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
{Ana.Cavalli, Christian.Rinderknecht, Fatiha.Zaidi}@int-evry.fr
° *Bell Laboratories, Lucent Technologies*
600 Mountain Avenue
Murray Hill, NJ 07974, USA
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 Intelligent Network (IN) architecture, including the Basic Call Service and five supplementary services.

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" [9].

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 [11, 13, 18, 1], on fault coverage [19], on the test of systems with semicontrollable and uncontrollable interfaces [5], or on-the-fly [6]. 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.

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 [11]. Our algorithm is designed to "jump" out of the "trap" and pursue the exploration further.

The Hit-or-Jump algorithm has been applied to the embedded testing of services on a telephone network. With the aid of ObjectGEODE tool, 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. Section 4 discusses the implementations. Section 5 reports the experimental results on IN. Section 6 concludes the paper with remarks on the generalization and variations of the algorithm and on related issues.

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., $P_t(\vec{x}) = \text{TRUE}$, 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.

We model the environment and component under test by EFSM's. During an execution the states and variable values can be determined as in the construction of reachability graphs for EFSM's [8, 12]. 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. $B \cong A$ implies $C \times B \cong C \times A$. 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.

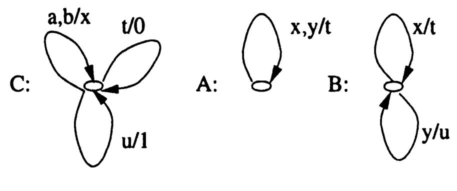

Figure 1.1

In Figure 1.1, 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 $B \not\cong A$. 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. This criterion can be specified by assigning a distinct color to each entity (transition or variable value, for instance) to be tested and by covering all the assigned colors by test sequences generated. On the other hand, we do not worry about the coverage of C, since it is assumed to be correctly implemented. For clarity, we assume that the system under test does not have deadlock nor livelock, which are well studied topics in validation [8].

3. EMBEDDED TESTING

We first briefly survey three commonly used and related methods and then present our Hit-or-Jump algorithm, which is a generalization and unification of all these three methods. For clarity, we describe a procedure that covers all the transitions of the component machine under test. This is a commonly used criterion in practice. As indicated earlier, other coverage criteria can be reduced to color covering of the component under test, and our procedure can be easily adapted to generating tests for the color covering; it is only a marking process. 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 that all the colors are covered at least once. This approach requires a construction of the reachability graph of $C \times A$, which leads to the well-known state explosion. Even if we could obtain such a graph the problem is still NP-hard. 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 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 [11] 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 they 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. Local search is used in the procedure, and it can be Depth-first or Breadth-first search.

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 $\vec{x}^{(0)}$.

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 in $C \times A$ until:
   (a) Reach an edge which is associated with unmarked colors 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 of A;
      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 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.

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 apparently quite different approaches.

4. IMPLEMENTATIONS OF HIT-OR-JUMP

In this section we describe the implementation of Hit-or-Jump. It was not possible to use the ObjectGEODE tool (Verilog) alone because our algorithm needed special handling of features of the tool that are not widely used (and hence poorly documented). For our purpose, we developed a software tool which drives the ObjectGEODE simulator. Due to space limit we do not include the details here, and the interested readers are referred to [3].

Our goal is to get a test 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.

Interface. We supply our tool with the following informations. The result is a file containing the test sequence for the embedded component. The sequence is a series of pairs of inputs and outputs. (1) A disjunctive stop condition modeling the set of the embedded system component transitions. It defines the embedded system. (2) A positive integer denoting a depth limit that will be passed along to the simulator; we stop when a search (DFS, BFS or B-DFS) reaches it. (3) The inputs that the simulator can fire from the environment in order to stimulate the whole system (here opposed to the *embedded* system). (4) The protocol-dependent variable for the simulator ("let" clauses, like number of users, actions allowed, maximum number of actions per user, service subscriptions etc.). (5) An initial scenario that starts all the processes and put then in their initial state.

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 the inputs that the simulator can fire from the environment, the protocol-dependent variables, the current scenario, and specifies the exhaustive and B-DFS mode. Then it runs the simulator. (2) stop_search.startup. It is devoted to the identification of the stop condition in the disjunction that actually interrupted the simulation. (3)

final.startup. It loads the inputs that the simulator can fire from the environment and replays the final scenario we got after hitting all the colors (transitions) of the embedded component. As a result we get an ObjectGEODE log file from which we extract the test sequence.

Simulation. We start the simulation with the main.startup file. There are two possible situations: (1) *The simulator outputs a scenario.* This file is output if and only if we Hit an uncovered transition of the embedded system. Two cases can occur: (A) *There was only one stop conditions remaining.* Then we have successfully completed the algorithm. We run the simulator with the final.startup and extract from the log file the test sequence. (B) *There were at least two stop conditions remaining.* Then we 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 prints the stop condition statuses and identifies the one assessed to true. (2) *The simulator does not output a scenario.* This means that the simulator stopped after reaching the depth limit- 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, as a result of the interrupted simulation, but we know neither the current state in the EFSM (SDL specification), nor the path from the initial state. 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 [10] as far as call treatment, service invocation and user management are concerned [4]. 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 1.2.

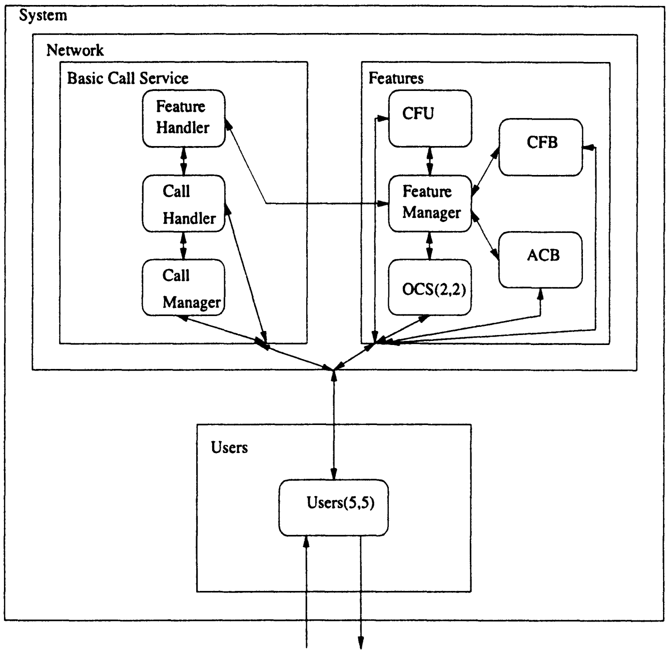

Figure 1.2 Global architecture

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 sends messages to the process Users, that are 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 1.2 the global architecture of the system and in Figure 1.3 some relevant metrics. The global system was simulated using exhaustive simulation in a mood to obtain the complete reachability graph. Figure 1.4 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.

Lines	3,098
Blocks	4
Process	9
Procedures	12
States	88
Signals	50
Macro definition	12
Timers	0

Figure 1.3 Metrics of the service specification

#states	674,814
#transitions	2,878,800
Max depth reached	28
Duration	43mn 49s
Transition coverage	46.07%
States coverage	70.37%

Figure 1.4 Partial simulation of the complete specification

We now report detailed results on test generation of OCS and CFU services modules. 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, which is depicted in figure 1.5. 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 1.6 illustrates the stop conditions of OCS module.

In order to perform the simulation of the system we configure a startup, that 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 these variables around 80 actions for each users.

The results are shown in Figure 1.7. The line "Number of transitions" indicates the number of fired transitions at each simulation (i.e., the size of the deployed automaton). Hence it is aimed to be a measure of resource consumption, not of the size of the corresponding test

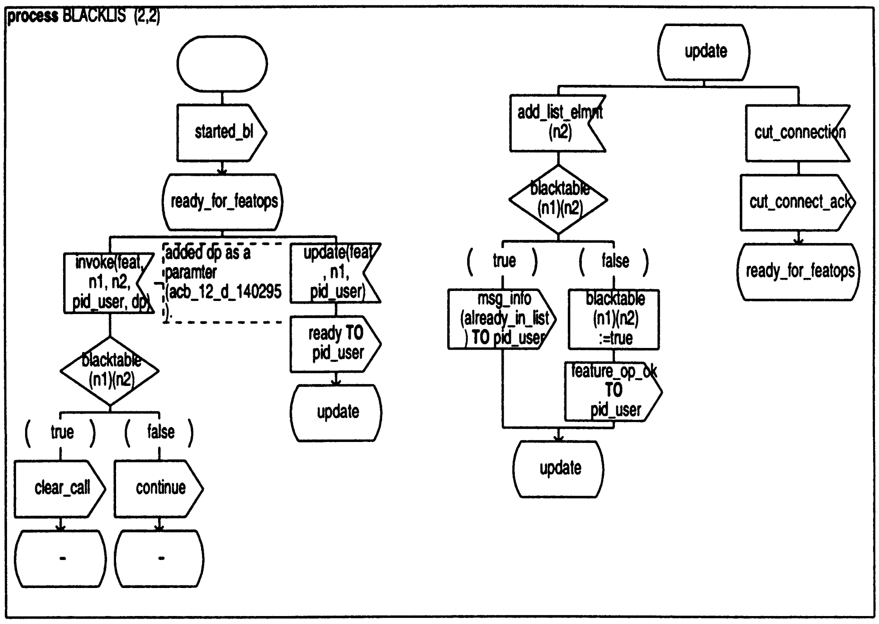

Figure 1.5 Blacklist process

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

Figure 1.6 Stop conditions of the Blacklist process

subsequence, which is a path in the deployed automaton. 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).

Stops	#1	#2	#3	#4	#5	#6
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 1.7 Simulation results for each stop condition

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.

We have exercised a Random Walk (see section 3.2) and got a test sequence of 1.402 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. Figure 1.8 illustrates the results obtained for OCS and CFU services. Moreover we have also applied the Hit-or-Jump algorithm to the process Responder of the INRES protocol [7]. The results for the module Responder of the INRES protocol are relevant, in fact we obtained a sequence of length 44 in a BFS mode. We have also obtained various test sequence lengths with hit-or-jump algorithm in different modes of search and a Random Walk (RW).

Modules	OCS				CFU			
Modes	DFS	BFS	B-DFS	RW	DFS	BFS	B-DFS	RW
Depths	50	50	50	-	100	100	100	-
#Stops	6	6	6	6	6	6	6	6
Sequences	834	150	167	1402	deadlock	137	261	586
#Jumps	18	1	0	-	< 70	0	0	-

Figure 1.8 Results of the Modules OCS and CFU

6. CONCLUSION

We have presented a new algorithm Hit-or-Jump to perform testing of components that are embedded in a complex communication system. It is a generalization and unification of Random and Guided Random Walks and also the structured algorithm. 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.

Hit-or-Jump is a new technique for system state search. We have applied it for embedded testing. It can also be used for verification and validation, which depend on system state search, and our method could help dealing with the state explosion problem there.

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 a few 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 [11], and then conduct a "Guided Jump" according to the leaf node priorities as in a Guided Random Walk. Another variation is: 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. The idea behind is: get back when one has gone "astray". 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. Also when constructing a search tree on-line, we can compress internal transitions of $C \times A$ [13] to further save space.

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. Several approaches [16], [15] use fault models. We have a general procedure, independent of any fault models. However, we can assign colors to the entities of the component machine under test for the coverage from fault models, and our procedure can be used for test generation associated with fault models.

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

References

[1] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. A guided incremental test case generation procedure for conformance testing for CEFSM specified protocols. In *IWTCS'98*, Tomsk, Russia, August 1998.

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

[3] A. Cavalli, D. Lee, C. Rinderknecht, and Fatiha Zaidi. Hit-or-jump: An algorithm for embedded testing with applications to in services. In *Tech. Memo, Bell Laboratories*, May 1999.

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

[5] M. A. Fecko, U. Uyar, A. S. Sethi, and P. Amer. Issues in conformance testing: Multiple semicontrollable interfaces. In *Proceedings of FORTE/PSTV'98*, Paris, France, November 1998.

[6] J.-C. Fernandez, C. Jard, T. Jeron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV, LINCS 1102*, USA, July 1996.

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

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

[11] 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.

[12] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *Proc. of the IEEE*, 84(8):1090–1123, August 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/PSTV'96*, Kaisersläutern, 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.