RAPPORT DE RECHERCHE

L R I

# USE OF VERIFICATION TECHNIQUES FOR COMPONENTS TESTING

ZAIDI F / LALLALI M

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

01/2007

**Rapport de Recherche N° 1465**

**CNRS – Université de Paris Sud**
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

# Use of Verification Techniques for Components Testing

Fatiha Zaidi+, Mounir Lallali++
+Université Paris-Sud XI - CNRS UMR 8623
laboratoire de Recherche en Informatique, LRI, Bat 490
91405 Orsay Cedex
{Fatiha.Zaidi@lri.fr}
++Institut National des Télécommunications-CNRS SAMOVAR
9 rue Charles Fourrier
91011 Evry Cedex
{Mounir.Lallali@int-evry.fr}

## Abstract

*In this paper, we report about the use of the model checker SPIN to generate test cases. We have adapted the SPIN tool for automatically generating test cases for an embedded component. The generation of the latter is based on the counter-example generation mechanism of SPIN. For that purpose, we have implemented inside SPIN our test generation algorithm that allows to generate a test case for each test objective to be covered. The strategy of our algorithm permits to manage the state space explosion in combination with the efficient existing algorithms of SPIN to handle large state spaces. We define in this paper how we characterize the behavior of the embedded component. Given a* PROMELA *model of the complete specification, i.e with the component, our method helps to generate efficiently tests from a test purpose written in Propositional Linear Temporal Logic Formulas. Finally, we exercise our prototype on a case study and obtain first promising results.*
**Keywords***: Testing, Verification, Model checking, Formal Method, PLTL, SPIN.*

## 1 Introduction

Conformance testing, that ensures correct protocol implementations, has become more and more essential for the development of reliable communicating systems. During these last decade, numerous testing generation methods have been proposed. Due to the complexity of the test of the systems, the designers use modularity to build their systems and opt for a "divide to conquer" strategy. Thus, to test a system as a whole becomes difficult due to the systems huge size. In this case, testers have to face the state space explosion problem. Moreover, to test a system in isolation is not always feasible due to the interactions among the system besides of not being realistic. Component-based testing has become a main research area in the testing realm. The goal of embedded testing is to check the conformance of the implementation of a component regarding its specification in the context of the other components. Different approaches have been published in the literature [2, 12] and they faced the combinatorial explosion problem. Among them, we can refer structured algorithms and non-structured one that are based on a random exploration of the model to generate tests, especially those that use test objectives as a guide during the state space exploration, *e.g.* the Hit-or-Jump algorithm [5]. This latter gave good results [1, 4] and have been applied on real protocols such as TCP and WAP [1,6]. Nevertheless, the implementation of such algorithms can be improved by taking advantage of existing techniques of model checking to reduce the state space. Model checking and testing are two complementary techniques.

The former works on a model of the implementation and checks properties of a system expressed by a temporal logic by exercising a model of it. The latter requires, as the verification techniques, the construction a system model. Nevertheless, the aim of testing is to exercise the real system with the test generated from the formal model. In this way, both techniques need the effort of the model construction, being the verified model, *i.e.* deadlocks and livelocks free model, used as input to generate tests. Recently, the research in this area produced papers that try to use model checker tool [3,7,9],*i.e.* especially the input model for these tools, to deviate it from its original goal in order to generate tests.

In this paper, we try to propose such an approach. Thus, the contributions of this paper is two folds: *(i)* we propose

an efficient method to generate component-based test cases and *(ii)* we adapt to an efficient model checker SPIN [10] an efficient generation testing algorithm. This tool can be used for validation and verification by automatically checking a property and by generating a counter-example in case of the violation of this property. It can also efficiently handle large state spaces as it employs various methods to reduce the complexity of search. The input language of the tool is the PROMELA language. Hence, our testing generation method will use this input model to generate our test cases, as we will consider this model as our specification of the expected behavior of the system. We propose to implement our embedded testing algorithm inside SPIN model checker.

The rest of the paper is organized as follows. In the Section 2, we present our testing generation algorithm and the SPIN model checking algorithms. In Section 3, we describe our proposed approach and we explain how we adapt SPIN to implement our algorithm. Moreover, We describe also how we formulate the test objectives with PLTL formulas. In Section 4 the case study and how we formulate our test objectives for this example are presented and we discuss about the obtained results. Finally, in Section 5 we give our conclusion and guidelines for future work.

## 2 Components Tests Generation and SPIN model checker algorithms

In this section, we propose an embedded test generation method , the Hit-or-Jump algorithm [5] which will be used inside the model checker SPIN [10]. We will propose in the next section a new algorithm to adapt the acceptance cycle detection algorithm of SPIN (see subsection 2.2) [8] by using the hit-or-jump method. Our new method, based on this two algorithms, uses test objectives to describe finite behavior and accept state detection algorithm. The integration of our algorithm in the SPIN tool permits to obtain a guided random walk to generate test sequences of an embedded component.

### 2.1 Outline of the Hit-or-Jump algorithm

The algorithm presented here allows to cover all the interactions of the component in its context. 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 accessibility graph. If an untested part of the component is found (a Hit), we keep it for the final test sequence, and then continue the search process from there. Otherwise, we move randomly to the frontier of the neighborhood searched (Jump), and resume the process from there. This procedure avoids the building of a whole system accessibility graph. Accordingly, the space required is determined by the user, *e.g.* a depth limit or a maximum number of states, and it is independent of the system 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. We build at each step a partial accessibility graph to avoid the state-number explosion problem mentioned before. The algorithm finally produces a test sequence as a transition tour of the component in its context.

**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 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) or (b) occurs:*
   (a) *Reach an edge which is associated with unmarked 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 transitions 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.*
   (b) *Reach a search depth or space limit without hitting any unmarked transition 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.*

This algorithm avoids looping on the same state, as far as we choose uniformly and randomly in the spanning tree. Notice that Hit-or-Jump assumes that the specification is correct in the sense that it does not imply run-time deadlocks, neither non accessible states, etc.

### 2.2 Optimized Nested Depth First Search for Acceptance Cycles (NDFS)

SPIN's verification procedure is based on an optimized depth first graph traversal method. The principle of the nested depth-first search algorithm is as follows. For an existing accepting cycle in the reachability graph at least one accepting state must be both reachable from the initial system state and from itself. The first depth first search establishes which accepting states are reachable from the initial system state. The second (nested) search starts at each accepting state previously detected, and checks whether or not that state is reachable from itself. If so, a complete execution sequence that includes the acceptance cycle is also constructed: It is the concatenation of all the steps that are both on the first and on the second depth first search stack. We note that the second search always explores the same states that are found in the first search and stops when reaching a state of the first search stack.

```
proc dfs(s) /* The first DFS*/
    if error(s) then report error fi
    add {s,0} to Statespace
    add s to Stack
    for each (selected) successor t of s do
        if {t,0} not in Statespace then dfs(t) fi
    od
    if accepting(s) then ndfs(s) fi
    remove s from Stack
end
proc ndfs(s) /* The second DFS*/
    add {s,1} to Statespace
    for each (selected) successor t of s do
        if {t,1} not in Statespace then ndfs(t)
        else if t is in Stack then report cycle fi
    od
end
```

## 3 Description of our method

This method uses the structure of SPIN based on temporal properties checking. We introduce the concept of the test objectives (*i.e.* the transitions of the component to be covered) for the needs of the components test generation. In this method, the test objectives describe a finite execution. The intersection between the language accepted by the system automaton (i.e all its executions) and the language accepted by the test objectives automaton is a finite execution (i.e finite transitions sequence). The *never claims* are used to detect behaviors that are considered undesirable or illegal (in our case the negation of the test objectives). The statements in a *never claim* are used to follow the executions of the system. The following is achieved by computing a synchronous product of the sequence specified in the temporal claim, with the interleaving sequences specified in the system. This synchronous product allows to build a new automaton, in which every state is defined as a pair *(s,n)* with *s* being a state from the global system, and *n* a being state from the temporal claim. Every transition in the new automaton is defined by a pair of transitions *(a,p)*, with the first element (i.e *a*) being a statement of the system, and the second (i.e *p*) being a proposition of the claim. In other words, every transition in this final synchronous product is defined as a joint transition of the system and the claim. Such a transition can only occur if the proposition *p* is valid in the originating state *s* of the transition. Instead of seeking an acceptance cycle as in the NDFS algorithm, we search just a path going from the initial state of the product automaton to a *hit* state *(s,n)* (*i.e.* a state that corresponds to an uncovered component transition) with *n* being an *accept* state of the automaton temporal claim. This path must contain all the *terminating* states of the transitions (*hit* states) that appear in the test objectives (described by the claim). We note that an *accept* state is a claim violation state. For our purpose as mentioned previously, we use only the first DFS of the NDFS algorithm by adapting the Hit-or-Jump algorithm for jumping from a current state (initialized at the beginning to the initial state and then to the reached *hit* states) if search depth is reached. The use of the SPIN structure of the temporal properties' checking for embedded test generation is presented in Figure 1. We express our test objectives as an undesirable property. This latter is converted to the Büchi automaton [14]. Furthermore, SPIN will produce a path as a counter-example if it exists that exhibits the violation of our property. Thus, this path is a sequence of events that corresponds to the satisfaction of our test objectives (*i.e.*$\neg property$).

### 3.1 Detection algorithm for accept state

This algorithm performs only the first DFS as we consider only finite execution. As said previously, the second DFS (the NDFS) is launched to search an acceptance cycle for infinite execution. We seek to reach an *accept* state from the initial state of the product automaton of the system and of the property. This state has as one component the *terminating* state of the property automaton. In this search, all the *terminating* states (i.e the *accept* states) of the transitions that appear in the test objective described by this property must to be reached. During this DFS, only the presence of the states in the accessibility graph is memorized by the addition of the state *s* in the *Statespace*. We keep track (in a transitions table indexed by their *terminating* states) of all the reached transitions, which are candidates to appear in the generated test sequence.

If a certain depth (*i.e.* a depth limit set by the user) is reached from a current state (initialized to the initial state)

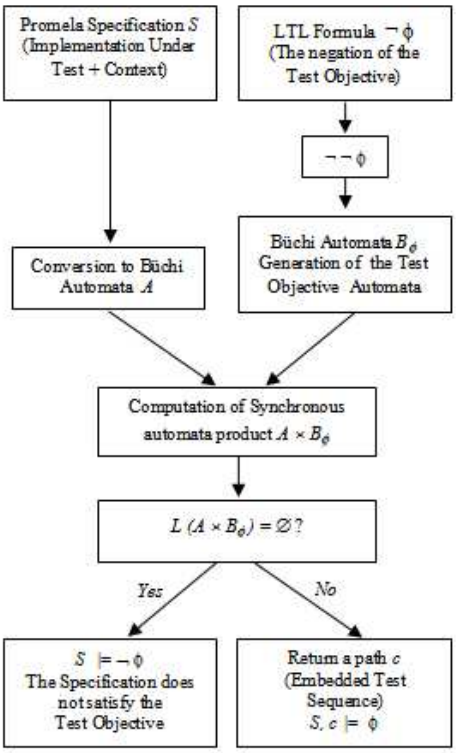


**Figure 1. SPIN structure for embedded test generation**

without detecting an *accept* or a *hit* state, a jump of depth $d$ is carried out by building a partial search tree and by choosing uniformly and randomly a leaf node of this tree. This tree has as root the current node and also has a depth $d$. Consequently, we restore the queues content of the current state and we update the states space and the transitions table. All the states reached before the jump and which are successors of the current node, will be replaced by all the states that form the path from this current node to the selected leaf node. In the same way, all the transitions occurred after this jump will be replaced by those occurred between this current node and the selected leaf node. This leaf node will be the new current node. We continue the search of an *accept* state or a *hit* state. The algorithm finishes if an *accept* state is reached or if the complete exploration of the states space is performed.

In the first case, this generation is carried out by going back to the intial state from the reached acceptance state. In the second case, only a partial test sequence is generated and contains only a part of the transitions to be covered.

```
current_node := initial_state
dfs(current_node);
proc dfs(s)
  if not(depth reached) then
    add(s) to Statespace
    if accepting(s) then
      report test sequence fi
    if s is Hit state then current_node:=s fi
    for each(selected) successor t of s do
      if t not in Statespace then
        add transition(s,t) to the reachedTransitions;
        dfs(t); fi
    od
  else if s = current_node then
    Build from s a partial exploration tree
    Choose uniformly and randomly a leaf node of this tree
    Initialize the partial Statespace
     Update the Statespace and the Transitionstable
    current_node := leafnode
    dfs(leafnode);
  fi
 end dfs
```

***The report function:*** To extract the test sequence beginning from the *initial* state to the *accepting* state and by using the reached transitions table *reachedTransitions*.

***The Build function:*** To build a partial exploration tree having depth $d$ from the *current node* and by using the partial *Statespace*.

***The Update function:*** To update the Statespace, the transitions table and the states sequence of the *dfs*.

### 3.2 The Test objectives

The Propositional Linear Temporal Logic (PLTL) [13] describes sets of paths in the computation tree. This logic allows to describe propositions linked by connectors and temporal operators. We use it to describe the test objectives that model the component transitions to be tested. We search to build in the accessibility graph (i.e part of the Büchi automaton resulting from the synchronous product) a path that checks the test objectives. This search is handled by the detection algorithm of an *accept* state. The formulation of the test objectives in PLTL obliges us to express it by a set of states that are the *originating* states of the transitions that we want to test. The SPIN tool accepts correctness properties expressed in PLTL, translates them into a *never claim* (in PROMELA language) and generates Büchi automaton. The *never claim* must express a negative property. For our embedded test generation method, we can not use the SPIN PLTL Property Manager to express test objectives. This editor translates automatically PLTL formulas

into *never claim* but it is not adequate with our test objectives formulation that has to describe states sequence. However, every state is associated to a property. A disjunction of these associated properties does not express this states suite. It is a necessary condition but not a sufficient one. Nevertheless, their conjunction has not any sense because it can be never satisfied. We express directly the test objectives by a *never claim* that describes the states sequence. This sequence is in fact, a sequence of occurred transitions between those states.

### 3.3 Test objectives formulation describing a finite undesirable behavior

The *never claim* describes the test objectives as a bad and finite behavior of the system. As said previously, the model checker SPIN permits to detect a system finite execution that checks these test objectives. This detection is in fact a search for an *accept* state. We generate the embedded test sequence of the component under test from this execution (states sequence from the root state until the *accept* state). This test sequence is a transitions sequence that occurs from the execution states (i.e from the *originating* states of these transitions). The *never claim* is noted $\neg$ *(*$\diamond$ *test_objective)* in PLTL. In PROMELA, it is in the following form:

```
never {
    do
    :: skip
    :: test_objective ⟶ break
    od
    accept : skip /* accept state */
}
```

We leave the loop when the test objective was checked and then we reach the *accept* state. A test objective formulation (in PROMELA syntax) of the *Receiver* component of the Alternating Bit Protocol is presented in the section 4.1.

## 4 Prototype Implementation

We realized an implementation of our test generation method. To construct our prototype, we introduced modifications to the states space exploration DFS procedure (i.e *new_state()*) and to the claim check procedure (i.e *check_claim()*). We have also used data structures (for example transitions matrix) to model the transitions system and states space.

### 4.1 Case Study: The Alternating Bit Protocol

The Alternating Bit Protocol (ABP) is a simple data link layer network protocol that retransmits lost or corrupted messages. Messages are sent from *Transmitter A* to *Receiver B*. Assume that the channel from *A* to *B* is initialized and that there are no messages in transit. Each message contains a data part, a checksum, and a one-bit sequence number, i.e a value that is 0 or 1. When *A* sends a message, it sends it continuously, with the same sequence number, until it receives an acknowledgment (ACK) from *B* that contains the same sequence number. When that happens, *A* complements (flips) the sequence number and starts transmitting the next message. When *B* receives a message from *A*, it checks the checksum. If the message is not corrupted *B* sends back an ACK with the same sequence number. If it is the first message with that sequence number then it is sent for processing. Subsequent messages with the same sequence bit are simply acknowledged. If the message is corrupted *B* sends back a negative-acknowledge character (NAK). This is optional, as *A* will continue transmitting until it receives the correct ACK. *A* treats corrupted ACK messages and NAK messages in the same way. The simplest behavior is to ignore them all and continue transmitting. We give an example of the formulation of the test objectives for the *Receiver* Component of the ABP.

### 4.2 Test objectives formulation for the receiver component

To test the *Receiver* component, we label the transitions that the test sequence must contain, in other words we instrument our PROMELA model. We use these labels for the formulation of the test objectives. In the example 1, *rec_state1* allows to label the control statel (i.e the *originating* state of the transition RACK!(B)). When this state is reached during the states space exploration, we are sure to have received a message and the correct control bit (*RDT?M,eval(B)* action) and to have sent the good ACK.

```
proctype Receiver ()
    int M; bool B = 0;
    do
    :: RDT?M,eval(B) ⟶ Get!M;
                        rec_state1 : RACK!B;
                        B = !B;
    :: RDT?M,eval(!B) ⟶ rec_state2 : RACK!(!B);
    :: RDTe?(Error) ⟶ rec_state3 : RACK!(!B);
    od
}
```

**Example 1: Receiver PROMELA specification**

In the example 2, we exhibit our *never claim* to test the transitions of the *Receiver*. We test that the *Receiver* executes the actions *RACK!(B)* (sending of a correct ACK after the reception of a correct control bit) and *RACK!(!B)* (sending of an incorrect ACK after the reception of an incorrect control bit). For that purpose, we define three propositions *Claim1*, *Claim2* and *Claim3*. For sake of simplicity, we will discuss only on the two first claims. The first (respectively the second) expresses that the *Receiver* has re-

ceived a correct control bit (respectively the incorrect control bit) and has gone to the *originating* state (i.e control state) of the action or transition *RACK!(B)* (respectively *RACK!(!B)*) labelled in the specification by *rec_state1* (respectively *rec_state2*). We define (in PROMELA) *Claim1* by "*#define Claim1 Receiver[rec_id]rec_state*". This expression *Claim1* is a boolean variable which takes the value *true* if the *Receiver* component identified by its process identity number *rec_id* is in the state labelled by *rec_state1*. Our test objectives (the *never claim*) express that the exploration must reach at least once (in an unspecified order) the states labelled by *rec_state1* (respectively *rec_state2*) that are the *originating* states of the transitions (*RACK!(B)* (respectively *RACK!( !B)*) which will form the final test sequence. In our embedded test generation method applied to the *Receiver* component, we must produce a path that reaches an *accept* state going through all the *hit* states (i.e. which are the reached states of the *Receiver* transitions to be tested).

The path (or the execution) to be produced have to meet our test objectives. To achieved this, we labelled the *hit* states by *accept* labels (*e.g accept_Hit_1* in Example 2) to distinguish them during the exploration of the automaton product (*i.e* system and property). They will be reached only once. Each property defined above (*i.e.* #define) is associated to a *hit* state of the *Receiver* automaton.

By checking these properties, at each time of the search, we can check if the associated *hit* state is reached. In the example 2, we used a Hit boolean array indexed by the *hit* states number and each element of this array is assigned to *true* whether the associated *hit* states is reached for the first time, by default the values are *false*. For instance, the condition *Claim1 && !Hit[0]* holds only if *Claim1* is valid (i.e the *Receiver* process instance is in a *hit* state labelled by *rec_state1*) and this state is reached for the first time. We initialize to *false* all the elements of the Hit array. Each time that an *accept hit* state is reached, we will marked this traversal. The *accept* state of the *never claim* automaton is reached, if all the *hit* states are reached during the search (i.e all the array elements are evaluates to *true*).

```
never {
    skip;
    Hit[0]=0; Hit[1]=0; Hit[2]=0;
    T0_init : if
              :: Claim1 && !Hit[0] ⟶
              accept_Hit_1 : atomic {Hit[0]= 1;
              if
                 :: (Hit[0] && Hit[1] && Hit[2])
                 :: else ⟶ goto T0_init
              fi
              }
              :: Claim2 && !Hit[0] ⟶
              accept_Hit_2 : atomic {Hit[1]= 1;
              if
                 :: (Hit[0] && Hit[1] && Hit[2])
                 :: else ⟶ goto T0_init
              fi
              }
              ...
} /* end of the never claim */
```

%endminipage

**Example 2: Receiver test objectives**

### 4.3 Discussion of results

This section presents a comparative table (see Figure 2) between the results obtained by the original SPIN and by our own prototype for the same case study in the same context. The case study is a well-known protocol, i.e the ABP, and allows us to exercise our test method. More precisely, we chose as test objectives the coverage of all the transitions of the *Receiver* component. This latter is not directly accessible as we have to go through the *Transmitter* which can initiate the communication and through the medium. We compared the stocked states number (in data structures), the visited states number, the analyzed or occurred transitions number, and the violation depth of the temporal claim (i.e the test objectives negation).

| Maximal Depth | Tool | Stocked States | Visited States | Analysed or occured Transitions | Violation Depth |
|---|---|---|---|---|---|
| 75 | Prototype | 150 | 96 | 246 | 74 |
| | SPIN | 150 | 106 | 256 | 74 |
| 80 | Prototype | 175 | 100 | 275 | 74 |
| | SPIN | 195 | 171 | 366 | 74 |
| 95 | Prototype | 58 | 13 | 71 | 94 |
| | SPIN | 57 | 17 | 74 | 90 |

**Figure 2. Discussions of results**

We note that for the two tools (SPIN and our own prototype), we obtained the same test sequence with the same depth. Nevertheless, in most of cases, the use of our prototype decreases all the evaluated parameters mentioned previously. We can explain this by the fact that SPIN DFS carries out a back-tracking if the maximum search depth is reached without reaching an *accept* state. Indeed, a maximal depth limit is predefined and the value is high. On the contrary, our prototype works on partial graph and traverses smallest path due to the small depth limit set to obtain a jump. In this case we only back-track to the last *hit* state (current node) reached by the depth limit and executes a jump. Moreover, the jump allows us to get out the "‘trap"’ and to search our test objectives in other part of the graph. We can avoid very long test sequences with uninteresting paths according to a good depth search choice.

## 5 Conclusion

This paper discussed a new approach to generate test cases for an embedded component. This method conducts a search in the partial product of the whole system and the automaton of the formula that represents the test objectives *i.e.* the transitions of the component to cover. For that purpose, we use the counter-example generation of the model-checker SPIN as our test cases. This tool explores all the possible states within the model and checks whether the property holds. Otherwise, it outputs a trace that illustrates the violation of the property, the counter-example. In our case, we negated our test objectives and fed the tool. Hence, the violation of the property in SPIN in that case will represent the meet in the specification of our test objectives.

We have implemented inside SPIN an efficient testing algorithm that allows to avoid in most cases the state spaces explosion. Moreover, SPIN permits to manage this problem by its optimization techniques. The combination of both gives good results in terms of number of states to visit and to store. We have exercised our prototype on the alternating bit protocol. We obtained first promising results for the test cases generation of embedded components, which allows us to consider huge systems by avoiding states space explosion and to access components that have no direct access from the environment.

An immediate line of future is to exercise our prototype with different criteria of coverage. We present in the paper a coverage criterion that is the transitions coverage in order to produce a transitions tour of the component in its context. This criterion can be rapidly changed , may one want only to test some critical functionalities of a module. We can also mention improvements on our prototype in order to have an integrate tester available with SPIN. We can modify the SPIN editor to adjust it for that purpose *i.e.* edition of the test objectives, posting of test cases, and so on. Moreover, we want to improve our prototype in order to deal with real protocols.

## References


[1] C. Besse, A. Cavalli, M. Kim, and F. Zaïdi. Automated Generation of Interoperability Testing. *IFIP TC6/WG6.1 Fourteenth International Conference on Testing and Communication Systems*, pages 169–184, march 2002.

[2] 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, Aug. 1998.

[3] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using modelchecking. In *Proceedings 1996 SPIN Workshop*, Aug. 1996.

[4] A. Cavalli, B. Defude, C. Rinderknecht, and F. Zaïdi. A Service-Component Testing Method adn Suitable CORBA Architecture. In I. C. Society, editor, *Proceedings of the Sixth IEEE Symposium on Computers and Communications*, pages 655–660, Tunisia, july 2001.

[5] A. Cavalli, D. Lee, C. Rinderknecht, and F. Zaïdi. Hit-or-Jump: An Algorithm for Embedded Testing with Applications to IN Services. In *Proceedings of FORTE/PSTV'99*, Beijing, China, Oct. 1999.

[6] A. Cavalli, A. Mederreg, F. Zaïdi, P. Combes, W. Monin, R. Castanet, M. MacKaya, and P. Laurençot. A Multi-Services and Multi-Protocol Validation Platform - Experimentations Results. In R. Hierons and R. Groz, editors, *The 16th IFIP International Conference on Testing of Communication Systems*, pages 17–32, Oxford, march 2004. Lectures Notes in Computer Science.

[7] R. de Vries and J. Tretmans. On-the-fly conformance testing using. In *4th International SPIN Workshop*, Paris, 1998.

[8] D. P. G. J. Holzmann and M. Yannakakis. On Nested Depth First Search. *Proc. Of the 2nd SPIN Workshop. The Spin Verification System.*, pages pp. 23–32, 1996.

[9] A. Gargantini and C. L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC / SIGSOFT FSE*, pages 146–162, 1999.

[10] G. J. Holzmann. *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley, Reading Massachusetts, 2004. 608 pgs.

[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] L. P. Lima and A. Cavalli. Exécution de tests de services sur une plate-forme distribuée. In *Proceedings NOTERE'97*, Pau, France, Nov. 1997.

[13] A. Pnueli. The temporal logic of programs. *Proc. 18th IEEE Symposium on Foundations of Computer Science*, pages pp. 46–57., 1977.

[14] M. Vardi and P. Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115:pp. 1–37., 1994.