Towards execution in automatic test suite generation

Yixin Zhao Jianping Wu
Department of Computer Science and Technology,
Tsinghua University, Beijing, 100084, China
Email: zhaoyx@csnet1.cs.tsinghua.edu.cn Fax: 8610-62788109
jianping@cernet.edu.cn

Abstract Only executable test suite generated automatically has practical usage. The authors devise and implement the algorithm of "parametrizing and executizing" in the TUGEN system and discuss the result, limitation and its reason Basing on these work and the study in the executability of the transition and the satiability problem of the predicate, the authors further present and implement another algorithm called "executable parametrizing" to overcome the deficiency of the previous one and further improve the practicability and efficiency of TUGEN. After analysis and comparison, the authors outline the focus of the future research.

Key words automatic test suite generation, parameterization, executization, complexity

1. Introduction

Automatic generating test suite is still a hot point of protocol test [1-3]. With the rapid development of network technology, the traditional way of writing test suite by hand has fallen far behind the actual requirement. Automatically generated test suites considerably save human labor while offering much more test cases, and, if cooperated with proper test system, such as PITS [4], it can realize the automatilization of the whole test procedure. Despite the fruitful research [5-7] done so far, much is left to do to make the automatically generated test suite executable. Non-executable test suite can not be used in actual test activity, and it is only useful for theoretically analysis. But since there are non-executable test cases in the suite, the accuracy of analysis is inevitably affected.

In the development of TUGEN [5], much of the emphasis lies in the executability of the automatically generated test suite. The algorithm of "Parametrizing and executizing" is designed and implemented, which guarantees that all the test cases generated are executable [8]. However, for some test cases, although there do exist proper parameters to make them executable, this algorithm can’t find these parameters efficiently. Further study shows that, it is the procedure of "parametrizing first, executizing second" that raises this problem. It is realized the problem of the executability of transitions and the complexity of predication satisfiability must be resolved well first in the work of automatic generating test suite. Based on all these, the algorithm of "executable parametrizing" is devised and implemented, which possesses maturity in theory and gets desirable results in practice.

This paper is composed of six parts. Chapter 2 briefs automatic generating test suite. Chapter 3 analyzes the algorithm of "parametrizing and executizing". Chapter 4 discusses the problems of the executability of transitions and the complexity of predicate satisfiability. Chapter 5 explains and analyzes the algorithm of "executable parametrizing". In the last chapter, a conclusion is given.

2 Brief of automatic generating test suite

In traditional procedure of conformance test, test suite generation is the most difficult part of all, which is always undertaken by experts familiar with protocols. Not only are subjective errors introduced inevitably, the number of test cases is also very limited. For example, there are only about 200 test cases in *ISO/IEC 8882-3 Packet layer conformance test suite*. One of the trends in protocol test is generating test suite automatically or computer aided. Researchers have devoted themselves to this field, devised models and algorithms, and implemented several tools or systems to generate test suites. Figure 1 is the main process of TUGEN [5], which is a typical automatic test suite generating system.

* Supported by National Natural Science Foundation of China under Grant No. 69682002 and No.69725003

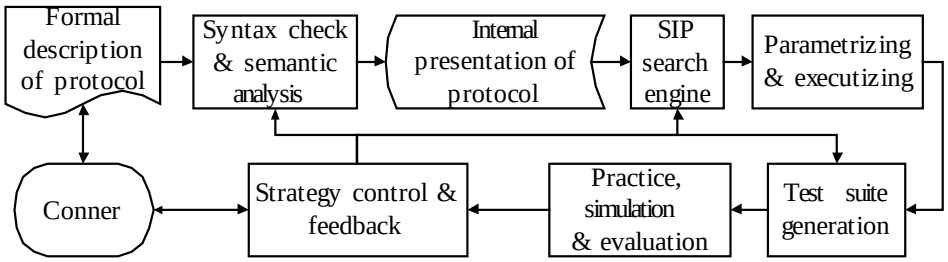

Figure 1. The process of TUGEN

The input of TUGEN is the External Behavior Expression (EBE) [9] of the protocol. The EBE is defined as below:

Def.1 *EBE is a four-tuple <**S**,s_0,**T**,**R**>, where **S** is the external state set, s_0 is the initial external state, **T** is the transition set of **S** and **R** is the logic relation and timing relation set of **T**.*

Def.2 *A transition between external states is the interactions exchanged between the system and its external environment in terms of input and/or output primitives and their parameters. The general form of a transition is given by T_{ij} = < I_p, O_q >, where $T_{ij} \in \boldsymbol{T}$, $I_p \in \boldsymbol{I}$, $O_q \in \boldsymbol{O}$, and:*

*1) **I** is a set of input primitives from the external environment, and each input primitive is denoted by : I_p (X_{p1}, . . . , X_{pn}), where I_p is the input primitive identifier, and X_{p1}, . . . , X_{pn} (n>=0) are parameters of the input primitive I_p.*

*2) **O** is a set of output primitives to the external environment, and each output primitive is denoted by: O_q (Y_{q1}, . . ., Y_{qm}), where O_q is the output primitive identifier, and Y_{q1}, . . ., Y_{qm} (m>=0) are parameters of the output primitive O_q.*

3) The absence of an input primitive or an output primitive is denoted by "--".

Def.3 *The set of logical relations of a transition R_{ij} = (**F**, **P**, **OP**) holds if and only if there exists a transition T_{ij} from state s_i to s_j.*

*1) **Z** is the group of elements that influence the output of transition T_{ij}. **Z**= PARAMETER ∪ TIMER ∪ VAR. Where PARAMETER is the set of input and/or output parameters in the service primitives and VAR is the set of protocol variables and TIMER is the set of time constructs used to specify time in the protocol representation,*

*2) **F** is a set of function relations of transition T_{ij}. The output parameters Y_{q1}, . . ., Y_{qm} (m>=0) of transition T_{ij} will be produced if and only if there exists Z which satisifies a set of function F(Z), that is, {Y_{q1}, . . ., Y_{qm} }=**F(Z)**.*

*3) **P** is a set of predicate relations of transition T_{ij}. The transition T_{ij} will happen if and only if there exists Z that satisfies the predicate **P(Z)**.*

*4) **OP** is a set of operation relations of transition T_{ij}. These operations are mainly used to deal with system global variables such as protocol parameters and timers and some additional actions that need to be done during a transition.*

*5) A transition T_{ij} can be executed if and only if the I associated with the transition (if any) is received, and the enabling predicate P_{ij} is true. When a transition T_{ij} fires, the associated function F_{ij} and operation OP_{ij} is executed automatically. When protocol variables and timers are set, **O** is assembled (their parameters are set) and sent.*

Def.2 and Def.3 describe the external behavior of the system, and process the values and their changes of

input/output primitives' parameters. An example described with EBE is given below.

S3 **[**CONF_REQ <PCO2>, CONF_ACK <PCO2>|

PREDICATE: REQ_SEND = TRUE **AND** CONF_REQ.LENGTH = CONF_PDU_LENGTH

AND CONF_REQ.MRU <= SYSTEM_MRU

OPERATION: REQ_RECEIVCED:=TRUE; RECEIVED_ID := CONF_REQ.ID;

RECEIVED_MRU := CONF_REQ.MRU;

FUNCITON: CONF_ACK.ID := RECEIVED_ID; CONF_ACK.MRU :=RECEIVED_MRU**;]** S4

The main process of TUGEN contains 4 steps:

Step 1 Input the formal description of the protocol. After syntax and semantic checking, enough information is extracted to construct the internal presentation of the protocol.

Step 2 Interactive Path (IP) or Sub Interactive Path (SIP) is acquired according to some searching strategy and constraint mechanism [5].

Even if the parametrizing and executizing step is omitted, test trees with correct, incorrect and protective branches can still be constructed using SIP. Each leaf node in the test tree will be assigned one of the three verdicts, on which verification and evaluation can be performed. However, there is no guarantee that all the test cases are executable, which is not only a waste but also a negative factor for the accuracy of test suite analysis and evaluation.

Step 3. Executizing process of the test cases.

Step 4. Evaluate the test suite by simulation or test results. When necessary, modify the formal description, searching strategy, and re-generate the test suite until we get satisfying results.

3 Analysis of the "Parametrizing and executizing" algorithm in TUGEN

The process of "Parametrizing and executizing" algorithm ("A.1" for short) is shown in figure 2.

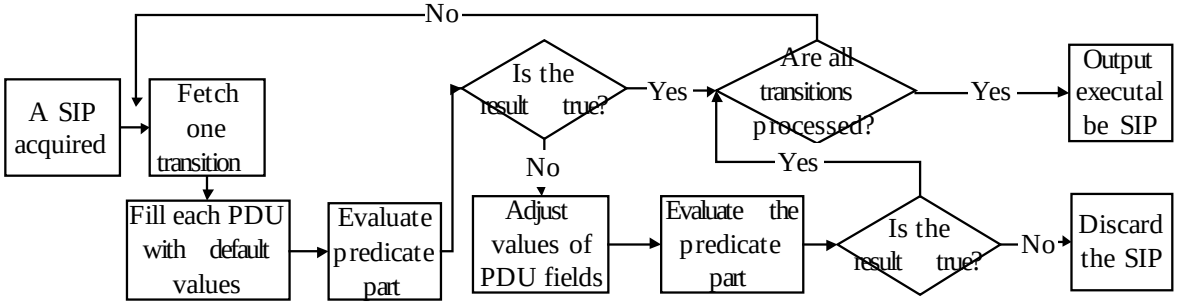

Figure 2. The process of A.1

3.1 Description and analysis of A.1

Suppose ***left_value rel_op right_value*** is a logical expression in the predicate part. **rel_op**∈ {=,>,>=,<,<=,<>} which is the logical operator. The **left_value** and **right_value** act as their names imply. When the type of **left_value** is a variable in protocol formal description, **right_value** will be constant or status flag. If the type of **left_value** is a field (depicted as $PDU_i.FIELD_j$) of PDU (Protocol Data Unit) in the description, **right_value** will be constant or variable. The key part of A.1 is "Adjust values of PDU fields", the algorithm of which is given below:

Input: One transition with default values filled and the predicate part evaluated false using these default values.

Output: One transition with adjusted values of PDU fields or none

Step1 Fetch first atomic logical expression of the predicate part

Step2 If the type of left_value is not a field of PDU, then goto *Step6*

Step3 Evaluate this expression

Step4 If result is TRUE, then goto *Step6*

Step5 CASE rel_op

= : $PDU_i.FIELD_j$:= right_value;

> : $PDU_i.FIELD_j$:= right_value + 1;

>= : $PDU_i.FIELD_j$:= right_value;

< : $PDU_i.FIELD_j$:= right_value - 1;

<= : $PDU_i.FIELD_j$:= right_value;

<> : $PDU_i.FIELD_j$:= right_value + 1;

Step6 If done, then return with adjusted transition

Step7 Fetch next atomic logical expression and goto *Step2*

Since the values of most PDU fields are constant, they can be filled with default values and be checked for executability, that is, evaluate the whole predicate part. If the result is true, then this transition is executable after parameterization. Otherwise, adjustment is needed to make its predicate part true.

We need to stress here that A.1 only adjusts the values of those fields of input PDU and does nothing with other variables. This is because that the formal description part depicts the action of the protocol implementation, and therefore the fields of input PDU can be filled by us, that is, determined by the testing part. On the other hand, variables of protocols are relevant to the procedure and the state of the protocol, so we can't change their values at will.

The adjustment module exploits a heuristic approximate algorithm: find all the expressions that are evaluated false and then adjust them until true. The defect of A.1 is that, it is not necessary to demand all the expressions to be true to make the whole predicate part true. Even if there are some expressions evaluated false, the whole predicate part can still be true. For example:

PREDICATE: PDU1.TYPE = TYPE1 **AND** PDU1.FIELD1 = VALUE1

AND(PDU_FIELD2 = VALUE2 **OR** PDU1.FIELD1 = VALUE3)

If the last expression is evaluated false while the previous three expressions are all true, then the predicate part is still true. On the other hand, A.1 is not sufficient either:

PREDICATE: PDU1.TYPE = TYPE1 **AND** PDU1.FILED1 = VALUE1

AND PDU1.FIELD2 = VALUE2 **OR** PDU1.TYPE = TYPE2

If the value of PDU1.FILED1 is changed to VALUE3 in order to satisfy PDU1.FIELD1 = VALUE, and if VALUE3<>VALUE1, then the whole predicate part is still evaluated false after the adjustment. The re-evaluation process in A.1 is to prevent such phenomena. Therefore, A.1 can't guarantee to generate all the test cases that can be excutized, but it does guarantee that those non-executable test cases will not appear in the test suite generated. That is, all the test cases generated are executable. The result of A.1 is shown in table 1.

Protocol Under Test	Set of All the searched cases S	Set of actually executable test cases E	Set of executable test cases filled with default values D	Set of executable test cases after adjustment A	Set of failure of adjustment F	Process time T*
TCP	|S| = 3122	|E| = 718	|D| = 437	|A| = 667	|F| = 51	17 min.
PPP	|S| = 4723	|E| = 853	|D| = 314	|A| = 793	|F| = 60	24 min.
OSPF	|S| = 45128	|E| = 11734	|D| = 5984	|A| = 11261	|F| = 473	13 hours

Table 1 Result of A.1 (*running platform is Ultra 1 , 167Mhz with 64M memory and Solaris 2.5.1)

4 Two open problems in executization

4.1 The executability of the transition

A SIP or a test case is executable, if and only if all the transitions in it are executable. A transition is executable, if and only if its predicate is satiable (can be evaluated true). But the executability of transition **T** always refer to its executability in some path **P**, that is, it is the executability in the context, which reflects the procedure and state of the protocol behavior. It is of no use to discuss the executability of one transition alone.

The predicate of each transition determines if this transition is executable, but the relation between predicate and transition is not simply determining and being determined. Instead, we believe each transition is executable (can occur), which is the primary factor, while the predicate describes the conditions under which the transition "would like" to happen.

In figure 3, as to state s_j ($s_j \neq s_0$), all the transitions ending at s_j are marked as T_{ij1}, T_{ij2} ,…, T_{ijm}, where $m=d^+(s_j)$; All the transitions starting from s_j are marked T_{oj1}, T_{oj2}, … ,T_{ojn}, $n=d^-(s_j)$.

For any transition $T_{ijp,}$ there must be a T_{ijp}, and under some condition $Cond_t$, T_{ijp}-T_{ojq} will be on some executable SIP, that is

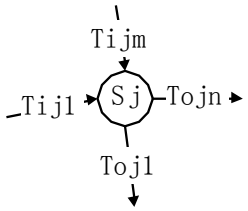

Figure 3. Executability of a transition

$$\forall \quad T_{ojq}, q \in [1..n] \; \exists T_{ijp}, p \in [1..m] \; \exists Cond_t \; Cond_t : T_{ijp} \rightarrow T_{ojq}$$

Otherwise, T_{ijp} is called a **transition without successor**.

On the other hand, for any transition $T_{ojq,}$ there must be a T_{ijp}, and under some condition $Cond_t$, T_{ijp}-T_{ojq} will be on some executable SIP, that is

$$\forall \quad T_{ojq}, q \in [1..n] \; \exists T_{ijp}, p \in [1..m] \; \exists Cond_t \; Cond_t : T_{ijp} \rightarrow T_{ojq}$$

Otherwise, T_{ojq} is called a **transition without predecessor**.

Obviously, there will be no transition without successor or without predecessor unless there is some error in protocol itself or its formal description.

4.2 The complexity of the satisfiability problem of predicate

The question of whether one transition is executable, that is, whether there is a set of values to make the predicate part evaluated true, will end at the satisfiability problem of logical expressions. If we can determine that the logical expressions of the predicate part are satiable, then we can further find a set of values to make the predicate part evaluated true. Otherwise, the transition is non-executable. Generally speaking, the satisfiability problem of conjunctive normal form is a NPC problem, that is:

For logical expression f, $f=C_1 \wedge C_2 \wedge \ldots \wedge C_k$, each C_i is composed of ORed logical variables(or NOTed variables).Then f is called conjunctive normal form, each C_i is a clause of f. The satisfiability problem of conjunctive normal form is referring that, for any f, if there exists a set of values to make f evaluated true. (*)

If we induce each logical variable to one atomic logical expression, then the satisfiability problem of conjunctive normal form is induced to the satisfiability problem of predicate part.

The satisfiability problem of disjunctive normal form is a P problem, which can be solved in polynomial time, that is:

For logical expression g, $g=d_1 \vee d_2 \vee \ldots \vee d_k$, each d_i is composed of ANDed logical variables(or NOTed varialbes). Then g is called a disjunctive normal form, each d_i is a clause of f. The satisfiability problem of disjunctive normal form is concerned about, for any g, if there exist a set of values to make g true. (**)

Obviously, it only takes polynomial time to solve the satisfiability problem of disjunctive normal form. If we induce each atomic logical expression to one logical variable defined as above, then the satisfiability problem of predicate part can be induce to the satisfiability problem of disjunctive normal form. The predicate part of the protocol description is always in the format of disjunction.

It can be concluded from (*) and (**) that for a general logical expression, the algorithm complexity of its satisfiability problem is between P and NPC. General logical expressions can be transformed to disjunctive normal form or conjunctive normal form. But this procedure can’t be guaranteed to accomplish in P time. For example, the transformation from conjunctive normal form to disjunctive normal form will generate exponential number of items. Therefore, it is not a proper way to determine the satisfiability problem through transformation,

for example, after exponential transformation, the satisfiability problem of disjunctive normal form that can be solved in P time becomes the satisfiability problem of conjunctive normal form, which is a NPC problem.

Since the predicate part only specifies the conditions under which the transition "wish" to occur, then requirement of description can be fulfilled just by disjunctive normal form. Each clause specifies one condition, all the possible conditions are ORed together. Then P time to solve the satisfiability problem is guaranteed from the beginning by algorithm complexity theory. Therefore, it is feasible and necessary to solve the satisfiability problem of predicate part from the stage of formal description of protocol.

5 Executable parametrizing Algorithm

The primary defect of A.1 is its procedure of processing. First parametrizing then executizing increase the complexity, instead. Secondly, A.1 is not complete in theory, which will definitely effect the quality of test suite and the exactness of the valuation.

The "Executable parametrizing Algorithm"(A.2 for short) settles these problems. Firstly, A.2 combines parametrizing and executizing together to process the test case. To be more exact, it is a parametrizing algorithm with an eye to the demand of executability, from which comes the "Executable parametrizing". After main problems are solved theoretically, this algorithm can be further exploited for an exact algorithm to guarantee the quality of the test suite.

To make A.2 more clear, a simple description of the symbols appeared in EBE is given below:

< protocol symbol> := <const>|<variable>

< variable> := <state variable>|<event variable>|<internal variable>|<PDU field variable>

The "const" defines constant value in protocol, such as C021, C023 in the "protocol type" of PPP [10], which will remain unchanged and can be used as right value of the operation and function parts or the right value of the atomic logical expression of the predicate part.

On the other hand, the value of the "variable" may be assigned or modified with the interaction of the protocol activity. The "state variable" records the information of protocol states and will be processed in the operation part. The "event variable" keeps track of the events that have happened, such as a "Time Out" event, for the convenience of the formal description of protocols. The "internal variable" records the values of input and output PDU, such as the ID value of a received PDU. A complete PDU is composed of all the necessary "PDU field variables". “Executizing” is to assign proper values to all these variables, especially the values of the field variables of input PDUs.

Each value concerning the PDU field is actually a 3-tuples (PDU, FIELD, VALUE). Different PDU and FIELD determines different value ranges of the VALUE. For example, the C021 denoting the protocol type should be depicted as <CONF_REQ, PROTOCOL, C021>.

5.1 Description of A.2

The 3-tuples of PDU field value is further extended to (PDU, FIELD, VALUE, DOMAIN), where DOMAIN is the set of the possible values of this PDU field and set to ANY initially, that is, all the possible values of this field.

Input: A searched SIP

Output: Executable SIP after parametrizing or none (when no executable values exist)

Step1 Fetch the first conjunctive clause

Step2 Initialize PDU_i.$FIELD_j$.DOMAIN to ANY

Step3 Take first expression of the conjunctive clause

Step4 If the left_value is of state variable then evaluate the expression

Step5 If result is false, then goto *Step12*

Step6 If left_value is of event variable then evaluate the expression

Step7 If result is false then goto *Step12*

Step8 If left-value is of PDU field variable(referred as $PDU_i.FIELD_j$), then

CASE rel-op

= :$PDU_i.FIELD_j.DOMAIN$ ={right_value}

> :$PDU_i.FIELD_j.DOMAIN$ = $PDU_i.FIELD_j.DOMAIN$ ∩(right-value,MAX_VALUE]

>=:$PDU_i.FIELD_j.DOMAIN$ = $PDU_i.FIELD_j.DOMAIN$ ∩[right-value,MAX_VALUE]

< :$PDU_i.FIELD_j.DOMAIN$ = $PDU_i.FIELD_j.DOMAIN$ ∩[MIN_VALUE,right-value)

<=:$PDU_i.FIELD_j.DOMAIN$ = $PDU_i.FIELD_j.DOMAIN$ ∩[MIN_VALUE,right-value]

<>:$PDU_i.FIELD_j.DOMAIN$ = $PDU_i.FIELD_j.DOMAIN$ ∩(right-value,MAX_VALUE]

Step9 If $PDU_i.FIELD_j.DOMAIN.DOMAIN$is null, then goto *Step12*

Step10 If this conjunctive clause is done, then select a value from $PDU_i.FIELD_j.DOMAIN.DOMAIN$ and assign to $PDU_i.FIELD_j.VALUE$. Successfully return with executable SIP.

Step11 Otherwise, take next expression and goto *Step4*

Step12 If the predicate part is processed, then return none with this SIP discarded.

Step13 Fetch the next conjunctive clause and goto *Step2*

5.2 Analysis and evaluation of A.2

A.2 combines parametrizing and executizing together, and is designed with executability as the goal. Therefore the defects in A.1 have been overcome. A.2 is a precise and P time algorithm with improved efficiency. A.2 is complete in theory, guaranteeing all the executable test cases will all generated, which improves the quality and efficiency of test suite generation. The result of A.2 is given in table 2.

Protocol Under Test	Set of All the searched cases S'	Set of actually executable test cases E'	Set of executable test cases after processing A'	Set of failure of adjustment F'	Process time T'
TCP	|S'| = 3122	|E'| = 718	|A'| = 718	|F'| = 0	10 min.
PPP	|S'| = 4723	|E'| = 853	|A'| = 853	|F'| = 0	18 min.
OSPF	|S'| = 45128	|E'| = 11734	|A'| = 11734	|F'| = 0	10 hours

Table 2 Result of A.2 (*the same running platform as shown in table 1)

Compared with A.1:

(1) S = S', E = E' This is because the searching algorithm remains the same

(2) In A.2 there is no longer any "Set of executable test cases filled with default values"

(3) A' = E' ,|A'| > |A| ,A' ⊃ A (4)F' = Φ ,|F'| < |F| ,F' ⊂ F (5)T' < T

From (2) to (5) embody the advantage of A.2 over A.1. Further comparison is given in table 3, where the correctness is referring to never generating non-executable test cases and the completeness is referring to generating all the test cases for which there do exist executable parameters. A.2 results in a set of possible values, and we further improve the test case coverage rate with different data selection strategy.

Name of the Algorithm	Completeness in theory	Complexity	Exactness	Correctness	Completeness
Parametrizing and Executizing	No	P	No	Yes	No
Executable parametrizing	Yes	P	Yes	Yes	Yes

Table 3. Comparison of A.1 and A.2

6. Conclusion

The test suite automatically generated ought to be executable, which is emphasized in TUGEN. There are

two algorithms implemented consequently. After the analysis of the defects in the "parametrizing and executizing algorithm" and the study of executability of the transition and the satisfiability problem of predicate part, the executable parametrizing algorithm is devised and analyzed with those executable test cases as the goal. Besides, it also possesses completeness in theory and further improves the feasibility of TUGEN. During the study, more improvements can be done to the procedure of first searching SIP then executizing it. The executizing can be performed during the SIP searching, that is, the executability of a transition will become a condition to control the search, therefore further improvement in the efficiency of TUGEN can be expected in later work.

Reference

1. S.T.Vuong, H.Janssen,Y.Lu, C.Mathieson and B.Do. TESTGEN: An environment for protocol test suite generation and selection, Computer communications Vol.17 number 14 April 1994:23-27
2. Samuel T. Chanson, Jinsong Zhu. A Unified Approach to Protocol Test Sequence Generation, Proc. IEEE INFOCOM, San Francisco, March 1993. 137-149
3. ISO/IEC ,JTC1/SC21 WG7. Information Retrieval, Transfer and Management for OSI, FMCT Guidelines on Test Generation Methods from Formal Descriptions, Annex A and Annex B, ISO, February 1995
4. Wu Jianping. PITS-The protocol integrated test system based on formal technology, Journal of Tsinghua University. 1998, 38(S1):26~29
5. Jianguo Wang, Reibin Hao and Jianping Wu. TUGEN : An automatic test case generator integrating data-flow and control-flow test methods. Proc. IEEE International Communication Conference, Atlanta.Session 8 Paper 7.1998. 332-343
6. JianPing Wu and Samuel T.Chanson. Testing Sequence Derivation Based on External Behavior Expression. Proc. 2nd International Workshop on Protocol Test Systems. Berlin. 1989. 172-184
7. Zhao Yi Xin. The PPP conformance test based on automatically generated test suite, Graduate thesis, Tsinghua University, 1998
8. Krishan Sabnani and anton Dahbura. A Protocol Test Generation Procedure. Protocol Specification, Testing and Verification:VIII,North Holland,1988. 173-191.
9. A.Guerroua and H.Konig. Automation of test case derivation in respect to test purposes,Proc. 7th International Workshop on Protocol Test Systems,Berlin,Germany,1996. 213-230
10. W. Simpson. The Point-to-Point Protocol (PPP). RFC1661, 1994