Generating Test Cases of Composite Services Based on OWL-S and EH-CPN

Bixin Li[1,2], Ju Cai[1], Dong Qiu[1], Shunhui Ji[1], and Yuting Jiang[1]
[1]School of Computer Science and Engineering, Southeast University
Nanjing 210096, Jiangsu Province, P.R.China Email: bx.li@seu.edu.cn
[2]Dept. of Computer Science and Engineering, University of California
Riverside, CA92521, USA. Email: lbxin@cs.ucr.edu

Abstract

In web service times, the techniques for composing services are the base of service reuse and automatic integration. A new web service will be generated by composing some existed web services, these web services cooperate each other to provide a new more complex function. It is needed and very important to test the interaction behavior between any two web services during composition. In this paper, a kind of enhanced hierarchical color petri-net (or EH-CPN) is introduced to generate test cases for testing the interaction, where EH-CPN is transformed from OWL-S document, and both control flow and data flow information in EH-CPN are analyzed and used to generate an executable test sequence, and further test cases are created by combining the test sequence and test data in an XML file.

1 Introduction

Web service technology has got widely and warmly welcomed in developing application software based on internet environment, but it has raised many new challenges for its testers, where two core challenges will be considered in this paper are: (1) source code of a web service is invisible to tester: clients of a web service can have functions provided by the service but they cannot get the source code of the service. It means that structure testing strategies are not able to be used to test your wanted single services, because it is impossible for testers who are not service providers themselves to generate test cases from source code of a web service, some informal specifications are explored to see the possibility for generating test cases, so both difficulty and complexity raise. The specifications which are being explored include WSDL, BPEL, and OWL-S, some necessary transforms are needed to generate test cases automatically and precisely; (2) many intermediate states of web service are also invisible to testers, it is hard to do testing manually, some automatic test techniques are needed.

In order to solve those challenge problems, researchers have introduced a variety of useful methods [1, 2, 3]. But most of these methods are based on WSDL or BPEL specification that specifies the location of the service and the operations (or methods), which the service will be exposed to clients. Test cases generated by these methods are more suitable for atomic web service or a composite service within an organization. In a wide composing service environment, web service is usually used to interact with other services from different organizations, it will play a different role. It is needed and very important to test the interaction behaviors among different web services when we compose some existed web services in a certain style to provide new functions.

In this paper, an EH-CPN based test case generation approach has been introduced, where EH-CPN is an enhanced hierarchical color Petri Net. The outline of the approach is summarized as follows: at first, OWL-S document is transformed to EH-CPN and further the process of web service composition is displayed by this kind of Petri Net; next, data flow and control flow information of EH-CPN are analyzed in detail to find all *output-input-define-use chains* (or *OI-du-chain*); next, *OI-du-chain* is extended to correspondent executable test sequences satisfying `ALL-DU-PATHS` criterion; finally, both test sequences and test data are combined to generate test cases.

2 Primaries

There are three important concepts will be used in this paper, let's see how they are defined.

Definition 1 *Multiset*[4]

A multiset bag is a function from a non-empty set A to non-negative integer set IN, $bag : A \rightarrow IN$. Let set $Bag(A) = \bigcup_{a \in A} bag(a)$ be the set of all the multisets that are defined in set A.

Definition 2 *E-CPN*

An extended color Petri Net (marked as E-CPN in this paper) is defined as follows: E-CPN is a 6-tuple $< P, T, C, Cd, Pre, Post >$

(1) P is a finite set of places;

(2) T is a finite set of transitions. There are five kinds of typical transitions in web service composition: (2.1) *service invoking transition:* When this transition is fired, it will invoke the corresponding web service; (2.2) *condition controlling transition:* When this transition is fired, it will invoke a condition checking function whose return value is Boolean, and which places the transition will go to depends on the return value; (2.3) *concurrent controlling transition:* This transition is used to assort with synchronization between transitions; (2.4) *interface transition:* This transition will invoke checking function to check whether or not the output from upper net equals to the input to fire sub net; (2.5) *empty transition:* This transition will invoke nothing. The aim to define this transition is to make the net satisfy the definition of Petri Net in some special condition.

(3) C is a finite set of colors.

(4) Cd is a color function $Cd : P \cup T \rightarrow C$

(5) $Pre, Post \in \beta^{|P|\times|T|}$, both are Incidence Matrixes, where Pre is a pre-Incidence Matrix and $Post$ is a post-Incidence Matrix satisfying following equations:

$$\forall(p,t) \in P \times T, Pre[p,t] : Cd(t) \rightarrow Bag(Cd(p)),$$

$$Post[p,t] : Cd(t) \rightarrow Bag(Cd(p));$$

β is a set of grouping functions in following form:

$$\beta : Cd(t) \rightarrow Bag(Cd(p))$$

Definition 3 *EH-CPN*

An enhanced hierarchical color Petri Net (marked as EH-CPN in this paper) is defined as follows: EH-CPN is a 4-tuple $< S, C, IC, I_0 >$

(1) S is a finite set of sub nets satisfying following features:

(1.1) $\forall s \in S, s = (E\text{-}CPN, Ci, Co)$, Ci is a finite set of input colors, Co is a finite set of output colors.

(1.2)$\forall s_i, s_j \in S$ and $s_i \neq s_j$, $(P_{s_i} \cup T_{s_i} \cup A_{s_i}) \cap (P_{s_j} \cup T_{s_j} \cup A_{s_j}) = \Phi$

(2) C is a finite set of color

(3) IC is an interface checking function which will check whether or not the output coming from the upper net equals to the input that can fire the sub net.

(4) I_0 is an initialization state

3 Transforming OWL-S to EH-CPN

OWL-S is a web service describing language based on ontology. OWL-S document contains some useful control flow and data flow information, but they all are hidden in the descriptive level document. In order to analyze and capture some useful information, it is needed to introduce a mechanism so as to transform OWL-S document to an EH-CPN in constructive way. In this mechanism: (1) all input variables and output variables are represented by *color tokens*, where each variable has its own *color*; (2) the services are transformed into *transitions*; (3) all input and output states are transformed into *places* containing *tokens*; (4) the precondition is represented by a *condition checking function* in condition controlling transition or a *guard function*; (5) the effect is represented by an *output arc*.

In OWL-S document, some outputs are conditional output, which means the output will contain different variables according to different conditions. To transform these outputs into corresponding places, we use a *condition controlling transition* to follow the place to control different outputs. We also change the condition expression to condition checking function and put the function into the condition controlling transition. In this way, different outputs will be dispatched to different places. The control flow and data flow relations will be captured by *connecting arcs, transitions* or *arc expressions*.

In OWL-S specification, the service process is divided into three kind of forms, where the atomic process and the composite process can be invoked, but the simple process can not be, and therefore the transformation of simple process isn't needed. So we only analyze atomic process and composite process in next sections.

3.1 Atomic process transformation

The atomic process describes the process of single service that means it can not be divided again. It also has no sub-services. When the input satisfies the firing rule, the process will be invoked and corresponding output will be produced. The construction of the atomic process of sub net is more complex than that of the upper net, because sub net must be connected to its supper net based on some conditions. In order to check whether the input tokens coming from upper net is conformance to the tokens required by sub net, we add an *interface transition*. In this transition there is an *interface checking function* used to check the input. But we do not need this transition in non-sub nets.

In this paper, the atomic processes are divided into four types according to the input and output: (1) *Input coming from single net*. In this case, the input only comes from upper net. The transformation refers to Figure 1(a). (2) *Input coming from multi-nets*. In this case, the input comes from the upper net and the local net. The transformation refers to Figure 1(b). (3) *Non-conditional output*. The transformation of the output part is the same as the output part of Figure 1(a). (4) *Conditional output*. We add condition con-

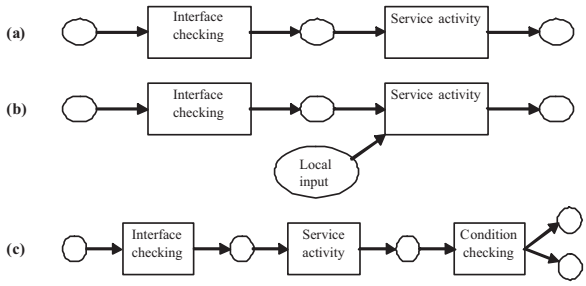

Figure 1. The construction of atomic process

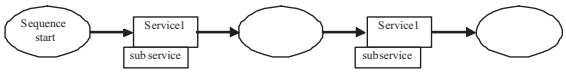

Figure 2. Sequence structure

trolling transition to dispatch different output according to the result of condition checking. The transformation refers to Figure 1(c).

3.2 Composite process transformation

A composite process can be decomposed into some atomic processes and/or other smaller composite processes. If we organize atomic processes or composite processes in a certain order using some control constructs, we will have new web services for providing new functions. The control constructs used in composite process include: sequence, split, split+join, choice, any-order, if-then-else, iterate, repeat-while, and repeat-until etc. Now we discuss how to transform composite processes to EH-CPN in detail.

Sequence: The processes contained in this construct will be invoked sequentially. The transformation refers to Figure 2

Split+join: In this construct, the concurrent processes are described too. All the processes have not only the same precursor but also the same subsequence. When all the concurrent processes end, they enter next state at the same time. We use a *concurrent controlling transition* to coordinate this kind of synchronization. The transformation refers to Figure 3

If-then-else: Three properties i.e., *ifCondition*, *then* and *else*, and two kind of services components are contained

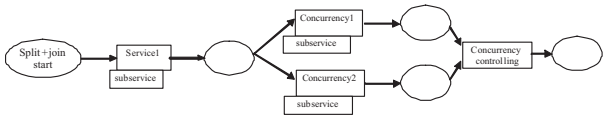

Figure 3. Split+Join structure

in this construct; If $ifCondition$ is true, the service in *then* branch will be executed; otherwise, the service in *else* branch will be executed. In our transformation, we map the property $ifCondition$ to a *condition controlling transition* in EH-CPN to dispatch different states.

Repeat-while: In this construct, one *testing condition* and one *loop-process* are contained. It tests the condition, then, does the loop-process if the result is true, exits else. So the loop-process is not executed if the condition is false. In our transformation, the testing condition is mapped to *condition checking function*, and then the function is put into a *condition controlling transition* to display this construct in EH-CPN.

Repeat-Until: This construct contains one testing condition and one *loop-process*, which is the same as Repeat-While construct. But there is a little difference of the execution process between them, it executes the loop-process first, then checks the condition, later the loop continues if the condition is true, exits else. So the loop-process will be executed at least once anyway. We also map the testing condition to a *condition checking function* and put it into *condition controlling transition* to display this construct in EH-CPN.

Any-order: This construct contains a list of processes which will be invoked in any order except for concurrency. All the processes must be executed at least once.

Using the above mechanism, we transform the control constructs in OWL-S document to EH-CPN in a constructive way. So the EH-CPN can intuitively describe the *control flow* of one process. But *data flow* is not very obvious.

4 Data flow in EH-CPN

In this section, we will discuss how to capture data flow and control flow in the EH-CPN.

4.1 Notations and definitions

Firstly, we need to clarify some useful notations and definitions that will be used in our analysis.

(1) During the transferring process from transition T_i to place P_j, the transition T_i will send some tokens $(x_1, x_2, ..., x_n)$ to its subsequence place P_j. In EH-CPN, these tokens are the interactive output of transition T_i and are defined in place P_j. We mark this relation as $T_i \cdot P_j(x_1, x_2, ..., x_n)$ in EH-CPN.

(2) During the transferring process from place P_i to transition T_j (non condition-controlling transition), the transition T_j will receive all needed tokens, which form the interactive input of transition T_j. Obviously, they are computation-use (or c-use) in T_j, and this relation is marked as $P_i \cdot T_j(x_1, x_2, ..., x_n)$ in EH-CPN.

(3) If transition T_j is a condition-controlling transition, it will judge all the tokens $(x_1, x_2, ..., x_n)$ from pre-place P_i. Obviously, these tokens are predicate-use (p-use) in T_j and this relation is also marked as $P_i \cdot T_j(x_1, x_2, ..., x_n)$ in EH-CPN.

(4) If a token (i.e., a variable) x is defined in place P_i and used in transition T_j (c-use or p-use), we call (P_i, T_j) a *define-use pair* of token x and mark it as $(P_i, T_j)_x$ in EH-CPN.

(5) A *path segment* in EH-CPN is defined as a sequence which is composed of places and transitions and marked as $PATH = (P_i, T_i, ..., T_j P_j, ...)$, where, $P_i, P_j \in P$, $T_i, T_j \in T$. If token v is defined in place P_i and used in transition T_j, we define $PATH(v)$ as a *define-use path segment* for v. If token v is defined only in a place P_i of $PATH(v)$ and no redefinition in any other places, we define this path as *def-clear-path segment* for v.

(6) From first three items, we know that all tokens are the output of their pre-transition, and meanwhile the input of their subsequence transitions. If a token v is defined in place P as the output of the pre-transition of P (marked as O) and is used in transition T as the input of the transition T (marked as I). Place P and transition T are in the same path (marked as $PATH_{PT}$). We define this path as the output-input-define-use chain (*OI-du-chain*) for token v and mark this relation as $(O, PATH_{PT}, I)$.

If a token is used in a transition, it will be consumed, so a token can be used only once in one process. Different services will produce different outputs, so every token will be defined only once according to first item. Therefore, we can conclude that every *OI-du-chain* is a *define-clear-path segment*.

(7) For a given set of test data, if there is a path in EH-CPN, each transition in this path will be triggered sequentially according to the order in path and arrive at final designated position. We regard this path as an *executable path*.

(8) There are two kinds of special positions in Petri Net. One is the position that has no output-edges; another is the position with *end* label. Both of them are regarded as *end position* in EH-CPN.

4.2 Data flow analysis

By analyzing the incidence matrixes of EH-CPN, we will have some useful data flow information for test case generation.

(1) we will have the *define-use pairs* of all tokens. On one hand, the post-incidence matrix records tokens which are produced after transitions have been fired. So a token v is defined at the place where token appears for the first time and this place should be added in the *define-use pair* of token v. On the other hand, the pre-incidence matrix records tokens which are needed to fire transitions. So the transition which requires token v for the first time and this transition should be added in the *define-use pair* of token v.

(2) After that, we can use those *define-use pairs* to find all *define-use-paths*, further we will have all *OI-du-chains* which contain all kinds of data flow information. In the *OI-du-chain*, we can see which services are affected by a certain variable. If we find all the *OI-du-chain*, we can get all the interaction influence between services.

In next section, we will discuss how to generate test case and illustrate how to get *OI-du-chain* in detail.

5 Test case generation

In EH-CPN based approach, test cases are generated in three phases: we will discuss how to produce test sequence in phase 1, then we discuss how to prepare test data in phase 2, finally we discuss how to generate test cases by combining test sequences and test data.

Phase 1: generation of test sequence Test sequences is generated according to following steps:

Step 1: Preprocessor of EH-CPN The main work is to identify the concurrent modules and modify them. In EH-CPN, the concurrent module is described in *split+join* structure. In this structure, there is a transition whose in-degree is one and out-degree is bigger than one. We call this transition *split transition*. There is a *synchronization controlling transition* whose out-degree is one and in-degree is bigger than one. Every concurrent module begins with a *split transition* and ends with a *synchronization controlling transition*. So we take this kind module as a black-box and use one transition to replace the module whose input is the input of *split transition* and whose output is the output of *synchronization controlling transition*. The algorithm for identifying all concurrent modules is omitted because the space limitation.

Step 2: OI-du-chain generation OI-du-chain is composed of three parts: O, I and $PATH$. By analyzing incidence matrixes of EH-CPN, we can find all define-use pairs, and further we can determine the O and I for every token. The algorithm for computing the $PATH$ of *OI-du-chain* consists of two phases: (1) the algorithm is used to find a sequence that begins with a transition where token v is used and ends with a place where the token v is defined; (2) the algorithm is used to reverse this sequence generated in (1). By this way, we get a sequence which is the path of the *OI-du-chain* for variable v.

Step 3: Pre-sequence and post-sequence generation To let the path of *OI-du-chain* be an executable path, we need to extend it with a *pre-sequence* and a *post-sequence*.

The computation of *pre-sequence* is easy, we can use the algorithm in step 2 to compute it as long as we use the start node of the EH-CPN and the first node in the path of *OI-du-chain* as the two input parameters respectively.

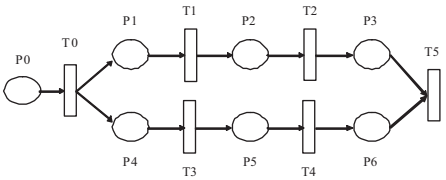

Figure 4. concurrent module

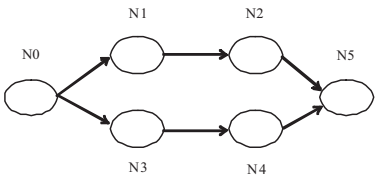

Figure 5. action graph

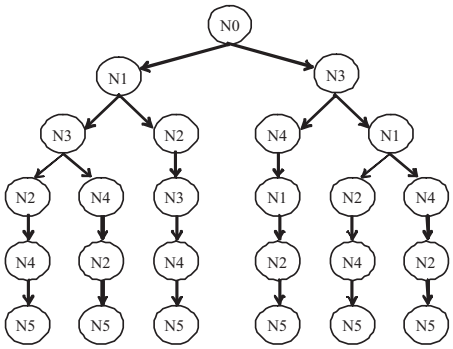

Figure 6. test sequence tree

Post-sequence can be found as follows: firstly, we will find every subsequence node starting from the last node of the path of the *OI-du-chain* until we arrive at *end position*, and then, collect all the subsequence nodes orderly and a post-sequence will be found. If a node has more than one post-sequence, this way will find all the post-sequences.

Because EH-CPN is a hierarchical Petri-net, it is likely that a complete path, consisting of the path segment of one *OI-du-chain* and its *pre-sequence* and *post-sequence*, will contain some transitions to sub nets. In this case, it is necessary to replace those transitions with a new path in their sub nets using above steps 1-3. The algorithm will repeat this replacement process until the path segment of one *OI-du-chain* has been found, where transitions do not contain sub nets. In our EH-CPN based method, the path segment of *OI-du-chain* with its pre-sequence and post-sequence altogether are regarded as a *test sequence*.

Step 4: Test sequence generation for concurrent module After steps 1-3, we have got a test sequence, but this sequence is generated based on modified EH-CPN in step 1. where we just regarded concurrent module which includes many transitions and places as a transition simply for easy to deal with. If we find a sequence for real concurrent modules, we will have a complete and precise sequence based on the primary EH-CPN. In order to get test sequences from concurrent modules, we can do as follows: (1) we combine every transition and its pre-places into one node, maintaining relations between transitions unchanged. In this way, the concurrent module has only one kind of nodes and we name this net as *action graph*.(2) we construct test sequence tree. The tree contains all the concurrent test sequences. (3) one path which is from root to one leaf is a *test sequence*.

The following process illustrates how to transfer an *action graph* into a *test sequence tree*: (1) Make the node which is composed of *split transition* and its pre-place be the root of a *test sequence tree*; (2) Delete above nodes and their post arcs in *action graph*. The nodes which will be deleted are in a path from root to node i in the $k^{th}(k \geq 0)$ level; (3) Find nodes which have no direct precursor and let them be the children of node i.

Figure 4 shows the concurrent module in EH-CPN and Figure 5 is an action graph of Figure 4, where we can see that some transitions and places in Figure 4 have been united as a new node in Figure 5. For example, $P1$ and $T1$ are united as node $N1$.

Figure 6 is a test sequence tree coming from Figure 5. $N0$ is composed of split transition and its pre-places. So it is the root of the test sequence tree. If we delete $N0$ and its post arcs, we will find $N1$ and $N3$ with no precursor. So $N1$ and $N3$ are children of $N0$ according to rule 2.

In Figure 6, the test sequence tree has six leaves, so it has six test sequences. If we map nodes in one test sequence of test sequence tree into corresponding places and transitions in EH-CPN, we will get the test sequence of concurrent modules. For example, ($N0$, $N1$, $N3$, $N2$, $N4$, $N5$) can be mapped to ($P0$, $T0$, $P1$, $T1$, $P4$, $T3$, $P2$, $T2$, $P5$, $T4$, $P6$, $P3$, $T5$). After generating test sequence of concurrent module, we use these sequences to replace the corresponding modified transitions in step 1. In this way, we will get test sequences based on primary EH-CPN.

Step 5: Executable test sequence generation If there are loops in EH-CPN, the test sequence is likely non-executable. Because it is impossible to know how many times the loop will execute exactly. In our EH-CPN based way, we borrow the heuristic method, which is proposed by C. Bourhfir[7], to solve this problem by finding an appropriate loop and inserting it into the non-executable test sequence to generate an executable sequence. Now we get executable test sequences satisfying `ALL-DU-PATH` criterion.

Phase 2: preparation for test data The main idea to generate test data is originated from the XPT method

(XML-based Partition Testing) method in [8], which is consisting of three parts: (1)map XML Schema which defines the structures and data types of input and output of all the web services to *Category Partition*. In this way, a set of final instances and intermediate instance frames have been got; (2) find all the preconditions which are included in the services in one test sequence and do AND operation on all those preconditions to get the value domain of the instances; (3) generate the test data using random methods.

Phase 3: generation of test case Test cases in EH-CPN based way is the combination of test data and test sequence, where test sequence can be generated in section 5.1 and test data can be got using the way in section 5.2. In EH-CPN based way, the test sequence is only composed of all the web services contained in a test sequence generated in section 5.1. Test cases are coded in an XML file and can be used as an input of a test tool.

6 Conclusion

There are many methods have been proposed to generate test cases for web service. These methods can be parted into two basic categories: one can generate test cases based on specifications, the other can generate test case based on model checking, such as [1], [2],[4], [5], [6], [9], [10], and [11] etc. In this paper, we introduced a method to generate test cases based on a kind of extended hierarchical colored Petri Net, where we transform OWL-S to EH-CPN for capturing more control flow and data flow information so that we can generate more precise test case. But the problem is if there are too many services in one concurrent module, the state explosion problem rises, so it is necessary to find an effective method to solve state explosion problem and improve our method in future work.

Acknowledgement

Bixin Li is now with University of California at Riverside as a visitor scholar and he thanks Prof. Rajiv Gupta in University of California Riverside for providing a very comfortable Lab. This work is partially supported by the National Nature Science Foundation of China under No.60773105, partially by the Natural Science Foundation of Jiangsu Province of China under Grant No.BK2007513, and partially by National High Technology Research and Development Program under Grant No. 2008AA01Z113.

References

[1] X. Y. Bai, W. L. Dong, W. T. Tsai, and Y. N. Chen. *WSDL-Based Automatic Test Case Generation for Web Services Testing*. Proceedings of the 2005 IEEE International Workshop on Service-Oriented System Engineering (SOSE'05).on 20-21 Oct. 2005 Page(s):207-212

[2] Y. B. Wang, X. Y. Bai, J. Z. Li, and R. B. Huang. *Ontology-Based Test Case Generation for Testing Web Services*. Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07).on 21-23 March 2007 Page(s):43-50.

[3] H. M. Sneed and S. H. Huang. *WSDLTest-A Tool for Testing Web Services*. Eighth IEEE International Symposium on Web Site Evolution, 2006. Sept. 2006. Page(s):14-21

[4] Y. P. Yang, Q. P. Tan, Y. Xiao, J. S. Yu, and F. Liu. *Exploiting Hierarchical CP-Nets to Increase the Reliability of Web Services Workflow*. In: Proceedings of the 2005 Symposium on Applications and the Internet (SAINT'06). on 23-27 Jan. 2006 Page(s):7 pp.

[5] Y. P. Yang, Q. P. Tan, J. S. Yu, and F. Liu. *Transformation BPEL to CP-Nets for Verifying Web Services Composition*. Proceedings of the International Conference on Next Generation Web Services Practices (NWeSP'05). On 22-26 Aug. 2005 Page(s):6 pp.

[6] S. Y. Wang, P. Yu, J. J. Huo, and C. Y. Yuan. *Petri Nets for Systems Engineering*. Publishing House of Electronics Industry.2005.

[7] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. *Automatic executable test case generation for extended finite state machine protocols*. In Proceedings of IWTCS'97 [17], pages 75-90.

[8] B. Antonia, J.H. Gao, M. Eda, and P. Andrea. *Automatic Test Data Generation for XML Schema-based Partition Testing*. Automation of software Test 2007. Second Internation Workshop on 20-26 May 2007 page(s):4-4.

[9] D. Martin, A. Ankoleka. *CongoProcess.owl document*. http://www.daml.org/services/owl-s/1.2/

[10] Y. Y. Zheng J. Zhou P. Krause. *A Model Checking based Test Case Generation Framework for Web Services[*. Fourth International Conference on Information Technology (ITNG'07). on 2-4 April 2007 Page(s):715 - 722.

[11] L. Hua, and Y. X. Ming *Generation Executable Test Sequence Based on Petri-net for Combined Control and Data Flow of Communication Protocol*. International Conference on Communication Technology. On 22-24 October,1998 Page(s):S48-02-1 - S48-02-5