Uncovering Unknown System Behaviors in Uncertain Networks with Model and Search-based Testing

Ruihua Ji*, Zhong Li*, Shouyu Chen*, Minxue Pan*†, Tian Zhang*†, Shaukat Ali‡†, Tao Yue‡†, Xuandong Li*

*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210093, China

‡Simula Research Laboratory, Oslo, Norway

†Corresponding author

{yiting.ji, remainxy}@gmail.com, lz_seg@163.com, {mxp, ztluck, lxd}@nju.edu.cn, {tao, shaukat}@simula.no

***Abstract*—Modern software systems rely on information networks for communication. Such information networks are inherently unpredictable and unreliable. Consequently, software systems behave in an unstipulated manner in uncertain network conditions. Discovering unknown behaviors of these software systems in uncertain network conditions is essential to ensure their correct behaviors. Such discovery requires the development of systematic and automated methods. We propose an online and iterative model-based testing approach to evolve test models with search algorithms. Our ultimate aim is to discover unknown expected behaviors that can only be observed in uncertain network conditions. Also, we have implemented an adaptive search-based test case generation strategy to generate test cases that are executed on the system under test. We evaluated our approach with an open source video conference application—Jitsi with three search algorithms in comparison with random search. Results show that our approach is efficient in discovering unknown system behaviors. In particular, (1+1) Evolutionary Algorithm outperformed the other algorithms.**

Keywords*— Uncertainty, Model-based Testing, Search-based Testing, Uncertain Networks

I. Introduction

Software systems are increasingly relying on information networks for communication. Such networks are inherently prone to uncertain conditions (e.g., packet loss). These conditions often lead software systems to exhibit behaviors that are unknown during their design. Such behaviors might not be necessarily safe and result in software system failure. Thus, it is important to discover these behaviors before their deployment with systematic and automated methods. In current literature, software systems are often tested with assumptions on information networks. These assumptions do not necessarily hold in the real operation. Thus, it is imperative to discover such unknown behaviors before deployment and improve software systems' implementation to avoid potential failures.

Our approach relies on Model-based Testing (MBT) [1-3] that relies on models for executable test case generation. These models (i.e., test models) represent abstract representation of the expected behavior of a System under Test (SUT). In our context, test models also include expected behaviors of a SUT in the presence of known uncertain network conditions. These test models are at the core of our approach to discover unknown expected test behaviors that are unknown at the design time. Our approach is an online testing approach, i.e., it combines test case generation and test case execution in an iterative process [4]. Test case execution results are used as feedback in the subsequent test generation cycles. Such information is also used to incrementally evolve test models to discover expected system behaviors that are previously unknown in the presence of uncertain network conditions.

We also propose an adaptive search-based test case generation strategy that is applied to generate test cases from a test model capturing the expected behavior of a SUT in the presence of uncertain network conditions. We proposed an adaptive fitness function, which is integrated with commonly used search algorithms. This fitness function guides the search algorithms to generate possible system behaviors relying on historical test execution information and known system behaviors. Our adaptive fitness function is developed to find possible system behaviors by maximizing their similarity with the known system behaviors. Also, our fitness function maximizes the diversity of all the possible system behaviors. Finally, our approach generates test cases from the generated possible system behaviors.

We evaluated three search algorithms to select which algorithm can help us to cost-effectively discover unknown behaviors in the presence of uncertain network conditions. These algorithms are (1+1) Evolutionary Algorithm (EA), Alternating Variable Method, Genetic Algorithm. Random Search (RS) is the baseline for comparison. We applied our approach to an open source video conference application—*Jitsi* [30]. The results of our evaluation show that our approach was effective to discover unknown system behaviors under uncertain network conditions. (1+1) EA turned out to be the best algorithm, which performed better than the other algorithms.

The rest of this paper is organized as follows. Section II introduces a running example. Section III introduces test models and related definitions of concepts. We present our approach in Section IV. The evaluation is given in Section V, followed by the related work in Section VI and conclusion in Section VII.

II. Running Example

We will use a running example of a videoconference system to illustrate our approach. The simplified state machine and class diagram of the running example are shown in Fig 1 and Fig 2. We use two key operations, i.e., dial and disconnect. The dial operation initiates a videoconference or adds a participant into an existing videoconference. The disconnect operation ends a videoconference or removes a participant from a videoconference. We use two system variables to define states of the videoconference software, i.e., *activecall* and *videoquality*. *Activecall* is used to count all the active calls in the current videoconference. The value of *activecall* is obtained from method *getActiveCall* in class *Jitsi*. *Videoquality* indicates the video quality of the current videoconference. The value of *videoquality* is obtained with method *getVideoQuality* in class *Jitsi*. The value of *videoquality* includes four types, *good*, *acceptable*, *bad*, and *nopics*.

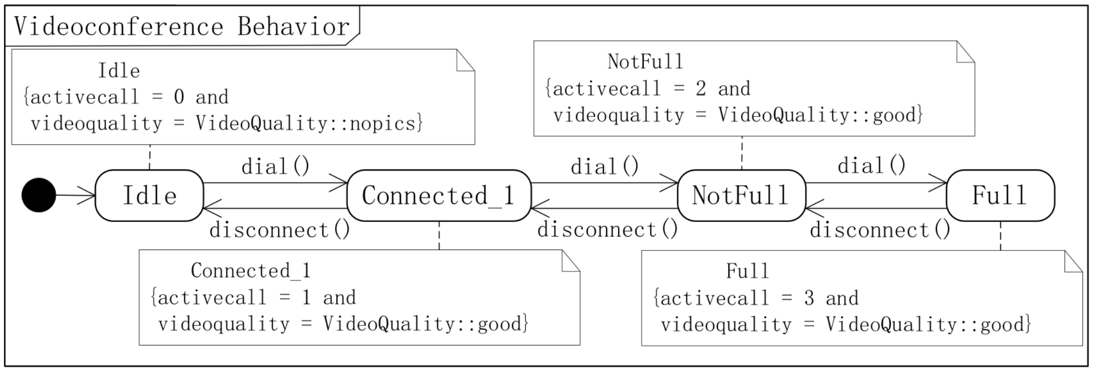

Fig. 1. A Running Example

In total, the test model, i.e., a state machine of the running example (Fig 2) has four states and six transitions. When a transition is fired, the input includes an operation (the trigger of the transition) and the network condition which is recorded separately in a list. There are no guards in our running example. We consider the state machine in Fig 2 as the initial test model and network conditions do not have any packet delay, loss, corruption, or duplication. Each transition has associated network conditions captured as values of packet delay, loss, corruption, and duplication.

III. Formalization and Definitions

In this section, we first introduce the modeling of SUT and the network conditions in Section III-A. We provide necessary definitions related to test model in Section III-B. Finally, in Section III-C, we provide definitions of test paths and test cases.

A. Test Modeling

Our test model is a simplified standard UML state machine, i.e., all the states are *simple* states, i.e., no hierarchy. Also, all transitions are without effects. Each transition has a trigger and an optional guard to fire the trigger.

We define inputs to the system in the test model consisting of two parts: user operations and network conditions. A user operation is the trigger of a transition in the test model. A network condition associated with a transition is in the list L which is maintained with the test model. User operations in the running example corresponding to dial and disconnect are represented as $UO = \{uo_1, uo_2\}$. All the network conditions maintained in the list L are represented as $NC = \{nc_1, nc_2, \cdots, nc_n\}$, i.e., the NC for the running example in Fig 2 has only one member nc_1 = *(packet delay = 0ms, packet loss = 0%, packet corruption = 0%, packet duplication = 0%)*. Inputs to the system can be represented as $input_j = (uo_k, nc_i)$. If uo_k is the trigger of one transition, the evaluation of the guard is true, and network conditions are held in the real network, the transition is fired.

Formally, let Σ be the set of all the inputs to the system modeled as the test model, the simplified state machine is a 4-tuple $M = (Q, T, \{q_0\}, \Sigma)$, where, Q is the set of all the states in M; T is the set of all the transitions in M; $q_0 \in Q$ is the initial state. Σ is the set of all the inputs, and $input_j = (uo_k, nc_i) \in \Sigma$.

We maintain a network condition list L. Each node in the list represents a transition with a network condition $node_k = (transition_m, nc_i)$. For each transition, there can be more than one node in the L list. This means that each transition can be triggered in these network conditions.

Note that packet delay, loss, corruption, and duplication represent common problems in networks [18-22]. Packet delay means packets take longer to get to their destination. Packet loss is a failure of one or more transmitted packets to arrive at their destination. Packet corruption is defined as the portion of data packets transmitted at the MAC layer which are interrupted at the receiver due to interference. Packet duplication is defined as part of the packets transmitted that are identical to others. Packet delay is measured as time in millisecond (ms) ranging from 0ms to 1000ms. Packet loss, packet corruption, and packet duplication are all measured in percentage, whose ranges are between 0% and 100 %. A possible network condition at any

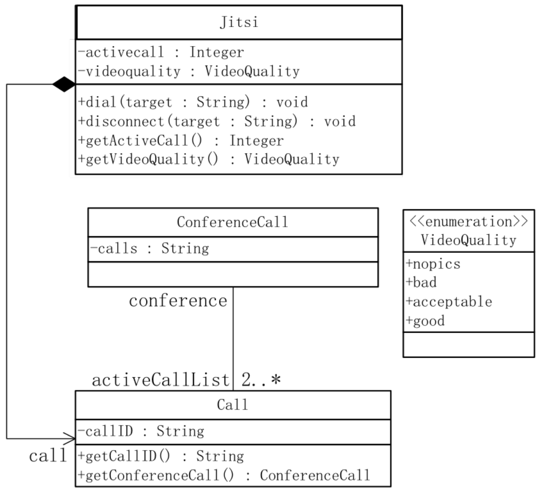
Jitsi
-activecall : Integer
-videoquality : VideoQuality
+dial(target : String) : void
+disconnect(target : String) : void
+getActiveCall() : Integer
+getVideoQuality() : VideoQuality
ConferenceCall
-calls : String
<<enumeration>>
VideoQuality
+nopics
+bad
+acceptable
+good
conference
activeCallList 2..*
Call
-callID : String
call
+getCallID() : String
+getConferenceCall() : ConferenceCall

Fig. 2. Class Diagram of the Running Example

particular time can be represented as a 4-tuple, ne_i = *(packet delay, packet loss, packet corruption, packet duplication)*. For example, ne_i = *(10ms, 1%, 2%, 2.5%)* shows that the current network conditions are 10ms packet delay, 1% packet loss, 2% packet corruption, and 2.5% packet duplication.

B. *System Behaviors, Possible System Behaviors and (previously) Unknown System Behaviors*

We define a *System Behavior* as *b = (q_1, guard, trigger, network condition, q_2)*. The *source state* is q_1 and *target state* is q_2. The *guard* is optional, the *trigger* is a user operation, and the *network condition* is defined by the four variables.

Suppose, we have a system behavior *b = (q_1, guard, trigger, network condition, q_2)* and a test model *M = (Q, T, {q_0}, Σ)*. If there is a $t_i \in T$ whose *source state* equals to q_1, *target state* equals to q_2, *trigger* equals to *b*'s *trigger*, *guard* equals to *b*'s *guard*, *network conditions* equal to *b*'s *network conditions*, then *b* is a *Known System Behavior*. Otherwise, *b* is an *Unknown System Behavior*. Recall that the *T* set is all the existing transitions in the test model, whereas q_1 belongs to *Q* of the test model and q_2 may belong to *Q*. If q_2 does not belong to *Q*, then q_2 is a previously unknown state, and as part of *b*, it is a newly discovered state. For example, in the *M* test model of the running example in Fig 2, a behavior *beh = (Connnected_1, null, dial, (52ms, 20%, 35%, 27%), New6)* is an unknown system behavior for *M* because *New6* is a state not belonging to *M*, whose activecall is *2* and videoquality is *nopics*.

Notice that, associated with its source, its target state, and one related network condition in *L*, one transition represents one system behavior. Since a transition, at least, has one related network condition in the *L* list, there is at least one system behavior that is represented. For example, given a transition *t = (Connected_1, null, dial, NotFull)* associated with *(t, ne_i) =(t, (0ms, 0%, 0%, 0%))* and *(t, ne_j) =(t, (20ms, 1%, 2%, 2.5%))* in the running example, the transition represents two related system behaviors, b_1 = *(Connected_1, null, dial, (0ms, 0%, 0%, 0%), NotFull)* and b_2 = *(Connected_1, null, dial, (20ms, 1%, 2%, 2.5%), NotFull)*. Therefore, every transition from *T* combining its source state, target state, and related network conditions represent, at least, one known system behaviors.

A *Possible System Behavior* is an unknown behavior having an associated probability of happening in the real world. We represent it as *pb = (q_1, guard, trigger, network condition, q_2)*. q_1 belongs to *Q* of the test model; q_2 may belong to *Q*. However, it is unknown, during the discovery process, if *pb* exists in reality. A possible system behavior is different from a normal system behavior since it has an attribute called *probability* indicating the likelihood of the existence of a system behavior. When the probability is 100%, the possible system behavior is an unknown system behavior. Therefore, a previously unknown possible system behavior with a high probability implies that there is a high probability of discovering a previously unknown system behavior. Taking the example of the M test model of the running example in Fig 2, and given an unknown system behavior *beh = (Connnected_1, null, dial, (52ms, 20%, 35%, 27%), New6)*, if *beh* has not been executed on SUT to check whether it exists, then *beh* is a possible unknown system behavior.

Testing a possible system behavior *pb* means that setting up an input to the system when it is in state q_1 and checking whether the system will end up in q_2. After testing, *pb* is revised according to execution results. If the revised *pb* is not covered by any $t_i \in T$ of the test model as mentioned above, *pb* is then an unknown system behavior.

C. *Test Paths and Test Cases*

Our objective is to discover unknown system behaviors by testing the SUT with generated possible system behaviors. To reach the source state of the possible system behavior, test cases used to discover unknown system behaviors should include the following two steps: 1) leading the SUT going from the initial state of the test model to the state where the source state of the possible system behavior will be reached, and 2) setting the input (i.e., user operation and network condition). Necessarily, a *Test Path* in the test model is a path starting from the initial state of the state machine with a possible system behavior starting from the last state of the path. It is an *Abstract Test Case* if the path of a *Test Path* has no concrete data; otherwise, it is *Executable Test Case*.

We apply the *Shortest Simple Path* strategy [38] to generate an abstract test case based on the given possible system behavior and test model. We take the test model as a map consisting of edges (transitions) and vertexes (states), and the strategy first uses the Dijkstra algorithm [5] to find the shortest path between the initial state and the source state of the possible system behavior. This path is part of a *Test Path*, which starts from the

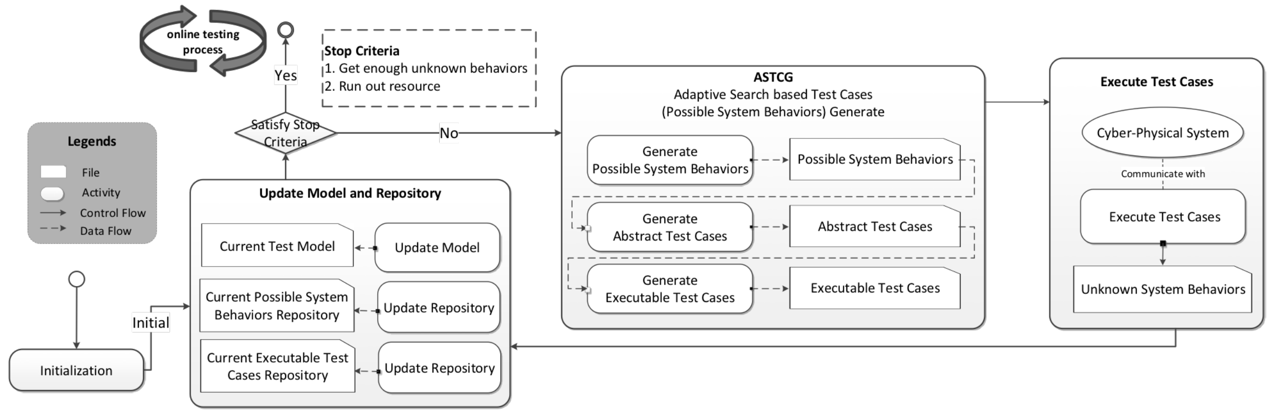

Fig. 3. The overall approach of ASUBE

initial state and ends at the source state of the possible system behavior. Tailing the possible system behavior to the path, a *Test Path* is constructed which is named as an *Abstract Test Case*. For example, in Fig 2, given a possible unknown system behavior *beh = (NotFull, null, dial, (52ms, 20%, 35%, 27%), New6)*, we first generate the path *<Idle> → <Connected_1> → <NotFull>*, and then the generated abstract test case is *atc = (<Idle> → <Connected_1> → <NotFull>, (<NotFull>, (null, dial, (52ms, 20%, 35%, 27%)), < New6 >))*.

We use a random strategy to generate the executable test cases. The process constructs a sequence of inputs, each of which is composed of a user operation specified by the trigger of the transition, a guard, and a network condition. The inputs before the last one are generated based on the transitions in the test path, and the last input is generated based on the possible unknown system behavior. To construct an input generated from a transition, we randomly construct the guard and randomly choose a network condition which is related to the transition. To construct an input generated from a possible unknown system behavior, we construct a guard as null, but use the network condition directly. Given the abstract test case used above, the executable test case *etc = (null, dial, (0ms, 0%, 0%, 0%)) → (null, dial, (0ms, 0%, 0%, 0%)) →(null, dial, (52ms, 20%, 35%, 27%))*.

IV. Our Approach

Our proposed approach, named as Adaptive Search Based Unknown Behavior Explorer (ASUBE), uses an incremental online testing process. It contains a sequence of test cycles to discover unknown system behaviors in uncertain network conditions. A test cycle consists of three activities: adaptive search-based test case generation, test case execution, and historical information update. In the first test cycle, we provide an initial test model associated with its network conditions list, an empty repository for collecting all the possible unknown system behaviors, and an empty repository for collecting all the generated executable test cases. In a test cycle, the test case generation step takes the current test model and current possible system behaviors repository updated in the last test cycle as inputs. It then uses a search algorithm (along with a fitness function) to generate a new set of test cases that are expected to have a high chance of observing unknown behaviors. The executable test case repository is used to store all the generated executable test cases as part of the outputs of our approach.

In the rest of the section, we provide an overview of ASUBE (Section IV-A), formal definitions of the key concepts (Section IV-B), the adaptive search-based test case generation steps (Section IV-C), and the other steps of ASUBE (Section IV-D).

A. Overview

Our overall approach is shown in Fig 3. The approach uses an incremental online testing process containing a sequence of test cycles to discover unknown system behaviors under uncertain network conditions. The process terminates (i.e., the number of test cycles) when any of the two conditions meet: 1) no new unknown system behaviors are observed; 2) having reached the ceiling of the budget and resources allocated.

The first test cycle is started with initialization, which has the following steps: 1) ASUBE loads the test model capturing known expected behaviors of SUT in known network conditions. Note that this model is constructed manually by test engineers. 2) ASUBE initializes a repository to collect all the possible system behaviors used to generate test cases that will be executed during each test cycle. Also, 3) ASUBE maintains an empty repository to collect all the generated executable test cases which are part of the output of ASUBE. As the test process goes on, the test model is updated with observed and previously-unknown system behaviors at the last phase of each test cycle. In each test cycle, the possible system behaviors repository is appended with newly generated possible system behaviors.

The key component of ASUBE is the Adaptive Search-Based Test Case Generation (ASTCG). ASTCG is executed until a predefined number of test cases in a test cycle (e.g., 5) are generated. When generating each test case, ASTCG first generates a possible system behavior by applying search algorithms, which is not validated but estimated to have a high probability of being a valid behavior, according to the heuristics we defined and implemented as the fitness function of the search algorithms. Then, ASTCG generates a test case based on the possible system behavior and the current test model, using the *Shortest Simple Path* strategy as described in Section III-C. At last, ASTCG transforms the generated abstract test case into an executable test case using a random test data generation strategy discussed in Section III-C.

We encode the test case generation problem (actually a possible system behavior generation problem) as a search problem and define an adaptive fitness function, together with the selected search algorithms, to generate test cases (possible system behaviors). The fitness function is adaptive in the sense that at each test cycle it adapts to the current test model associated with the network condition list and the set of all the generated possible system behaviors, which are updated at each test cycle. The overall aim is to generate possible system behaviors containing specific network conditions with a high probability of being valid, then use them to generate test cases to find the unknown behaviors caused by some network condition. Consequently, executing them leads to a high chance of discovering unknown system behaviors caused by the network environment. An applied search algorithm hence aims to maximize the probability of a possible system behavior being valid. More details about ASTCG are provided in Section IV-C.

Abstract test cases newly generated by ASTCG are then transformed to executable test cases executed together with the SUT. Section III-C presents further details. Test case execution results, i.e., whether we find unknown system behaviors or

TABLE I. Definitions

Symbol	**Explanation**
T_i	The set of test cases generated in $cycle_i$
B_i	The set of possible system behaviors generated by a search algorithm and used to generate Ti in $cycle_i$
M_{i-1}	The input test model in $cycle_i$
M_i	The updated test model in $cycle_i$
ET_{i-1}	The set of all the generated test cases before $cycle_i$
ET_i	The set of all the generated test cases after $cycle_i$
EB_{i-1}	The repository of all the generated possible system behaviors in the previous test cycles of $cycle_i$
EB_i	The repository of all the generated possible system behaviors after $cycle_i$

known system behaviors, are obtained automatically. Observed system behaviors might be existing (therefore known) or (previously) unknown. Previously-unknown system behaviors are therefore added to the test model.

B. Formal Definitions

Given a test case generation process p_{ASTCG}, p_{ASTCG} has n sequential test cycles and the i^{th} cycle of p_{ASTCG} can be represented as $cycle_i$. Table I provides a set of definitions to describe the test case generation process.

To start a process p_{ASTCG}, two inputs are given as mentioned in Section III-A, i.e., the initial test model represented as $M_0 = (Q_0, T_0, \{q_0\}, \Sigma)$, an empty possible system behavior repository represented as $EB_0 = \emptyset$, and an empty test case repository represented as $ET_0 = \emptyset$. We use F_{TCG} to represent the functionality of every test cycle. Since the inputs include the test model M_{i-1}, and the possible system behavior repository EB_{i-1}, the i^{th} test cycle, $cycle_i$, could be represented as $(M_i, EB_i) = F_{TCG}(M_{i-1}, EB_{i-1})$. In $cycle_i$, a set of possible system behaviors, B_i, is generated. Using B_i and EB_{i-1}, we could get $EB_i = B_i \cup EB_{i-1}$. At the same time, the set of test cases executed in this test cycle, T_i, will be generated from B_i, and all the generated test cases will be collected by ET_i through $ET_i = T_i \cup ET_{i-1}$.

In general, the test process is dynamic and receives feedback (via test case execution and model update) and works as the following sequence:

$$(M_1, EB_1) = F_{TCG}(M_0, EB_0)$$
$$(M_2, EB_2) = F_{TCG}(M_1, EB_1)$$
$$\cdots\cdots$$
$$(M_n, EB_n) = F_{TCG}(M_{n-1}, EB_{n-1}).$$

C. Adaptive Search-Based Test Case Generation (ASTCG)

As discussed in Section IV-A, when using ASTCG to generate test cases, we first generate the specified number of possible system behaviors by repeatedly applying search algorithms. Running search algorithm once at most generates one possible system behavior. Afterwards, we use the *Shortest Simple Path Strategy* (Section III-C) and the random test data generation strategy (Section III-C) to generate executable test cases based on the possible system behavior. Therefore, generating possible system behaviors becomes the key step of ensuring the effectiveness of exploring unknown behaviors of the system in diverse network conditions.

1) Overview of ASTCG

Suppose that the process is in the i^{th} test cycle ($cycle_i$) of a process p_{ASTCG}, the current test model M_{i-1} and all the generated possible system behaviors BT_{i-1} are the inputs to this test cycle. ASTCG follows the following three steps. Note that the following three steps will be repeated sometimes in ASTCG so that enough test cases are obtained in every test cycle. It is due to the following reasons: 1) going through the three steps one time produces one possible system behavior (representing one test case) and a predefined number of test cases are needed in one test cycle; 2) a possible system behavior generated is either a possible known system behavior or a possible unknown one. We use the possible unknown since we want to find the unknown system behaviors; and 3) repeating the three steps could produce the same possible system behaviors (i.e., the same test cases will be produced from these behaviors). In this case, only one of the possible behaviors is kept and others are discarded.

Step1: Obtain T_{i-1} (i.e., all the transitions of M_{i-1} in Table I) and BT_{i-1} (i.e., the possible system behaviors in BT_{i-1} are generated in the past cycles, and have been transformed into test cases in ET_{i-1} to be executed) as inputs to the selected search algorithm to generate new possible system behaviors. **Step2:** Apply the adaptive fitness function together with the selected search algorithm, i.e., GA, (1+1) EA to generate new possible system behaviors. More details are provided in Section IV-C-2)c). **Step3:** Repeat Step2 until a required number of possible system behaviors is generated. Executing Step2 can generate one possible system behavior every time. If it is a known system behavior according to M_{i-1} or a same one as the previous, it is discarded, and we repeat the step.

2) Adaptive Search-based possible behavior generation

a) Problem Representation

ASUBE constructs possible system behaviors which are previously unknown but estimated to have a high probability of being a valid behavior in every test cycle. These behaviors are then used to generate test cases.

A possible system behavior consists of a source state, a target state, a trigger, a guard, and a network condition. The source state could be any known state in the current test cycle. The target state could be any known or unknown state. When constructing a target state, we directly give random values to the state variables, so the constructed target state could be a known state or an unknown state. The trigger is the user operation, the guard is null, and the network condition is the tuple used to describe the network environment (Section II-A). In the context of *Jitsi*, operations include “dial()”, “disconnect()” and “keep()”. The “keep()” operation means that user does not do anything, and we try to observe the system behavior when the inputs only contain network conditions. The trigger could be any one of the operations. The network condition contains four variables, “Packet Delay” (1~100ms), “Packet Loss” (1~100%), “Packet Corruption” (1~100%) and “Packet Duplication” (1~100%). When constructing a network condition, we assign a value randomly to each variable.

Combining all the parts, we can get a complete possible system behavior. So, a possible system behavior is represented as a vector $pbv = (e_1, e_2, \ldots, e_9)$. In pbv, e_1 and e_2 represent the two variables of source state, e_3 represents the user operation, $(e4, e5, e6, e7)$ represents the network conditions *(PacketDelay, PacketLoss, PacketCorruption, PacketDuplication)*, and, e_8 and e_9 represent the two variables of the target state. All the possible *pbvs* construct the possible system behavior space, *BSpace*.

We defined and implemented heuristics “Obj()” (contains Similarity and Diversity, details in Section IV-C-2)b)) as the fitness function to find possible system behaviors. In the i^{th} test cycle, we need to find a system behavior B_k from the possible behavior space, *BSpace*, such that for:

$$Any\ B_j \in BSpcae, B_j \neq B_k, Obj(B_k) \geq Obj(B_j).$$

b) Definition of Adaptive Fitness Function

ASUBE aims to discover unknown system behaviors in uncertain network conditions. ASUBE uses test case execution information from previous cycles, i.e., the test model and the generated possible system behaviors, to generate new possible system behaviors, from which new test cases will be generated and executed. Search has two objectives in our case.

First, as discussed in Section III-B, a transition from T of the test model represents more than one known behavior. Therefore, all the transitions of the test model combining their source state, target state, and network conditions represent all known system behaviors. If a possible system behavior generated by search is similar to known system behaviors (T_{i-1} extracted from M_{i-1}, in $cycle_i$), it is considered that there is a high similarity between the newly generated possible behavior and all the known behaviors, implying that there is a high chance that the newly generated possible system behavior is valid. Second, newly generated possible behaviors should be different from each other such that there would be a higher chance for the generated possible behaviors to cover (via a new set of to-be-generated test cases from them) diverse situations. In general, we represent the two objectives as O1: maximizing the similarity between possible system behaviors and known system behaviors, and O2: maximizing the diversity among all the possible system behaviors.

O1: Maximizing the Similarity between Possible System Behaviors with Known System Behaviors.

We measure the objective as a Similarity Measure. To calculate the Similarity Measure, we calculate the similarity between a newly generated possible behavior and a set of known behaviors, which are represented as the transitions (combining its source state, target state, and network conditions) in the test model. To implement the Similarity Measure, we first need to encode the possible system behaviors and the transitions with network conditions to vectors, and then calculate similarity values according to a similarity metric. **Encode:** If a possible behavior is $pb = (q_1, guard, user\ operation, network\ condition, q_2)$, states q_1 and q_2 consist of two variables. In the running example, these are "number of active calls" and "video quality", and the input consists of a ne_j and a uo_k. Here, we use packet delay (delay), packet loss (loss), packet corruption (corruption) and packet duplication (duplication) to describe the network conditions. *pb* is represented as a vector $pbv = (e_1, e_2, ..., e_9)$, in which we have $q_1 = (e_1, e_2)$, $q_2 = (e_8, e_9)$, and guard = null. In the running example, there is no guard. $uo_k = e_5$, and $ne_i = (e_6, e_7, e_8, e_9)$. Therefore, one transition could generate some vectors, which represent some behaviors. **Metric:** We apply Euclidean Distance [6] to calculate the similarity value between a Possible Behavior and the behaviors derived from transitions. Given two vectors $pbv = (e_1, e_2, ..., e_n)$ and $pbv = (e'_1, e'_2, ..., e'_n)$, we get the distance between them with the formula below:

$$\mathrm{D}(pbv, pbv'\ ) = \sqrt{\sum_{i=1}^{n}(e_1 - e_2)^2}$$

The similarity is then the reverse of the distance. After normalization (with the normalization function nor() [24][25]), the range of similarity values is 0 to 1: $nor(Similarity) = 1 - nor(Diversity)$. We use the average of the all the Similarity values as the final output.

O2: Maximizing the Diversity among all Possible System Behaviors.

This objective is measured by a Diversity Measure. If a newly generated possible system behavior has a high Diversity value, it contributes to cover more space in the possible system behavior space, along with the other possible system behaviors. The implementation of the Diversity Measure is similar to the Similarity Measure. **Encode:** Objects used to calculate Diversity are possible system behaviors, and we use the method mentioned in Similarity Measure to transform all the possible system behaviors into vectors. **Metric:** Euclidean distance is directly the Diversity value. After calculating the Diversity value of a new possible system behavior and other generated possible system behaviors, we use the average of the all the Diversity values as the final output.

Based on the two objectives, we define our fitness function. A lower value of the fitness function represents better fitness. In the ith test cycle, the fitness function is:

$$\boldsymbol{fitnessvalue} = 2 - (nor(Similarity(B, M_{i-1})) + nor(Diversity(\mathrm{B}, BT_{i-1})))$$

Since similarity and diversity values are not within the same range, comparing them without normalization is inappropriate. Therefore, we use the following normalization function [24][25]:

$$\mathrm{nor(x)} = \frac{\mathrm{x}}{\mathrm{x} + 1}$$

c) The Selected Search Algorithms

We choose (1+1) Evolutionary Algorithm ((1+1) EA), Genetic Algorithm (GA), and Alternating Variable Method (AVM). AVM is chosen as the representative of local search algorithms. GA is the most commonly used global search algorithm. (1+1) EA is simpler than GA. However, it has shown better performance than GA in many works [10][23]. Random Search (RS) is used as the baseline for comparison.

D. Other Activities

Besides ASTCG, every test cycle also includes test case execution and historical information update activities (Fig 3).

1) Test Case Execution

Test case execution activity executes test cases generated by the test case generation activity of ASUBE. To support the execution together with its result reporting, we need test setup, test data generation, and test verdicts.

Test case execution involves providing inputs to trigger the behaviors of the SUT. Test setup includes developing APIs to control network environment and control specific actions of SUT. APIs to control network environment are provided by a tool named *NetEM* [29]—a network emulator. By sending string commands (i.e., *<Terminal1 dial Terminal2>*) to *NetEM*, NetEM could set the packet delay, packet loss, packet corruption and duplication of specified network equipment, such that specific network conditions can be enforced. The APIs used to control specific actions of the SUT are designed by the tester, and these APIs work by sending specific string commands to the remote clients and requiring the remote clients act according to the string commands. In the context of a videoconference system, the actions include *<Terminal1 dial Terminal2>*, *<Terminal1 addConfmember Terminal3>*, *<removeConfmember Terminal3>*, *<Terminal1 disconnectConf>*, and so on. When we have more than two participates of a video call, the video call becomes a conference, and the "dial" operation becomes "addConfmember" operation to add one member to a conference.

Based on the above explanation, the test data generated from a test case actually is a sequence of string commands. For example, in the context of a videoconference system, given an example of an executable test case is: *etc = (dial, (0%, 0ms, 0%, 0%))→(dial, (0%, 100ms, 2%, 5%))*. The sequence of string commands could be represented as:

(⟨Terminal1 dial Terminal2⟩ , SetNetworkEnvrionment(0%, 0ms, 0%, 0%))
→ (WaitToBeStable(25s))
→ (⟨Terminal1 addConfmember Terminal3⟩,
SetNetworkEnvironment(0%， 100ms, 2%， 5%))
→ (WaitToBeStable(25s)).

The "dial" and "addConfmember" commands are sent to videoconference system clients, and the "networkenvironment" commands are sent to *NetEM*. The "WaitToBeStable" commands are used locally, to allow the latter command to be sent after the previous command finished, i.e., before *terminal1* adds *terminal3* to the current conference, *terminal1* needs to construct the conference with *terminal2*, which takes some time.

The behaviors of the SUT after acting as the input test data are recorded automatically. It relies on sending system information after actions by the clients to the control unit. The information constructs the system behaviors on the control unit. We define two test verdicts to evaluate the unknown system behaviors. We compare the system behaviors constructed by the information from clients with the current test model. If the test model does not cover the behavior, it is an unknown system behavior. In this case, it needs to be delivered to the model update activity automatically.

2) Historical Information Update

We maintain a repository of possible system behaviors when starting the ASUBE exploration process which is empty at that time, as well as a repository of executable test cases. The repository of possible system behaviors is used as historical information to the next test cycle, while the repository of executable test cases is only used to record and output the executable test cases as part of the final output. In test cycle $cycle_i$, the set of newly generated possible system behaviors PBS_i are combined with the repository of possible system behaviors EB_{i-1}. Moreover, test model M_{i-1} is used to store the discovered previously-unknown behaviors. To achieve this, ASUBE implements the following two steps:

First, ASUBE transforms the newly-discovered previously-unknown behaviors to transitions. More specifically, ASUBE takes the source state and target state of the behavior as a transition's source state and target state; takes the user operation input of the behavior as the transition's trigger, and put the network conditions to the network condition list *L*. Second, if the source and target states of the new transition transformed from the behavior exist in the test model, ASUBE adds the new transition between the two states to the test model. If the source state of the new transition belongs to the test model and the target state of the new transition is a new state, ASUBE constructs the new state and add it to the test model, and then adds the new transition. Since we start from a known state as the source state when validating a new possible behavior by testing, we only consider the above two situations about updating the

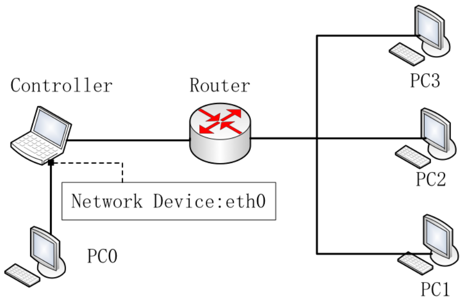

Fig. 4. The Network Topology

test model using new transitions. Finally, when the above steps are finished, we output the updated test model.

V. Evaluation

In this section, we introduce the case study, experiment environment (Section V-A), research questions (Section V-B), experiment design (Section V-C), experiment execution (Section V-D), results and analysis (Section V-E), discussion (Section V-F), and threats to validity (Section V-G).

A. Case Study

We selected an open source videoconferencing and instant messaging application, *Jitsi* [30]. Its implementation is available on multiple platforms including Windows, Linux, Mac OS X, and Android. Users implement clients on their platforms to obtain audio/video conference services. More information about *Jitsi* can be found at [30], including released versions and source code. We use Jisti as the case study to test its videoconference related functionalities such as making a call, with the ultimate goal of discovering unknown system behaviors when Jisti is operating under uncertain network environment.

For the experiment, we set up *Jitsi* on an internal network, whose topology is shown in Fig 4. Five PCs were deployed in the internal network: four of them (*PC0*, *PC1*, *PC2*, and *PC3*) have *Jitsi* deployed on each of them and *Jitsi* deployed on *PC0* is the SUT. The test program is deployed on the *Controller*. *PC0* communicates with *PC1*, *PC2*, and *PC3* through the *Controller* and the *Router*, and the network device *eth0* of the *Controller*

TABLE II. Definitions of Evaluation Metrics

Metric		Definitions
Effective-ness	**NUB**	Total number of behaviors discovered that are previously unknown.
	NUS	Total number of states discovered that are previously unknown.
Cost	**SR**	Running time of a search algorithm to generate test cases in a test process (Section IV).
	ST	The total time (in minutes) used to search for test cases (Section IV).
	ET	The total test case execution time of a test process (Section IV).
	AT	The total time cost of a test process (Section IV), AT = ET + ST.
Efficiency	**BOSR**	NUB/SR
	BOST	NUB/ST
	BOET	NUB/ET
	BOAT	NUB/AT

connects directly to *PC0*. *PC0*, *PC1*, *PC2*, and *PC3* have Windows 7 operating system installed and have Intel Core i5-4210M 2.6G Hz with 8G RAM. The *Controller* is installed with the Ubuntu 12.10 operating system with Intel Core L9400 1.86G Hz with 3.8G RAM.

The test program deployed on *Controller* automatically generates test cases (Section III-C), executes test cases (Section IV-D), conducts online testing of the *Jitsi* application deployed on *PC0*, and processed testing results (Section IV-D). To support the automatic process of ASUBE, we developed required testing APIs based on the source code of *Jitsi*.

The state machine of the initial test model of the case study is shown in Fig 2, which consists of four states and six transitions. The two key system variables used to define system states are "number of active calls" and "video quality of calls". For example, the *Connected_1* state is represented as: *activecall = 1*, *videoquality = good*. The *activecall* variable stores the number of active calls at a given time and the *videoquality* stores video quality of the current active call(s) measured with the Peak Signal to Noise Ratio (PSNR) metric [28]. Simply speaking, PSNR compares a transmitted figure with the original figure and give the quality value of the transmitted one. To assess the video quality, we take 10 screenshots of both the videoconference in an uncertain network condition and the one in a normal network condition, get 10 PSNR values by calculating corresponding to the 10 couples of screenshots, and finally use the average of the 10 PSNR values as the output videoquality value. We implemented such videoconference video quality assessment.

Let's take an example of behavior *b = (Idle, null, dial, network condition, Connected_1)*. The observed behavior is that the client deployed on *PC0* dials the client on *PC1* in the network condition specified by *network condition* of *b*, and a call of high video quality (measured by PSNR to get a high value, such as 3 in Fig 2, of the average quality of the pictures from the video) between the two clients is created. The original test model

TABLE III. RESULTS FOR TEST CASES PER CYCLE (10 TEST CASES)

Metric	RS	AVM	(1+1) EA	GA
NUB	20.56	19.5	**33.56**	21.75
NUS	7	6.5	7.44	**8**
SR	627	1021.67	**189.33**	342.5
ST(minutes)	11.979	11.685	**2.534**	4.432
ET(minutes)	255	**227**	306	285
AT(minutes)	267	**239**	309	289
BOSR	0.0328	0.0191	**0.1773**	0.0630
BOST	1.7163	1.6688	**13.2439**	4.9075
BOET	0.0804	0.0857	**0.1095**	0.0762
BOAT	0.0768	0.0815	**0.1086**	0.0751

TABLE IV. RESULTS FOR TEST CASES PER CYCLE (5 TEST CASES)

Metric	RS	AVM	(1+1) EA	GA
NUB	19.71	14.875	**20.25**	18.67
NUS	7.286	6.5	6.5	**7.67**
SR	527	459.625	**74.625**	645
ST(minutes)	5.66	3.919	**0.7045**	4.752
ET(minutes)	173	**146**	162	178
AT(minutes)	178	**150**	163	182
BOSR	0.0374	0.0324	**0.2714**	0.0289
BOST	3.4823	3.7956	**28.7437**	3.9289
BOET	0.1138	0.1013	**0.1249**	0.1048
BOAT	0.1102	0.099	**0.1243**	0.1021

describes that a video call made by *Jitsi* should be in high video quality when the network environment is normal. Otherwise, *Jitsi* makes calls in low video quality or even fails to make video calls, which are uncertain (and previously-unknown) behaviors that need to be automatically discovered with ASUBE.

B. *Research Questions*

Our overall objective is to assess the cost-effectiveness of ASUBE to discover unknown system behaviors. However, combining ASUBE with different search algorithms leads to various degrees of cost-effectiveness. We selected three search algorithms to be integrated with ASUBE: (1+1) Evolutionary Algorithm ((1+1) EA), Genetic Algorithm (GA), and Alternating Variable Method (AVM). Random Search (RS) was used as the comparative baseline. We will answer the following two research questions: **RQ1**: Are AVM, (1+1) EA and GA effective regarding discovering unknown behaviors comparing with RS? **RQ2**: Which of the three selected search algorithms fits the best regarding discovering unknown behaviors?

C. *Experiment Design*

To address RQ1 and RQ2, we integrated the selected search algorithms ((1+1) EA, GA, and AVM) with ASUBE and compared results with the ones obtained when combining ASUBE with RS. To answer RQ1 and RQ2, the results were evaluated based on the cost, effectiveness, and efficiency metrics defined in Table II.

D. *Experiment Execution*

In our experiments, all the algorithms were run up to 2000 generations each time. We collected all observed system behaviors from test case execution, test cases generated by ASTCG (Section IV-C), and time taken by each test process. We configured GA with a population size of 100, a crossover rate of 0.75, and a 1.5 bias for rank selection, as recommended by [10]. A standard one-point crossover was used, and mutation of a variable was set according to the standard probability 1/n, where n is the number of variables. Such mutation strategy was also used in (1+1) EA. We generated 5 and 10 test cases in every test cycle to apply the four search algorithms. We ran every test process (in total 8, i.e., four algorithms and two choices of the number of test cases to generate in one test cycle) 10 times to reduce the effect of the randomness of the algorithms on the results.

E. *Results and Analysis*

To answer RQ1 and RQ2, we applied the four algorithms individually with an option in either generating 5 test cases per cycle or generating 10 per cycle. Each test process has 10 test cycles. The whole experiment took about 305 hours to finish the total 80 test processes. The average of ten runs of the experiment is presented in Table III and Table IV. Based on the guidelines for reporting results for search-based software engineering problems [7], we applied Vargha and Delaney statistics ($\widehat{A_{12}}$) and Mann Whitney U Test (*p*-value) to compare the four selected search algorithms for uncovering unknown behaviors in uncertain networks. In our context, $\widehat{A_{12}}$ is used to compare all the effectiveness and efficiency measures of a pair of search algorithms. If $\widehat{A_{12}}$ is 0.5, the two algorithms are equivalent. If $\widehat{A_{12}}$ is greater than 0.5, the first algorithm in the pair has higher chances to obtain better efficiency than B. For Mann Whitney U

Test, we choose the significance level of 0.05, i.e., if p-value is less than 0.05, there is a significant difference. The results of the comparison are presented in Table V. In Table V, the number 5 or 10 used in the first column means the number of test cases generated per test cycle, and the A and p in the second column mean $\widehat{A_{12}}$ and p-value respectively.

Through rows SR and ST(min) in Table III, one can see that ASUBE with (1+1) EA used the least time of running each algorithm and searching for unknown possible system behaviors. Also, note that, in Table III, though AVM used the most time among the search algorithms, RS still took more time than AVM. It is because the time cost of once search is increasing for the accumulated historical information. Similar results were obtained when generating 5 test cases in every test cycle (see rows SR and ST(min) in Table IV).

Through rows BOST, BOSR, BOET, and BOAT in Table III and Table IV, it can be seen that (1+1) EA performed the best regarding all the efficiency measures. Through columns BOSR, BOST, BOET and BOAT in Table V, one could find the following key observations :1) (1+1) EA performs significantly better than RS in terms of BOST and BOSR, 2) (1+1) EA performs significantly better than AVM and GA in terms of all the efficiency measures, 3) RS, AVM, and GA do not have a significant difference when generating 10 test cases per test cycle, 4) The same results were obtained when generating 5 test cases per test cycle.

Based on the results, we answer RQ1 as follows: (1+1) EA significantly outperforms RS, which is not the case for AVM and GA. For RQ2, we recommend using (1+1) EA with ASUBE since (1+1) EA performed significantly better than the rest of the algorithms regarding the efficiency measures.

F. Discussion and Experience

TABLE V. RESULTS OF COMPARISON AMONG ALGORITHMS

		NUS	NUB	BOST	BOSR	BOET	BOAT
AVM-RS-10	**A**	0.37	0.62	0.75	0.58	0.16	0.25
	p	0.72	0.73	0.42	0.85	0.28	0.42
GA-RS-10	**A**	0.55	0.65	1.00	0.90	0.40	0.70
	p	>0.99	>0.99	>0.99	>0.99	>0.99	>0.99
EA-RS-10	**A**	0.52	1.00	1.00	1.00	0.88	0.88
	p	>0.99	**0.04**	**0.03**	**0.03**	0.14	0.14
AVM-EA-10	**A**	0.31	0.01	0.00	0.00	0.01	0.00
	p	0.24	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**
EA-GA-10	**A**	0.50	1.00	0.91	0.93	0.86	0.86
	p	>0.99	**<0.01**	**0.01**	**<0.01**	**0.02**	**0.02**
AVM-GA-10	**A**	0.31	0.35	0.13	0.16	0.26	0.23
	p	0.33	0.46	0.05	0.08	0.24	0.17
AVM-RS-5	**A**	0.28	0.18	0.51	0.37	0.35	0.33
	p	0.15	**0.04**	0.95	0.46	0.39	0.33
GA-RS-5	**A**	0.61	0.35	0.52	0.38	0.33	0.42
	p	0.62	0.56	>0.99	0.66	0.51	0.83
EA-RS-5	**A**	0.34	0.57	0.98	0.83	0.60	0.64
	p	0.33	0.68	**<0.01**	**0.02**	0.53	0.39
AVM-EA-5	**A**	0.46	0.14	0.00	0.00	0.21	0.20
	p	0.81	**0.01**	**<0.01**	**<0.01**	0.06	**0.04**
EA-GA-5	**A**	0.25	0.62	1.00	1.00	0.75	0.75
	p	0.24	0.60	**0.01**	**0.01**	0.27	0.27
AVM-GA-5	**A**	0.18	0.22	0.50	0.58	0.37	0.41
	p	0.13	0.21	>0.99	0.77	0.63	0.77

Based on the results reported in Table III, Table IV, and Table V, we see that regarding the efficiency measures, (1+1) EA performed significantly better than all the other algorithms. One of the key reasons is that (1+1) EA is a global search algorithm that uses only mutation operator to explore the search space rather than looking for solutions nearby to the existing ones. A GA, on the other hand, uses a mutation operator to explore and also uses a crossover operator to exploit nearby solutions and hence require more time to explore the search space. Thus, we may conclude that in our case best solutions are scattered across the search space and thus (1+1) EA was successful in finding the best solutions quickly. Due to the same reason, the performance of AVM was not good since it is a local search algorithm and it focused on searching the best solutions nearby the existing ones.

G. Threats to Validity

To reduce construct validity threats, we compare the four algorithms with the same metrics, i.e., new states and new transitions discovered via ASUBE. The same stop criteria, same mutation operator, and the same number of generations are used in search algorithms to avoid any bias. A possible threat to internal validity is that we conducted the experiments with only one set of settings for the search algorithms. However, these current settings fit with the common guidelines in the literature [37]. The generalization of the result is a typical external threat to validity. Currently, we only evaluated our approach with one case study. In the future, more case studies are needed to generalize the results.

VI. RELATED WORK

In this section, we discuss techniques that are closely related to our work, from the aspects of online testing, model evolution, and uncertainty-aware testing.

A. Online Testing

Larsen et al. [4][35] presented an online black-box testing tool, named as T-UPPAAL, for model-based testing of real-time embedded systems. T-UPPAAL generates one test case from a state machine and its assumed environment specifying required and allowed observable behaviors of the SUT, at a time, and simultaneously executes it. T-UPPAAL generates test inputs iteratively using the implemented randomized testing algorithm, which randomly chooses input events of the SUT and randomly sets the delay used to wait for the output until an input event finishes the execution. Hielscher et al. [31] presented the PROSA framework, which aimed to enable proactive self-adaptation of service-based applications by using online testing to detect changes and deviations between SUT and its specifications, based on which adaptations could be triggered. Their online testing process includes: generating test cases from specifications, executing test cases, and adapting the application based on the test results. Similarly, Sammodi et al. [25][32] used online testing to provide enhanced proactive adaptation capabilities to service-based applications. The authors proposed a usage-model updating and adaptation mechanism based on test results. The usage model records if the services are used, and the test cases generated online are to enhance the coverage of the usage model. To compare with these works, ASUBE aims a different objective, i.e., dynamically discovering unknown system behaviors under uncertain networks, addressed by implementing an adaptive search-based approach during the online testing process.

Walkinshaw et al. [34] applied online testing to reverse engineer behavior models of software systems. In every cycle, their approach generated a specified number of test cases from a Partial LTS model (PLTS), ran the generated test cases, and inferred the PLTS from test execution traces. Meinke and Sindhu [33] applied learning-based testing (LBT) techniques to test reactive systems. They modeled reactive systems as Kripke structures [36] and introduced a learning algorithm named as Incremental Kripke Learning (IKL) for model inference using a model checker. To compare with these works, ASUBE discovers unknown system behaviors under uncertain network environments. Same test cases could repeatedly be generated by these approaches across test cycles, which might lead to low efficiency of the overall approaches. Our approach, however, maintains all the historical test information as ASUBE maintains a test case repository and a system behavior repository. The adaptive search-based testing approach of ASUBE utilizes the information adequately.

B. *Model Evolution*

There are some works related to model evolution. For example, Ghezzi et al. [12] proposed an approach to recover the specification of a black-box software component from its run-time behaviors. It uses a deterministic finite state machine to model the partial behaviors of SUT, and then a finite state machine is generalized via graph transformation rules. Walkinshaw et al. [14] proposed an extended finite state machine inference technique, which was based on WEKA [26] and Daikon. However, none of these pieces of work evolves models with uncertainty captured.

The work that is most relevant to our approach is the *UncerTolve* framework by Zhang et al. [11] for interactively evolving Belief Test Ready Models (BMs) with uncertainty information explicitly captured. Such uncertainty information needs to be specified by test engineers to capture their beliefs about model elements of a BM, associated with one or more uncertainties due to "lack of knowledge" about a BM. Taking initial BMs of Cyber-Physical Systems (CPSs), known subjective uncertainty, and real data from the operation of CPSs as inputs, *UncerTolve* validates the syntactic correctness and conformance of BMs against real operational data, evolves objective uncertainty measurements, and evolves state invariants and guards of transitions with model execution and Daikon [13]. To compare with *UncerTolve*, we, however, take an online testing approach, which is dynamic and incremental, and particularly focus on uncertain network environments.

C. *Uncertainty-aware Testing*

Walkinshaw and Fraser [27] proposed a test generation approach that inferred a behavioral model of SUT using Genetic Programming (GP) from test execution. Test cases whose predictions elicit the highest degree of uncertainty concerning the current model are generated. Test case execution results were collected to update the behavioral model. Qin at al. [8] proposed sample-based interactive testing (SIT) approach for testing self-adaptive applications, focusing on inadequate consideration of environmental dynamics and uncertainty. Their proposed approach makes the assumptions that the input space of a self-adaptive application could be systematically split, adaptively explored, and mapped to the testing of the application's different behaviors. Their approach relies on an interactive application model represented by a tuple capturing interactions between an application and its environment and generates test cases with adaptive sampling (splitting the input space and mapping the splitting results to the application's behaviors under test).

Uncertainty-wise test case generation and minimization strategies relying on test ready models explicitly specifying subjective uncertainty were proposed by Ali et al. [3]. Test ready models are BMs developed with the Uncertainty Modeling Framework (*UncerTum*), which defines a set of UML Profiles. BMs are composed of two types of UML diagrams: belief class diagrams and belief state machines (BSMs). Two test case generation strategies were defined for the test case generation. Because the number of generated test cases might be large and test resource is often limited, the authors of [3] also proposed four uncertainty-wise, multi-objective search-based test case minimization strategies, which share the objectives of minimizing the number of test cases, maximizing the transition coverage, and maximizing four different uncertainty-related objectives.

To compare with the above-mentioned related works, our work presented in this paper focuses on uncertainty inherent in information networks. Testing software systems (e.g., CPS) in the presence of uncertain network conditions is not focused by existing works.

VII. Conclusion

These days, software systems commonly use information networks for communication. Given the inherent uncertainty in such networks, it is necessary to discover behaviors of software systems under such uncertainty. To this end, we proposed an online, iterative, and incremental model-based testing approach to evolve test models with search algorithms. Our ultimate goal was to systematically and automatically discover previously unknown expected behaviors of a software system in uncertain network conditions. We proposed an adaptive search-based test case generation strategy to generate test cases dynamically. We evaluated our approach with an open source video conferencing case study—*Jitsi*. Also, we evaluated four commonly used search algorithms. (1+1) Evolutionary Algorithm (EA) turned out to be the best search algorithm to discover unknown behaviors caused by uncertain network conditions. In particular, (1+1) EA performed better than the other algorithms.

Acknowledgment

This work is partially supported by National Natural Science Foundation (Grant Nos. 61690204, 61472180 and 61502228) of China and Natural Science Foundation of Jiangsu Province under Grant No. BK20150589 of China, and partially funded by the MBT4CPS project (Number:240013) funded by Research Council of Norway. Tao Yue and Shaukat Ali are funded by

MBT4CPS and Zen-Configurator projects in Norway.

REFERENCES

[1] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[2] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, “A survey on model-based testing approaches: a systematic review,” in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. ACM, 2007, pp. 31– 36.

[3] S. Ali, H. Lu, S. Wang, T. Yue, and M. Zhang, “Uncertainty- wise testing of cyber-physical systems,” in *Advances in Computers*. Elsevier, 2017, vol. 107, pp. 23–94.

[4] M. Mikucionis, K. G. Larsen, and B. Nielsen, “T-uppaal: Online model-based testing of real-time systems,” in *Proceedings of the 19th IEEE international conference on Automated software engineering*. IEEE Computer Society, 2004, pp. 396–397.

[5] M. Noto and H. Sato, “A method for the shortest path search by extended dijkstra algorithm,” in *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 3. IEEE, 2000, pp. 2316– 2320.

[6] P.-E. Danielsson, “Euclidean distance mapping,” *Computer Graphics and image processing*, vol. 14, no. 3, pp. 227–248, 1980.

[7] A. Arcuri and L. Briand, “A practical guide for using statistical tests to assess randomized algorithms in software engineering,” in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 1–10.

[8] Y. Qin, C. Xu, P. Yu, and J. Lu, “Sit: Sampling-based interactive testing for self-adaptive apps,” *Journal of Systems and Software*, vol. 120, pp. 70–88, 2016.

[9] S. Elbaum and D. S. Rosenblum, “Known unknowns: testing in the presence of uncertainty,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 833–836.

[10] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, “Generating test data from ocl constraints with search techniques,” *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1376–1402, 2013.

[11] M. Zhang, S. Ali, T. Yue, and R. Norgre, “Uncertainty-wise evolution of test ready models,” *Information and Software Technology*, vol. 87, pp. 140–159, 2017.

[12] C. Ghezzi, A. Mocci, and M. Monga, “Synthesizing intensional behavior models by graph transformation,” in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 430–440.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[14] N. Walkinshaw, R. Taylor, and J. Derrick, “Inferring extended finite state machine models from software executions,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 811–853, 2016.

[15] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, “Auto- mated test suite generation for time-continuous simulink models,” in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 595–606.

[16] S. Ali and H. Hemmati, “Model-based testing of video conferencing systems: challenges, lessons learnt, and results,” in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 353–362.

[17] Y.-M. Baek and D.-H. Bae, “Automated model-based android gui testing using multi-level gui comparison criteria,” in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on.* IEEE, 2016, pp. 238–249.

[18] K. Xu, M. Gerla, and S. Bae, “How effective is the ieee 802.11 rts/cts handshake in ad hoc networks,” in *Global Telecommunications Conference, 2002. GLOBECOM’02. IEEE*, vol. 1. IEEE, 2002, pp. 72–76.

[19] G. Huston, “Tcp in a wireless world,” *IEEE Internet Computing*, vol. 5, no. 2, pp. 82–84, 2001.

[20] R. Ludwig and M. Meyer, “The eifel detection algorithm for TCP,” *RFC*, vol. 3522, pp. 1–14, 2003.

[21] M. Andrews, K. Kumaran, K. Ramanan, A. Stolyar, P. Whiting, and R. Vijayakumar, “Providing quality of service over a shared wireless link,” *IEEE Communications magazine*, vol. 39, no. 2, pp. 150–154, 2001.

[22] W. Jiang and H. Schulzrinne, “Modeling of packet loss and delay and their effect on real-time multimedia service quality,” in *PROCEEDINGS OF NOSSDAV’2000*. Citeseer, 2000.

[23] A. Arcuri, “It really does matter how you normalize the branch dis- tance in search-based software testing,” *Software Testing, Verification and Reliability*, vol. 23, no. 2, pp. 119–147, 2013.

[24] Y. Zhang, A. Finkelstein, and M. Harman, “Search based require- ments optimisation: Existing work and challenges,” *Lecture Notes in Computer Science*, vol. 5025, pp. 88–94, 2008.

[25] D. Greer and G. Ruhe, “Software release planning: an evolutionary and iterative approach,” *Information and software technology*, vol. 46, no. 4, pp. 243–253, 2004.

[26] G. Holmes, A. Donkin, and I. H. Witten, “Weka: A machine learning workbench,” in Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on. IEEE, 1994, pp. 357–361.

[27] N.WalkinshawandG.Fraser,“Uncertainty-drivenblack-boxtestdata generation,” in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 253–263.

[28] Q. Huynh-Thu and M. Ghanbari, “Scope of validity of psnr in image/video quality assessment,” *Electronics letters*, vol. 44, no. 13, pp. 800–801, 2008.

[29] S. Hemminger *et al.*, “Network emulation with netem,” in *Linux conf au*, 2005, pp. 18–23.

[30] https://jitsi.org.

[31] J. Hielscher, R. Kazhamiakin, A. Metzger, and M. Pistore, “A framework for proactive self-adaptation of service-based applications based on online testing,” *Towards a Service-Based Internet*, pp. 122–133, 2008.

[32] O. Sammodi, A. Metzger, X. Franch, M. Oriol, J. Marco, and K. Pohl, “Usage-based online testing for proactive adaptation of service-based applications,” in *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*. IEEE, 2011, pp. 582–587.

[33] K. Meinke and M. A. Sindhu, “Incremental learning-based testing for reactive systems,” in *International Conference on Tests and Proofs*. Springer, 2011, pp. 134–151.

[34] N. Walkinshaw, J. Derrick, and Q. Guo, “Iterative refinement of reverse-engineered models by model-based testing,” in *International Symposium on Formal Methods*. Springer, 2009, pp. 305–320.

[35] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, “Testing real-time systems using uppaal,” in *Formal methods and testing*. Springer, 2008, pp. 77–117.

[36] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without bdds,” *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 193–207, 1999.

[37] A. Arcuri and G. Fraser, “On Parameter Tuning in Search Based Software Engineering,” Proc. Int’l Symp. Search Based Software Eng., 2011.

[38] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec 1959.