(S_1)}$, $q^{\triangle(S_2)} \in Q^{\triangle(S_2)}$, $q^{ITP} \in Q^{ITP}$, f_1 (resp. f_2) represents the input queue of $\triangle(S_1)$ (resp. $\triangle(S_2)$). $q_0^{STP} = (q_0^{\triangle(S_1)}, q_0^{\triangle(S_2)}, q_0^{ITP}, \varepsilon, \varepsilon)$ is the initial state of STP, where $q_0^{\triangle(S_1)}$, $q_0^{\triangle(S_2)}$, q_0^{ITP} are the initial states in $\triangle(S_1)$, $\triangle(S_2)$ and ITP respectively. ε denotes an empty input queue.
- $\left\{Accept^{\triangle(STP)}, Refuse^{\triangle(STP)}\right\} \subseteq Q^{\triangle(STP)}$ where $Accept^{\triangle(STP)} = Q^{\triangle(STP)} \cap (Q^{\triangle(S_1)} \times Q^{\triangle(S_2)} \times Accept^{ITP})$ and $Refuse^{\triangle(STP)} = Q^{\triangle(STP)} \cap (Q^{\triangle(S_1)} \times Q^{\triangle(S_2)} \times Refuse^{ITP})$
- $\triangle(STP)$ is controllable: No choice is allowed between several stimuli sent by TS or between a stimulus and an output of IUT. The controllable property implies that in a state with a conflict, some transitions must be pruned: Either one stimulus is kept and all other stimuli and IUT outputs are pruned, or all IUT outputs are kept but stimuli are pruned.
- $T^{\triangle(STP)}$ is the set of transitions.
- $Q^{\triangle(STP)}$ and $T^{\triangle(STP)}$ are obtained in the following way: Let $q^{\triangle(STP)} = (q^{\triangle(S_1)}, q^{\triangle(S_2)}, q^{ITP}, f_1, f_2)$ be a state in $\triangle(STP)$, and let a be a possible action at state $q^{\triangle(S_1)}$or $q^{\triangle(S_2)}$ or q^{ITP}. Notice that a can be an output, an input, or a quiescence. A new transition $(q^{\triangle(STP)}, a, p^{\triangle(STP)}) \in T^{\triangle(STP)}$ is created according to the following conditions: *(i)* If ITP is in neither *Accept* nor *Refuse* state, check the *asynchronous interaction operation rules* below. If one rule is executable, a new transition in $\triangle(STP)$ labeled by a is created according to the rule. *(ii)* If ITP is in *Accept* or *Refuse* state, no new transition in created. i.e., *Accept* and *Refuse* inhibit unnecessary global behavior calculation.

Asynchronous interaction operation rules

Rule 1 If a is an upper interface action or a quiescence of $\triangle(S_i)$ *(i={1,2})*, a new transition in $\triangle(STP)$ labeled by a is created. These events do not influence the communication channels of the IUTs.

(1.1) If $a \in \Sigma_U^{\triangle(S_1)}$, $(q^{\triangle(S_1)}, a, p^{\triangle(S_1)}) \in T^{\triangle(S_1)}$, $(q^{ITP}, a, p^{ITP}) \in \triangle^{ITP} \Longrightarrow (q^{\triangle(STP)}, a, p^{\triangle(STP)}) \in T^{\triangle(STP)}$ where $p^{\triangle(STP)} = (p^{\triangle(S_1)}, q^{\triangle(S_2)}, p^{ITP}, f_1, f_2)$.

(1.2) If $a \in \Sigma_U^{\triangle(S_2)}$, $(q^{\triangle(S_2)}, a, p^{\triangle(S_2)}) \in T^{\triangle(S_2)}$,$(q^{ITP}, a, p^{ITP}) \in \triangle^{ITP} \Longrightarrow (q^{\triangle(STP)}, a, p^{\triangle(STP)}) \in T^{\triangle(STP)}$ where $p^{\triangle(STP)} = (q^{\triangle(S_1)}, p^{\triangle(S_2)}, p^{ITP}, f_1, f_2)$.

Rule 2 If a is a lower interface output of $\triangle(S_i)$ (except quiescence), a new transition in $\triangle(STP)$ labeled by a is created. a is put into the tail of the input queue of $\triangle(S_j)$ (i,j={1,2}, i≠j) :

(2.1) If $a \in \Sigma_L^{\triangle(S_1)^O} \backslash \delta$, $(q^{\triangle(S_1)}, a, p^{\triangle(S_1)}) \in \triangle^{(S_1)}$, $(q^{ITP}, a, p^{ITP}) \in \triangle^{ITP} \Longrightarrow (q^{\triangle(STP)}, a, p^{\triangle(STP)}) \in T^{\triangle(STP)}$ where $p^{\triangle(STP)} = (p^{\triangle(S_1)}, q^{\triangle(S_2)}, p^{ITP}, f_1, f_2')$ with $f_2' = f_2.a$.

(2.2) If $a \in \Sigma_L^{\triangle(S_2)^O} \backslash \delta$, $(q^{\triangle(S_2)}, a, p^{\triangle(S_2)}) \in T^{\triangle(S_2)}$, $(q^{ITP}, a, p^{ITP}) \in \triangle^{ITP} \Longrightarrow (q^{\triangle(STP)}, a, p^{\triangle(STP)}) \in T^{\triangle(STP)}$ where $p^{\triangle(STP)} = (q^{\triangle(S_1)}, p^{\triangle(S_2)}, p^{ITP}, f_1', f_2)$ with $f_1' = f_1.a$.

Rule 3 If a is a lower interface input, and a is the first element in the input queue of $\triangle(S_i)$, a new transition in $\triangle(STP)$ labeled by a is created.

(3.1) If $a \in \Sigma_L^{\triangle(S_1)^I}$, $a \in head(f_1)$, $(q^{\triangle(S_1)}, a, p^{\triangle(S_1)}) \in T^{\triangle(S_1)}$, $(q^{ITP}, a, p^{ITP}) \in \triangle^{ITP} \Longrightarrow (q^{\triangle(STP)}, a, p^{\triangle(STP)}) \in T^{\triangle(STP)}$ where $p^{\triangle(STP)} = (p^{\triangle(S_1)}, q^{\triangle(S_2)}, p^{ITP}, f_1', f_2)$ with $f_1 = a.f_1'$.

(3.2) If $a \in \Sigma_L^{\triangle(S_2)^I}$, $a \in head(f_2)$, $(q^{\triangle(S_2)}, a, p^{\triangle(S_2)}) \in T^{\triangle(S_2)}$, $(q^{ITP}, a, p^{ITP}) \in \triangle^{ITP} \Longrightarrow (q^{\triangle(STP)}, a, p^{\triangle(STP)}) \in T^{\triangle(STP)}$ where $p^{\triangle(STP)} = (q^{\triangle(S_1)}, p^{\triangle(S_2)}, p^{ITP}, f_1, f_2')$ with $f_2 = a.f_2'$.

- Rule 1 deals with upper interface events or quiescence: These events do not depend on the state of queues between the two specifications, but only the current state of the $\triangle(S_i)$ concerned by the event. i.e., the event is executable during the interaction if it is executable for the concerned $\triangle(S_i)$ at its state $q^{\triangle(S_i)}$. In addition, the execution of these events does not affect the status of queues. Notice that in this rule, $q^{\triangle(S_i)}$ can be equal to $p^{\triangle(S_i)}$, in the case of a is a quiescence. Moreover, q^{ITP} can be equal to p^{ITP} for the following reason: We consider that the ITP is complete, i.e., each state in ITP allows all actions. The completeness

of ITP is realized by the "*" loops. "*" stands for all actions at a state except the events which label the outgoing transitions at this state.

- Rule 2 deals with lower interface output events. An output is executable by the system during asynchronous interacting if $\triangle(S_i)$ is in a state where the output is specified. The message a is then added to the end of the input queue of $\triangle(S_j)$. Notice that normally the FIFO queue is bounded. Therefore, an output is executable if the queue is not full.
- Rule 3 deals with lower interface input events. For that an input at a lower an interface be executable, the receiver $\triangle(S_i)$ should be at a state where the reception is expected and the corresponding message must have been sent by $\triangle(S_j)$. i.e. The message a to be received must be at the head of the FIFO queue of its input lower interface. $\triangle(S_i)$ then removes the message from the queue.

3.3.1 Depth-first Suspension STP Construction

The rules defined above allow building a composite product of $\triangle(S_1)$, $\triangle(S_2)$ and ITP recursively from the initial state $q_0^{\triangle(STP)}$. The attributes *Accept* or *Refuse* allow computing only the necessary part of the global behavior of the specications w.r.t the ITP. Indeed, *Accept* means the interaction of the specications has reached the ITP. While *Refuse* means the interaction of specications lead to an option in the specifications and will not reach the *Accept* state of the ITP. Therefore it is no need to continue further global behavior calculation.

The algorithm of $\triangle(STP)$ construction is depth-first traversal. From the initial state $q_0^{\triangle(STP)}$, at each step we check if a transition could be created according to the *asynchronous interaction operation rules*. Three cases are possible:

1. At state q, only one rule can be applied: The corresponding rule is applied to generate a new transition in $\triangle(STP)$.
2. At state q, several rules can be applied. i.e., at least two transitions can be generated. q is then called *branching state* and stored in a stack. Then we choose one of the applicable rules to carry out $\triangle(STP)$ construction until no more rules can be applied. i.e., *Accept* or $Refuse$ attribute is reached.

   - If $Accept^{\triangle(STP)}$ is reached, the algorithm calls a function *Accept_path_calculation* to calculate the corresponding path that traverses $q_0^{\triangle(STP)}$ and the $Accept^{\triangle(STP)}$. Then the algorithm backtracks to state q and choose another possible rule to continue STP construction.
   - If $Refuse^{\triangle(STP)}$ is reached, the algorithm backtracks to state q and choose another possible rule to continue $\triangle(STP)$ construction.
   - If all the possible rules at state q have been applied, q is removed from the stack.

   In order to satisfy the controllable property of $\triangle(STP)$. i.e., each state of $\triangle(STP)$ has no choice between several stimuli or a stimulus and other actions, the following rules are defined:

   - At branching state q, if no rule has been applied yet, we apply one of the applicable rules to generate a new transition in $\triangle(STP)$.
   - At branching state q, if (at least) one transition has already been generated, and there exists another possible rule that has not been applied yet. We check if this rule conflicts with the existing labels $\Gamma(q)$ at state q. Two cases are possible:
     - The rule does not conflict any of the existing labels at state q. Then we apply the rule and continue $\triangle(STP)$ construction until no more rules can be applied.
     - The rule conflicts (at least one) existing actions at state q. Then we check whether there is a label (event) $l \in \Gamma(q)$ that has reached $Accept^{STP}$. If not, we apply the rule and continue $\triangle(STP)$ construction until no more rules can be applied. Otherwise we will not apply the rule.

   By checking conflict conditions, some conflict transitions can be avoided being developed. However, not all the conflict transitions can be avoided in this way. To prune all the conflict transitions, a backtracking procedure will be performed when $\triangle(STP)$ is finished being constructed.
3. At state q, no rule can be applied. Then we check if the stack that stores branching states is empty. If it is not empty, we jump to state stored in the top position of the stack and continue $\triangle(STP)$ construction. On the contrary, if the stack is empty. i.e., no more state can be explored. The algorithm stops.

The $\triangle(STP)$ construction algorithm and *Accept Path Calculation* algorithm can be found in Appendix A.

3.3.2 Application to a simple connection protocol

Consider the example of a simple connection protocol illustrated in Fig.3. Suspension specifications S_1 and S_2 represent respectively a client and a server. $U1?CNR$ is a connection request from the upper layer, while $U1?Close$ is a disconnection request. $l1!cnr$ (resp. $l2?cnr$) is the request sent to (resp. received by) the peer IUT. $l2!ack$ (resp. $l2!nack$) is the positive (resp. negative) response of the server, and $U1!ACK$ (resp. $U1!NACK$) is the response forwarded to the upper layer following the reception of $l2!ack$ (resp. $l2!nack$). Moreover, quiescence states are explicitly modeled: In S_1, two states contain outputlocks $\{0, 2\}$; while in S_2, state 0 contains an outputlock.

The ITP in this example aims at testing a connection request CNR sent to IUT_1 leads to a positive response $l2!ack$, while $l2!nack$ is defined as $Refuse$ behavior. i.e., all events subsequent to $l2!nack$ are not the target of test case and should not be taken into account.

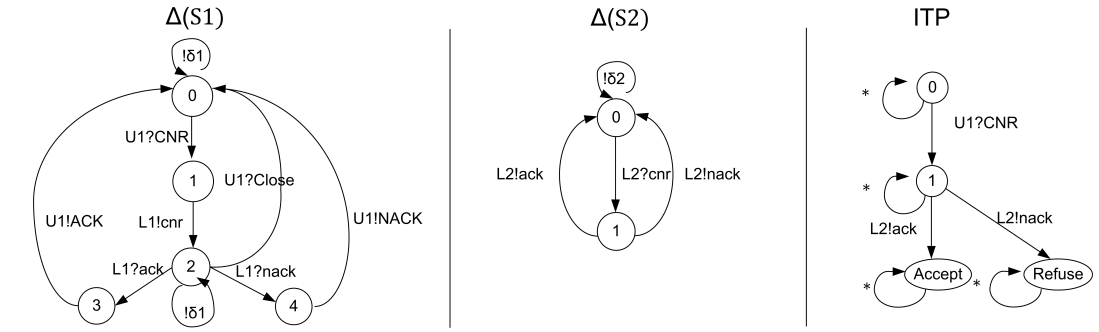

Figure 3: Connection protocol specifications and iop test purpose

In practice, a maximal depth can be used during $\triangle(STP)$ construction, which can be quite useful when there are cycles in specification. In this example we apply the depth-first $\triangle(STP)$ construction algorithm with a maximal depth configured to 5. The obtained graph $\triangle(STP)$ is illustrated in Fig.4.

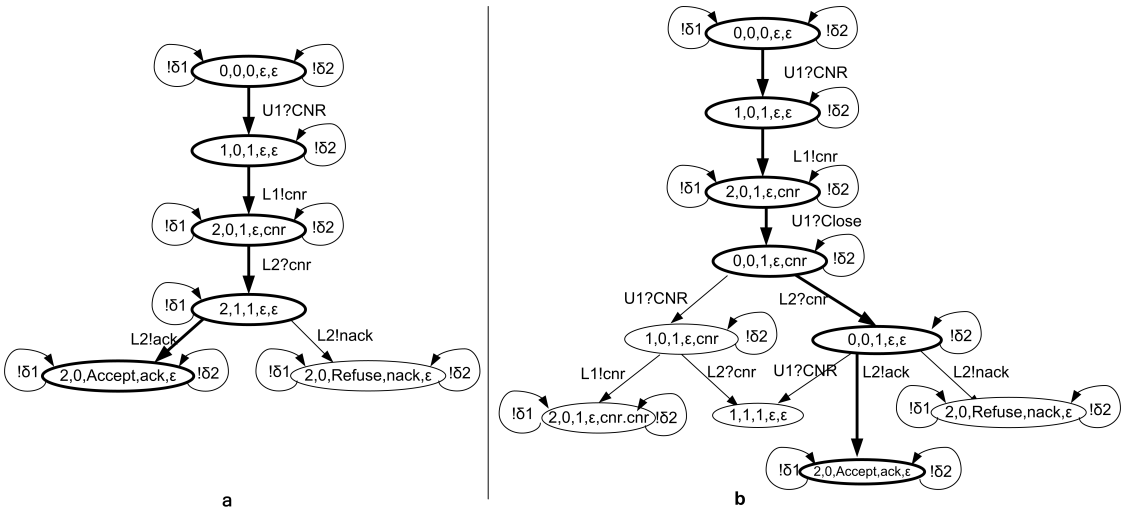

Figure 4: Composite product STP

There are two possible $\triangle(STP)$ generation orders: Fig.4-a corresponds to the case where in branching state $(2, 0, 1, \varepsilon, cnr)$, $l2?cnr$ is generated first other than $U1?Close$. As the transitions following $l2?cnr$ have reached $Accept^{\triangle(STP)}$, there is no need to further generate its conflict action $U1?Close$. On the contrary, Fig.4-b illustrates the case where $U1?Close$ is explored before $l2?cnr$. In this case, more states are explored to reach $Accept^{\triangle(STP)}$. In both figures, the bold parts are the $Accept\_path\_transitions$ obtained by $Accept\_path\_calculation$. Indeed, $Accept\_path\_transitions$ represents the detailed interaction of the specifications from their initial states to reach the objective defined by the ITP ($Accept$ attribute). We can see that in Fig.4-b, state $(0, 0, 1, \varepsilon, cnr)$ contains conflict actions $U1!CNR$ and $L2?cnr$. Conflict transitions will be pruned later in a backtracking way.

Besides, all the quiescences are preserved during asynchronous specifications interaction, indicating that in these composite states, specified quiescences are authorized.

3.4 Test Case Minimization

The derivation of an executable iop test case involves in performing the following instructions to extract a subgraph from $\triangle(STP)$.

Observable behavior extraction First, iop test case minimization involves in extracting the observable behavior from $\triangle(STP)$ while masking the non-observable actions. As in iop testing, each implementation is seen as a black box, an input of IUT is in fact an internal event that cannot be directly observed. Therefore, we replace these actions by τ. Then, observable behavior of $\triangle(STP)$ is extracted by applying τ - reduction [14], which hides internal actions τ from the composite product followed by determinization.

Conflict pruning Possible conflicts in $\triangle(STP)$ should be pruned, which can be done in a backtracking manner. The transitions that can not lead to *Accept* attributes are pruned.

Quiescence reduction Not all the quiescences are kept. Quiescence is only kept at states containing outputs but not stimuli (stimuli are sent by the tester). Moreover, trap states do not need to indicate quiescence, because the trap states in an ITC are the verdicts.

3.5 Verdicts Assignment

To obtain an executable interoperability test case, verdicts should be assigned. A verdict is the result of the execution of a test case. It is determined by the comparison between the actual behavior of the SUT during test case execution and its expected behavior. In general, there exist the following types of verdicts.

1. *Pass:* The observed behavior of SUT is correct and the iop test purpose is reached.
2. *Inconclusive:* The observed behavior is correct, however does not correspond to the test purpose
3. *Fail:* At least an error is detected.

To associate the graph with verdicts, the algorithm takes *Accept_path_transitions* and $\triangle(STP)$ as inputs. For each state $v \in$ *Accept_path_transitions*, the following conditions are checked:

1. For $v \in$ *Accept*$^{\triangle(STP)}$, v is associated with verdict *Pass*: *Pass* $=\{v|\ u \in$*Accept_path_transitions,* $v \in$ *Accept*$^{\triangle(STP)}$, $a \in \Sigma^{\triangle(STP)}$, $u \xrightarrow{a}_{\triangle(STP)} v\}$.
2. For $v \in$ *Accept*$^{\triangle(STP)}$ that allows an output reaching a state in $\triangle(STP)$ but not contained in *Accept_path_transitions* itself, a new transition with verdict *Inconclusive* and the allowed output is added. *Inconclusive* $=\{u|\exists v \in$ *Accept_path_transitions* $,u \in Q^{\triangle(STP)} \cap u \notin$ *Accept_path_transitions*, $a \in \Gamma(v), v \xrightarrow{a}_{\triangle(STP)} u\}$.
3. If a state v which could have an unexpected output in either S_1 or S_2. A new transition with event "*otherwise*" and verdict *Fail* is added. $(v, otherwise, Fail)$ where *Fail* $=\{u|v \in$ *Accept_path_transitions*,*otherwise* $\notin \Sigma^{\triangle(STP)^O}$,$\neg(v \xrightarrow{otherwise}_{\triangle(STP)} u)\}$. The label *Otherwise* represents unexpected outputs as well as the timeout non specified at these states. i.e., specified timeout is tolerated.

3.6 Iop Test Case

By applying the $\triangle(STP)$ construction, iop test case derivation and minimization, as well as verdict assignment algorithms, an interoperability test case is obtained, which stands for the detailed set of instructions that need to be taken in order to perform the test w.r.t the ITP.

Definition 5 Interoperability test case is a suspension IOLTS $\triangle(ITC)$=($Q^{\triangle(ITC)}$, $\Sigma^{\triangle(ITC)}$, $T^{\triangle(ITC)}$, $q_0^{\triangle(ITC)}$) where

- $Q^{\triangle(ITC)}$= *Accept_path_states*$\cup$*Pass*$\cup$*Inconclusive*$\cup$*Fail* . {*Pass, Fail, Inconclusive*}are the trap states of $\triangle(ITC)$, representing the possible verdicts of a test execution.
- $\Sigma^{\triangle(ITC)} \subseteq \{(\Sigma_U^{\triangle(S_1)} \cup \Sigma_U^{\triangle(S_2)}) \cup (\Sigma_L^{\triangle(S_1)^O} \cup \Sigma_L^{\triangle(S_2)^O}) \cup$ *otherwise*$\}$ is the set of alphabets.
- $T^{\triangle(ITC)}$= *Accept_path_transitions* $\cup\ T_{Inconclusive} \cup T_{Fail}$.

An iop test case should meet the following properties:

1. From any state in interoperability test cases, a verdict must be accessible: $\forall q \in Q^{\triangle(ITC)}$, $\exists \sigma$ such that for $p \in \{Pass, Fail, Inconclusive\}$,$q \xrightarrow{\sigma}_{\triangle(ITC)} p$.

2. A test case is *non-biased* w.r.t the specifications and global iop criterion: $\forall I_1$, I_2 two IUTs, if $\sigma \in STraces\ (I_1 \| _A I_2)$ is produced by SUT during the execution of $\triangle(ITC)$, verdict Fail is raised iff $\neg(I_1 iop_G I_2)$.

Example 3 Applying the verdict assignment rules and test case minimization on Fig.4-b, an iop test case is generated: Conflict transitions that can not lead to the *Accept* attribute are pruned, while verdicts are associated.

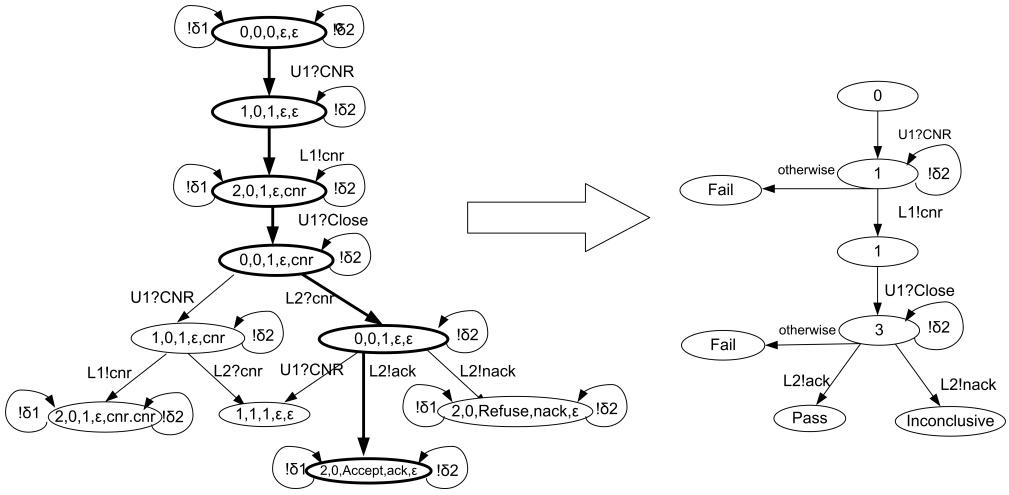

Figure 5: A simplified connection protocol ITC

4 Results

Given an interoperability test purpose ITP and two suspension specifications $\triangle(S_1)$, $\triangle(S_2)$. The proposed approach calculates the composite product $\triangle(STP)$=$\triangle(S_1)$×$\triangle(S_2)$×ITP. Meanwhile, the observable expected behavior to be verified is extracted. At last, an iop test case is derived and minimized while *Pass*, *Inconclusive* and *Fail* verdicts are assigned.

We have made experiments by implementing all the algorithms, including STP construction rules, accept paths extraction, test case minimization and verdicts assignments. Applying these algorithms on Example 1, we obtain an $\triangle(STP)$ of 12 states and 13 transitions. Whereas by reachable analysis, 66 states and 79 transitions are generated. In fact, an ITP that constraints the behaviors by the use of *Accept* and *Refuse* allows quickly synthesize an iop test case without calculating all the global behavior.

Besides, by calculating the joint behavior of the specifications, design errors such as unspecified inputs, deadlock states and overflow can be detected. Also, the consistency of ITP can be verified.

Using the approach proposed, the generated iop test case is *non-biased*. i.e., $\forall I_1$, I_2 two IUTs, if $\sigma \in Traces\ (I_1 \| _A I_2)$ is produced by SUT during the execution of iop test case, verdict *Fail* is raised if and only if $\neg(I_1 iop_G I_2)$. The *non-biased property* can be divided into the following two propositions:

Proposition 1 $\forall I_1$, I_2 two IUTs, if the application of a generated interoperability test case produces a *Fail* verdict, then $\neg(I_1 iop_G I_2)$.

Let *iop test case* an interoperability test case generated from specifications S_1, S_2, and a valid ITP. State *Fail* in *iop test case* specifies that *Fail* verdict is only produced by an observable action which is not allowed in the specifications. Let σ a trace produced by the SUT during the execution of *iop test case, verdict* (σ, *SUT)=verdict (iop test case after* σ*)*. As *iop test case* is deterministic, thus *iop test case after* σ is unique. Moreover, according to the formal interoperability definition (*cf. Definition 2*), if $I_1 iop_G I_2 = \forall \sigma \in STraces\ (S_1 \| _A S_2)$, $Out(I_1 \| _A I_2, \sigma) \subseteq Out(\triangle(S_1 \| _A S_2), \sigma)$. Therefore, *verdict (iop test case after* σ*)*$\neq$*Fail. verdict* (σ, *SUT)*$\neq$*Fail.*

Proposition 2 $\forall I_1$, I_2 two IUTs, if $\neg(I_1 iop_G I_2)$, then the application of *iop test case* produces a *Fail* verdict.

On the contrary of *iop test case*, which is deterministic, implementations can be non deterministic. Thus the application of the same *iop test case* may produce different verdicts. Thus, we assume a bounded fairness hypothesis on implementations, which implies that a bounded number of executions of a non deterministic implementation will show all its behavior.

Therefore, for a trace σ which contradicts the interaction of S_1 and S_2, *verdict (iop test case after σ)=Fail* will be raised by repeating *iop test case*.

However, the proposed method has some limitations: In the worst case, the method does not reduce the global state space. Moreover, the approach needs that *Accept* attribute must not be assigned to two conflict actions, otherwise one of them will not be developed (conflict transitions are cut during STP generation). i.e., the approach is only suitable for single-stimuli iop testing.

5 Conclusion and Future Work

This report presents an approach of automatic interoperability test generation. Based on formal interoperability specifications, the approach associates the asynchronous calculation of specifications with respect to an interoperability test purpose. For each iop test purpose, an iop test case is developed. We have formally defined the rules allowing building a composite product $\triangle(STP)$ of specifications and the ITP, from which an iop test case can be derived. To relieve state exploration, we have applied a suite of strategies: *(i)* Each test objective is assigned with *Accept and Refuse attributes* to limit state exploration. *(ii)* Depth-first $\triangle(STP)$ construction keeps only the events relevant to the test objective *(iii)* States with conflicts are applied rules to avoid generating certain conflict transitions. Finally, we have defined verdicts assignment and test case minimization rules to construct an executable iop test case. Besides, the constructed iop test case respects the unbiased property. i.e. verdict $Fail$ is raised iff $\neg(I_1 iop_G I_2)$.

However, the approach is suitable for single stimuli iop test case generation. Moreover, in the worst case, the state-space explosion problem still exists. Future work will be focus on researching better methods to solve state-space explosion problem, as well as on extending this solution to multi-component interoperability testing.

Appendix A

A.1 Depth-first $\triangle(STP)$ construction:

- *Current* : The current state under construction. Initially, Current: $= q_0^{\triangle(STP)}$.
- *Rules(q)*: Function *Rules(q)* returns two values *q.succ* and *q.nb_succ*. *q.succ* is the set of the successor states of state q that can be generated according to the *asynchronous interaction operation rules;* While *q.nb_succ* is the number of the successor states of q.
- *q.visited* : The set of the transitions at state q that have already been generated.
- *Accept_path_states / Accept_path_transitions*: The set of states / transitions in the path traversing $q_0^{\triangle(STP)}$ and $Accept^{\triangle(STP)}$.
- *Create_transition (STP, (q, p))*: Generate a transition from state q to state p in STP.
- *End_Exploration*: Boolean equal to *TRUE* iff no more state can be explored.
- Conflict: Boolean equal to *TRUE* iff the currently examined rule at state q is conflict with any of the existing labels at q.
- *Branch*: A stack to store branching states. *Branch* involves the following operations: *Push* adds a new item to the top of *Branch*. *Pop* removes an item from the top of *Branch*. *Stacktop* return the value of the item from the top most position of *Branch* without deleting it.
- *add (set, x):* add an item x to the set
- *clear set:* remove all elements in a set

Algorithm 1: Depth-first $\Delta(STP)$ Construction

Input: $\triangle(S_1)$, $\triangle(S_2)$,ITP
Output: $\triangle(STP)$
Initialization: $Current$=q_0^{STP}, $Branch$=[], $Current.visited$=[ ], $End\_Exploration$=$False$;
while *not End_Exploration* **do**
  Rules (Current) ;
  // If only one rule can be applied
  if *Current.nb_succ ==1* **then**
    Create_transition ($\triangle(STP)$, (Current, Next)) where Next∈ Current.succ;
    Current = Next
  end
  // If at least two rules can be applied
  if *Current.nb_succ>=2* **then**
    if *Current ∉ Branch* **then**
      Push(Branch,Current)
    end
    if ∃ *Next where Next ∈ Current.succ ∧ (Current, Next) ∉ Current.visited* **then**
      add (Current.visited, (Current, Next)) ;
      if Γ*(Current) = Ø* ***or*** Γ*(Current) ≠Ø ∧ not* Conflict ***or*** *(*Γ*(Current) ≠Ø ∧* Conflict ***and*** ∃ z∈Γ(Current) ∩Accept_path_states *)* **then**
        Create_transition ($\triangle(STP)$, (Current, Next));
        Current = Next
      end
    end
    // All rules at current states have been checked:
    else
      Pop(Branch, Current);
      Clear *Current.visited*;
      if *Branch ≠ [ ]* **then**
        Current =Stacktop (Branch);
      else
        End_Exploration = True
      end
    end
  end
  // No rule can be applied;
  else
    if $q_{Current}^{ITP} \in Accept^{ITP}$ **then**
      Accept_path_transitioins=Accept_path_calculation(Current,$\triangle(STP)$);
    end
    if *Branch ≠ [ ]* **then**
      Current =Stacktop (Branch);
    end
    else
      End_Exploration = True
    end
  end
end

A.2 Accept Path Search

Each time a state $q \in Accept^{\Delta(STP)}$ is reached, function *Accept_path_calculation* is called to search and mark the path that traverses the initial state of $\Delta(STP)$ and the state q. *Accept_path_calculation* is based on backtracking. i.e., from state q, a backward search is carried out to find all transitions traversing $q_0^{\triangle(STP)}$ and q. The calculation plays two roles: *(i)* It inhibits conflict transition generation. If at branching state q, there already exist some transitions that lead to $Accept^{\Delta(STP)}$, then there is no need to generate other transitions that conflict with the existing ones. *(ii) Accept_path_transitions* are important interoperability test case generation because they stand for the behavior of the SUT which leads to *Pass* verdict.

The algorithm requires a time complexity linear w.r.t the size of the transition relation of the composite product $\Delta(STP)$ and a space complexity linear w.r.t state space.

- *end_marking*: Boolean value to break out of the while loop. *end_marking* = *False* means not all *Accept_path_transitions* have been found yet.

Algorithm 2: Accept Path Calculation

```
Input: Current, △(STP)
Output: Accept_path_states, Accept_path_transitions
Begin
Accept_path_states.add(Current)
end_marking=False
while not end_marking do
    end_marking=True ;
    for (q, a, p) in T^△(STP) do
        if p∈Accept_path_states then
            Accept_path_transitions.add((q, a, p)) ;
            if q ∉ Accept_path_states then
                Accept_path_states.add(q) ;
                end_marking=False
            end
        end
    end
end
Return Accept_path_states
Return Accept_path_transitions
End
```

References

[1] ISO. Information Technology - Open System Interconnection Conformance Testing Methodology and Framework, Parts 1-7. International Standard ISO/IEC 9646/1-7,1994.

[2] N. Arakawa, M. Phalippou, N. Risser, and T. Soneoka. Combination of conformance and interoperability testing, in Formal Description Techniques, M. Diaz and R. Groz, Eds. New York: Elsevier, vol. C-10, pages 397–412, 1993.

[3] Rafiq, O., Castanet, R.: From conformance testing to interoperability testing. In Proc. 3rd Int. Workshop Protocol Test System, pp. 371–385, 1990.

[4] S. Schulz, A. Wiles and S. Randall. TPLan-A notation for expressing test purposes. ETSI, TestCom/FATES, LNCS 4581, pages 292-304, 2007.

[5] C.Jard and T.Jéron. TGV: theory, principles and algorithms. Internatioinal Journal on Software Tools for Technology Transfer, 2004.

[6] L. Verhaard, J. Tretmans, P. Kars, Ed. Brinksma. On asynchronous testing. In Gregor von Bockmann, Rachida Dssouli, and Anindya Das, editors, Protocol Test Systems, volume C-11 of IFIP Transactions, pages 55-66. North-Holland, 1992.

[7] K.El-Fakih, V.Trenkaev, N.Spitsyna and N.Yevtushenko. FSM based interoperability testing methods for multi stimuli model. In Roland Groz and Robert M. Hierons editors, TestCom, volume 2978 of Lecture Notes in Computer Science, pages 60-65. Springer, 2004.

[8] S.Seol, M.Kim, S.Kang, and S.T.Chanson. Interoperability test generation and minimization for communication protocols based on the multiple stimuli principle. IEEE Journal on selected areas in Communications, 22 (10), pages 2062-2074, 2004.

[9] O. Koné and R.Castanet. Test generation for interworking systems. Computer Communicqtions, 23(7), pages 642-652, 2000.

[10] A.Desmoulin, Test d'interopérabilité de Protocoles: de la Formalisation des Critères d'interopérabilité à la Génération des Tests, Doctor thesis, University Rennes 1, 2007.

[11] S.Seol, M. Kim, S.Kang and J.Ryu. Fully automated interoperability test suite derivation for communication protocols. Computer Networks, 43(6): 735-759, 2003.

[12] S.Kang, J.Shin, and M.Kim. Interoperability test suite derivation for communication protocols. Computter Networks, vol. 32, no.3, pages 347-364, 2000.

[13] N.Griffeth, R.Hao, D.Lee, R.K.Sinha. Integrated system interoperability testing with applications to VoIP. IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification, 2000.

[14] Cormen, Thomas H, Leiserson, Charles E., Rivest, Ronald L. Introduction to Algorithms (1st ed.). MIT Press and McGraw-Hill. ISBN 0-262-03141-8. Section 26.2, "The Floyd–Warshall algorithm", pages. 558–565. 1990.

[15] C. Viho, L. Tanguy and S. Barbin, Towards a formal framework for interoperability testing, 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems, 2001.