HIT-OR-JUMP: AN ALGORITHM FOR EMBEDDED TESTING WITH APPLICATIONS TO IN SERVICES

Ana Cavalli* David Lee** Christian Rinderknecht*
Fatiha Zaïdi*

*Institut National des Telecommunications
9 rue Charles Fourier
F-91011 Evry Cedex, France
Email : {Ana.Cavalli, Christian.Rinderknecht, Fatiha.Zaidi}@int-evry.fr
**Bell Laboratories, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974, USA
Email : lee@research.bell-labs.com

Abstract This paper presents a new algorithm, Hit-or-Jump, for embedded testing of components of communication systems that can be modeled by communicating extended finite state machines. It constructs test sequences efficiently with a high fault coverage. It does not have state space explosion, as is often encountered in exhaustive search, and it quickly covers the system components under test without being "trapped", as is experienced by random walks. Furthermore, it is a generalization and unification of both exhaustive search and random walks; both are special cases of Hit-or-Jump. The algorithm has been implemented and applied to embedded testing of telephone services in an IN architecture, including the Basic Call Service (BCS) and five supplementary services: Originating Call Screening (OCS), Terminal Call Screening (TCS), Call Forward Unconditional (CFU), Call Forward on Busy Line (CBL) and Automatic Call Back (ACB).
Keywords: conformance testing, embedded testing, communicating extended finite state machines, IN.

1 Introduction

With the advanced computer technology and the increasing demand from the users for sophisticated services, communication protocol systems are becoming more complex yet less reliable. Conformance testing, which ensures correct protocol implementations, has

become indispensable for the development of reliable communication systems. Traditional testing methods tend to test these systems as a whole or to test their components in isolation. Testing these systems as a whole becomes difficult due to their formidable size. On the other hand, testing system components in isolation may not be always feasible due to the interactions among the system components. Embedded testing or testing in context has become one of the main focuses of conformance testing research in recent years. The goal of embedded testing is to test whether an implementation of a system component conforms to its specification in the context of other components. It is generally assumed that the tester does not have a direct access to the component under test; the access is obtained through other components of the system. According to the standard: "if control and observation are applied through one or more implementations which are above the protocol to be tested, the testing methods are called embedded" [10].

Different approaches for embedded testing have been proposed in the published literature. They are based on fault models [15], on reducing the problem to testing of components in isolation [16], on test suite minimization [12, 13, 18], on fault coverage [19], and on the test of systems with semicontrollable and uncontrollable interfaces [4]. Most of these approaches resort to reachability graphs to model the joint behaviors of all the system components, and are exposed to the well-known state space explosion.

Communication systems can be properly modeled by communicating extended finite state machines (CEFSM). We propose a general procedure for embedded testing of CEFSM's. The reason that we choose the CEFSM model is for the clarity of presentation; our algorithm can be easily adapted to other mathematical models such as Transition Systems, Labeled Transition Systems, and Petri Nets.

Our goal is to test pre-specified parts of a system component that is embedded in a complex communication system. The pre-specified parts are determined by practical needs or by system certification requirements. For instance, for a given system component, we may want to test all the transitions or certain boundary values of system variables. We can first construct a reachability graph, which is the Cartesian product of all the system components involved, and then derive a test that covers all the pre-specified parts of the component under test. Unfortunately, this exhaustive search technique is often impractical; it is impossible to construct a reachability graph for practical systems due to the state space explosion. To avoid this problem random walks have been proposed; at any moment we only keep track of the current states of all the components and determine the next step of test at random. This approach indeed avoids the state space explosion but it may repeatedly test covered parts and take a long time to move on to the untested parts.

We propose a new technique: Hit-or-Jump. It is a generalization and unification of both the exhaustive search technique and random walks, yet it does not have the drawbacks of the two approaches. The essence of our approach is as follows. At any moment we conduct a local search from the current state in a neighborhood of the reachability graph. If an untested part is found (a Hit), we test that part and continue the process from there. Otherwise, we move randomly to the frontier of the neighborhood searched (Jump), and continue the process from there. This procedure avoids the construction of a complete system reachability graph. As a matter of fact, the space required is determined by the

user - the local search, and it is independent of the systems under consideration. On the other hand, a random walk may get "trapped" at certain part of the component under test [12]. Our algorithm is designed to "jump" out of the "trap" and pursue the exploration further.

The algorithm has been implemented and drives the ObjectGEODE tool, taking advantage of some of its functionalities, such as the construction of a searched neighborhood of the reachability graph that is used to produce the test scenarios in case of a Hit or to determine a Jump otherwise.

The Hit-or-Jump algorithm has been applied to the embedded testing of services on a telephone network. This case study is on a real system that has been specified using the SDL language. It describes telephone services in an Intelligent Network (IN) architecture. In addition to the Basic Call Services (BCSs), five other services are included: Originating Call Screening (OCS), Terminating Call Screening (TCS), Call Forward Unconditional (CFU), Call Forward on Busy Line (CBL) and Automatic Call Back (ACB).

The paper is organized as follows. Section 2 introduces the basic concepts and testability of embedded components. Section 3 describes the test generation algorithm Hit-or-Jump for embedded components. In Section 4, the software tool of Hit-or-Jump in conjunction with ObjectGEODE is presented. Section 5 reports the experimental results, and section 6 concludes this paper.

2 Basics

In this work we use extended finite state machines to model system components: the environment, the components under test and their implementations. It is only for the convenience of presentation; our technique can be adapted to other mathematical models, such as Transition Systems [14], Petri Nets [17] and Labeled Transition Systems [2].

Definition 1. An *extended finite state machine* (EFSM) is a quintuple $M = (I, O, S, \vec{x}, T)$ where I, O, S, $\vec{x}$, and T are finite sets of input symbols, output symbols, states, variables, and transitions, respectively. Each transition t in the set T is a 6-tuple:

$$t = (s_t, q_t, a_t, o_t, P_t, A_t)$$

where s_t, q_t, a_t, and o_t are the start (current) state, end (next) state, input, and output, respectively. $P_t(\vec{x})$ is a predicate on the current variable values and $A_t(\vec{x})$ defines an action on variable values.

Initially, the machine is in an initial state $s^{(0)} \in S$ with initial variable values: $\vec{x}^{(0)}$. Suppose that the machine is at state s_t with the current variable values $\vec{x}$. Upon input a_t, if $\vec{x}$ is valid for P_t, i.e., $P_t(\vec{x}) =$ TRUE, then the machine follows the transition t, outputs o_t, changes the current variable values by action $\vec{x} := A_t(\vec{x})$, and moves to state q_t.

For each state $s \in S$ and input $a \in I$, let all the transitions with start state s and input a be: $t_i = (s, q_i, a, o_i, P_i, A_i)$, $1 \leq i \leq r$. In a deterministic EFSM the sets of valid variable values of these r predicates are mutually disjoint, i.e., $X_{P_i} \cap X_{P_j} = \emptyset$, $1 \leq i \neq j \leq r$.

Otherwise, the machine is nondeterministic. In a deterministic EFSM there is at most one transition to follow at any moment, since at any state and upon each input, the associated transitions have disjoint valid variable values for their predicates and, consequently, current variable values are valid for at most one predicate. On the other hand, in a nondeterministic EFSM there may be more than one possible transition to follow. In this paper we only consider deterministic EFSM's.

There are two types of communications among the system components: (1) Synchronous communication by rendez-vous [6] without channels between; and (2) Asynchronous communication with channels between system components. Channels can be bounded or unbounded and both can be modeled by EFSM's. The interactions between the channels and system components are synchronous. On the other hand, we can use an additional variable to "encode" different system components. Therefore, we can model the whole environment, including the channels to and from the system component under test, as one EFSM [1] which communicates with the system component synchronously.

From now on we use the following notation : C is the environment EFSM, A is the specification EFSM under test, and B is the implementation of A. Machine C and $A(B)$ communicate synchronously. We represent A in the context of C by the following notation : $C \times A$.

We want to test the conformance of B to A in the context of C where C and A are known and B is a "black-box". It should be noted that $C \times A$ may not be minimized or strongly connected even if C and A are. Also they can be partially (incompletely) specified.

In general, it is not always possible to test for isomorphism of embedded components, even in the case of FSM's. Assume that A and B are FSM's. Denote machine isomorphism by $A \cong B$. Then we have:

Proposition 1. $B \cong A \quad$ implies $\quad C \times B \cong C \times A$. However, the converse is not true in general.

The first part of the proposition is trivial. We show the second part by an example.

Example 1.

[1] As a matter of fact, extended finite state machine has a same computing power as Turing machine.

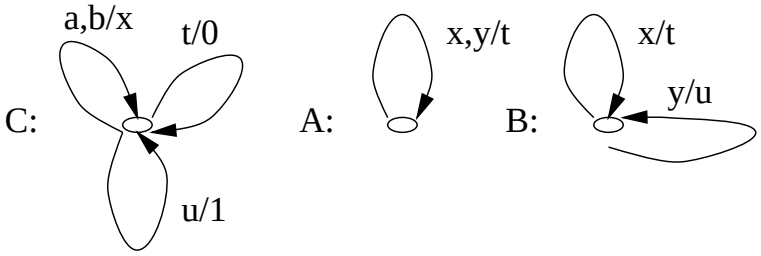

Figure 1:

In Figure 1, a and b are external inputs, 0 and 1 are external outputs, and x, y, t, u are internal input/outputs. Obviously $C \times B \cong C \times A$. But $B \ncong A$. Therefore, it is impossible to test $A \cong B$ in the context of C.

In practice, what we want is that B "behaves correctly" in the context of C. That is, $C \times B \cong C \times A$. Therefore, the problem is reduced to testing if $C \times B \cong C \times A$. However the real goal is to test the component A, assuming that the environment machine C is correctly implemented. Suppose that we test A in isolation. Then we may want to test all the transitions of A. That is, we want to obtain a testing sequence such that all the transitions of A are exercised. Similarly, in embedded testing we want to obtain test sequences (with external inputs) such that all the transitions of the component A are exercised. Specifically, we want to derive tests for $C \times A$ such that all the transitions of A are tested. We may want different coverage of A than testing all the transitions. For instance, we often want to test the boundary values of the variables. In general, we want to obtain a test sequence for $C \times A$, i.e., for testing the component A in the context of C, such that the component machine A is covered according to a pre-specified criterion. On the other hand, we do not worry about the coverage of C, since it is assumed to be correctly implemented.

Note that systems may or may not have reset. We can consider EFSM's with reset transitions as a special case by inserting a reset edge from each state to the initial state, independent of the variable values. For clarity, from now on we only consider machines without reset.

A livelock in $C \times A$ is a loop without external inputs, and it is reachable from the initial state. When the system $C \times A$ gets into a livelock, it will loop around indefinitely without any external inputs and may or may not produce external outputs. Under such circumstances, $C \times A$ can "move-on" by itself without any external control, and the external tester cannot drive the system $C \times A$ further to fulfill the testing task. This is undesirable system behavior in terms of embedded testing. Another undesirable system behavior is deadlock. During test generation, if we find a sink node (no outgoing transition) in $C \times A$, we abort the process and declare a deadlock detected. As a matter of fact, both livelock and deadlock are to be checked for system designs, and are well studied in protocol validation and verification research [8]. We shall not digress here and proceed with our discussion of testing with an assumption that the system under consideration $C \times A$ is free of livelock

and deadlock.

3 Test Generation Methods for Embedded Testing

We now present our Hit-or-Jump algorithm. We first briefly survey three commonly used and related methods and then present our procedure, which is a generalization and unification of all these three procedures.

In the discussion, we aim at covering all the transitions of the component machine under test. This is a commonly used criterion. Our technique can be easily modified to generate tests for different coverages; it is only a marking issue. We shall further elaborate on this issue when describing the algorithm.

3.1 A Structured Algorithm

From the initial state we want to generate a test sequence such that all the transitions of component machine A are covered at least once. The algorithm includes three steps: (1) Assign a distinct color to each transition of A; (2) Construct a reachability graph of $C \times A$ where each edge of $C \times A$ is marked with a color from A if it is derived from that transition of A; (3) From the initial node of $C \times A$, find a path of minimal length such all the colors are covered at least once.

We can reduce the Rural Postman problem to this covering path problem. Therefore the problem is NP-hard [5]. There are various heuristic procedures for solving this problem. However, these algorithms require the construction of the reachability graph of $C \times A$. It is often impossible in practice due to the state explosion. Consequently, unstructured algorithms such as random walks are considered, which do not require the construction of reachability graphs.

3.2 Random Walk

Starting from the initial node $(s_C^{(0)}, s_A^{(0)}, \vec{x}^{(0)})$ where $s_C^{(0)}, s_A^{(0)}$ and $\vec{x}^{(0)}$ are the initial state of C and A and initial variable values, respectively. Among all the possible outgoing edges in the reachability graph from the initial node, we select one uniformly at random, and follow that edge to the next node in the reachability graph. Suppose that after k steps we arrive at a node $(s_C^{(k)}, s_A^{(k)}, \vec{x}^{(k)})$. We examine all the outgoing edges from this node and select one uniformly at random to follow. Meanwhile, if there are colors associated with the chosen edges that have not been marked (exercised), we mark them off. We repeat the process until all the colors are marked off. During the walk, we only keep track of: (1) The current node $(s_C^{(k)}, s_A^{(k)}, \vec{x}^{(k)})$; (2) The colors that have not been marked off; and (3) The edges that have been walked through with the associated external I/O sequence, and that is the test sequence obtained from this walk. Obviously, there is no need to construct a whole reachability graph of $C \times A$.

3.3 Guided Random Walk

The procedure is the same as the random walk in Section 3.2 except for the following. When we arrived at a node $(s_C^{(k)}, s_A^{(k)}, \vec{x}^{(k)})$ among all the possible outgoing edges, we classify them: (1) With transitions of A involved, some of which are not marked; (2) With transitions of A involved and all of them are marked; and (3) Without any transitions of A involved. If the set (1) is not empty, we select one uniformly at random and follow that edge; else if (2) is not empty, we select one uniformly at random and follow that edge; and, finally, if none of the above is true, (3) must be non-empty, and we select one uniformly at random and follow that edge.

Guided random walks favor transitions of the embedded component under test, and among them give first priority to the transitions that have not been tested.

3.4 Hit-or-Jump Algorithm

The problems with random walks are: (1) To be "trapped" in a small neighborhood; (2) With a low probability to cross a "narrow bridge" to test the parts beyond the bridge; and (3) To miss the unmarked transitions of A even if them are nearby (more than one step from the current node). The Hit-or-Jump algorithm is designed to avoid these problems yet without the construction of a reachability graph. It does not require a construction of a reachability graph of $C \times A$ either, and performs better than pure random walks [12].

ALGORITHM HIT-OR-JUMP

initial condition. The environment machine C is in an initial state $s_C^{(0)}$, the component machine under test A is in an initial state $s_A^{(0)}$, and the system variables have initial values $\vec{x}^{(0)}$. [2]

termination. The algorithm terminates when all the colors (transitions) of A have been marked off.

execution.

1. Hit
   From the current node $(s_C^{(k)}, s_A^{(k)}, \vec{x}^{(k)})$ conduct a search (Depth-first or Breadth-first) in $C \times A$ until:

   (a) Reach an edge which is associated with unmarked colors (transitions) of the component machine A – a Hit. Then :

      i. Include the path from the current node to the edge (inclusive) in the test sequence under construction;

      ii. Mark off the newly exercised colors (transitions) of A;

[2] Here the initial states and variable values refer to the information known to the tester, and they are not necessarily the states and variable values after a system reset, which is often hard to obtain in practice.

iii. Arrive at a node $(s_C^{(k+1)}, s_A^{(k+1)}, \vec{x}^{(k+1)})$;

iv. Erase the searched graph;

v. Repeat from 1.

or

(b) Reach a search depth or space limit without hitting any unmarked colors (transitions) of A. Then move to 2.

2. Jump

(a) We have constructed a search tree, rooted at $(s_C^{(k)}, s_A^{(k)}, \vec{x}^{(k)})$.

(b) Examine all the leaf nodes of the tree, and select one uniformly at random.

(c) Include the path from the root to the selected leaf node in the test sequence.

(d) We arrive at the selected leaf node $(s_C^{(k+1)}, s_A^{(k+1)}, \vec{x}^{(k+1)})$ – a Jump.

(e) Repeat from 1.

3.5 Remarks on Hit-or-Jump Algorithm

For clarity we have presented one version of the Hit-or-Jump algorithm. There are various variations and generalizations. It is indeed a generalization and unification of the seemingly different algorithms: structured algorithms, random walks and guided random walks. We also comment on the space requirement, fault models and coverages.

(1) *Guided Hit-or-Jump.* For clarity we have presented a straightforward version of Hit-or-Jump algorithm. It has a number of variations and generalizations, and their implementations are simple modifications of the version presented. We briefly describe one here.

For a Jump we select uniformly at random a leaf node of the locally searched graph (tree) and proceed from there. Instead, we can enforce certain priorities in selecting the leaf nodes as in a Guided Random Walk [12] (Section 3.3). For instance, we can examine the outgoing edges from each leaf node and classify them with the priority: (A) With transitions of A involved, some of which are not marked; (B) With transitions of A involved and all of them are marked; and (C) Without any transitions of A involved. We then conduct a "Guided Jump" according to the leaf node priorities as in a Guided Random Walk.

(2) *Generalization and Unification of Structured Algorithms, Random Walks and Guided Random Walks.* Suppose that the local search depth is set to one. Then, obviously, Hit-or-Jump becomes a Random Walk. If we enforce priorities then it becomes a Guided Random Walk. On the other hand, if we do not set any bound on the local search depth then

we construct a reachability graph in the worse case; Hit-or-Jump becomes a structured algorithm. Therefore, Hit-or-Jump is a generalization of Random and Guided Random Walks and also the structured algorithm. Furthermore, this technique unifies these three seemingly quite different approaches.

(3) *Space Requirement.* Often the environment machine C (and also A) is very large. It is impossible to construct a reachability graph of C, A, or $C \times A$. Our algorithm does not need any of them. For each Hit or Jump step, we construct a local search graph on-line, which is a subgraph of $C \times A$. This can be easily done by a Depth-first search, for instance. The size of the subgraph, hence the space requirement, is determined by the users, and is independent of the machines A and C.

When constructing a search tree on-line, we can compress internal transitions of $C \times A$ [13] to further save space.

(4) *Fault Models.* Several approaches [16], [15] use fault models. We have a general procedure, independent of any fault models. We are interested in covering the component machine A, and our procedure can also be used for test generation associated with fault models.

(5) *Coverage.* We have been focused on covering all the transitions of A. The algorithm can be easily extended to: (A) Covering some (not necessarily all) transitions of A, which are specified by users or the testers; (B) Covering some states of A; and (C) Covering some transitions and states of A along with specified variable values such as boundary values. We can assign a distinct color to each entity to be covered, and run Hit-or-Jump until all the colors are covered.

(6) *Local Testing.* Local testing [15] is a search in a component machine with very restrictive assumptions, basically reducing to testing isolated machines. Hit-or-Jump conducts a local search within $C \times A$ without any assumption on A or C.

4 Implementation of Hit-or-Jump

In this section we describe the implementation of Hit-or-Jump. We develop a software tool, which also drives the ObjectGEODE simulator.

4.1 ObjectGEODE Features

The simulation in exhaustive mode was used for our implementation. We needed to take into account the advantages and limitations of the ObjectGEODE simulator. The following is a list of the main features and limitations:

1. A simulation in exhaustive mode can be stopped on a condition. A *stop condition* is a boolean expression. If during the simulation it becomes true, then the simulator

stops. In this case we can get two files: one containing the partially *deployed automaton* and another one with the *scenario* (in a file whose extension name is .b1.scn). If the simulation is actually completed, these two files are available too.

2. A stop condition may be a disjunction of other stop conditions (thus modeling a set of conditions). In case it becomes true in the middle of simulating, the *verdict* (the result) will not point out all the conditions in the disjunction that has become true.

3. A scenario is a series of signal inputs (a *sequence*) fired by the simulator from the initial state, aiming at achieving the exhaustive simulation of the specification. Each SDL state from which the input is accepted by the SDL process is given in the scenario.

4. A deployed automaton corresponds to the contents of the file associated to the simulator's variable "edges_dump" in an exhaustive mode. This automaton is logically an FSM, and contains hooks to the SDL specification (and to the scenarios), and the SDL values (local variables and signal arguments) have been instantiated, either partially (interrupted simulation) or fully (exhaustive running).

5. Each simulation in exhaustive mode produces a new deployed partial automaton; when an interrupted simulation is resumed a new partial automaton is produced, which generally has nothing to do with the previous one. Nevertheless the newly produced scenario includes the previous one: this property along the simulations will be our Ariadne's clue.

6. It is possible to make the exhaustive simulation to be a depth-first search (DFS) or a breadth-first search (BFS) in the SDL specification, and to stop it on a depth limit value. In this last situation, we do *not* get any scenario.

4.2 Implementation of Hit-or-Jump

The aim is to get a unique sequence in the fully deployed automaton, corresponding to a path starting at the initial state, that contains all the transitions of the embedded component under test yet without constructing the fully deployed automaton.

4.2.1 Interface

Our tool needs the following command-line options:

- `-sdl` SPEC.pr
  This input file is the protocol specification in SDL textual syntax. It must not be empty.

- `-stop` SPEC.stop
  This input file contains a disjunctive stop condition, modeling the set of the embedded

system component transitions. It defines the embedded system and hence must not be empty.

- `-depth-lim` l
  The given positive integer l is a depth limit that will be passed to the simulator; we stop when a search (DFS or BFS) reaches a depth of l.

- `-feed` SPEC.feed
  This input file only contains the inputs that the simulator can fire from the environment in order to stimulate the whole system. Hence no input of the embedded component appears in this file. It may be empty only if SPEC.scn is not (see below).

- `-init` SPEC.init
  This input file only contains protocol-dependent variable initializations for the simulator ("let" clauses). It may be empty.

- `-scn` SPEC.scn
  This input file contains an initial scenario only made of to-be-fired transitions for the embedded systems (in order to directly stimulate them, since they have no connection with the environment and hence have no associated feed clauses). It may be empty only if SPEC.feed is not (see above).

- `-seq` SPEC.seq
  This output file contains a test sequence for the embedded system component. The sequence is a series of pairs of inputs and outputs.

4.2.2 Configuration

The first step of our tool is to configure and produce three start-up files that will be used to drive the simulator.

1. main.startup
   It loads SPEC.feed, the initial conditions (SPEC.init), the current scenario, and specifies the exhaustive and DFS mode, together with the depth limit value (l).

2. stop_search.startup
   It is devoted to the identification of the stop condition in the disjunction (initially in SPEC.stop) that actually interrupted the simulation, and thus work around the limitation of ObjectGEODE we mentioned in 4.1, item 2.

3. final.startup
   It loads SPEC.feed and replays the final scenario we got after hitting all the colors (transitions) of the embedded system component in order to make ObjectGEODE output a SPEC.log file, from which we extract the test sequence (into SPEC.seq).

4.2.3 Simulation

We start the simulation with the main.startup file. There are two possible situations:

1. *The simulator outputs a SPEC.b1.scn file.*
   This file is output if and only if we Hit an uncovered transition of the embedded system (see section 4.1, item 1) - a Hit.

   We then run again the simulator with the stop_search.startup file in order to identify the stop condition that corresponds to the Hit transition among the current disjunction (i.e., the set of uncovered transitions). This start-up drives a dichotomous search in the following way.

   Let E be the set of the candidate stop conditions. There are only two possible cases:

   - *E is a singleton.*
     Then there is no ambiguity. We have successfully completed the Hit-or-Jump process.

     We run the simulator with the final.startup file (see section 4.2.2) and extract from the SPEC.log the test sequence. Exit.
   - *E has at least two elements.*
     We divide E into two non empty sets E_1 and E_2 with approximatively the same cardinality.

     We undo one step in the current scenario and start a simulation with E_1 in BFS mode with a depth limit of two. The idea is that we want the simulator to build all the transitions starting at the previous state where it stopped. Only two cases can occur:

     – *The simulator stops at depth 1*
       This means that it Hits again the transition of the component under test because it did not try to build the deployed automaton until depth two (and we did know that the transition we were looking for was at depth one). Thus the transition to be Hit belongs to the subset E_1. We resume the search with E_1 instead of E.

     – *The simulator stops at depth 2*
       This means that it built the deployed automaton till depth two - therefore without encountering the transition to be Hit (we did know that it was at depth one). Therefore, the transition we have been looking for does not belong to E_1. We resume the search with E_2 instead of E.

   Note: The cost of this dichotomous search is logarithmic in the number of stop conditions in the disjunction.

2. *The simulator does not output a SPEC.b1.scn file.*
   This means that the simulator stopped after reaching the depth limit l - a Jump is to be made.

In other words, it did not find any transition that satisfies one of the stop conditions in the disjunction.

We nevertheless got a file, containing the partially deployed automaton (see section 4.1, item 4), as a result of the interrupted simulation (see section 4.1, item 6), but we know neither the current state in the EFSM (SDL specification), nor the path from the initial state (see section 4.1, item 3).

Thus we parse the deployed automaton and conduct a DFS on it. We choose uniformly at random a leaf node and find a (shortest) path for the current state to the selected leaf node. We append the path at the end of the constructed scenario and resume the simulation.

5 Case Study: IN Telephone Services

In this section we report experimental results of applying the Hit-or-Jump test generation technique to Intelligent Network (IN) telephone services. The service integrates the supplementary services: Originate Call Screening (OCS), Terminal Call Screening (TCS), Call Forward Unconditional (CFU), Call Forward on Busy Line (CBL) and Automatic Call Back (ACB). The system has been described using the SDL language [11] as far as call treatment, service invocation and user management are concerned [3]. It is located at the Global Functional Plane (GFP), taking some concepts of the Distributed Functional Plane (DFP). It consists of different functional entities that are represented by the Network block. The Network block is composed by two blocks: the Basic Service, which represents the Basic Call Service (BCS) and a Features Block (FB) that represents the services. The BCS block contains three processes: the Call Manager (deals with the management of a call); the Call Handler (which takes in charge the call itself) and the Feature Handler (which allows to access to services). The FB block is composed of five processes that represent the services: Black List which is instantiated twice in order to obtain a black list on calls start, the OCS service, and a black list at calls arrival, the TCS. The other services are CFU, CBL and ACB as mentioned above. This block includes also a process: Feature Manager (which establishes a link between the Feature Handler and the services). The architecture of this specification is depicted in Figure 2.

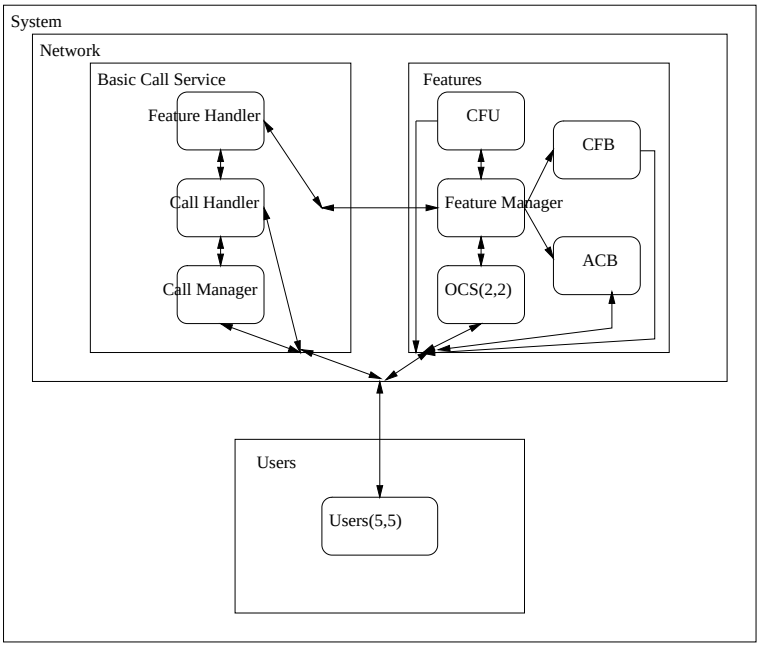

Figure 2: Global architecture

The model is described in such a way that it allows the execution of different calls in parallel and also calls initiated by the network.

The environment, i.e., the users, are also modeled as SDL process instances that composed the Users Block. The user process represents a combination of a phone line, a terminal and a user. It is relatively complete with respect to the service-usage life-cycle, with user-activations, deactivations, updates and invocations all modeled.

In order to provide a general idea of the complexity of the SDL system specifications, we present in Figure 2 the global architecture of the system and in Figure 3 some relevant metrics. The global system was simulated using exhaustive simulation in a mood to obtain the complete reachability graph. Figure 4 gives some information concerning the numbers of states, transitions, etc, obtained after a manual stop of the exhaustive search/simulation. It is impossible to construct the whole reachability graph due to the formidable state space requirement.

5.1 Embedded Testing of the OCS Module

In this paper we only report results on test generation of OCS service module. It is a system component that is embedded in the Features block and does not possesses any link with the environment. For the embedded testing of this module, we want to traverse at least once each of its branches. Stop conditions are used to represent the characteristics of each branch. To distinguish each branch of the component, we hand-crafted the stop conditions. Figure 6 illustrates the stop conditions of OCS module.

Lines	3098
Blocks	4
Process	9
Procedures	12
States	88
Signals	50
Macro definition	12
Timers	0

Figure 3: Metrics of the service specification

Number of states	674814
Number of transitions	2878800
Maximum depth reached	28
Duration	43 mn 49 s
Transition coverage rate	46.07 %
States coverage rate	70.37 %

Figure 4: Partial simulation of the complete specification

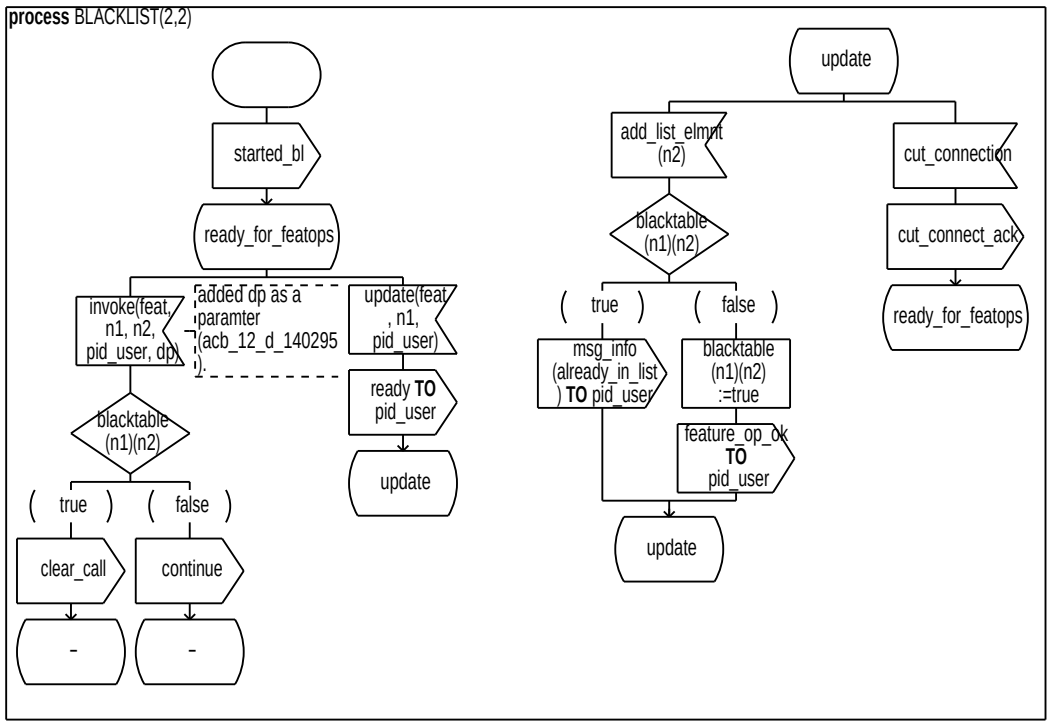

Figure 5: Blacklist process

In order to perform the simulation of the system we configure a startup that plays the role of the environment: it starts each process by firing the first transition. We initialize some variables and some services: the subscribers that invocate the services and actions each subscribers can do (eg. hangups, activations, disactivations, normal dialing). For this case study, and the results obtained, we have set this variables around 80 actions for each users.

stop if	output continue from blacklist	# 1
or	output clear_call from blacklist	# 2
or	output ready from blacklist	# 3
or	input add_list_elmnt to blacklist and output feature_op_ok from blacklist	# 4
or	trans blacklist : from_update_input_cut_connection	# 5
or	input add_list_elmnt to blacklist and output msg_info from blacklist	# 6

Figure 6: Stop conditions of the Blacklist process

The results are shown in Figure 7. Note that in the worst case, when finding the stop condition input add_list_elmnt to blacklist and output msg_info from blacklist (stop # 6), the simulator only passed through 103 transitions. It clearly shows that Hit-or-Jump algorithm effectively finds untested transitions without constructing the reachability graph. Furthermore, the total test sequence is short. Note that the time corresponds to the CPU real user time (Sun Sparc Ultra-1).

	Stop #1	Stop #2	Stop #3	Stop #4	Stop #5	Stop #6
Number of states	12	66	96	4	5	104
Number of transitions	11	65	95	3	4	103
Max depth reached	11	50	50	3	4	50
Duration (seconds)	2.5	4.4	6.7	8.6	10.4	11.1

Figure 7: Stop conditions

Once all the transitions of the embedded component OCS module have been traversed, we obtain a single test sequence, which corresponds to the total path that has been traversed from the environment to the last transition of the module. The obtained sequence is of length 150; we only need to take 150 transitions to cover the whole OCS module in the context.

The segment of the sequence in Figure 8 exhibits the invocation of services.

```
(50,"dial_tone/dialed[2]",51)
(51,"dialed[2]/net_trigger[addressed,1,2]",52)
(52,"net_trigger[addressed,1,2]/invoke[ocs,1,2,user(1),addressed]",53)
(53,"invoke[ocs,1,2,user(1),addressed]/invoke[ocs,1,2,user(1),addressed]",54)
(54,"invoke[ocs,1,2,user(1),addressed]/continue",55)
(55,"continue/continue",56)
```

Figure 8: Invocation of services

Figure 9 shows a series of transitions, which allow us to reach the transition of the embedded module in the case of the stop output ready (stop #3).

```
(137,"line_free/user_ack",138)
(138,"user_ack/cm_ack",139)
(139,"cm_ack/handler_exit[call_handler(4)]",140)
(140,"handler_exit[call_handler(4)]/schedule_user[true]",141)
(141,"schedule_user[true]/offhook",142)
(142,"offhook/dial_tone",143)
(143,"dial_tone/feature_op[ocs,updat]",144)
(144,"feature_op[ocs,updat]/user_trigger[ocs,updat,1,1]",145)
(145,"user_trigger[ocs,updat,1,1]/update[ocs,1,user(1)]",146)
(146,"update[ocs,1,user(1)]/update[ocs,1,user(1)]",147)
(147,"update[ocs,1,user(1)]/ready",148)
```

Figure 9: Finding the stop condition ready

The complete test sequence of 150 transitions that cover the whole embedded OCS service module is in Appendix.

We have exercised a Random Walk (see section 3.2) and got a test sequence of 1402 transitions. It is clear that Hit-or-Jump produces a test sequence with a same fault coverage as a Random Walk but is an order of magnitude shorter.

We have also performed experiments on the embedded testing of the service CFU. Moreover we have also applied the Hit-or-Jump algorithm to the process Responder of the INRES protocol[7]. We have also obtained various test sequence lengths with different mode of search, BFS (Breath-first-search), DFS (Depth-first-search), Merge (a DFS and then a BFS), and a Random Walk. The following figures contain the results.

Module	OCS			
Mode	DFS	BFS	Merge	RWalk
Depth	50	50	50	
Stops	6	6	6	6
Sequence	834	150	167	1402
Jumps	18	1	0	

Figure 10: Module OCS

Module	CFU			
Mode	DFS	BFS	Merge	RWalk
Depth	100	100	100	
Stops	6	6	6	6
Sequence	deadlocks	137	261	586
Jumps	< 70	0	0	

Figure 11: Module CFU

Module	INRES			
Mode	DFS	BFS	Merge	RWalk
Depth	100	100	100	
Stops	4	4	4	4
Sequence	368	36	101	6856
Jumps	4	0	0	

Figure 12: Module Responder

Module	INRES			
Mode	DFS	BFS	Merge	RWalk
Depth	100	100	100	
Stops	5	5	5	5
Sequence	stopped	44	186	stopped
Jumps		0	0	

Figure 13: Module Responder

6 Conclusion

We have presented a new algorithm to perform testing of components that are embedded in a complex communication system. It is a natural generalization and also a unification of random walk and guided random walk algorithms and structured search algorithms. Yet it does not have the state space explosion problem as is encountered by the structured algorithms, and it generates high coverage test sequences that are much shorter than that from random walks.

For convenience, we present the algorithm using extended finite state machine model. The algorithm can be adapted to other mathematical models such as transition systems and labeled transition systems. For a similar reason, we conducted experiments on IN with SDL [10] specification because of its availability. Due to the simplicity and generality of the algorithm, we believe that it can also be adapted to embedded testing of systems specified by other languages such as LOTOS [1] and ESTELLE [9]).

The algorithm has been implemented and drives the ObjectGEODE tool. It has been applied to embedded testing of services of Intelligent Networks (IN). The experimental results are promising. It avoids the construction of a complete reachability graph, which is impossible for IN; it conducts a local search only with a space requirement independent of the systems under test. It effectively covers the whole embedded components of the IN services under test with a rather short test sequence of only 150 transitions, which is an order of magnitude shorter than that from random walks.

We have presented a basic version of the Hit-or-Jump algorithm, and have described briefly a generalization - Guided Hit-or-Jump. Other variations or generalizations can also be explored. For instance, if there has been no Hit for a large number of Jumps, one might "backtrack" to the previous Hit, and Jump to a different node to proceed with testing. Even though in our experiments with IN we have not encountered such problem, it might not be a surprise for testing components that are embedded in a complex system.

We have not specified the depth of local search for a Jump in case there is no Hit. For IN we tested on a few depth values, i.e., 50 and 100. Intuitively, a larger depth value increases the probability of hitting an uncovered part of the component under test. However, it requires more space and time for each step. Furthermore, a long "Jump" implies a longer subsequence in the test for this step. We believe that it depends on the system under test to choose a good depth value. As indicated earlier, one can always choose a depth value that is within the limit of affordable memory space.

We have tested both Breadth-first-search and Depth-first-search for the local search for a Hit or Jump. Breadth-first-search seems to perform better; it is "unbiased" and makes an "equi-distance" random Jump.

References

[1] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. In *Computer Networks and ISDN Systems*, volume 14(1), 1987.

[2] E. Brinksma. A theory for the derivation of tests. In *Proc. IFIP WG6.1 8th Int. Symp. on Protocol Specification, Testing and Verification*. Horth-Holland, 1988.

[3] P. Combes and B. Renard. Service validation, tutorial. In *SDL Forum'97*, France, 1997.

[4] M. A. Fecko, U. Uyar, A. S. Sethi, and P. Amer. Issues in conformance testing: Multiple semicontrollable interfaces. In *Proceedings of FORTE'97*, Paris, France, November 1998.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, New York, 1985.

[7] D. Hogrefe. Osi formal specification case study: the inres protocol and service, revised. Technical report, Institut für Informatik Universität Bern, may 1992.

[8] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey, 1991.

[9] International Standards Organization. *ISO/IEC 9074(E), Estelle: a Formal Description Technique based on a finite state machine transition model*, 1997.

[10] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework, International Standard IS-9646.*, 1991.

[11] ITU. *Recommendation Z.100 : CCITT Specification and Description Language (SDL)*, 1992.

[12] D. Lee, K. Sabnani, D. Kristol, and S. Paul. Conformance testing of protocols specified as communicating finite state machines - a guided random walk based approach. In *IEEE Transactions on Communications*, volume 44, No.5, May 1996.

[13] L. P. Lima and A. Cavalli. A pragmatic approach to generating test sequences for embedded systems. In *Proceedings of IWTCS'97*, Cheju Island, Korea, September 1997.

[14] R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[15] A. Petrenko, N. Yevtushenko, and G. V. Bochmann. Fault models for testing in context. In *Proceeding of FORTE*, Kaiserslatern, Germany, October 1996.

[16] A. Petrenko, N. Yevtushenko, and G. V. Bochmann. Testing faults in embedded components. In *Proceedings of IWTCS'97*, Cheju Island, Korea, September 1997.

[17] A. A. Petri. *Kommunikation mit Automaten*. Ph. D. thesis, Universitat Bonn, 1962.

[18] N. Yevtushenko, A. Cavalli, and L. P. Lima. Test suite minimization for testing in context. In *IWTCS'98*, Tomsk, Russia, August 1998.

[19] J. Zhu and S. T. Vuong. Evaluation of test coverage for embedded system testing. In *IWTCS'98*, Tomsk, Russia, August 1998.

Appendix

A complete test sequence of 150 transitions that cover the whole embedded OCS service module in IN:

```
des(0,150,151)
(0,"NULL/started_acb",1)
(1,"NULL/started_cfb",2)
(2,"NULL/started_cfu",3)
(3,"NULL/started_bl",4)
(4,"NULL/started_bl",5)
(5,"started_acb/NULL",6)
(6,"started_cfb/NULL",7)
(7,"started_cfu/NULL",8)
(8,"started_bl/NULL",9)
(9,"started_bl/user_started",10)
(10,"NULL/user_started",11)
(11,"NULL/user_started",12)
(12,"NULL/user_started",13)
(13,"NULL/user_started",14)
(14,"user_started/user_started_ack",15)
(15,"user_started/user_started_ack",16)
(16,"user_started/user_started_ack",17)
(17,"user_started/user_started_ack",18)
(18,"user_started/user_started_ack",19)
(19,"user_started_ack/NULL",20)
(20,"user_started_ack/NULL",21)
(21,"user_started_ack/NULL",22)
(22,"user_started_ack/NULL",23)
(23,"user_started_ack/schedule_user[true]",24)
(24,"schedule_user[true]/offhook",25)
(25,"offhook/dial_tone",26)
(26,"dial_tone/feature_op[ocs,updat]",27)
(27,"feature_op[ocs,updat]/user_trigger[ocs,updat,1,1]",28)
(28,"user_trigger[ocs,updat,1,1]/update[ocs,1,user(1)]",29)
(29,"update[ocs,1,user(1)]/update[ocs,1,user(1)]",30)
(30,"update[ocs,1,user(1)]/ready",31)
```

```
(31,"ready/add_list_elmnt[1]",32)
(32,"add_list_elmnt[1]/feature_op_ok",33)
(33,"feature_op_ok/onhook",34)
(34,"onhook/cut_connection",35)
(35,"cut_connection/cut_connect[ocs]",36)
(36,"cut_connect[ocs]/cut_connection",37)
(37,"cut_connection/cut_connect_ack",38)
(38,"cut_connect_ack/cut_ack",39)
(39,"cut_ack/cut_connect_ack",40)
(40,"cut_connect_ack/net_trigger[o_end_call,1,1]",41)
(41,"net_trigger[o_end_call,1,1]/resume_call[o_end_call,disconnecting,1,1]",42)
(42,"resume_call[o_end_call,disconnecting,1,1]/end_leg[1,1]",43)
(43,"end_leg[1,1]/line_free",44)
(44,"line_free/user_ack",45)
(45,"user_ack/cm_ack",46)
(46,"cm_ack/handler_exit[call_handler(1)]",47)
(47,"handler_exit[call_handler(1)]/schedule_user[true]",48)
(48,"schedule_user[true]/offhook",49)
(49,"offhook/dial_tone",50)
(50,"dial_tone/dialled[2]",51)
(51,"dialled[2]/net_trigger[addressed,1,2]",52)
(52,"net_trigger[addressed,1,2]/invoke[ocs,1,2,user(1),addressed]",53)
(53,"invoke[ocs,1,2,user(1),addressed]/invoke[ocs,1,2,user(1),addressed]",54)
(54,"invoke[ocs,1,2,user(1),addressed]/continue",55)
(55,"continue/continue",56)
(56,"continue/resume_call[addressed,connecting,1,2]",57)
(57,"resume_call[addressed,connecting,1,2]/net_trigger[analysed,1,2]",58)
(58,"net_trigger[analysed,1,2]/invoke[cfu,1,2,user(1),analysed]",59)
(59,"invoke[cfu,1,2,user(1),analysed]/invoke[cfu,1,2,user(1),analysed]",60)
(60,"invoke[cfu,1,2,user(1),analysed]/proceed[3]",61)
(61,"proceed[3]/proceed[3]",62)
(62,"proceed[3]/resume_call[analysed,completing,1,3]",63)
(63,"resume_call[analysed,completing,1,3]/phonestat_req[1,3]",64)
(64,"phonestat_req[1,3]/called_isfree[user(3)]",65)
(65,"called_isfree[user(3)]/ringback_tone",66)
(66,"ringback_tone/user_ack",67)
(67,"user_ack/ring_start",68)
(68,"ring_start/user_ack",69)
(69,"user_ack/net_trigger[completion,1,3]",70)
(70,"net_trigger[completion,1,3]/resume_call[completion,ringing,1,3]",71)
(71,"resume_call[completion,ringing,1,3]/action[false]",72)
(72,"action[false]/user_ack",73)
(73,"user_ack/action[false]",74)
```

```
(74,"action[false]/offhook",75)
(75,"offhook/call_completed",76)
(76,"call_completed/conv_caller",77)
(77,"conv_caller/call_completed",78)
(78,"call_completed/conv_called",79)
(79,"conv_called/net_trigger[acceptance,1,3]",80)
(80,"net_trigger[acceptance,1,3]/resume_call[acceptance,call_in_progress,1,3]",81)
(81,"resume_call[acceptance,call_in_progress,1,3]/call_switch",82)
(82,"call_switch/schedule_user[true]",83)
(83,"schedule_user[true]/offhook",84)
(84,"offhook/dial_tone",85)
(85,"dial_tone/dialled[1]",86)
(86,"dialled[1]/net_trigger[addressed,5,1]",87)
(87,"net_trigger[addressed,5,1]/resume_call[addressed,connecting,5,1]",88)
(88,"resume_call[addressed,connecting,5,1]/net_trigger[analysed,5,1]",89)
(89,"net_trigger[analysed,5,1]/resume_call[analysed,completing,5,1]",90)
(90,"resume_call[analysed,completing,5,1]/phonestat_req[5,1]",91)
(91,"phonestat_req[5,1]/called_isbusy[user(1)]",92)
(92,"called_isbusy[user(1)]/net_trigger[occupation,5,1]",93)
(93,"net_trigger[occupation,5,1]/resume_call[occupation,disconnecting,5,1]",94)
(94,"resume_call[occupation,disconnecting,5,1]/busy_tone",95)
(95,"busy_tone/onhook",96)
(96,"onhook/net_trigger[o_end_call,5,1]",97)
(97,"net_trigger[o_end_call,5,1]/resume_call[o_end_call,disconnecting,5,1]",98)
(98,"resume_call[o_end_call,disconnecting,5,1]/end_leg[5,1]",99)
(99,"end_leg[5,1]/line_free",100)
(100,"line_free/user_ack",101)
(101,"user_ack/cm_ack",102)
(102,"cm_ack/handler_exit[call_handler(3)]",103)
(103,"handler_exit[call_handler(3)]/schedule_ch",104)
(104,"schedule_ch/action[true]",105)
(105,"action[true]/onhook",106)
(106,"onhook/net_trigger[t_end_call,1,3]",107)
(107,"net_trigger[t_end_call,1,3]/resume_call[t_end_call,disconnecting,1,3]",108)
(108,"resume_call[t_end_call,disconnecting,1,3]/busy_tone",109)
(109,"busy_tone/onhook",110)
(110,"onhook/net_trigger[o_end_call,1,3]",111)
(111,"net_trigger[o_end_call,1,3]/resume_call[o_end_call,disconnecting,1,3]",112)
(112,"resume_call[o_end_call,disconnecting,1,3]/end_leg[1,1]",113)
(113,"end_leg[1,1]/line_free",114)
(114,"line_free/user_ack",115)
(115,"user_ack/cm_ack",116)
(116,"cm_ack/end_leg[3,1]",117)
```

```
(117,"end_leg[3,1]/line_free",118)
(118,"line_free/user_ack",119)
(119,"user_ack/cm_ack",120)
(120,"cm_ack/handler_exit[call_handler(2)]",121)
(121,"handler_exit[call_handler(2)]/schedule_user[true]",122)
(122,"schedule_user[true]/offhook",123)
(123,"offhook/dial_tone",124)
(124,"dial_tone/dialled[3]",125)
(125,"dialled[3]/net_trigger[addressed,1,3]",126)
(126,"net_trigger[addressed,1,3]/invoke[ocs,1,3,user(1),addressed]",127)
(127,"invoke[ocs,1,3,user(1),addressed]/invoke[ocs,1,3,user(1),addressed]",128)
(128,"invoke[ocs,1,3,user(1),addressed]/clear_call",129)
(129,"clear_call/clear_call",130)
(130,"clear_call/resume_call[addressed,disconnecting,1,3]",131)
(131,"resume_call[addressed,disconnecting,1,3]/busy_tone",132)
(132,"busy_tone/onhook",133)
(133,"onhook/net_trigger[o_end_call,1,3]",134)
(134,"net_trigger[o_end_call,1,3]/resume_call[o_end_call,disconnecting,1,3]",135)
(135,"resume_call[o_end_call,disconnecting,1,3]/end_leg[1,1]",136)
(136,"end_leg[1,1]/line_free",137)
(137,"line_free/user_ack",138)
(138,"user_ack/cm_ack",139)
(139,"cm_ack/handler_exit[call_handler(4)]",140)
(140,"handler_exit[call_handler(4)]/schedule_user[true]",141)
(141,"schedule_user[true]/offhook",142)
(142,"offhook/dial_tone",143)
(143,"dial_tone/feature_op[ocs,updat]",144)
(144,"feature_op[ocs,updat]/user_trigger[ocs,updat,1,1]",145)
(145,"user_trigger[ocs,updat,1,1]/update[ocs,1,user(1)]",146)
(146,"update[ocs,1,user(1)]/update[ocs,1,user(1)]",147)
(147,"update[ocs,1,user(1)]/ready",148)
(148,"ready/add_list_elmnt[1]",149)
(149,"add_list_elmnt[1]/msg_info[already_in_list]",150)
```