A Methodology of Testing High-Level Petri Nets

Hong Zhu
School of Computing and Mathematical Sciences
Oxford Brookes University
Wheatley Campus
Oxford OX3 0BP, UK
Email: hzhu@brookes.ac.uk

Xudong He
School of Computer Science
Florida International University
University Park, Miami,
FL 33199, U.S.A.
Email: hex@cs.fiu.edu

Abstract

Petri nets have been extensively used in the modelling and analysis of concurrent and distributed systems. The verification and validation of Petri nets are of particular importance in the development of concurrent and distributed systems. As a complement to formal analysis techniques, testing has been proven to be effective in detecting system errors and is easy to apply. An open problem is how to test Petri nets systematically, effectively and efficiently. An approach to solve this problem is to develop test criteria so that test adequacy can be measured objectively and test cases can be generated efficiently, even automatically. In this paper, we present a methodology of testing high-level Petri nets based on our general theory of testing concurrent software systems. Four types of testing strategies are investigated, which include state-oriented testing, transition-oriented testing, flow-oriented testing and specification-oriented testing. For each strategy, a set of schemes to observe and record testing results and a set of coverage criteria to measure test adequacy are defined. The subsumption relationships and extraction relationships among the proposed testing methods are systematically investigated and formally proved.

Keywords: Software Testing methods, Concurrent systems, High-level Petri nets, Test criteria, Behaviour observation

1. Introduction

Since 1970s, Petri nets have been extensively used in the modelling and analysis of concurrent and distributed systems. Although there are several formal analysis techniques of Petri nets such as coverability tree (or graph) technique and invariant techniques, formal verification and validation are not always applicable or effective, and are often very difficult to use. On the other hand, testing has been proven to be effective in detecting system errors and is easy to apply. We believe that a testing technique for Petri nets can be a cost-effective approach complementing other more formal analysis techniques in revealing errors in Petri nets. An open problem is how to test Petri nets systematically and effectively. An approach to solve this problem is to develop test criteria so that test adequacy can be measured objectively, test cases can be generated efficiently even automatically, and testing processes can be controlled effectively. It is the theme of this paper.

Generally speaking, testing methods can be classified into *program-based*, which select test cases according to the information contained in the program, and *specification-based*, which derive test cases from the requirements specification. Petri nets can play two different roles in the development of concurrent systems. A Petri net can be used as a formal specification of a concurrent system. Testing a concurrent system against a Petri net belongs to the catalogue of specification-based methods. In the past few decades, a great amount of research has been reported in the literature on specification-based testing methods. There are works on derivation of test cases from algebraic specifications [1, 2, 3, 4, 5, 6], Z specifications [7, 8, 9], finite state machines *cf.* [12], and other specification languages such as Estelle, LOTOS, and SDL in conformance testing of communication protocols [16]. On the other hand, a Petri net can be considered as an executable model of a concurrent system. It can also be tested against another specification. In this sense, a Petri net testing method also has the features of program-based testing. There are extensive literatures on program-based testing. Existing program-based testing methods include structural testing methods such as control flow testing and dataflow testing, fault-based testing methods such as mutation testing, and error-based testing. Readers are referred to [10] for a survey of researches on software testing methods.

However, there are very few works on testing concurrent systems, especially testing Petri nets. In [11], a hierarchy of test criteria of structural testing of concurrent programs was proposed. The criteria were defined on the basis of concurrency graphs. A concurrency graph is a directed graph whose nodes represent concurrency states and edges represent state transitions. An edge from concurrency state *A* to *B* represents that the execution of the program can progress directly from state *A* to state *B*. A concurrency state is a vector of the synchronisation related nodes in the flow graph of the concurrent tasks of the program. The strongest criterion in the hierarchy is the *all-concurrency-path* criterion, which requires that an adequate test of a concurrent program should cover all paths in the concurrency graph. It subsumes the *all-proper-cc-histories* criterion, which requires that an adequate test should cover all elementary paths (i.e. the paths on which no node occurs more than once) in the concurrency graph. The *all-edges-between-cc-states* criterion is less strict than the all-proper-cc-histories criterion. It requires an adequate test should cover all edges in the concurrency graph. An even less strict test criterion is the *all-cc-states* criterion, which requires that an adequate test should cover all states in the concurrency graph. These criteria resemble the path coverage, elementary path coverage, branching coverage and statement coverage criteria of control flow testing of sequential programs. An additional criterion, called the *all-possible-rendezvous* criterion, specific to Ada programs was also proposed, which requires that an adequate test should cover those particular nodes in the concurrency graph which involve a rendezvous. This criterion is, therefore, subsumed by the all-cc-states criterion.

State oriented testing methods can also be found in the research on testing finite state machines. The development of such testing methods has initially been driven by problems arising from functional testing of sequential circuits, and later re-boosted in the need of testing communication protocols [12]. More recently, such testing methods attracted much attention of researchers of object-oriented software testing, *cf.* [13]. Traditional testing methods based on finite state machines rely on the model of completely specified deterministic finite state machines [14]. Recently, more complex models of finite state machines have been studied such as communicating finite state machines [15]. Two types of testing problems have been studied in the theory of testing finite state machines. The first is the state identification and verification problem, which tests a finite state machine whose description is known to the tester. It intends to identify which state it is in or to verify it is in a certain state. The second is the conformance testing problem, which tests an implementation modelled by a finite state machine, whose description is unknown, against a specification which is modelled by another finite state machine whose description is known by the tester. It intends to verify that the implementation is isomorphic to the specification. For example, in the conformance testing of communication protocols, a finite state machine as implementation of a protocol is often tested against another finite state machine that serves as the specification [16]. A common feature of conformance testing methods is that they are fault-based, that is, they are targeted to detect or eliminate a specific type of faults, such as transition faults, and rely on a fault model. Test adequacy criteria are almost all defined (usually implicitly) as completeness with respect to certain fault models. For both state identification and verification testing problems and conformance testing problems, a common assumption is that the tester can only observe the input and output sequences of the finite state machine under test. Therefore, solutions of testing problems cannot be based on direct observations on which state the machine is in. Although these theories and methods are relevant to the testing of Petri nets, the characteristics of black-box testing of the testing problems and the simplicity of the computation model of finite state machine limited their applicability to solve complicated problems in testing concurrent software systems, such as

Petri nets. There are three fundamental differences between the models of finite state machines and Petri nets. First, Petri nets treat both state and state transitions equally, which thus can be dealt with explicitly in testing. Second, Petri nets define states distributedly while finite state machines define states globally. Third, the basic models of finite state machine are sequential while Petri nets are concurrent. Although, in recent years, research on testing communicating finite state machines starts to appear in the literature [15], the concurrent nature of such models has not been fully explored because of the state explosion problem.

In [17], a set of coverage criteria for ER nets (a type of high-level Petri nets) was proposed and their subsumption relations were proved. These criteria include: (1) *Firing Sequences*: an adequate testing must include all possible firing sequences from the given initial marking. A firing sequence is defined by control and data flows. This amounts to the exhaustive testing and is thus impractical; (2) *Firing*: each possible firing of any transition must belong to some firing sequence in test executions; (3) *Transition Sequences*: all possible transition sequences must be covered. A transition sequence concerns only the control flow. It may correspond to multiple firing sequences. Thus, a testing only needs to contain one of the firing sequences corresponding to the same transition sequence. The multiple firing sequences can be viewed as an equivalent class of a common transition sequence; (4) *N-Times*: an adequate testing covers only those transition sequences such that none of them contains the same transition more than n times; (5) *N-Notable*: an adequate testing covers only those transition sequences such that none of them contains the same transition more than n times and no more than one contains a notable subsequence of interest. Here, a subsequence γ of a firing sequence σ is called a *notable firing subsequence of* σ, if and only if, for all firing sequence σ' containing γ we have that that σ' has the same length as σ and σ' has the same initial marking as σ imply that σ' is a permutation of σ. A notable subsequence identifies a class of firing sequences that can be considered equivalent from the testing point of view. Thus, only one representative is included in testing from a set of firing sequences such that their underlying transition sequences contain the same notable subsequence; and (6) *Transition*: an adequate testing contains enough firing sequences that contain at least one firing of all transitions.

In the testing of concurrent systems, observing dynamic behaviour is of particular importance, but is more complicated than testing sequential systems. As pointed out by Carver and Tai [18], the non-deterministic behaviour of concurrent programs makes the replay of a testing process and regression testing uncertain. Observing various possible dynamic behaviours on test cases and controlling the executions of non-deterministic programs to demonstrate all possible behaviours have been a major research topic in testing concurrent systems [18]. Despite of the practical difficulties in observing and controlling the dynamic behaviours of concurrent systems, behaviour observations were used in the theoretical studies of process algebra, such as in defining equivalence relations between CCS processes [19]. This idea was developed into a formal framework for defining an equivalence relation based on testing [20] and further extended in [21]. These theoretical works only use the externally observable behaviours in the form of event sequences, and thus correspond to the practical methods of black-box testing.

Our work shares the same viewpoint with the above research that the semantics of a concurrent process can neither be simply defined nor adequately tested as a partial function from inputs to outputs although it is appropriate for a sequential program. Instead, it must be defined in terms of its dynamic behaviour and tested by observing the dynamic behaviour. Unfortunately, existing theories of software testing have mostly focused on test adequacy criteria [22, 23, 24, 25, 26, 27, 28, 29, 30], but neglected the aspect of behaviour observation in software testing. Many important and fundamental questions still remain unanswered. For example, what should be observed, i.e. in what sense an observation method is appropriate? What can be observed, i.e. what are the varieties of observation methods? What are the implications of using different observation methods and how to compare them? etc. In [31], we developed a general theory of testing concurrent systems to answer such questions. We argued that a well-defined testing method should contain at least two components, a method of observing a system's dynamic behaviour during a testing process and a criterion for selecting test cases and determining when testing can stop. We used complete partially ordered sets to formally define what are appropriate methods of behaviour observation and recording, and introduced the notion of observation schemes. We proposed and investigated the desirable properties that a scheme needs to satisfy. We also identified some common constructions of observation schemes in existing software testing methods and studied their properties [32]. Test criteria were defined as predicates or measurement functions on observed behaviour during testing.

The above work forms the basis of the work reported here. In this paper, the formal theory of observation scheme [31] is applied to testing high-level Petri nets. Four testing strategies are systematically investigated with various test adequacy criteria and behaviour observation schemes. The subsumption and extraction relationships among various proposed testing methods are proved. The paper is organised as follows. Section 2 gives the preliminaries of predicate transition nets, which is a kind of high-level Petri nets. Section 3 develops the work proposed in [17] into transition-oriented testing strategy. Section 4 abstracts and generalises the work of [11] into state-oriented testing strategy. Section 5 adapts and generalises data flow testing results from sequential software systems [33, 34, 35, 36] for testing Petri nets. Section 6 discusses specification-oriented testing. Section 7 is the conclusion of the paper.

2. Testing Predicate Transition Nets

The high-level Petri net model used in this paper is predicate transition net (PrT nets in the sequel) [37]. However, the results presented in this paper are also directly applicable to other high-level Petri net models as well as low-level Petri nets. In a PrT net, tokens can have structures and can be individually distinguished. Labels can be expressions containing variables, and constraints can be logical expressions. Thus, PrT nets are powerful enough to define control, data, functionality, and dynamic behaviour of underlying concurrent systems. Many variants and extensions of PrT nets have been proposed in the past decade. In this paper, PrT nets are defined with an underlying algebraic specification, which is also called algebraic Petri nets elsewhere [38].

2.1 The Syntax and Static Semantics

A PrT net consists of (1) a finite net structure (P, T, F), (2) an algebraic specification *SPEC*, and (3) a net inscription (φ, L, R, M_0). (P, T, F) is the essential net structure, where $P\cup T$ is the set of nodes satisfying the condition $P\cap T=\emptyset$. P is called the set of *predicates* (graphically represented by circles) and T is called the set of *transitions* (represented by boxes). F is the set of arcs and is called the *flow relation*, which satisfies the condition: $F \subseteq P\times T \cup T\times P$.

For example, Figure 1 shows a PrT net of the well-known problem of dining philosophers. There are three predicate nodes: *Thinking*, *Eating* and *Chopstick*. Tokens at the predicate node *Thinking* represent the philosophers in the state of thinking. Tokens at the predicate node *Chopstick* represent the chopsticks available to use. Each token at predicate note *Eating* consists of one philosopher and two chopsticks and represents the state that the philosopher is eating using the chopsticks. There are two transition nodes: *Pickup* and *Putdown*. The transition *Pickup* changes the state of a philosopher from *Thinking* to *Eating* and at the same time changes the state of two chopsticks from available to occupied. The transition *Putdown* represents that a philosopher finishes eating and puts down a pair of chopsticks. It changes the state that a philosopher is eating using a pair of chopsticks to the state that the philosopher is thinking and that the chopsticks become available. This example will be used throughout the remainder of the paper to illustrate the testing methods.

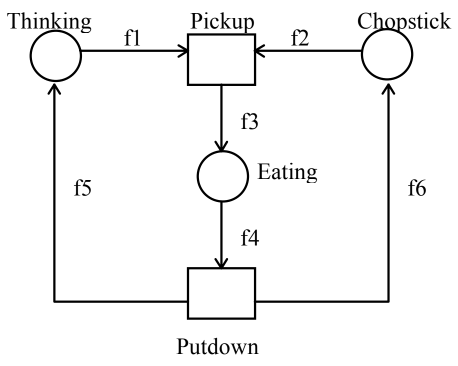


Figure 1 - A PrT Net Specification of Dining Philosophers' Problem

The algebraic specification *SPEC* is a meta-language to define the tokens, labels, and constraints of a PrT net. The underlying specification *SPEC* = (*S*, *OP*, *Eq*) consists of a signature Σ = (*S*, *OP*) and a set *Eq* of Σ-equations. Signature Σ = (*S*, *OP*) includes a set of sorts *S* and a family $OP = (OP_{s_1,...,s_n, s})$ of sorted operations for $s_1, ..., s_n, s \in S$. For each $s \in S$, we use CON_S to denote $OP_{,s}$ (the 0-ary operation of sort *s*), i.e. the set of constant symbols of sort *s*. The Σ-equations in *Eq* define the meanings and properties of operations in *OP*. In this paper, we often simply use familiar operations and their properties without explicitly listing the relevant equations in the examples.

For example, the specification *SPEC* = (*S*, *OP*, *Eq*) underlying the dining philosophers' PrT net contains the following elements.

(1) *S* includes elementary sorts PHIL to represent philosophers, CHOP to represent chopsticks and Boolean. PHIL and CHOP are derived from Integer. *S* also includes structured sorts such as set and tuple obtained from the Cartesian product of the elementary sorts.

(2) *OP* includes standard arithmetic and relational operations on Integer, logical connectives on Boolean, set operations, and selection operation on tuples[1].

(3) *Eq* includes known properties of the above operators.

The net inscription (φ, *L*, *R*, M_0) associates each graphical symbol of the net structure (*P*, *T*, *F*) with an entity in the underlying *SPEC*, and thus defines the static semantics of a PrT net.

Each predicate in a PrT net is a data structure and a component of the overall system state. The sort of each predicate is a member of *S* in *SPEC*. It defines its valid values, i.e. proper tokens, which are ground terms of the signature Σ, written $MCON_S$. Therefore, we associate each predicate *p* in *P* with a subset of sorts in *S*, and give the sort assignment $\varphi : P \rightarrow \wp(S)$.[2]

The flows in a PrT net are labelled with the sorts of the tokens that can pass through. The set of labels is denoted by $Label_S(X)$, where *X* is a set of sorted variables disjoint with *OP*. Each label can be an expression of the form $(k_1x_1 + ... + k_nx_n)$. Mapping $L: F \rightarrow Label_S(X)$ is a sort-respecting labelling of PrT net.

Each transition in a PrT net is associated with a constraint to define its functionality and processing. Constraints of a PrT net belong to a subset of first order logic formulas whose quantifiers range over finite domains and free variables appear in the label of some connecting

[1] We use A[i] to denote the i'th component of tuple A.
[2] $\wp(X)$ is the power set of *X*.

arc of the transition. Thus, constraints are essentially propositional logic formulas defined in the underlying algebraic specification. In particular, the subset of first order logical formulas contains the Σ-terms of sort *bool* over *X*, denoted as $Term_{OP,bool}(X)$. In general, a constraint contains two parts - the pre-condition part involving only label variables in incoming arcs and the post-condition part specifying the relationships between the variables of the incoming arcs and label variables of the outgoing arcs. The pre-condition specifies the required tokens and the post-condition defines the values of generated token in terms of the selected tokens. Therefore the pre-condition is essentially the guard of the functionality (processing) defined by the post-condition. The canonical form of the constraint $R(t)$ of a transition t can be written as $\mathrm{Pre}(t) \wedge \mathrm{Post}(t)$. Mapping $R: T \to Term_{OP,bool}(X)$ is a well-defined constraining mapping.

M_0: $\{m_0: P \to MCONS_S\}$ is a set of sort-respecting initial markings. Each initial marking assigns a multi-set of tokens to each predicate p in P. We view M_0 as a set of markings instead of a single marking for two reasons. First, we have the complete input domain explicitly. Second, we can easily distinguish multiple markings and study them separately.

For example, the net inscription (φ, L, R, M_0) for the dining philosophers problem given in Figure 1 is as follows.

(1) Sorts of predicates:
   φ(Thinking) = φ(Eating) = $\wp$(PHIL),
   φ(Chopstick) = $\wp$(CHOP).
(2) Arc definitions:
   L(f1) = {ph},
   L(f2) = {ch1, ch2},
   L(f3) = <ph, ch1, ch2>,
   L(f4) = <ph, ch1, ch2>,
   L(f5) = {ph},
   L(f6) = {ch1, ch2}.
(3) Constraints of transitions:
   R(Pickup) = (ch1 = ph) $\wedge$ (ch2 = ph $\oplus$ 1),
   R(Putdown) = true.
(4) The initial marking:
   $M_0 = \{m_k \mid k = 2, 3, \ldots\}$, where m_k is defined as follows.
   m_k(Thinking) = {1, 2, ..., k},
   m_k(Eating) = { },
   m_k(Chopstick) = {1, 2, ..., k}.

where $\oplus$ is modulus k addition.

2.2 Dynamic Semantics and Observable Behaviour

A marking m of a PrT net is a mapping $P \to MCONS_S$ from the set of predicates to multi-sets of tokens. A transition is enabled if its pre-set contains enough tokens and its constraint is satisfied with an occurrence mode. The firing of an enabled transition consumes the tokens in the pre-set and produces tokens in the post-set. Two transitions including the same transition with two different occurrence modes can fire concurrently if they are not in conflict, where two transitions are conflict if the firing of one transition disables another. Conflicts are resolved non-deterministically. The firing of an enabled transition is atomic. We define the behaviour of a PrT net to be the set of all possible execution sequences. Each execution sequence represents consecutively reachable markings from the initial marking, in which a successor marking is obtained through firing of a subset of enabled non-conflicting transitions from the predecessor marking. The semantic model used in this paper is thus the interleaving-set model (also called step sequence model in [39], which is capable to capture the non-sequential behaviours. Other well-known semantic models of Petri nets include interleaving and partial order [39].

For example, the specification of dining philosophers' problem in PrT net given in Figure 1 allows concurrent executions such as multiple non-conflicting (non-neighbouring) philosophers picking up chopsticks simultaneously, and some philosophers picking up chopsticks while others putting down chopsticks. The constraints associated with transitions Pickup and Putdown also ensure that a philosopher can only use two designated chopsticks defined by the implicit adjacent relationships. Table 1 and Table 2 below give the details of two partial executions of the five dining philosophers' PrT net. The partial execution given in Table 1 only involves firing one transition in each step. In the sequel, we call such executions flat. The execution given in Table 2 contains steps that two transitions are fired simultaneously.

Table 1. A Flat Execution of the Dining Philosophers' PrT Net

Markings m_i			Transitions n_i	
Thinking	Eating	Chopstick	Fired Transition Set	Token(s) consumed
{1,2,3,4,5}	{ }	{1,2,3,4,5}	Pickup	ph=1, ch1=1, ch2=2
{2,3,4,5}	{<1, 1, 2>}	{3,4,5}	Putdown	<ph, ch1, ch2>=<1,1,2>
{1,2,3,4,5}	{ }	{1,2,3,4,5}	Pickup	ph=2, ch1=2, ch2=3
{1,3,4,5}	{<2, 2, 3>}	{1,4,5}	Pickup	ph=4, ch1=4, ch2=5
{1, 3, 5}	{<2, 2, 3>, <4, 4, 5>}	{1}	Putdown	<ph, ch1, ch2>=<2,2,3>
{1, 2, 3, 5}	{<4, 4, 5>}	{1,2,3}	Putdown	<ph, ch1, ch2>=<4,4,5>
{1,2,3,4,5}	{ }	{1,2,3,4,5}	Pickup	ph=5, ch1=5, ch2=1
{1,2,3,4}	{<5, 5, 1>}	{2,3,4}	Pickup	ph=3, ch1=3, ch2=4
{1,2,4}	{<5, 5, 1>, <3, 3, 4>}	{2}	Putdown	<ph, ch1, ch2>=<3,3,4>
{1,2,3,4}	{<5, 5, 1>}	{2,3,4}	Putdown	<ph, ch1, ch2>=<5,5,1>
{1,2,3,4,5}	{ }	{1,2,3,4,5}	…	…

Table 2. A Non-flat Execution of the Dining Philosophers' PrT Net

Markings m_i			Transitions n_i	
Thinking	Eating	Chopstick	Fired Transitions	Token(s) consumed
{1,2,3,4,5}	{ }	{1,2,3,4,5}	Pickup	ph=1, ch1=1, ch2=2
{2,3,4,5}	{<1, 1, 2>}	{3,4,5}	Putdown	<ph, ch1, ch2>=<1,1,2>
{1,2,3,4,5}	{ }	{1,2,3,4,5}	Pickup	ph=2, ch1=2, ch2=3
			Pickup	ph=4, ch1=4, ch2=5
{1, 3, 5}	{<2, 2, 3>, <4, 4, 5>}	{1}	Putdown	<ph, ch1, ch2>=<2,2,3>
			Putdown	<ph, ch1, ch2>=<4,4,5>
{1,2,3,4,5}	{ }	{1,2,3,4,5}	Pickup	ph=3, ch1=3, ch2=4
{1,2,4,5}	{<3, 3, 4>}	{1,2,5}	Pickup	ph=5, ch1=5, ch2=1
			Putdown	<ph, ch1, ch2>=<3,3,4>
{1,2,3,4}	{<5, 5, 1>}	{2,3,4}	Putdown	<ph, ch1, ch2>=<5,5,1>
{1,2,3,4,5}	{ }	{1,2,3,4,5}	…	…

Definition 1 (Interleaving-Set Semantics of PrT Nets)

Let N be a PrT net, and M_0 be the set of initial markings of N, *which* is thus the set of test cases for N. An *execution* of N on a test case m_0 is a sequence of reachable markings starting from m_0 and linked by transition firings. We denote such an execution as follows:

$$e: m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots,$$

where n_i, called *steps*, are non-empty subsets of transitions that are not in conflict with each other, $m_0 \in M_0$ is an initial marking, m_i, i= 1, 2,…, are markings such that m_i is obtained from m_{i-1} by firing transitions in the subset n_{i-1}. $<n_0, n_1, ..., n_k, ...>$ is called a *step sequence*. An execution is *flat*, if the subsets n_i, i=0, 1, …, are singleton sets and $<n_0, n_1, ..., n_k, ...>$ is called a *transition sequence*. A flat execution e can be obtained from a non-flat execution e' by flattening, if

(a) $e: m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} m_{k+1} \cdots$,

(b) $e': m'_0 \xrightarrow{n'_0} m'_1 \xrightarrow{n'_1} m'_2 \xrightarrow{n'_2} \cdots \xrightarrow{n'_{k-1}} m'_k \xrightarrow{n'_k} m'_{k+1} \cdots$, and

(c) there are natural numbers $j_0 < j_1 < \ldots < j_k < \ldots$ such that $n'_0 = \bigcup_{i=0}^{j_0} n_i$, $m'_0 = m_0$, $n'_{k+1} = \bigcup_{i=j_k+1}^{j_{k+1}} n_i$, $m'_{k+1} = m_{j_{k+1}}$, k=0, 1, 2, …. □

Informally, flattening an execution into a flat execution is to execute the concurrent firings of transitions by interleaving. Of course, there may exist many different orders to execute concurrent transition firings by interleaving. The relationship between the interleaving semantics (transition sequences) and interleaving-set semantics (step sequences) was studied and given in [39].

As an important activity in software testing practice, the observation and recording of system's dynamic behaviour during testing process must be systematically and consistently performed.

Definition 2. (Observation Scheme) [31]

A *scheme* of behaviour observation and recording, or simply an *observation scheme,* is an ordered pair $<B, \mu>$, where $B=\{<B_p, \le_p> \mid p$ is a concurrent system$\}$, and $\mu=\{\mu_p \mid p$ is a concurrent system$\}$. B is called the *universe of phenomena*. For all concurrent systems p, $<B_p, \le_p>$ is a complete partially ordered set. B_p is called the *universe of phenomena* on p. The mappings in μ are called the *recording functions*. For all concurrent systems p, the recording function μ_p maps a test set T to a non-empty consistent subset of B_p. □

Informally, each element in B_p is a *phenomenon* observable from testing a concurrent system p. $\sigma_1 \le_p \sigma_2$ means that phenomenon σ_1 is a part of phenomenon σ_2. The least element $\perp_p$ of B_p denotes that nothing is observed. Notice that, because of the non-determinism and concurrency, two execution of a concurrent system on the same input may demonstrate two different behaviours and produce two different results. Sometimes, it is necessary to execute the system on the same test case twice or even more times in order to test all possible behaviours and outputs. Therefore, we define a test set of a concurrent system p as a multi-set of input to p to represents multiple executions on test cases. In particular, a test set of a PrT net is a multi-set of its initial markings M_0, which is the input domain of the PrT. Given a concurrent p and a test set T, the phenomenon observable by executing p on test set T may not be unique. We use $\mu_p(T)$ to denote the set of all such possible phenomena. In other words, $\sigma \in \mu_p(T)$ means that σ is a phenomenon that is observable by an execution of *p* on test set *T*. As a theory of testing PrT nets, we define the observable phenomena directly on the bases of dynamic semantics.

Definition 3 (Complete Scheme of Behaviour Observation)

In the *complete* observation scheme Ψ, the universe of phenomena B_N^{Ψ} and recording function Ψ_N for any given PrT net *N* are defined as follows:

(1) Let $R_{N,m} = \{e \mid e$ is an execution of N on $m\}$ for all $m \in M_0$, and $R_N = \bigcup_{m \in M_0} R_{N,m}$. We define:

(a) $B_N^{\Psi} = \wp(R_N)$ (the power set of R_N), and

(b) The partial ordering $\leq_{\Psi}$ on B_N^{Ψ} is the set inclusion relation $\subseteq$.

(2) For all $m \in M_0$, $\Psi_N(\{m\}) = \{\{e\} \mid e \in R_{N,m}\}$.

(3) For any test set $M \subseteq M_0$ and $m \in M_0$, $\Psi_N(M \cup \{m\}) = \{u \cup \{e\} \mid u \in \Psi_N(M) \wedge e \in R_{N,m}\}$.

□

Intuitively, the complete scheme records every detail of the executions of a concurrent system.

For many concurrent systems, a maximum sequence of markings can be infinite, i.e. the execution does not terminate. However, in software testing practice, we cannot observe and record an infinite execution within a finite period of time. Therefore, we often stop execution manually and observe and record a partial execution. This practice can be defined as an observation scheme. We first define a mapping *Truncation* as follows:

$$Truncation_n(e) = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k,$$

where $k=n$, if $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots$ or $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{s-1}} m_s$, $s \geq n$; and $k=s$, if $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{s-1}} m_s$ and $s<n$.

Definition 4 (Partial Executions Up to n Steps)

In the *partial execution up to n steps* observation scheme Π_n, the universe of phenomena $B_N^{(\Pi,n)}$ and recording functions Π_N for any given PrT net *N* are defined as follows.

(1) $B_N^{(\Pi,n)} = Truncation_n(B_N^{\Psi})$;[3]

(2) For any test set $M \subseteq M_0$, $\Pi_N(M) = \{Truncation_n(u) \mid u \in \Psi_N(M)\}$. □

The universal scheme Ω_N defined below contains both complete and partial executions.

Definition 5 (Universal Scheme)

In the *universal* observation scheme Ω, the universe of phenomena B_N^{Ω} and recording function Ω_N for any given PrT net N are defined as follows.

(1) $B_N^{\Omega} = \wp(R'_N)$, where $R'_N = \bigcup_{n=1}^{\infty} Truncation_n(R_N)$. The partial ordering $\leq_{\Omega}$ is the set inclusion relation $\subseteq$.

(2) For any $m \in M_0$, $\Omega_N(\{m\}) = \{\{e\} \mid e \in R'_{N,m}\}$.

(3) For any test set $M \subseteq M_0$ and $m \in M_0$, $\Omega_N(M \cup \{m\}) = \{u \cup \{e\} \mid u \in \Omega_N(M) \wedge e \in R'_{N,m}\}$. □

In the universal scheme, an observable phenomenon is a set of complete or partial executions of the PrT net under test. It is worth noting that partial executions in such a phenomenon can have

[3] Notice that, here we extended the function to apply on a set of executions, i.e. $Truncation_n(B_N^{\Psi}) = \{Truncation_n(x) \mid x \in B_N^{\Psi}\}$. In general, let *f* be any given function defined on a domain *D*, for all $X \subseteq D$, we write $f(X)$ to denote the set $\{f(x) \mid x \in X\}$. Such extensions of functions will be used in the sequel when there is no risk of confusion.

different number of steps, which are determined by the tester. This forms an additional dimension of non-determinism of testing. Without the loss of generality, we call both complete executions and partial executions as test executions in the sequel. The scheme given in Definition 5 is universal in the sense that every observation scheme can be extracted from it. The notion of extraction is formally defined as follows [31]. Let $\mathscr{A}$=<A, μ^A> and $\mathscr{B}$=<B, μ^B> be two schemes.

Definition 6. (Extraction Relation between Schemes) [31]

Scheme $\mathscr{A}$ is an *extraction* of scheme $\mathscr{B}$, written $\mathscr{A} \lhd \mathscr{B}$, if for all $p \in P$, there is a homomorphism φ_p from $<B_p, \leq_{B,p}>$ to $<A_p, \leq_{A,p}>$, such that (1) $\varphi_p(\sigma)=\bot_{A,p}$ if and only if $\sigma=\bot_{B,p}$, and (2) for all test sets T, $\mu_p^A(T) = \varphi_p(\mu_p^B(T))$. □

Informally, scheme $\mathscr{A}$ is an *extraction* of scheme $\mathscr{B}$ means that scheme $\mathscr{B}$ observes and records more detailed information about dynamic behaviours than scheme $\mathscr{A}$ does. The phenomena that scheme $\mathscr{A}$ observes can be extracted from the phenomena that $\mathscr{B}$ observes.

3. Transition-Oriented Testing

As argued in [31], a test method contains two main components, an observation scheme and an adequacy criterion. The adequacy criterion determines how to select test cases before testing and how to analyse test results after test executions. The observation scheme determines how to observe and record a system's dynamic behaviour during the test executions. A transition-oriented testing method observes the transitions fired during test executions and analyses test adequacy according to the transitions covered by the testing.

The most basic transition-oriented testing method uses the following observation scheme and adequacy criterion.

Definition 7 (Fired Transitions Scheme)

The fired transitions' scheme Ξ_N is extracted from the universal scheme by the mapping: $Firing(e) = \bigcup_{i=0,1,\ldots} n_i$, where $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots$. Its recording function Ξ_N is:

$$\Xi_N(M)=\left\{ \bigcup_{x \in u} Firing(x) \middle| u \in \Omega_N(M) \right\}, \text{ for any } M \subseteq M_0. \quad \Box$$

The following is the transition coverage criterion for adequacy analysis.

Definition 8 (Transition Coverage)

Let E be a collection of test executions of a PrT net N. E satisfies *transition coverage* criterion if $\bigcup_{e \in E} Firing(e) = T_N$, where T_N is the set of transitions of N. The following function is called the transition coverage measurement.

$$TransitionCoverage(N,E) = \left\| \bigcup_{e \in E} Firing(e) \right\| \Big/ \left\| T_N \right\|. \quad \Box$$

For example, both executions of the dining philosophers' PrT net given in Table 1 and Table 2 satisfy the transition coverage criterion. All the transitions in the PrT net (i.e. Pickup and Putdown) are fired.

Multiple transitions may be enabled in a marking and fired in one step of execution due to the existence of non-determinism and concurrency. However, not all subsets of transitions can be fired. The following defines the notion of feasibility of a subset of transitions and the feasibility of a sequence of such subsets.

Definition 9 (Feasibility and Concurrency Degrees of Transition Traces)

Let $n \subseteq T$ be a subset of the transitions of a PrT net *N*. The subset *n* is *feasible*, if and only if there exists an execution $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots$ of the PrT net *N* such that for some $i \in \{0, 1, ..., k, ...\}$, $n \subseteq n_i$. The *concurrency degree* of a feasible transition subset *n* is the size of the set.

A *transition trace* is a sequence $<n'_1, n'_2, ..., n'_L>$, $L>0$, of subsets of the transitions of a PrT net *N*. A transition trace $<n'_1, n'_2, ..., n'_L>$ is *feasible*, if and only if there is an execution $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots$ of the PrT net *N* such that for some $i \in \{0, 1, ..., k, ...\}$, $n'_j = n_{i+j}$, for all $j = 1, 2, ..., L$, and we say that the execution *e covers* the transition trace. The *concurrency degree* of transition trace is the maximum of the concurrency degrees of its elements. □

When a transition trace's concurrency degree equals to 1, that is, the subsets in a transition trace $<n'_1, n'_2, ..., n'_L>$, $L>0$, are all singleton sets, it is a transition sequence. For example, consider the partial execution in Table 1 of the dining philosophers' PrT net. The set of length-2 feasible transition sequences includes <Pickup, Putdown>, <Pickup, Pickup>, <Putdown, Putdown>, and <Putdown, Pickup>. The set of feasible length-3 sequences of transitions includes <Pickup, Pickup, Putdown>, <Putdown, Pickup, Putdown>, <Putdown, Pickup, Pickup>, <Pickup, Putdown, Pickup>, <Pickup, Putdown, Putdown>, and <Putdown, Putdown, Pickup>. The transition trace of the partial execution of the dining philosophers' PrT net given in **Table 2** is <{Pickup}, {Putdown}, {Pickup, Pickup}, {Putdown, Putdown}, {Pickup}, {Pickup, Putdown}, {Putdown}>.

Let *Trace* be the mapping defined as follows.

$$Trace(e) = < n_0, n_1, \cdots, n_k, \cdots >, \text{ where } e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots.$$

The following scheme records the sequences of transition firings in the executions of a concurrent system.

Definition 10 (Transition Trace Scheme)

The transition trace scheme Γ_N is extracted from the universal scheme by the mapping *Trace* defined above. Its recording function has the property that for any test set $M \subseteq M_0$, $\Gamma_N(M) = \{e \mid e = \{Trace(x) \mid x \in u\} \wedge u \in \Omega_N(M)\}$. □

The transition trace testing method requires the sequence of transition subsets fired during test executions be recorded. That is, it requires the transition trace scheme to be used. A hierarchy of adequacy criteria can be defined for transition trace testing.

Definition 11 (K-Concurrency Length-L Trace Coverage)

Let *E* be a collection of executions of PrT net *N*. Let *K*, $L>0$ be any natural numbers. *E* is said to satisfy the *K-concurrency length-L transition trace coverage* criterion, if and only if for any feasible transition trace *q* with length less than or equal to *L* and concurrency degree less than or equal to *K*, there is at least one $e \in E$, such that *q* is covered by *e*. In particular, *K*-concurrency length-1 trace coverage is called *K-concurrency transition coverage*. The K-concurrency length-*L* trace coverage measurement is defined by the formula:

$$ConTraceC_{K,L}(N,E) = \|CV_{N,K,L}(E)\| / \|Trc_{K,L}(N)\|,$$

where $Trc_{K,L}(N)$ is the set of all feasible transition traces with length less than or equal to L and concurrency degree less than or equal to K, and $CV_{N,K,L}(E)$ is the set of transition traces in $Trc_{K,L}(E)$ that is covered by at least one element in E. □

For example, the partial execution in Table 1 satisfies the 1-concurrency length-2 transition trace coverage criterion. However, it does not satisfy the 1-concurrency length-3 transition trace coverage criterion because the sequence of transitions <Putdown, Pickup, Putdown> is not covered by the partial execution. The 1-concurrency length-2 adequacy measurement of the partial execution is $5/6$. The partial execution given in **Table 2** satisfies the 2-concurrency transition coverage criterion (i.e. 2-concurrency length-1 coverage).

The following lemmas prove the subsumption relationships between *k*-concurrency length-*l* adequacy criteria. An adequacy criterion *A* subsumes criterion *B* if for all test executions *E*, *E* satisfies criterion *A* implies that *E* also satisfies criterion *B*. The subsumption relationships between test adequacy criteria are closely related to testing methods' fault detecting ability and test cost [40].

Lemma 1.

For all natural numbers k, l_1 and l_2, $l_1 \geq l_2$ implies that k-concurrency length-l_1 transition trace coverage subsumes k-concurrency length-l_2 transition trace coverage.

Proof. Let $k>0$ be any given natural number. By Definition 9, for any natural numbers $l_1 \geq l_2$, the set S_1 of feasible k-concurrency length-l_1 transition traces is a subset of the set S_2 of feasible k-concurrency length-l_2 transition traces. By Definition 9, for all sets E of executions, E covers S_2 implies E also covers S_1. By Definition 11, E is adequate according to k-concurrency length-l_2 coverage implies that E is also adequate according to k-concurrency length-l_1. Therefore, the statement is true. □

Lemma 2.

For all natural numbers l, k_1 and k_2, $k_1 \geq k_2$ implies that k_1-concurrency length-l transition trace coverage subsumes k_2-concurrency length-l transition trace coverage.

Proof. It is similar to the proof of Lemma 1. □

Note: (1) k-concurrency transition coverage is k-concurrency length-1 trace coverage. Therefore, by Lemma 2, for all $k, l>0$, k-concurrency length-l transition trace coverage criterion subsumes k-concurrency transition coverage. (2) Transition coverage is 1-concurrency length-1 trace coverage. Therefore, by Lemma 1 and Lemma 2, all $k, l>0$, k-concurrency length-l transition trace coverage criterion subsumes transition coverage.

Moreover, the following *all transition trace coverage* criterion subsumes all these criteria.

Definition 12 (All Transition Trace Coverage)

Let E be a collection of executions of a PrT net N. The test execution E is said to satisfy the *all transition traces coverage* criterion, if and only if for any feasible transition trace q of N, there is at least one $e \in E$ such that q is covered by e. The transition trace coverage measurement is defined by formula: $TTC(N,E) = \|CV_N(E)\| / \|Trc(N)\|$, where $Trc(N)$ is the set of all feasible transition traces of N, and $CV_N(E)$ is the set of feasible transition traces of N that is covered by at least one element in E. □

Lemma 3.

For all k, l >0, the all transition traces coverage criterion subsumes the k-concurrency length-l transition trace coverage criterion.

Proof. It is similar to the proof of lemma 1. □

It is worth noting that most concurrent systems in practical use contain an infinite number of feasible transition traces. To satisfy such an adequacy criterion, we may need an infinite amount of computational resources.

In software testing, one may consider one concurrent execution of a PrT net as several interleaved executions of the same PrT net. Let q = $<t_0, t_1, \ldots, t_L>$, L>0, be a sequence of transitions. We say that an execution *e logically covers* sequence q, if a flattening of e contains q as a consecutive subsequence of transition firings. For example, the transition trace of the partial execution given in Table 2 logically covers the following two transition sequences.

<Pickup, Putdown, Pickup, Pickup, Putdown, Putdown, Pickup, Pickup, Putdown, Putdown>

<Pickup, Putdown, Pickup, Pickup, Putdown, Putdown, Pickup, Putdown, Pickup, Putdown>

Definition 13 (Interleaving Length-L Transition Sequence Coverage)

Let E be a collection of executions of PrT net N. Let L>0 be any natural number. E is said to satisfy the *Interleaving length-L transition sequence coverage* criterion, if and only if for any feasible transition sequence q with length less than or equal to L, there is at least one $e \in E$ that logically covers q.

The Interleaving length-L transition sequence coverage measurement is defined by the formula:

$$InterleaveC_L(N,E) = \left\| InterleaveCV_{N,L}(E) \right\| \big/ \left\| Seq_L(N) \right\|,$$

where $Seq_L(N)$ is the set of all feasible transition sequences with length less than or equal to L, and $InterleaveCV_{N,L}(E)$ is the set of transition sequences in $Seq_L(E)$ that is logically covered by at least one element in E. □

For example, the partial execution given in Table 2 satisfies the interleaving length-3 transition coverage criterion. The two transition sequences that are logically covered by the execution contain all transition subsequences of length less than or equal to 3. The partial execution given in Table 1 does not satisfy the interleaving length-3 transition coverage criterion.

Notice that, in interleaving semantics, the interleaving length-L transition sequence coverage criterion is equivalent to the length-L trace coverage criterion proposed and investigated in [41]. Also, interleaving length-1 transition sequence coverage is transition coverage.

Lemma 4.

For all natural numbers l_1 and l_2, $l_1 \geq l_2$ implies that interleaving length-l_1 transition sequence coverage subsumes interleaving length-l_2 transition sequence coverage.

Proof. It is similar to the proof of lemma 1. □

Lemma 5.

For all natural numbers l>0, that 1-concurrency length-l transition trace coverage subsumes interleaving length-l transition sequence coverage.

Proof. Let $q=<n_1, n_2, ..., n_L>$ be any 1-concurrency transition trace of length less than or equal to L. By Definition 9, every element n_i of q is a singleton set. Let $n_i=\{t_i\}$, i =1, 2, …, L. let $q'=<t_1, t_2, …, t_L>$. An execution e covers q implies that there is a consecutive subsequence of transition firings $n'_{k+1}, n'_{k+2}, …, n'_{k+L}$ such that $n_i=n'_{k+i}$, i =1, 2, …, L. Therefore, a flattening of e contains a consecutive subsequence of transitions $t_1, t_2, …, t_L$. The statement follows due to the fact that a sequence q' of transitions is feasible if and only if the corresponding concurrency degree 1 transition trace is feasible. □

Notice that, 1-concurrency length-L transition trace coverage is not equivalent to interleaving length-L transition sequence coverage criterion, because the former forces test executions to fire one transition at a time to cover an 1-concurrency transition trace while the later does not. For example, the partial execution given in Table 2 does not satisfy 1-concurrency length-3 transition trace coverage criterion, but it satisfies interleaving length-3 transition sequence coverage criterion.

The following diagram summarises the subsumption relationships between the transition oriented testing methods.

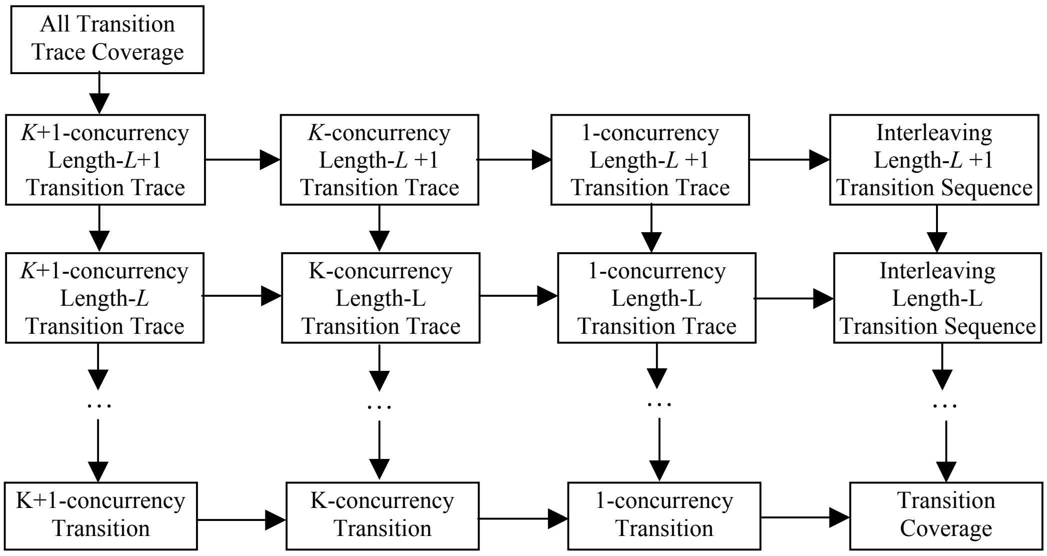

All Transition Trace Coverage
K+1-concurrency Length-L+1 Transition Trace
K-concurrency Length-L +1 Transition Trace
1-concurrency Length-L +1 Transition Trace
Interleaving Length-L +1 Transition Sequence
K+1-concurrency Length-L Transition Trace
K-concurrency Length-L Transition Trace
1-concurrency Length-L Transition Trace
Interleaving Length-L Transition Sequence
…
K+1-concurrency Transition
K-concurrency Transition
1-concurrency Transition
Transition Coverage

Figure 2 - The hierarchy of transition oriented testing methods

4. State-Oriented Testing

In contrast to transition-oriented testing, a state-oriented testing method records the states (i.e. the markings of a PrT net) of the concurrent system under test and analyses test adequacy according to such recorded information.

Definition 14 (State Scheme)

The state scheme Σ_N is extracted from the universal scheme by the mapping:

$$Markings(e)=\{m_0,m_1,\cdots,m_k,\cdots\}\text{, where } e=m_0\xrightarrow{n_0}m_1\xrightarrow{n_1}m_2\xrightarrow{n_2}\cdots\xrightarrow{n_{k-1}}m_k\xrightarrow{n_k}\cdots.$$

Its recording function has the property:

$$\Sigma_N(M)=\{\bigcup_{x\in u}Marking(x)\mid u\in\Omega_N(M)\}\text{ for any test set } M\subseteq M_0.\quad\square$$

Let N be a given PrT net. $Mark(N)$ is defined to be the set of reachable markings on N, i.e.

$m \in Mark(N)$ if and only if there is an initial marking $m_0 \in M_0$ and an execution e of N on m_0 such that $m \in Markings(e)$. The concept of *abstract states* of a concurrent system consists of: (1) a finite set AS_N of abstract states of N, and (2) a mapping $State_N$: $Mark(N) \rightarrow AS_N$ that defines how markings are associated to states.

For example, we can define $AS_{DP}=\{0, 1, 2, \ldots, \lfloor k/2 \rfloor\}$ to be the set of abstract states in the dining philosophers PrT net (k is a given natural number), where $n \in AS_{DP}$ means that "*n philosophers are eating*". The mapping $State_{DP}$ is defined as follows: $State_{DP}(m)=n$, if the predicate *Eating* contains n tokens in the marking m. It is worth noting that for a given PrT net, we can define more than one abstract state space. For example, the following is another abstract state space for the dining philosophers PrT net. Let $AS'_{DP}=\{think, eat\}$, and the mapping

$$State'_{DP}(m) = \begin{cases} think, & \text{if the token "1" is included in the predicate } Thinking; \\ eat, & \text{if the token "<1,1,2>" is included in the predicate } Eating. \end{cases}$$

where, informally, the state "*think*" represents the situation that philosopher 1 is thinking, and the state "*eat*" represents that philosopher 1 is eating.

The state testing method uses the state scheme and the following state coverage criterion.

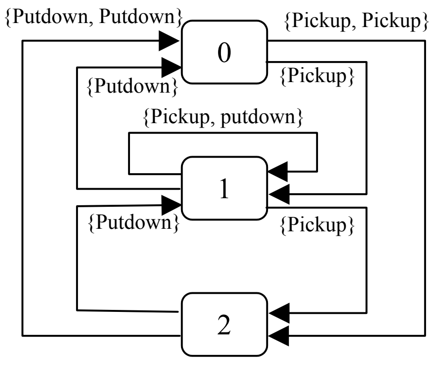
Definition 15 (State Coverage)

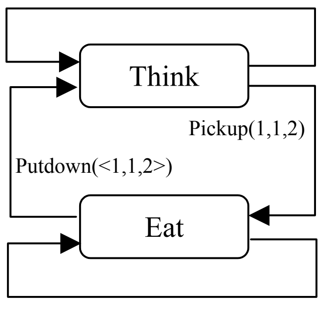
A collection E of executions of N satisfies the state coverage criterion with respect to the concept of state (AS_N, $State_N$), if for all feasible state $s \in AS_N$, there is at least one execution e in E such that there is at least one $m \in Markings(e)$ such that $State_N(m)=s$. The state coverage measurement is defined by the formula:

$$StateCoverage(N,E) = \left\| State_N\left(\bigcup_{e \in E} Markings(e)\right) \right\| \Big/ \left\| AS_N \right\|. \quad \Box$$

For example, the partial execution given in Table 1 satisfies the state coverage criterion with respect to $State_{DP}$. It also satisfies the state coverage criterion with respect to $State'_{DP}$.

A pair $<s_1,s_2>$ of states, s_1, $s_2 \in AS_N$, is a feasible state transition, if there is an execution $m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots$ such that for some j, $State(m_j)=s_1$ and $State(m_{j+1})=s_2$. For example, the feasible state transitions for the five dining philosophers' PrT net (k=5) is given in the following diagrams.





(a) (b)

Figure 3 - State transition diagrams for the dining philosophers PrT net

Definition 16 (State Trace Scheme)

The state trace scheme Φ_N is extracted from the universal scheme by the mapping:

$$MarkingTrace(e) = < m_0, m_1, \cdots, m_k, \cdots > \text{, where } e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots .$$

Its recording mapping has the property:

$$\Phi_N(M)=\{MarkingTrace(u) \mid u \in \Omega_N(M)\} \text{ for any test set } M \subseteq M_0. \quad \square$$

The state transition testing method uses the state trace scheme. The adequacy criterion is defined as follows.

Definition 17 (State Transition Coverage)

Let E be a collection of test executions. E is said to satisfy the *state transition coverage* criterion, if for all feasible state transitions $<s_1,s_2>$, there is at least one execution e in E such that e covers the state transition, i.e. there is j such that $State(m_j)=s_1$ and $State(m_{j+1})=s_2$. The state coverage measurement is defined by the following formula:

$$STC(N,E) = \|CT_N(E)\| / \|ST_N\|,$$

where $CT_N(E)$ is the set of state transitions that are covered by the collection e of test executions, ST_N is the set of feasible state transitions. □

For example, the partial execution given in Table 1 satisfies the state transition coverage criterion with respect to the state space $State'_{DP}$. It does not satisfy the state transition coverage criterion with respect to the state space $State_{DP}$ because it does not cover the state transition from 0 to 2 and transition from state 2 to 0.

Compared with state testing, state transition testing requires to record not only more detailed information during the testing process, but also more test executions because state transition coverage subsumes state coverage.

Lemma 6.

Provided that for all states s_1 in AS_N, there is $s_2 \in AS_N$ such that either $<s_1,s_2>$ or $<s_2,s_1>$ is a feasible state transition, we have that a collection E of test executions satisfies the state transition coverage criterion implies that E also satisfies the state coverage criterion.

Proof. It is straightforward from the definition. □

A sequence q of states $<s_1,s_2,\ldots,s_k>$ is a state transition path of length k, if for all i=1,2,…,k–1, $<s_i,s_{i+1}>$ is a state transition. An execution $m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots$ covers the path, if and only if for some u, $State(m_{u+i})=s_i$, i=1,2,…,k. If there is an execution e such that e covers q, we say that the path q is feasible. For example, the set of length 3 AS_{DP} state transition paths of include <0,1,2>, <0,1,0>, <1,0,1>, <1,2,1>, <2,1,0>, and <2,1,2>.

State transition path testing also uses the state trace scheme. There is a hierarchy of adequacy criteria that can be used for its adequacy analysis.

Definition 18 (State Transition Path Coverage)

Let k>1 be a given natural number. Let E be a collection of test executions, we say that E satisfies the *length-k state transition path coverage* criterion, if and only if for any feasible state transition path q of length less than or equal to k, there is an execution e in E such that e covers the path q. The length-k state transition path coverage measurement is defined by the following formula.

$$LSTP_k(N,E) = \|CU_{N,k}(E)\| / \|STPath_k(N)\|,$$

where $STPath_k(N)$ is the set of feasible state transition paths of length less than or equal to k, $CU_{N,k}(E)$ is the subset of $STPath_k(N)$ that is covered by test executions in E. □

For example, the partial execution given in Table 2 does not cover the state transition path <2,1,2>.

Lemma 7.

For all natural numbers k_1 and k_2, $k_1 \geq k_2$ implies that length-k_1 state transition path coverage subsumes length-k_2 state transition path coverage.

Proof. It is similar to the proof of lemma 1. □

5. Flow-Oriented Testing

In transition oriented and state oriented testing, information about the flows of tokens in the system is ignored, which are reflected in the definitions of the relevant observation schemes. To consider token flows in testing, we develop a new type of testing method based on the similar ideas of data flow testing of sequential programs [33~36].

A flow-oriented testing method requires the recording of tokens passing through the arcs in a PrT net.

Let (P, T, F) be the net structure of a given PrT net N and $t \in T$ be a transition node of N, a flow $f \in F$ is called an *inward* flow of t, if f=<t', t> for some predicate node $t' \in P$. A flow $f \in F$ is called an *outward* flow of t, if f=<t, t'> for some predicate node $t' \in P$. An inward flow f of t is said to be covered by an execution if the execution contains a firing of t such that at least one token on flow f is consumed by the firing. An outward flow f of t is said to be covered by an execution if the execution contains a firing of transition t such that at least one token is produced on the flow f as the result of the firing.

Definition 19 (Inward Flow, Outward Flow and Flow Schemes)

Let $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots$ be any given execution, F_i and G_i be the sets of inward flows and outward flows participated in the transition firing of n_i, i=1,2,…,k,…, respectively. We define the following mappings:

$$InwardFlow(e) = < F_0, F_1, \cdots, F_k, \cdots >,$$

$$OutwardFlow(e) = < G_0, G_1, \cdots, G_k, \cdots >,$$

$$Flow(e) = < F_0 \cup G_0, F_1 \cup G_1, \cdots, F_k \cup G_k, \cdots >.$$

(1) The *inward flow* scheme *IN* is extracted from the universal scheme by the mapping *InwardFlow*, and the recording function is
$IN_N(M)=\{e \mid e=\{InwardFlow(x) \mid x \in u\} \wedge u \in \Omega_N(M)\}$, for any test set $M \subseteq M_0$.
(2) The *outward flow* scheme *OUT* is extracted from the universal scheme by the mapping *OutwardFlow*, and the recording function is:
$OUT_N(M)=\{e \mid e=\{OutwardFlow(x) \mid x \in u\} \wedge u \in \Omega_N(M)\}$, for any test set $M \subseteq M_0$.
(3) The *flow* scheme *FL* is extracted from the universal scheme by the mapping *Flow*, and the recording function is:
$FL_N(M)=\{e \mid e=\{Flow(x) \mid x \in u\} \wedge u \in \Omega_N(M)\}$, for any test set $M \subseteq M_0$.

□

Definition 20 (Inward Flow, Outward Flow, and Flow Coverage)

Let E be a set of executions of PrT net N.

(1) E is said to satisfy the *inward flow coverage* criterion if for all inward flow f in N, f is covered by at least one execution in E.

(2) E is said to satisfy the *outward flow coverage* criterion if for all outward flow f in N, f is covered by at least one execution in E.

(3) E is said to satisfy the *flow coverage* criterion, if it satisfies both inward flow coverage criterion and outward flow coverage criterion. □

From the definition, it is easy to see the following lemma.

Lemma 8.

The flow coverage criterion subsumes both inward flow coverage and outward flow coverage criteria. □

In hierarchical predicate transition nets (HPrT nets [42]), a flow can be labeled with an expression in the form of $X + Y + \ldots + Z$, hence allow different types of tokens to flow through. The label constructor + indicates non-deterministic flow relation. The expression $X + Y$ means that either a token X, or a token Y, or both tokens of X and Y may pass through the flow. Each term X in the expression is called a possible choice of tokens on the flow.

Definition 21.

Let expression $X_1+X_2+\ldots+X_k$ be the label of an inward flow f of a transition t in a PrT net N. If an execution of N contains at least one firing of transition t that consumes at least one token of X_i type on the flow f, we say that the execution covers the X_i choice of tokens on the inward flow f.

Let expression $X_1+X_2+\ldots+X_k$ be the label of an outward flow f of a transition t in a PrT net N. If an execution of N contains at least one firing of transition t that produces at least one token of X_i type on the flow f, we say that the execution covers the X_i choice of tokens on the outward flow f.

Let expression $X_1+X_2+\ldots+X_k$ be the label of a flow f of a transition t in a PrT net N. Let $\{X_{i1},X_{i2},\ldots,X_{is}\}$, $0<s\le k$, be a subset of $\{X_1,X_2,\ldots,X_k\}$. The set $\{X_{i1},X_{i2},\ldots,X_{is}\}$, $0<s\le k$, is called a *combination of tokens* on the flow f. An execution of N covers the combination $\{X_{i1},X_{i2},\ldots,X_{is}\}$, $0<s\le k$, of tokens on inward flow f of transition t, if there is at least one firing of transition t that consumes tokens $X_{i1},X_{i2},\ldots$, and X_{is}. Similarly, we define the notion of covering a combination of tokens on an outward flow.

□

Definition 22 (Input Token, Output Token and Token Schemes)

Let $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots$ be any given execution, U_i and V_i be the set of input tokens and output tokens of the transition firing of n_i, i=1,2,…,k,…, respectively. We define mappings *InputToken*, *OutputToken* and *Token* as follows.

$$InputToken(e) = < U_0, U_1, \cdots, U_k, \cdots >,$$
$$OutputToken(e) = < V_0, V_1, \cdots, V_k, \cdots >,$$
$$Token(e) = < U_0 \cup V_0, U_1 \cup V_1, \cdots, U_k \cup V_k, \cdots >.$$

(1) The *Input Token* scheme TIN is extracted from the universal scheme by the mapping

InputToken and the recording function is:

$$TIN_N(M)=\{e \mid e=\{InputToken(x) \mid x\in u\}\wedge u\in\Omega_N(M)\}, \text{ for any test set } M\subseteq M_0.$$

(2) The *Output Token* scheme *TOUT* is extracted from the universal scheme by the mapping *OutputToken*, and the recording function is:

$$TOUT_N(M)=\{e \mid e=\{OutputToken(x) \mid x\in u\}\wedge u\in\Omega_N(M)\}, \text{for any test set } M\subseteq M_0.$$

(3) The *Token* scheme *TK* is extracted from the universal scheme by the mapping *Token*, and the recording function is:

$$TK_N(M)=\{e \mid e=\{Token(x) \mid x\in u\}\wedge u\in\Omega_N(M)\}, \text{ for any test set } M\subseteq M_0. \quad \square$$

Definition 23 (Flow Choice Coverage, and Flow Combination Coverage)

Let *E* be a set of executions.

(1) *E* satisfies the *input choice* coverage criterion, if and only if for all inward flow *f* and all possible choices *X* of tokens on *f* , there is at least one execution in *E* that covers the choice of tokens on *f*.

(2) *E* satisfies the *output choice* coverage criterion, if and only if for all outward flow *f* and all possible choices *X* of tokens on *f*, there is at least one execution in *E* that covers the choice of tokens on *f*.

(3) *E* satisfies the *flow choice* coverage criterion, if it satisfies both the inward choice coverage criteria and the outward choice coverage criterion.

(4) *E* satisfies the *input combination* coverage criterion, if for all inward flow *f* and all combinations of tokens on *f* there is at least one execution in *E* that covers the combination of tokens on *f*.

(5) *E* satisfies the *output combination* coverage criterion, if for all outward flow *f* and all combinations of tokens on *f* there is at least one execution in *E* that covers the combination of tokens on *f*.

(6) *E* satisfies the *flow combination* coverage criterion, if it satisfies both input combination coverage criterion and the output combination coverage criterion.

□

Lemma 9.

(1) The input choice coverage subsumes the inward flow coverage;
(2) The output choice coverage subsumes the outward flow coverage;
(3) The input combination coverage subsumes the input choice coverage;
(4) The output combination coverage subsumes the output choice coverage;
(5) The flow choice coverage subsumes both the input choice coverage and the output choice coverage;
(6) The flow combination coverage subsumes both the input combination coverage and the output combination coverage;
(7) The flow choice coverage subsumes the flow coverage;
(8) The flow combination coverage subsumes the flow choice coverage.

Proof. It is straightforward from the definitions. □

Lemma 10.

The flow coverage subsumes the transition coverage.

Proof. In a well-formed PrT net, each transition must have at least one inward flow or one outward flow. Therefore, by definition, covering all flows implies covering all transitions. □

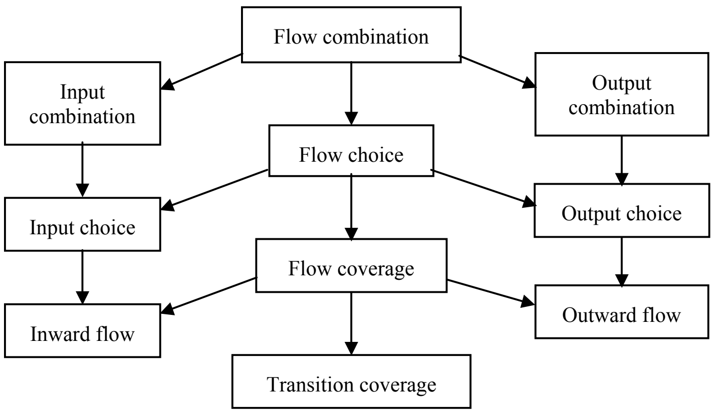


Figure 4 - The hierarchy of data flow test adequacy criteria

Let $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \cdots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \cdots$ be an execution. We say that a token ξ is defined by transition firing n_k and used by transition firing n_j, if (a) ξ is an output of transition firing n_k, (b) ξ is also an input of transition firing of n_j, and (c) ξ does not participate in any transition n_i, $k<i<j$, where $j>k$. Let $n_{i_0}, n_{i_1}, \cdots, n_{i_k}$ be a sub-sequence of the firings of execution e. It is called a *data flow chain*, if there are tokens $\xi_0, \xi_1, \ldots, \xi_{k-1}$ such that ξ_j is defined by n_{i_j} and used by $n_{i_{j+1}}$, j=0,1,…, k–1. A path <Tr_0, Pr_0, Tr_1, Pr_1, …, Tr_k> in a PrT net N is covered by a data flow chain $n_{i_0}, n_{i_1}, \cdots, n_{i_k}$, if n_{i_j} is a firing of transition node Tr_j, j=1, 2, …, k. For example, the first two transition firings in Table 1 cover the paths <*Pickup*, *Eating*, *Putdown*> in the dining philosophers' PrT net.

Definition 24 (Flow Path Coverage)

Let k>1 be a given natural number. Let E be a collection of test executions, E satisfies the *length-k data flow path coverage* criterion, if and only if for any path q in the PrT net N of length less than or equal to k, there is an execution e in E such that e contains at least one data flow chain that covers the path q. The length-k flow path coverage measurement is defined by the following formula.

$$DFP_k(N,E) = \left\|CDF_{N,k}(E)\right\| / \left\|Path_k(N)\right\|,$$

where $Path_k(N)$ is the set of paths in PrT net N of length less than or equal to k, $CDF_{N,k}(E)$ is the subset of $Path_k(N)$ that is covered by data flow chains of test executions in E. □

Lemma 11.

For all natural numbers k_1, k_2>0, k_1>k_2 implies that length-k_1 flow path coverage subsumes length-k_2 flow path coverage.

Proof. It is similar to the proof of Lemma 1. □

6. Specification-Oriented Testing

The testing methods discussed above only concern with the structure of PrT nets. In this section, we discuss specification-oriented testing methods. By specification-oriented testing, we mean both specification-based testing and testing the specification itself. The discussion below will be applicable to both types of testing.

The formal algebraic specification of a PrT net defines a meta-language for the net. Each transition is associated with a constraint in the language to define its function. It can be considered as a function that takes tokens on the inward flows as input and produces tokens on the outward flows as output. The operators used in the specification of the transitions are defined by an algebraic specification. Therefore, testing a PrT net or a concurrent system that implements a PrT net can be carried out at two levels.

At the lower level, the correctness of the algebraic specification or the implementation of the operations used in the PrT nets is tested. Software testing methods based on algebraic specifications have been proposed by Gaudel *et al.* [1, 2] and further developed for testing object-oriented software by Doong and Frankl [3, 4] and Chen and Tse, *et al.* [5, 6]. These methods assumed that each operator in the signature of an algebraic specification is implemented by a corresponding function/procedure of an abstract data type or a method of a class in object-oriented systems. The basic idea of the methods is to use each equation of an algebraic specification to generate two sequences of method (or procedure/function) calls and then to check the equivalence between the two results. This method can achieve a high degree of test automation in the validation of object-oriented software systems against the final algebra semantics of algebraic specifications [6, 43]. Therefore, the method can be applied if the implementation of the algebraic specification uses abstract data types or object-oriented techniques. However, it is not always valid for testing against initial algebra semantics [6, 43]. Other methods for testing algebraic specifications have also been proposed in the literature, such as mutation testing of algebraic specifications proposed and investigated in [44, 45].

At a higher level, once the correctness of the operations is verified, the correctness of the functions associated to the transitions is tested. Since the function of a transition is specified by an expression constructed from the operators of the algebraic specification, i.e. a term of the signature, functional testing methods and adequacy criteria can be applied. Let n be a transition node in a PrT net N. Each time the transition n is fired, it consumes a number of tokens a_1, a_2, ..., a_k from its inward flows and produces tokens b_1, b_2, b_m on its outward flows. The tuple of tokens $<a_1, a_2, ..., a_k>$ then constitutes an input to the function F_n associated to the transition node n, and tuple $<b_1, b_2, ..., b_m>$ forms the corresponding output. The set X_n of the inputs that a transition node n consumed during the test executions of PrT net N is then the test set of the function F_n. Let C be an adequacy criterion that is applicable to the testing of the functions associated to transitions in a PrT net.

Definition 25 (Criterion of Transition Constraint Adequacy)

Let E be a collection of executions of PrT net N, E is *transition constraint adequate* with respect to C, if and only if for any feasible transition node n in N, $C(F_n, X_\mathrm{n})$ is adequate, where F_n is the function associated with transition node n, X_n is the set of inputs consumed by the firings of the node n in the executions E. □

An important property of test adequacy criteria is the axiom of inadequacy of empty testing [23, 27], which requires that it is inadequate according to the criterion if the software is not tested.

Lemma 12.

The criterion of transitional adequacy with respect to C subsumes transition coverage, if C satisfies the axioms of inadequacy of empty testing.

Proof. Assume that E is transition constraint adequate with respect to C. By the definition of inadequacy of empty testing, each feasible transition node is fired at least once in E. Therefore, by Definition 8, E is adequate according to transition coverage. □

The application of this adequacy criterion requires a tester to observe and record the computation of the output tokens inside each transition firing. For example, to analyse if the underlying algebraic formal specification has been adequately tested, we can record the equations used in the computation of output tokens and/or in the proof of the correctness of the transitions fired during test executions. The following adequacy criterion can then be defined based on such observations.

Definition 26 (Equation Coverage)

Let *Eq* be the set of equations of the formal specification *SPEC* underlying a PrT net *N*. A set *E* of executions of *N* is said to satisfy the *equation coverage* criterion, if and only if for all equations *w* in *Eq*, there is at least one execution *e* in *E* such that *e* contains at least one transition firing that the equation *w* is used in the proof of the correctness of the output tokens of the transition firing (or used in the evaluation of the output tokens in the transition firing). □

Notice that, it is possible that the proof of the correctness of a transition firing can be done using different equations in the algebraic specification. There is non-determinism in the select and use of the equations in such cases. The test adequacy criterion defined in Definition 26 requires that each equation is actually used at least once in one proof as observed during a testing process.

In general, let *C* be any given adequacy criterion for testing algebraic specification.

Definition 27 (Specification Coverage with respect to C)

Let *SPEC* be the algebraic formal specification underlying a PrT net *N*. A set *E* of executions of *N* is said to be *Specification adequate with respect to C*, if and only if the specification *SPEC* is adequately tested according to *C* in the executions of the transition firings in $e \in E$. □

This family of adequacy criteria links the testing of PrT nets at two different levels. It also enables the application of existing works on testing algebraic specifications to testing PrT nets. An obvious shortcoming of the method is that it neglected the network structure of PrT nets. Therefore, it should be used together with the methods discussed in sections 3~5.

7. Conclusion

In this paper, we proposed a methodology of testing high-level Petri nets based on our general theory of testing concurrent systems. We presented four groups of testing methods for high-level Petri nets: transition-oriented testing, state-oriented testing, data flow oriented testing and specification-oriented testing. Each method is formally defined by an observation scheme and an adequacy criterion. In addition to the subsuming relationships among the criteria, there are extraction relations between the observation schemes, which are summarized in Figure 5.

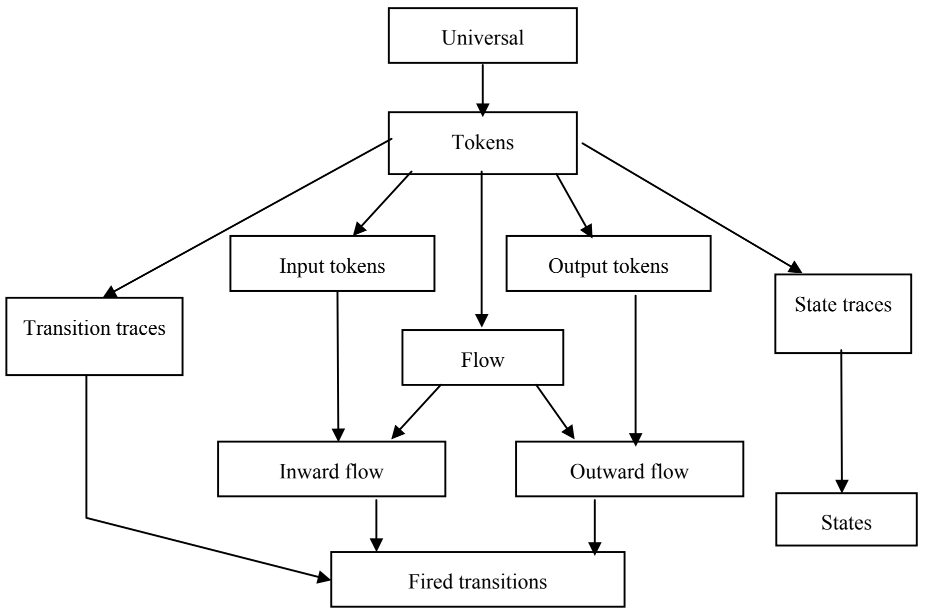


Figure 5 - Extraction relationships between of behaviour observation schemes

Our work shares the same viewpoint with the researchers on testing finite state machines and testing theories of process algebra that equivalence relations between two computation systems, i.e. finite state machines in the former and concurrent processes in the later, cannot be simply defined as a partial function from inputs to outputs (although this is appropriate for testing sequential programs), rather it must be defined in terms of its dynamic behaviour. However, there are several significant differences, including the goals, methods, and results, between our work and the above works. First, as discussed in section 1, the above works focused on a specific goal - behaviour equivalences defined in a formal specification technique such as CCS, LOTUS, or finite state machine using black-box testing approach. Our work is based on a general formal framework for testing concurrent systems, which covers black-box testing and white-box testing approaches, as well as the testing of different software artefacts including formal specifications with an underlying operational semantics and programs. In this paper, we have applied our framework to high-level predicate transition nets and developed a hierarchy of white-box testing methods. Second, the above works used a very simple observation scheme with the assumption that a program will provide a simple yet intelligent response such as a stable state being reached from each external stimulus. Our work provides extensive results on observation schemes and explores relationships (strengths and weaknesses) among the observation schemes. The observable behaviours are determined by the chosen observation scheme. By designing an appropriate observation scheme, we can cover known behaviour or semantic models such as the ones used in the above works and others such as failure semantics [46].

Although the testing methods are defined in terms of algebraic predicate transition nets with interleaving-set semantics, we believe that the testing methods should also be applicable to other models of Petri nets and other semantic models including the partial order semantics [39].

Acknowledgement

This work was partially supported by the National Science Foundation of the U.S.A. under grant INT-0096143.

References

[1] Bouge, L., Choquet, N., Fribourg, L., and Gaudel, M. C., Test set generation from algebraic specifications using logic programming, Journal of System and Software, Vol. 6, 1986, pp343-360.

[2] Beront, G., Gaudel, M. C. and Marre, B., Software testing based on formal specifications: A theory and a tool, Software Engineering Journal, Nov., 1991, pp387-405.

[3] Doong R. K. and Frankl, P. G. (1991), Case studies on testing object-oriented programs; Proceedings of the Symposium on Testing, Analysis, and Verification, 1991, pp165 - 177.

[4] Doong R. K. and Frankl, P. G. (1994), The ASTOOT approach to testing object-oriented programs; ACM Trans. Softw. Eng. Methodol. Vol. 3, No. 2, (Apr.), pp101 – 130

[5] Chen, H. Y. Tse, T. H. and Chen, T. Y., TACCLE: a methodology for object-oriented software testing at the class and cluster levels, ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 1, 2001,

[6] Chen, H. Y., Tse, T. H. and Deng, Y. T., ROCS: an object-oriented class-level testing system based on the relevant observable contexts technique, Information and Software Technology, Vol. 42, No. 10, 2000, pp677-686.

[7] Amla, N. and Ammann, P., Using Z specifications in category partition testing, Proc. of 7th IEEE Annual Conference on Computer Assurance, June 1992, pp3-10.

[8] Stocks, P. A., and Carrington, D. A., Test templates: A specification-based testing framework, Proc. of 15th International Conference on Software Engineering, May 1993, pp405-414.

[9] Ammann, P. and Offutt, J., Using formal methods to derive test frames in category-partition testing, Proc. of the 9th IEEE Annual Conference on Computer Assurance, June 1994, pp69-79.

[10] Zhu, H., Hall, P. and May, J., Software unit test coverage and adequacy, ACM Computing Survey, Vol. 29, No. 4, Dec. 1997, pp366~427.

[11] Taylor, R. Levine, D. and Kelly, C., Structural testing of concurrent programs, IEEE TSE, vol.18, no.3, 1992, pp206-215.

[12] Lee, D. and Yannakakis, M., Principles and methods of testing finite state machines -- a survey, Proc. of the IEEE, Vol. 84, pp. 1090--1123, Aug 1996.

[13] Binder, R., Testing Object-oriented Systems: Models, Patterns and Tools, Addison Wesley, 2000.

[14] Fujiwara, S., Bockmann, G., Khendek, F., Amalou, M. and Ghedamsi, A., Test selection based on finite state models, IEEE TSE Vol. 17, No. 6, June 1991, pp591-603.

[15] Hierons, R. M., Checking states and transitions of a set of communicating finite state machines, Microprocessors and Microsystems, Special Issue on Testing and testing techniques for real-time embedded software systems, vol. 24, No. 9, pp443-452, 2001.

[16] Bochmann, G., and Petrenko, A., Protocol testing: Review of methods and relevance for software testing, Proc. of ISSTA'94, Aug. 1994, Seattle, Washington, USA, pp109~ 124.

[17] Morasca, S. and Pezze, M., Using high-level Petri nets for testing concurrent and real-time systems, in Real-Time Systems, Theory and Applications, H. Zedan ed., North Holland, 1990.

[18] Carver, R. H. and Tai, K. C., Replay and testing for Concurrent Programs, IEEE Software, March 1991, pp66-74.

[19] Hennessy, M. and Milner R., Algebraic laws for nondeterminism and concurrency, Journal of ACM, Vol. 32, No. 1, January 1985, pp137~161.

[20] De Nicola, R. and Hennessy, M. C. B., Testing equivalences for processes, Theoretical Computer Science, Vol. 34, 1984, pp83-133.

[21] Phillips, I., Refusal testing, Theoretical Computer Science, Vol. 50, 1987, pp241-284.

[22] Baker, A. L., Howatt, J. W., and Bieman, J. M., Criteria for finite sets of paths that characterize control flow, Proceedings of the Nineteenth Annual Hawaii International Conference on System, 1986.

[23] Weyuker, E. J., Axiomatizing software test data adequacy, IEEE TSE, Vol.SE_12, No.12, 1986, pp1128-1138.

[24] Weyuker, E. J., The evaluation of program-based software test data adequacy criteria, Communications of the ACM, Vol.31, No.6, 1988, pp668-675.

[25] Parrish, A. & Zweben, S. H., Analysis and refinement of software test data adequacy properties, IEEE TSE, Vol. SE_17, No. 6, 1991, pp565-581.

[26] Parrish, A. S. and Zweben, S. H., Clarifying some fundamental concepts in software testing, IEEE TSE, Vol. 19, No.7, 1993, pp742~746.

[27] Zhu, H. & Hall, P., Test data adequacy measurement, SEJ, Vol. 8, No.1, 1993, pp21~30.

[28] Zhu, H., Hall, P., and May, J., Understanding software test adequacy -- An axiomatic and measurement approach, in Mathematics of Dependable Systems, Edited by Mitchell, C., and Stavridou, V., Oxford University Press, 1995, pp275~295.

[29] Zhu, H., Axiomatic assessment of control flow based software test adequacy criteria, Software Engineering Journal, Vol. 10, No. 9, 1995, pp194-204.

[30] Zhu, H., A formal interpretation of software testing as inductive inference, Journal of Software Testing, Verification and Reliability, Vol. 6, No.1, 1996, pp3~31.

[31] Zhu, H., and He, X., A Theory of Behaviour Observation in Software Testing, Technical Report CMS-TR-99-05, School of Computing and Mathematical Sciences, Oxford Brookes University, Sept. 1999.

[32] Zhu, H. and He, X., Constructions of behaviour observation schemes in software testing, Proc. 5th IEEE Symposium on HASE'2000, 15~17 Nov. 2000, Albuquerque, New Mexico, USA, pp7~16.

[33] Laski, J. and Korel, B, A data flow oriented program testing strategy, IEEE Transaction on Software Engineering, Vol. SE-9, 1983, pp33-43.

[34] Ntafos, S. C., On required element testing, IEEE Transaction on Software Engineering, Vol. SE_10, No. 6, 1984, pp795-803.

[35] Rapps, S. & Weyuker, E.J., Selecting software test data using data flow information, IEEE

TSE, Vol.SE_11, No.4, 1985, pp367-375.

[36] Frankl, P.G. & Weyuker, J.E., An applicable family of data flow testing criteria, IEEE TSE, Vol.SE_14, No.10, 1988, pp1483-1498.

[37] Genrich, H. and Lautenbach, K. System modeling with high-level Petri nets, Theoretical Computer Science, vol.13, 1981, pp109-136.

[38] Kan, C. and He, X. High-level algebraic Petri nets, Information and Software Technology, vol. 37, no.1, 1995, pp23-30.

[39] Reisig, W., On Semantics of Petri Nets, Formal Models in Programming, Neuhold E. and Chroust G., (eds.), Elsevier Science Publishing, North Holland, 1985, pp347-372

[40] Zhu, H., A formal analysis of the subsume relation between software test adequacy criteria, IEEE Transactions on Software Engineering, Vol. 22, No. 4, 1996, pp248~255.

[41] Zhu, H. and He, X., A theory of testing high-level Petri nets, Proc. of International Conference on Software – Theory and Practice, IFIP World Computer Congress 2000, Beijing, August 21-25, pp443~450.

[42] He, X. A formal definition of hierarchical predicate transition nets, in Lecture Notes in Computer Science, vol.1091, 1996, pp212-229.

[43] Zhu, H., Validating Algebraic Class Testing in Final Algebra, Submitted to ISSTA'2002.

[44] Gopal, A. & Budd, T., Program testing by specification mutation, Technical report TR 83-17, University of Arizona, November 1983.

[45] Woodward, M.R., Erros in algebraic specifications and an experimental mutation testing tool, SEJ, July 1993, pp211-224.

[46] Brookes, S. D., Hoare, C. A. R. and Roscoe, A. W., A theory of communicating sequential processes, Journal of the ACM, Vol.31, 1984, pp560-569.