1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

A Petri Net Model for Object-Oriented Class Testing

Chun-Chia Wang, Wen C. Pai, and Jung D. Chiang

Department of Information Management
Kuang Wu Institute of Technology and Commerce
PeiTou, Taipei, Taiwan 112, R.O.C.
phone: Intl. (02) 28927154 ext. 852
Email: CCWang@mine.tku.edu.tw

Abstract

In an object-oriented model, a class is considered to be a basic unit of testing. Methods (member functions) of a class can have different types of inter-method relationships. The causal relationships between methods specify the sequence in which the methods can be executed. In this paper, we propose a petri net model for specifying this causal relationship and present a test case generation technique based on petri nets. In our technique, petri nets are transformed into a reachability tree from which we generate test cases using the paths of the tree.

1 Introduction

Recently, the object-oriented paradigm is gaining acceptance for developing software. However, most research and practical experience in the object-oriented field has been in the analysis, design, and implementation phases of a tipical software life cycle, little work being carried out to examine the effects using object-oriented techniques has on the rest parts of the software life cycle such as testing and metrics. Although many procedure-oriented software testing techniques have been proposed [6, 7], these conventional methods, despite their efficiency, cannot be applied without adaptation to object-oriented systems due to new data and control abstractions such as inheritance, polymorphism, and message passing mechanisms introduced. In order to analyze programs coded in object-oriented programming languages, there is a need for developing object-oriented software testing techniques.

In an object-oriented model, the fundamental entities are objects, rather than procedures and functions as in imperative programming languages. Objects with similar data and operations can be specified by their common class. A class describes all the instance methods and attributes of these objects. It is considered to be a basic unit of testing in object-oriented testing literature. The majority of object-oriented class testing techniques generate test cases based on either algebraic specifications or model-based specifications [4, 8, 9, 15]. In this paper, we illustrate the application of petri nets (PNs) for the class testing of object-oriented designs. A major feature of classes provides the desired behavior by interacting with the data representation. The behavior of methods is determined by the values of attributes and in turn, methods manipulate attributes. This interaction can be properly modeled with PNs. Each PN can be implemented as a class; attributes represent the allowable places of a class and methods represent a change of places, i.e., a transition of PNs. Therefore, many object-oriented analysis and design (OOA/OOD) methods using *petri net* diagrams as a specification of classes have been proposed in recent years. Although the main function of an OOD is a state-transition diagram, PNs in OOD methods lack formal semantics to enable automated test cases generation. In this paper, we propose a technique that transforms the state-transition diagram of an OOD into a PN model and construct a corresponding *reachability tree* to the PN. Based on the paths (from the root to the terminal nodes) of the tree, we generate test cases for the class testing of object-oriented designs.

This paper is organized as follows: Section 2 discusses definitions and basic properties of petri nets. Section 3 introduces class specification based on finite-state machine (FSM), and then transforms the FSM into an equivalent class petri net machine (CPNM). Section 4 describes our testing technique based on the CPNM. Section 5 compares our work with related works. Conclusion and future works are given in section 6.

2 Definitions and Basic Properties of Petri Nets

Petri nets (PNs) [5, 10] are a graphical and mathematical modeling tool applicable to many systems. As a graphical tool, Petri nets can be used as a visual communication aid similar to flow charts, block diagrams, and networks. This section gives a formal definition of PNs, and introduces a notation of state in order to discuss behavioral analysis problems for PNs.

Definition 2.1. A *Petri Net* (PN) is a bipartite graph, i.e., a graph whose set of nodes is partitioned into two subsets, called the set of *places* $V = \{P_1...P_n\}$ and the set of *transitions* $T = \{t_1...t_m\}$, each arc connecting only nodes of different types. It is shown in figure 1.

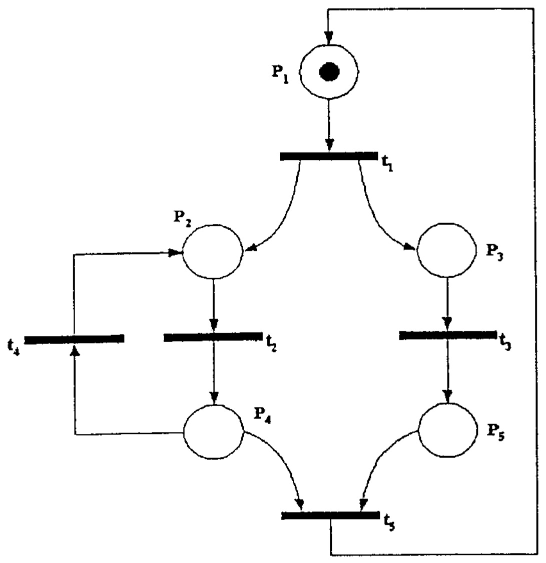

Figure 1: Example of a marked PN

Definition 2.2. A *marking* (or *state*) of a PN is a mapping $M : V \rightarrow N$ (numerical numbers); numerically it is represented by a vector with $|V|$ nonnegative components and graphically by $M(P_i)$ *tokens* in the place P_i.

Definition 2.3. A transition t_i is *firable* for a given marking M if, for every place P_i, $M(P_i) \geq$ (Number of arcs connecting P_i to t_i). A marking M' is reached from M by firing t_i if t_i is firable for M and, for every P_j,

$M'(P_j) = M(P_j)$ - (Number of arcs connecting P_j to t_i) + (Number of arcs connecting t_i to P_j).

Definition 2.4. A marking M' is said to be *immediately reachable* from M if M' can be obtained by firing a transition enabled in M. The PN execution allows a sequence of markings $\{M_0, M_1, M_2, ...\}$ and a sequence of transitions $\{t_1, t_2, ...\}$ to be defined. The firing of t_1, enabled in M_0, changes the PN state from M_0 to M_1 and so on. A marking M'' is said to be *reachable* from M if there exists a sequence of transition firings that moves the PN state from M to M''.

Definition 2.5. A *reachability set* $R(M_0)$ of a PN as the set of all markings that are reachable from M_0. The reachability set of a PN can be represented by a tree. It can be shown that the tree in figure 2 represents the reachability set of the PN in figure 1.

To represent a finite reachability tree, we stop to expand the tree when we reach an already considered marking. We shall call such markings duplicate nodes.

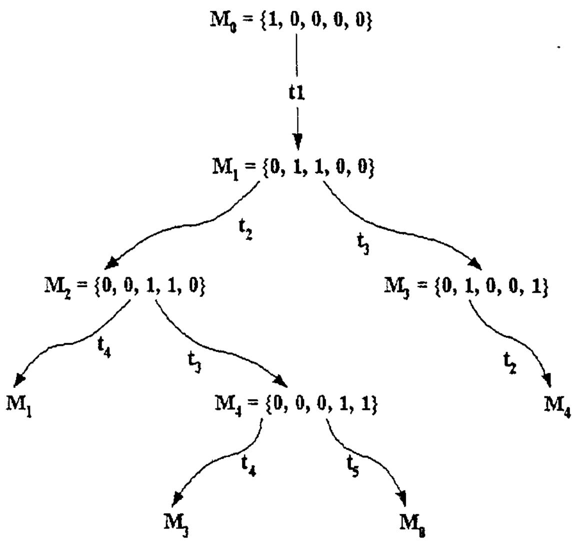

Figure 2: The reachability tree of the PN in figure 1

During the construction of the reachability tree, we can also find *dead* markings, i.e., marking in which no transition is enabled. Dead and duplicate markings constitute the frontier markings (leaves) of the tree.

3 Class Petri Net Machine

3.1 A Finite State Machine for Class Specification

Many OOD techniques propose FSMs for modeling the dynamic behavior of classes [12, 13, 14]. The FSM model represents all possible states of the class, the events that can cause state transitions, and the actions that result from the state change. Different states of an object are due to different value combinations of its attributes. The states are linked to one another through state transitions. State transitions are caused by events, which are the stimulus received from the environment. At each state, all the events are unique and events can be received in any state. The state transition depends on the current state and the message received. Thus, the class specification can be derived automatically from the FSM of the class. For the sake of deriving class specification, we use only the events different from the FSM model of OOD. The FSMs can be modeled as 5-tuple $M = (Q, \Sigma, \delta, s, f)$, where Q is a finite set of states, Σ is the set of methods accessible from the outside environment, δ is the transition function mapping $Q \times \Sigma$ to Q, $s \in Q$ is an initial state corresponding to the state after a class receives a *constructor* message, and $f \in Q$ is a final state corresponding to a class receiving a *destructor/delete* message.

3.2 Transforming FSM into CPNM

In the analysis of petri net models, there are two subclasses of petri nets commonly considered, state machines (FSMs) and marked graphs [1]. FSMs are

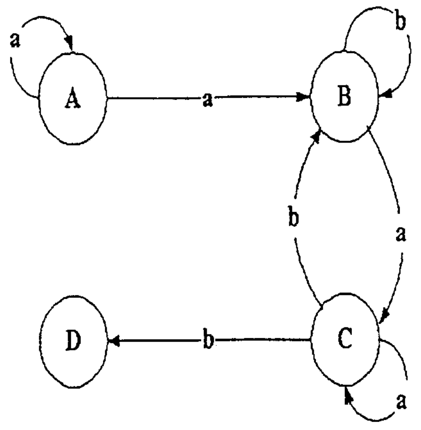

Figure 3: A FSM model

petri nets which are restricted so that each transition has exactly *one* input and *one* output. These nets are obviously finite-state. In fact, they are exactly the class of finite-state machines. This is clearly shown by considering the state graph of a finite-state machine, as in figure 3. The nodes of this graph represent the states of the finite-state machine. An arc from state i to state j labeled x indicates that there is a transition from state i to state j with input x. The graph of figure 3 can be converted to an equivalent petri net by simply making each state a place, and making each arc between two places a transition. This is illustrated in figure 4.

As an example of a finite-state machine [10], consider a vending machine which accepts either nickels or dimes and sells 15c or 20c candy bars. For simplicity, sppose the vending machine can hold up to 20c. Then, the state diagram of the machine can be represented by the petri net shown in figure 5, where the five states are represented by the five places labeled with 0c, 5c, 10c, 15c, and 20c, and transformations from one state to another state are shown by transitions labeled with input conditions, such as "deposit 5c" The initial state is indicated by initially putting a token in the place p_1, with a 0c label in this example. Note that each transition in this net has exactly one incoming arc and exactly one outgoing arc.

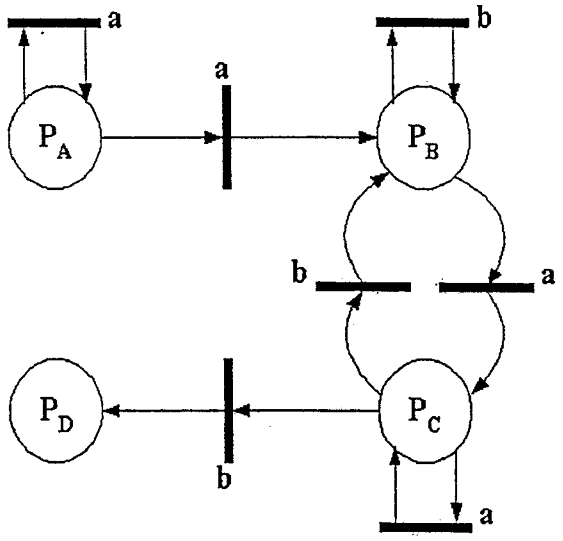

Figure 4: A CPNM model corresponding to figure 3

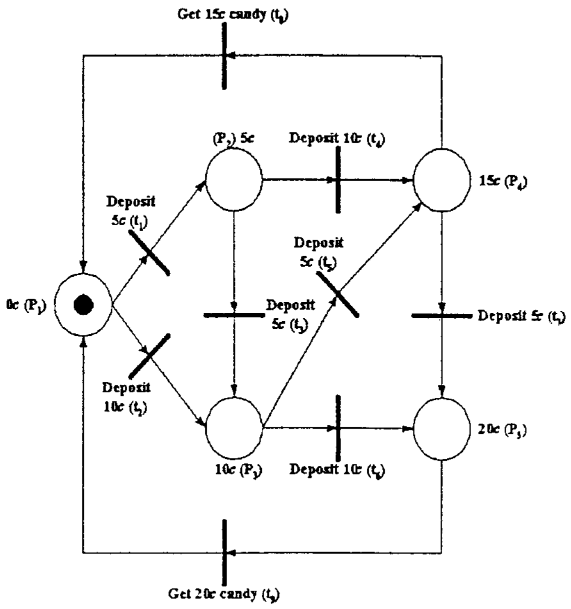

Figure 5: A petri net representing the state diagram of a vending machine, where coin return transitions are omitted

4 Test Cases Generation from CPNMs

Software testing is used for ensuring the correctness of a program against its specification by executing a program on a test case and comparing the result with the expected result [3]. The steps involved in testing are as follows:

1. generation of test cases for an identified portion of software
2. execution of the program using the test inputs, and

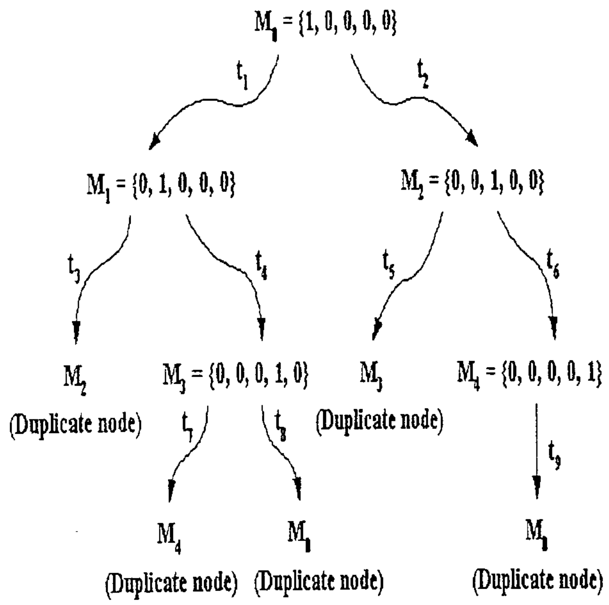

Figure 6: Reachability tree diagram of a class used for generating test cases

3. evaluation of the test results

In our technique, we focus on test case generation from the reachability tree of the CPNM of a class.

In an object-oriented program, testing a class corresponds to testing the methods supportd by the class. Individual methods are similar to conventional procedures. Therefore, methods can be tested similarly to conventional procedures using black-box and white-box testing. In an object-oriented paradigm, there is extensive interaction between the methods of a class. These method interactions must be tested for correctness. In this section, we discuss generating test cases for the method sequences of a class using *coverage criteria*.

In figure 6, a reachability tree diagram corresponding to the figure 5 is shown. Using different coverage criteria, different sequences can be constructed for generating test cases. To satisfy the *all-edge* coverage criteria, five sequences (paths of the tree, i.e., $R(M_0)$) $M_0 \cdot M_1 \cdot M_2$ $(t_1 \cdot t_3)$, $M_0 \cdot M_1 \cdot M_3 \cdot M_4$ $(t_1 \cdot t_4 \cdot t_7)$, $M_0 \cdot M_1 \cdot M_3 \cdot M_0$ $(t_1 \cdot t_4 \cdot t_8)$, $M_0 \cdot M_2 \cdot M_3$ $(t_2 \cdot t_5)$, and $M_0 \cdot M_2 \cdot M_4 \cdot M_0$ $(t_2 \cdot t_6 \cdot t_9)$ can be constructed and test cases corresponding to them can be generated.

For each method in the sequence, data input of each parameter and the expected output must be determined. If a method invokes methods of other classs, then during testing test-stubs may be required to supply the proper return values from those methods. Sequence-based testing of a class can be performed either during design as a walkthrough or implementation by actually executing the program with the test cases. Because sequence-based testing depends on the correctness of individual methods, the testing of individual methods must precede testing the sequences. Those paths of the reachability tree helps determine the interactions between methods of a class.

5 Related Works

In general, PN models of software are applied to many areas of software engineering, such as test case generation, software complexity measure, etc. In object-oriented testing literature, several techniques have been proposed for class testing. Most of them is a specification-based technique using either algebraic specifications or model-based specifications. An algebraic specification consists of signatures which define the syntactic properties and axioms which define the properties of member functions. A model-based specification specifies the pre-condition and post-condition of each member functions using well-defined mathematical models such as sets or sequences.

In [4, 8] test cases are generated as sequences of member functions based on the axioms in algebraic specifications. However, in algebraic specifications member functions are treated as a mathematical mapping without side-effects. Therefore, data flows between data members and member functions cannot be explicitly represented in these techniques. On the other hand, in [9, 15] test cases are generated based on the pre-condition and post-condition of each member function. Though member functions are tested whether or not they use and define data members correctly, data flows between member functions are not considered in these testing techniques.

Harrold and Rothermel [16] proposed data flow testing techniques for classes. They identity three levels of class testing: (1) intra-method testing which tests member functions individually, (2) inter-method testing which tests a member function together with other member functions that it calls, and (3) intra-class testing which tests the interactions of member functions when they are called in various sequences. To support each data flow in the three levels, they construct a flow graph which represents every possible sequences of member functions from the class's code. Then they generate test cases using inter-procedural data flow testing techniques.

Our technique is specification-based, i.e., we specify the behavior of classes using PNs and a reachability tree is constructed from PNs. There exist some of feasible paths are determined by PNs. Therefore, the works in [16] and our work are complementary each other.

6 Conclusion

In this paper, we have proposed a class testing technique for specifying the causal relation that exists between methods of a class in object-oriented designs. In our technique, we illustrated a finite-state machine model of a class, then transformed the FSM into an equivalent class petri net machine (CPNM). In order to test the behavior of classes effectively, we have generated test cases by transforming CPNM into a reachability tree of the model and applying all-edge testing technique upon the paths of the tree. We are currently working on integrating class specification with an object-oriented CASE tool. We are also working on verifying the class testing application and usefulness.

References

[1] A.W. Holt, and F. Commoner, *Events and condition,* Applied Data Research N.Y., 1970; also in *Record Project MAC Conference Concurrent Systems and Parallel Computation,* (Chapters I, II, IV, and VI) ACM, pp. 3-52, N.Y., 1970.

[2] T. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. on Software Eng.*, Vol. SE-4, No. 3, pp. 178-187, May 1978.

[3] G. Myers, *The Art of Software Testing,* Prentice Hall, Englewood Cliffs, NJ, 1979.

[4] J. Gannon, P. McMullin, and R. Hamlet, "Data Abstraction Implementation, Specification, and Testing," *ACM Trans. on Programming Languages and Systems*, Vol. 31, No. 3, pp. 211-223, 1981.

[5] J.L. Peterson, *Petri Net Theory and The Modeling of Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[6] S.C. Ntafos, "On Required Element Testing," *IEEE Trans. on Software Eng.*, Vol. SE-10, No. 6, pp. 795-803, November 1984.

[7] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. on Software Eng.*, Vol. SE-11, No. 4, pp. 367-375, April 1985.

[8] L. Bouge, N. Choquet, L. Fribourg, and M.C. Gaudel, "Test Sets Generation from Algebraic Specifications Using Logic Programs," *Journal of Systems and Software*, Vol. 6, pp. 343-360, 1986.

[9] I. Hayes, "Specifiaction Directed Module Testing," *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 1, pp. 124-133, January 1986.

[10] Tadao Murata, "Petri Nets: Properties, Analysis, and Applications," *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 541-580, April 1989.

[11] B. Beizer, *Software Testing Techniques*, Van Nonstrand Reinhold, 1990.

[12] G. Booch, *Object Oriented Design with Applications*, Benjamin Cummings, 1991.

[13] B. Bosik and M.Ü. Uyar, "Finite State Machine Based Formal Methods in Protocol Conformance Testin: from Theory to Implementation," *Computer Networks and ISDN Systems*, Vol. 22, pp. 7-33, 1991.

[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[15] S. Zweben, W. Heym, and J. Kimich, "Systematic Testing of Data Abstraction Based on Software Specifications," *Journal of Software Testing, Verification, and Reliability*, Vol. 1, pp. 39-55, 1992.

[16] M.J. Harrold and G. Rothermel, "Performing Data Flow Testing on Classes," in *Proceedings of the second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 154-163, December 1994.

[17] S. Kirani and W.T. Tsai, "Method Sequence Specification and Verification of Classes," *J. Object Oriented Programming*, pp. 28-38, October 1994.