Evaluating Test Adequacy Coverage of High Level Petri Nets Using Spin

Junhua Ding, Peter J. Clarke, Gonzalo Argote-Garcia, and Xudong He
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
email: {jding01, clarkep, gargo001, hex}@cis.fiu.edu

Technical Report FIU-SCIS: 2006-05-02

Abstract

High level Petri nets have been extensively used for system modeling; however their strong expressive power cost their analyzability. Currently, there are no effective general formal analysis techniques for high level Petri nets. Fortunately, high level Petri nets are executable and as a result they can be tested. In recent years, some theoretical testing adequacy coverage measurements have been proposed. In this paper, we propose an approach to validate the above theoretical results through experiments using the simulation functionality of the model checker Spin.

1 Introduction

Software systems have increasingly become more complex and safety and mission critical. How to ensure the dependability of complex software systems is a grand research challenge. Modeling especially based on a well-defined formal method plays an essential and critical role in large system development.

High level Petri nets [7] are a formal computation model well suited for concurrent and distributed systems, and have been extensively applied to system modeling in almost every branch in computer science as well as in many other scientific and engineering disciplines. There have been over 10000 publications related to Petri nets world wide in the past few decades, and the number is increasing rapidly. The benefits of high level Petri nets as a modeling method are particularly significant due to the nature of today's software systems that are operating concurrently in a distributed and networked environment. The strong modeling power of high level Petri nets covering functionality, data, behavior, and structure makes them difficult to analyze. Despite various attempts in extending traditional Petri net reachability tree analysis technique and adapting model checking techniques to high level Petri nets in the past decades, there are no effective general formal analysis techniques for them. Fortunately, high level Petri nets are executable and thus they can be tested as programs. Since high level Petri nets are often more abstract and concise, they are often much more simple than programs. In the past few years, some testing theories and methods for concurrent systems in general and Petri nets in particular have been proposed [16]. However the testing theories and methods proposed in [16] have not been validated for their practical use due to a lack of suitable tools for recording the dynamic behavior and for measuring test adequacy coverage. Recently, we exploited the simulation capability of the well-known model checker Spin [9], and found its applicability in validating the testing methods of high level Petri nets.

In this paper, we present our new results on how to use the simulation capability of model checker Spin to validate the testing methods of high level Petri nets. We use the model checker Spin [9] to control the execution of a high level Petri net expressed as a Promela [9] process with a global state and use a monitor (another Promela process) to record and evaluate the events generated by the net process against test coverage criteria [16]. We

present two approaches used to evaluate the Petri net: (1) external evaluation whereby net events are recorded in the monitor and the evaluation is done by a program external to Spin, and (2) an internal evaluation process whereby the evaluation is done in a monitor process during the simulation of the net using Spin. We present a case study to show how our approach can be scaled to models consisting of more than one high level Petri net.

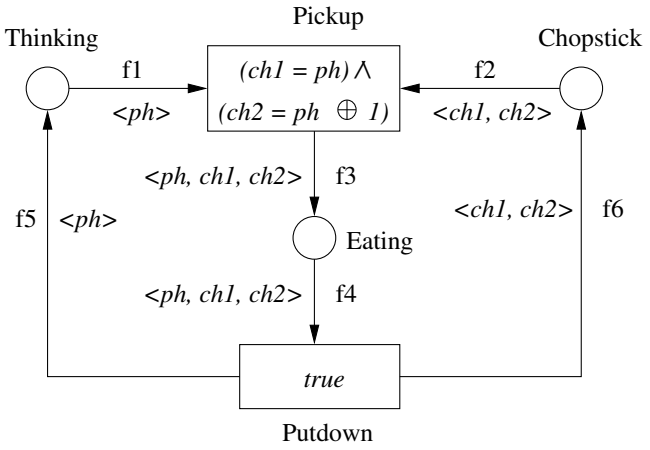
In the next section we present concepts essential to high level Petri nets, testing of high level Petri nets, and the simulation capability of Spin. Section 3 describes how we record net events and evaluate the test coverage criteria. Section 4 describes how we use the Spin model checker to simulate the behavior of a high level Petri net and to evaluate testing coverage criteria. Section 5 presents a case study. In Section 6 we provide a discussion on our approach. Section 7 describes related work followed by concluding remarks in Section 8.

2 Background

Predicate transitions nets (PrT nets) are a class of high level Petri nets, and can be used to define other high level Petri nets easily. We choose PrT nets in this paper due to the facts that the formal definitions of PrT nets are very close to the newly proposed international standards on high level Petri nets [7] and the testing theory and methods proposed in [16] are based on PrT nets.

In this section we overview several key concepts related to predicate transition nets (PrT nets), testing PrT nets, and the Spin model checker.

2.1 Predicate Transition Nets



$\varphi(Thinking) = \wp(PHIL)$
$\varphi(Chopstick) = \wp(CHOP)$
$\varphi(Eating) = \wp(PHIL, CHOP, CHOP)$
$PHIL$ and $CHOP$ are sorts to represent philosophers and chopsticks respectively.
Where $\oplus$ is modulus k addition and $\wp(A)$ is the power set of A.

Figure 1. A PrT net for dining philosophers.

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) [7, pp. 459-476]. P and T are the set of predicates and transitions, respectively, where $P \cap T = \emptyset$. F is the flow relation where $F \subseteq P \times T \cup T \times P$. $SPEC$ 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 $\Sigma = (S, OP)$ and a set Eq of Σ-equations. S is a set of sorts and OP is a family of sorted operations. Tokens of a PrT net are ground terms of the signature Σ, written $MCON_S$. The set of labels is denoted by $Label_S(X)$, where X is the set of sorted variables disjoint with OP. Each label can be a multiset expression of the form

Markings m_i			Transitions n_i	
Thinking	Eating	Chopstick	Fired Transition	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
$\{1,3,5\}$	$\{< 2,2,3 >, < 4,4,5 >\}$	$\{1\}$	...	...

Table 1. A flat execution of the dinning philosophers' PrT net.

$\{k_1x_1, ..., k_nx_n\}$. Constraints of a PrT net are a subset of first order logic formulas containing the S-terms of sort *bool* over X, denoted as $Term_{OP,\,bool}(X)$.

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. Mapping $\varphi : P \rightarrow \wp(S)$ assigns a subset of sorts to each predicate p in P, which defines its valid values, i.e. proper tokens. Mapping $L : F \rightarrow Label_S(X)$ is a sort-respecting labeling of flows. Mapping $R : T \rightarrow Term_{OP,\,bool}(X)$ associates each transition t in T with a constraint expressed in a first order logic formula in the underlying algebraic specification. The constraints define a transition in terms of pre-condition and post-conditions. The pre-condition specifies the constraints on the incoming arcs and the post-conditions specify the relationships between the variables of the incoming arcs and label variables of the outgoing arcs.

A marking m of a PrT net is a mapping $P \rightarrow MCON_S$ from the set of predicates to multi-sets of tokens. M_0 is a set of initial markings, which are thus the test cases. A transition is enabled if its pre-set contains enough tokens and its constraint is satisfied with an occurrence mode. The pre-set $({}^\bullet t)$ for a transition are the set of input places for that transition. Similarly, the post-set $(t^\bullet)$ for a transition are the set of output places for that transition. 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) fire concurrently if they are not in conflict. Conflicts are resolved non-deterministically. The firing of an enabled transition is atomic. We define the behavior of a PrT net to be the set of all possible execution sequences E. Each execution sequence $e \in E$ represents consecutively reachable markings from the initial marking, in which a successor marking is obtained through a step (firing of some enabled transitions) from the predecessor marking. We denote an execution as

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

where n_i is a set of transitions, $m_0 \in M_0$ is an initial marking, $m_i, i = 1, 2, ...$, are markings such that m_i is obtained from m_{i-1} by firing transition set n_i. The execution sequence e is said to be *flat* if all the n_i's are singletons, otherwise e is said to be *non-flat*. We can obtain a flat execution sequence from a non-flat execution sequence by interleaving the transitions in the non-singleton n_i's.

Figure 1 shows a PrT net model for the dinning philosophers problem. The PrT net in Figure 1 consists of three predicates - *Thinking*, *Eating* and *Chopsticks*, and two transitions - *Pickup* and *Putdown*. The flow relation in Figure 1 consists of the six labeled arcs *f1* through *f6*, and annotated with variables of the sorts shown in italics.

Table 1 shows the markings and tokens consumed after the transitions *Pickup, Putdown, Pickup* and *Pickup* are fired. The first three columns of Row 1 in Table 1 show the initial marking m_0 of the PrT net. The first transition fired ($n_0 = \{Pickup\}$) is shown in Column 4 of Row 1 and the tokens consumed are shown in Column 5 of Row 1. For example, Row 1 of Table 1 states that the initial marking of the net in Figure 1 consists of Thinking $= \{1,2,3,4,5\}$, Eating $= \{\}$, and Chopstick $= \{1,2,3,4,5\}$. After firing transition *Pickup* the tokens consumed are $ph = 1$, $ch1 = 1$ and $ch2 = 2$, giving the marking in the first three columns of Row 2. The partial execution sequence in Table 1 is a flat execution sequence since multiple transitions do not fire at any point in the execution. An example of a non-flat partial execution is given in [16].

2.2 Testing PrT Nets

PrT nets are formal models that use graphical and mathematical notations, and are well suited for modeling and analysis of concurrent and distributed systems. Zhu and He [16] state that PrT nets can play two different

roles in the development of concurrent systems: (1) as a formal specification, and (2) as an executable model. In general hese two roles provide the developer with the opportunity to combine both verification and testing of the PrT net model thereby providing a higher level of confidence in the correctness of the system. Verification of the PrT net model can be performed by using a model checker e.g., Spin, to check various properties of the net [6]. The properties of a PrT net allow the application of both specification-based and program-based testing techniques. This is possible since a PrT net is considered as both a specification and an executable model. In this paper we will focus on structural test coverage criteria of PrT nets.

Beizer [2] defines test coverage as any metric of completeness with respect to a test selection criterion. Zhu et al. [15] further explains the notion of test data adequacy criteria by describing the three definitions of test data adequacy criteria. These include: (1) test data adequacy as stopping rules, (2) test data adequacy criteria as measurements, and (3) test data adequacy criteria as generators. In this paper we focus on the second definition presented by Zhu et al. [15], test adequacy criteria as measurements. The formal definition presented by Zhu et al. [15] states that test data adequacy criteria as measurements is a mapping from the cross product of a set of programs, a set of specifications, and a class of test sets to a real number in the range of zero and one. The greater the real number the more adequate the test set is. Applying the dual nature of PrT nets to the concept of test data adequacy criteria as measurements imply that: (1) PrT nets, execution models, map to the set of programs, (2) PrT nets map to the set of specifications, and (3) the initial markings for each PrT net represent the test sets.

Zhu and He [16] provide a methodology of testing high-level PrT nets based on general theory of testing concurrent software systems. They identified four classes of testing strategies: *transition-oriented testing, state-oriented testing, flow-oriented testing*, and *specification-oriented testing* [16]. 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 authors formally define the concept of an *observational scheme* for concurrent system p as the ordered pair $<B, \mu>$ where B is the set of partial orders of events generated by p, and μ represents the mapping from a test set to a non-empty consistent subset of all partial orders for p [16]. Note that due to non-determinism and concurrency, two or more partial orders may be generated by the same test input for a given p. Zhu and He [16] state that unlike test data adequacy criteria (used to measure the adequacy of a test set), an observation scheme determines how to observe and record a system's dynamic behavior during test executions.

In this paper we focus on coverage criteria for (1) transition-oriented testing i.e., *transition coverage*, and (2) state-oriented coverage i.e., *state coverage*. These coverage criteria are defined below. We use N to represent a PrT net and E to represent a collection of test executions of N.

1. *Transition coverage* (Transition-oriented testing) - is the ratio of transitions fired during an execution of a PrT net to the total number of transitions.

$$TransitionCoverage(N, E) = \frac{\left\| \bigcup_{e \in E} Firing(e) \right\|}{\|T_N\|}$$

   where $Firing(e) = \bigcup_{i=0,1,...n_i}$, $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} ... \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} ...$, and n_i are non-empty subsets of transitions that are not in conflict with each other. T_N is the set of transitions of N. Note that a $Trace$ of an execution e is defined as $Trace(e) = < n_0, n_1, ...n_k, ... >$. In this paper we use the flatten version of the execution sequence.

2. *State coverage* (State-oriented testing) - is the ratio of of the reachable markings associated with abstract states during an execution to the finite set of abstract states for the PrT net. An *abstract state* (AS) is one of a finite set of states that is reachable in a PrT net given an initial marking.

$$StateCoverage(N, E) = \frac{\left\| State_N \left(\bigcup_{e \in E} Markings(e) \right) \right\|}{\|AS_N\|}$$

   where $State_N : Mark(N) \rightarrow AS_N$ defines how markings are associated with states. AS_N is a finite set of abstract states of N. Examples of abstract states for the dinning philosopher's problem are given in Section 3.2.

2.3 Spin

Spin [13] is a generic model checking tool to formally analyze the logical consistency of distributed systems, which are defined using Promela [9]. Spin has three basic functions: (1) As an exhaustive state space analyzer for rigorously proving the validity of user-specified correctness requirements. (2) As a system simulator for rapid prototyping. (3) As a bit-state space analyzer that can validate large protocol systems with maximal coverage of the state space.

Promela [9] is a verification modeling language with C programming language style. It provides a way for making abstractions of distributed systems that suppress details that are unrelated to process interaction. A Promela program consists of processes, message channels, and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which processes run. In this paper we will focus on the simulator component of the Spin tool.

Spin offers three options for performing simulation these include: (1) random, (2) interactive, and (3) guided. The simulation mode in Spin is intended primarily for the debugging of a model. The random simulation option allows a user to monitor the behavior of a model by printing any output produced by the model to the console. Interactive simulation allows a user to resolve non-deterministic choices during the simulation of the model by selecting an option during the simulation process. If there is only one option then Spin will immediately select that option and continue the simulation. Guided simulation uses a specially encoded trail file generated by the verifier, after a correctness violation, to guide the search. The execution sequence stored in the trail file represents the events leading up to the error.

We use the random simulation option to monitor the behavior of PrT net models. It is possible for a Spin simulation to be executed indefinitely, making it difficult to monitor the behavior of the model from the console. Therefore we pipe the output from the simulation to a file for analysis at a later time. To limit the output of the simulation on the model we use different combinations of the `-u` and `-j` options. The `-u`N option limits the simulation to the first N steps, and the `-j`N option skips over the first N steps. There are other options provided by the Spin tool [13] that provides for additional flexibility in managing the output from the random simulation facility.

3 Recording Dynamic Behavior for Test Coverage Evaluation

In this section we describe our approach to evaluating the event stream generated by the execution of PrT nets using the two classes of testing strategies identified by Zhu and He [16]. These classes of testing coverage strategies are: *transition* and *state*, introduced in Section 2.2. We also explain how our technique may be extended to other variants of *transition-oriented* and *state-oriented* testing strategies described by Zhu and He [16].

3.1 Transition-Oriented Coverage

Transition coverage criteria is one of three forms of transition-oriented described by Zhu and He [16]. The others coverage criteria are *K-concurrency length-L trace coverage*, *all transition trace coverage* and *interleaving length-L transition sequence coverage*. A hierarchy of transition-oriented coverage criteria is presented in [16]. We will focus on transition coverage criteria.

Transition coverage is defined over a collection of executions E in terms of the total number of transitions fired F_E and the total number of transitions in the PrT net T_N see Section 2.2. Our approach executes the PrT net using simulation for a finite set of executions recording the transitions fired. We then determine the transition coverage using the quotient stated in Section 2.2. During execution of the PrT net we record the transitions fired in a log. Note that since each PrT net has a finite set of transitions it is possible to get 100% test coverage if the model is allowed to execute for a long enough time and the initial markings (test cases) are adequate. However, If the transition coverage was less than the adequacy required by the tester the information in the log is used to perform other forms of validation such as manual inspection. In Section 4.2 we describe the two approaches used to evaluate the coverage criteria during the simulation process. The simulation environment provided by Spin is adequate to determine transition coverage for PrT net models.

K-concurrency length-L trace coverage over a collection of executions E of a PrT net is defined as, the existence of at least one $e \in E$ covered by the transition trace q with length less than or equal to L and concurrency degree less than or equal to K [16]. K and L are natural numbers greater than zero. The concurrency degree of a transition trace is the maximum number of transitions that may fire between any successive markings in that trace. We can use a similar approach to the one described in the previous paragraph to test whether a feasible transition trace q of N is covered by at least one execution e of the test set $E(e \in E)$, and to decide the coverage of any trace coverage. The added difficulty is how to determine the concurrency degree for a transition trace, since we have not yet found a way to record multiple transition firings in Promela.

Interleaving length-L transition sequence coverage over a collection of executions E of a PrT net is defined as, given any feasible transition sequence q with length less than or equal to L there is at least one $e \in E$ the logically covers q [16]. A execution e logically covers sequence q if a flattening of e contains q as a consecutive subsequence of transition firings. Using Spin it is possible to perform interleaving length-L transition sequence coverage. To obtain the best results it is better to chose the longest possible feasible transition sequence q since the longer sequence coverage subsumes the shorter sequence coverage [16]. *All transition trace coverage* requires that there is at least on $e \in E$ that covers any feasible transition trace q. In general all transition trace coverage is infeasible.

3.2 State-Oriented Coverage

The state-oriented test coverage described by Zhu and He [16] includes *state coverage*, *state transition coverage*, and *state transition path coverage*. We focus on state coverage defined in Section 2.2. Recall that state coverage is defined over a collection of executions E for a PrT net N, if for all feasible states $s \in AS_N$, there is at least one execution e in E such that there is at least one $m \in Markings(e)$ and $State_N(m) = s$ [16]. The main challenge in our approach is the identification of the abstract states to be used during model simulation. The concept of the abstract states provides a way to reducing the state space of the model by identifying a set of finite states to be used during state coverage testing. Note that if the state space is small enough then there can be a one to one mapping to the states to the abstract states. In general a mapping needs to be defined from the marking in the net to the abstract states, due to the state explosion problem.

An example of a set of abstract states for the dining philosophers problem is the number of philosophers eating. In this cases the state are 0, 1, 2. One set of possible markings associated with these state are: 0 - *Thinking* = {*1,2,3,4,5*}, *Eating* = {}, *Chopsticks* = {*1,2,3,4,5*}, 1 - *Thinking* = {*2,3,4,5*}, *Eating* = {<*1,1,2*>}, *Chopsticks* = {*3,4,5*}, and 2 - *Thinking* = {*1,3,5*}, *Eating* = {<*2,2,3*>, <*4,4,5*>}, *Chopsticks* = {*1*}. Therefore by inspecting the content of the variables that store the markings of the net it is possible to deduce which abstract states have been covered. Note that in the above example there is one possible marking for the abstract state 0, five possible markings for the abstract state 1, and five possible markings for abstract state 2. In our example in Section 4.3 we use the abstract states previously described in this paragraph.

State transition coverage over a collection E of test executions is satisfied, if for all feasible state transitions $< s_1, s_2 >$ there is at least one execution e in E such that e covers that state transition. We can extend our current approach for state transition coverage. For example, using the abstract states described in the previous paragraph it is possible to track if the transition from the abstract state 0 to the abstract state 1 fires or not. To handle the situation where two concurrent transitions fire would require additional modification to the implementation of the net since Spin does not automatically handle concurrent state changes.

The final state-oriented testing criteria considered by Zhu and He [16] is *state transition path coverage*. State transition path coverage, more specifically *length-k state transition path coverage*, is defined over a set of execution E and is satisfied, 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 path q. Assume k is a natural number greater than 1. A state transition path of length k is defined as a sequence of states $< s_1, s_2, ..., s_k >$. In general it is not practical to handle state transition path coverage, however, the restriction of specifying a length of the path makes it more practical. We do not address this coverage criteria in our current approach, but with a small enough k it would be possible to create a state machine that could be used to keep track of the state transition paths covered.

4 Evaluating Test Coverage Criteria of PrT Net Models

In this section we describe two approaches that use Spin to analyze PrT net models for various test coverage criteria. We also provide insight on the translation process from a PrT net into a Promela program using the dinning philosopher problem. In addition, we use this example to show how Spin analyzes PrT net models for state and transition test coverage criteria.

4.1 Transforming PrT Nets to Promela

In order to simulate PrT nets using Spin, it is necessary to translate PrT net models into Spin models - specified using Promela. Several researchers have used Spin to check models specified using Petri nets [3, 4, 6]. The basic idea is to translate Petri nets into equivalent Spin models - Promela programs, and their properties into assertions or never claims in Promela programs. We provide a general procedure and rules for translating a PrT net into a Promela program in Spin.

The Promela program structure. Each individual net is translated into a process in the Promela program, assuming the PrT net is a composition of other PrT nets. The sorts of a PrT net are translated into integer types and structured types in Promela. Predicates (places) in the PrT net are translated into fixed-length array variables. The transitions in the PrT net are translated into a process. The init process is used to assign initial values for the program according to the PrT net initial markings.

Translating predicates into Promela. Each predicate in a net is translated into a global variable in the Promela program. The type of the predicate is translated into an equivalent variable type in the Promela program. If a type is not a predefined type in the Promela program, then the type needs to be defined in the Promela program, which has the same domain, range and operations as the type in the PrT net. This may not be possible because of the restricted types in Promela.

Given a predicate T is defined in the PrT net: $\varphi(T) = \wp(PHIL, CHOP, CHOP)$, then a corresponding type will be defined for the type of predicate T: typedefine T {byte ph, byte ch, byte ch} in the Promela program. The value range of each variable represents the possible markings of the predicate in the PrT net. Therefore, the number of possible values of a variable is the number of possible markings of the corresponding place. If a place p is k-bounded, the declaration statement for place p is an array with k elements and its type is the predicate type. Thus, we treat a predicate symbol as a set of proposition symbols. This can be done when each p is bounded and $—\varphi(p)—$ is finite [8].

Translating transitions into Promela. The transitions in the PrT net are translated into a Promela process. The transitions are enclosed in a do .. od statement. Each transition is defined as a guarded atomic statement within that process. This atomic statement defines the firing rules of the transition. The combination of the do .. od statement and the guarded atomic statements ensures the non-deterministic firing of the transitions. Global variables and channel variables can be used to synchronize and communicate between different processes.

Defining the initial marking. Each global variable of the Promela program is initialized via the init process with a value that is the initial marking of the corresponding predicate in the PrT net.

The Dinning Philosopher problem shown in Figure 1 is translated into the Promela program shown in Appendix I.

4.2 Evaluation of Test Criteria

Figure 2 shows a high level representation of the two approaches that use Spin to analyze PrT nets for various test coverage criteria. We refer to the tool in Figure 2 as the *Test Coverage Criteria Analyzer* or *TCC_analyzer*. In both approaches the PrT net model is converted into Promela programs using the steps outlined in the previous section. In addition, the Promela program is instrumented with statements to collect events, E_{TC}, for test coverage analysis to a special process, referred to as the *Monitor* [11].

Figure 2(a) shows the approach that evaluates the test coverage criteria of the PrT net external to Spin. In the external evaluation approach, the monitor, $Monitor_R$, is used to record the events, E_{TC}, that are later analyzed

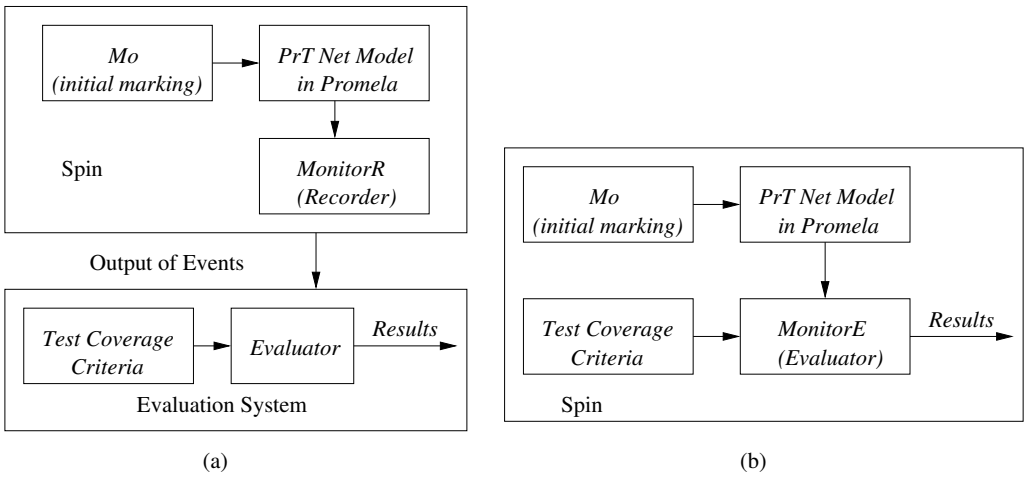


Figure 2. Test coverage criteria analyzer. (a) External evaluation. (b) Internal Evaluation.

using the various test coverage criteria by a program external to Spin. The external evaluation process, shown in Figure 2(a) as the box labeled *Evaluation System*, uses the criteria provided by the box labeled *Test Coverage Criteria* and checks the event trace supplied from the Spin tool for the level of coverage. This is done by simply searching the event trace for the events associated with a specific test coverage criterion.

Figure 2(b) shows the approach that internally evaluates the PrT net model against the test coverage criteria. The internal evaluation process is done in the Spin tool by a runtime monitor. The test coverage criteria are supplied to the monitor during process initialization and as events E_{TC} are accepted by the $Monitor_E$ process the evaluation of the test coverage criteria occur.

The results generated from both approaches shown in Figure 2 parts (a) and (b) identify the coverage obtained by the initial marking M_0 supplied to the net. In Section 3 we describe the test coverage criteria and the associated events generated for each test criterion.

4.3 Analyzing a PrT Net Model

An outline of the Promela program used to analyze test coverage criteria for the Dining Philosopher PrT net (see Figure 1) is shown in Figure 3. Recall that the PrT net for the dinning philosophers problem consist of three predicates $(Thinking, Eating, Chopsticks)$ and two transitions $(Pickup, Putdown)$. The flow relation in Figure 1 consists of six arcs labeled $f1$ through $f6$, each arc is also labeled with a tuple representing the types of the tokens consumed when a transition fires. In this example we present an approach that evaluates the test coverage criteria for transition and state coverage. When a transition is fired in an execution of the net we conclude that the transition has been covered. We identify three abstract states for state coverage, these are *not-eating* - no philosopher is eating, *one-eating* - one philosopher is eating, and *two-eating* - two philosophers are eating.

The Promela program in Figure 3 consists of four main sections: (1) global declarations to define user-defined types, global variables and other macros, lines 1 through 8, (2) process DP representing the PrT net in Figure 1, line 10 through 24, (3) process Monitor, representing the monitor in Figure 2(b), lines 26 through 38 and (4) process init used to initialize global variables and start each process. The complete listing of the Promela program is shown in Appendix I.

The global declarations section contain the constant N that holds the number of philosophers and chopsticks, the type declaration tokenE for tokens in the eating predicate, and arrays for the predicates thinking, chopstick and eating. There are also two arrays for the transitions and the states used by the monitor during test coverage criteria analysis. The process DP consists of two atomic statements, which are used to model the transitions *pickup* and *putdown*. Each atomic statement consist of a guard condition, statements representing the execution of the transition, and statements used to update the global variables used during test coverage analysis. When the transitions fire according to PrT net firing rules the variables representing the predicate are updated to reflect the

```
#define N 5
typedef tokenE{ byte ph; byte ch1; byte ch2;}
/*token type used in Eating predicate*/

 byte thinking[N]; byte chopstick[N]; tokenE eating[N];
 /*PrT net predicates - philosophers, chopsticks, eating*/
 byte trans[2]; /*Monitor - transitions*/
 byte states[3]; /*Monitor - states*/

proctype DP(){
   ....
   do
   /*transition pickup*/
   :: atomic { /*guard for transition pickup is enabled*/
              -> /*execute transition pickup*/
                 /*update global variables used in Monitor*/
   }
   /*transition putdown*/
   :: atomic { /*guard for transition putdown is enabled*/
              -> /*execute transition putdown*/
                 /*update global variables used in Monitor*/
   }
   od
}

proctype Monitor(){
/*Monitor checks transition and state coverage criteria*/
  do
   :: atomic { /*guard for transition coverage criteria*/
        /*if guard condition fails then skip*/
       }
   :: atomic { /*guard to check state coverage criteria*/
        /* if guard condition fails then skip */
       }
   :: else -> break
   od;
   assert(false) /* Stops simulation */
}
init {
/*Initialize global variables */
/*Initialize initial marking */
/*Run processes DP and Monitor */
}
```

Figure 3. An outline of Promela program used to analyze test coverage criteria.

behavior of the tokens. If multiple transitions are enabled at the same time, the statement selected to be executed in the Promela program is chosen non-deterministically.

The approach shown in Figure 3 is an internal evaluation method (see Figure 2(b)). During the simulation of the PrT net, events are recorded in the global variables trans and states for the transitions and states respectively. Initially the contents of the arrays trans and states are set to 0. When the transition *pickup* is fired trans[0] is assigned 1 and state[1] (one-eating) is assigned 1. When the transition *putdown* is fired trans[1] is assigned 1 and the value of state[0] is assigned to 1 if there are no philosophers eating, i.e., all the philosopher fields in the variable eating are zero. If we are one-eating state and a the transition *pickup* fires then state[2] is assigned to 1, signifying that we have entered the two-eating state. We have implemented several variations of the monitor we discuss further in Section 6, including the external evaluation approach shown in Figure 2(a).

5 Case Study

In this section, we use our approach to evaluate the testing coverage criteria of the PrT net model for the Alternating Bit Protocol (ABP). The ABP is a protocol that consists of a sender, a receiver, and two channels, for reliable transmission over channels that may corrupt, but not duplicate, messages. The channels can detect a corrupted message or acknowledgment, and then a message or acknowledgment is resent when the corrupted message or acknowledgment is detected. The protocol guarantees that (1) an accepted message will eventually be delivered, (2) an accepted message is delivered only once, and (3) the accepted messages are delivered in order.

5.1 Specifying ABP

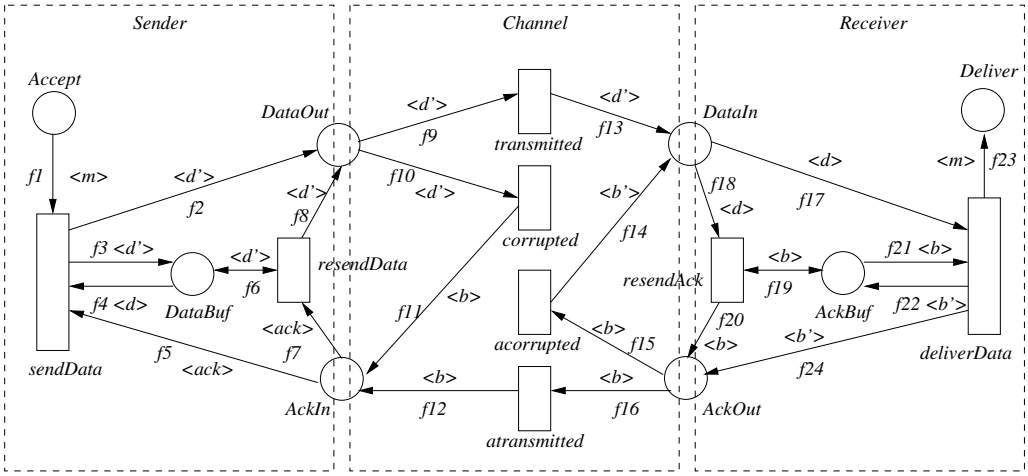


Figure 4. A PrT net model of ABP protcol.

The PrT net for the Alternation Bit Protocol (ABP) is shown in Figure 4. The net model has three components, the *Sender*, the *Channel* and the *Receiver*. The *Sender* component of the net accepts messages from the environment via the *Accept* predicate, shown on the upper left of Figure 4, and send them to the *Channel* component via the *DataOut* predicate, shown on the dotted lines between the *Sender* and *Channel*. These messages are passed from the *Channel* component via the *DataIn* predicate to the *Receiver* component and delivered to the environment via the *Deliver* predicate shown in the upper right of Figure 4.

The *Sender* component of the net has four predicates (*Accept, DataBuf, DataOut, Ackin*), two transitions (*sendData, resendData*) and eight arcs ($f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8$). Two of the predicates (*DataOut, AckIn*) are shared with the Channel component of the net. The arcs in the *Sender* component of the net are annotated based on the tokens consumed or created. These annotations include: $<m>$ - original message, $<d>$, $<d'>$ - representing the message and a bit value, and $<ack>$ a bit acknowledgment. The *Channel* component of the net contains four predicates (*DataOut, Ackin, DataIn, AckOut*), four transitions (*transmitted, corrupted, acorrupted, atransmitted*), and eight arcs ($f_9, f_{10}, f_{11}, f_{12}, f_{13}, f_{14}, f_{15}, f_{16}$). The predicates *DataIn* and *AckOut* are shared with the *Receiver* component of the net. The *a* prefixing the transitions represents the acknowledgment e.g., *acorrupted* - corrupted acknowledgment. The annotations on the arcs are similar to those in the *Sender* component except that *b* is used to represent corrupted or acknowledgment tokens. The *Receiver* component of the net is similar to the *Sender* component, except that the message is delivered to the environment, and the acknowledgment generated in that *Receiver* component.

The inscriptions for the ABP net shown in Figure 4 are given below.

1. Net inscription of the *Sender* model:
   $\varphi(Accept) = \wp(MESSAGE)$, where $MESSAGE$ is the type of string
   $\varphi(AckIn) = BIT \cup corrupted$, where $BIT = \{0, 1\}$
   $\varphi(DataOut) = \varphi(DataBuf) = BIT \times MESSAGE$
   $R(sendData) =$
   $(ack \in BIT \ \wedge \ ack = d[1] \ \wedge \ d'[1] = 1 - ack \ \wedge \ d'[2] = m)$
   $R(resendData) = (ack = corrupted)$

2. The net inscription of the *Receiver* model is as follows:
   $\varphi(Deliver) = \wp(MESSAGE)$
   $\varphi(AckOut) = \varphi(AckBuf) = BIT$
   $\varphi(DataIn) = (BIT \times MESSAGE) \cup corrupted$
   $R(deliverData) =$

$(d \in BIT \times MESSAGE \ \wedge \ d[1] = 1 - b \ \wedge \ b' = d[1] \ \wedge \ m = d[2])$
$R(resendAck) = (d = corrupted)$

3. The net inscription of the *Channel* model is as follows:
   $R(corrupted) = (b = corrupted)$
   $R(acorrupted) = (b' = corrupted)$

We provide a Promela implementation of the ABP in Appendix II. Using the guidelines in Section 4.1 we identify the ABP model as consisting of four PrT nets. These are the *Sender*, *Receiver*, *Mchannel* and the *Achannel*. The *Channel* component of the net in Figure 4 is partitioned into the *Mchannel* - message channel and the *Achannel* - acknowledgment channel. This view of the ABP net results in four Promela proctype constructs for each of the sub-nets *Sender, Receiver, Mchannel* and *Achannel*. Each transition is represented as an atomic statement in the proctype representing their sub-nets. In the next section we describe how the test criteria were applied to the ABP net.

5.2 Evaluating Test Coverage of ABP Net

In order to adequately perform transition and state test coverage on the ABP, specified by the PrT net in Figure 4, we need to model the transitions of the ABP net model, as well as identify and model the appropriate abstract states in Promela. Recall we use the definition of the an execution e as the basis for or definition of transition coverage. In Section 2.1 we defined an execution e of a PrT net as $e : m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} ... \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} ...$ where n_i is a set of transitions, $m_0 \in M_0$ is an initial marking, $m_i, i = 1, 2, ...$, are markings such that m_i is obtained from m_{i-1} by firing transition set n_i. Since we are using flat execution sequences in our simulation $n_i...n_k$ are all single transitions.

The transitions in the ABP net shown in Figure 4 consists of *sendData, resendData, transmitted, corrupted, acorrupted, atransimitted, resendAck* and *deliverData*. In the Promela program for the ABP, shown in Appendix II, we use a byte array of size 8 to store the information for transition coverage. Each element in the array is initialize to a value of 0. As the transitions are fired the corresponding array elements are assigned a value of 1 by invoking the inline function set_trans_covered(idx) with the appropriate parameter value (see line 94 of the ABP Promela program in Appendix II).

In Section 2.2 we defined abstract states of a PrT net as a set of finite states where each state in this set is reachable from the initial marking of the net. In addition there is the relation, $State_N : Mark(N) \rightarrow AS_N$, that defines how markings are associated with states. In the version of the ABP Promela program shown in Appendix II we use three abstract states. These states include *ready to send*, *sending* and *received*. The information for the abstract states are stored in a byte array of size 3. Whenever a transition is fired we call the inline function *set_state_covered* to update the variable holding the abstract state information (see line 94 of the ABP Promela program in Appendix II).

Using the Promela program for the ABP model in Appendix II all the transitions and states were adequately covered during the simulation. We ran the simulation for 939 execution steps before all the transitions were covered and 2246 steps before all the abstract states were covered. There was no need to use different values for the initial markings since the monitor terminated the program after complete coverage of the criteria. Figure 5 shows a screen shot of the bar chart generated by XSpin [13] when the ABP Promela program is executed in simulation mode. This simulation was performed with a random seed value of 1. Each bar in the bar chart represents a process, from left to right they are: init (line 200 in the ABP Promela program in Appendix II), sende represents sender (line 58), mchan represents mchannel (line 99), achan represents achannel (line 119), recei represents receiver (line 135), and monit represents monitor (line 167). From Figure 5 the monitor executed 11% of the total number of steps. Note that they are additional execution steps in the other processes used to update the global variables for monitoring.

In Section 6 we discuss some of the major issues on using simulation to test coverage criteria of PrT nets. In particular we look at some of the issues we faced when we use simulation to investigate test coverage of the ABP model.

5.3 Observations of the Case Study

Suppose the system accepts 8 different messages from the environment to be delivered and each message is represented as a number. Then the initial marking is as follows: $M_0(Accept) = \{128, 129, 130, 131, 132, 133, 134, 135\}$,

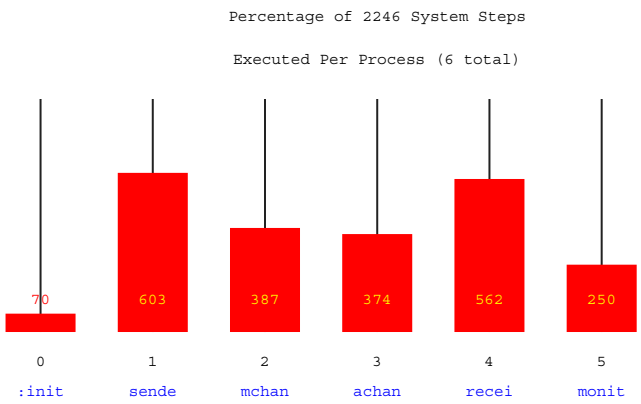


Figure 5. A screen shot showing a bar chart representing the number of system steps executed per process.

$M_0(Deliver) = \{\}$,
$M_0(DataBuf) = \{< 1, emp >\}$,
$M_0(AckIn) = \{1\}$,
$M_0(DataOut) = \{< 0, emp >\}$,
$M_0(DataIn) = \{< 0, emp >\}$,
$M_0(AckOut) = \{1\}$,
$M_0(AckBuf) = \{1\}$

where (emp represent the absence of a message, 0 and 1 are the alternating sequence numbers.

We ran the Promela program until all messages were transmitted from place *Accept* to place *Deliver* under simulation mode. During the simulation, transition events and state events are logged in global variables. The $Monitor$ periodically checks the values of the global variables to determine the coverage criteria. For this initial input, we found that both the transition and state coverage criteria were covered. However, if there is only one message in the test case such as $M_0(Accept) = 128$, we found that some transitions such as $resendACK$ are not covered by this input. Note however, that all the abstract states will eventually be covered if the system accepts one message for the environment.

Varying the random seed value in simulation mode produces different numbers of execution steps before all transitions and abstract states are covered. For example, using a seed of 5 all transitions were covered after two messages are sent requiring 359 execution steps. In Section 5.2 the number of execution steps required to cover all transitions were 939. There was also a slight change in the number of execution steps required to cover all the abstract states. We discuss other related issues to testing the ABP using our approach in Section 6.

6 Discussion

In this section we present a discussion of our approach using the Spin simulation mode to perform test coverage on PrT nets. We discuss our approach on both of the PrT nets presented in this paper, the net for the dining philosopher's problem and the ABP problem.

6.1 Translation from PrT Net to Promela

Handling Infinite State spaces: In Section 2.2 we stated that a marking defines a state in a PrT net. The state space for a PrT net can be infinite, and even it is finite, it can be very large; thus making it hard to cover all

the states in the state space. Bounding all places of a PrT net is necessary but not sufficient to limit the number of states; we also need to limit the possible values a token in each place can take. In our translation from a PrT net to Promela we perform the following. Each place is translated to a fixed-length array (bounding the place), and the types defined for the PrT net are mapped to bounded integers and structured types (bounding the values for tokens). For example, in our case study we encode strings as integers.

In testing PrT nets, we use abstract states, where each abstract state represents some configuration of interest to be observed in the execution of the net. The set of abstract states is finite, and we map a set of possibly infinite markings (concrete states) to an abstract state. If we want to observe the occurrence of an abstract state, i.e. that the net is at a given abstract state, then we need to observe one or more of the concrete states that maps to that abstract state. In online monitoring we provide a means for tracking both the concrete states and the abstract states. The abstract state space is mapped to a fixed-length array in Promela, and we use that array to record the states covered during simulation. Testing PrT nets using abstract states provides useful information. For example, in the five-dining philosopher problem, we defined three states: one for the situation in which no philosopher is eating (s_0), one philosopher is eating (s_1), and two philosophers are eating (s_2). If during the testing process, we never reach s_1 or s_2, we don't need to ask further questions such as whether every philosopher or a specific one gets the chance to eat or not.

Alternative Translation Approaches: We performed additional experiments using two variations on how PrT net transitions are translated to Promela. One in which each PrT maps to a unique process (see Sections 4 and 5) and another in which each transition is mapped to a process. Similar results in terms of coverage were obtained in both cases. However, for the first case, we had to add an extra variable for controlling the enabling of each transition, we discuss this below.

Handling Dynamic Semantics: In Spin simulation mode, the execution sequence of the translated PrT net can be very large, and we may have to stop the simulation at some point in the execution of the program. In the case that we comply with the coverage criteria (e.g. the state coverage criteria metric yields 1 or 100% coverage) then the simulation is terminated using an `assert` statement. Otherwise, after executing the simulation for a number of steps, the states and the transitions covered during the simulation can help in the refinement of the Prt net model.

One challenge regarding the "one Promela process per net" implementation used, was how to decide which transition to fire and which substitution to use for testing the enabling of a transition given the tokens at the input places. The Promela program non-deterministically chooses both a transition and the substitution to be used for checking its enabling condition. To realize this behavior each transition has the form:

```
:: atomic{ test_to_pick_transition && test_substitution
         -> fire_transition }
```

The condition `test_to_pick_transition` contains a variable whose value randomly changes. If a transition (the *atomic* construct) is chosen to be evaluated by Spin and it is enabled, its firing depends on whether `test_to_pick_transition` is true or not. This decision was particularly important for testing the transition coverage for the ABP model. The initial approach used to implement the *mchannel* process in the ABP model consisted of two transitions (*trasmitted* and *corrupted*). An outline of the implementation is shown below:

```
proctype mchannel() {
  do
    ::atomic{guard_transmitted -> fire_transmitted}
    ::atomic{guard_corrupted -> fire_corrupted}
    ::else -> skip
  od
}
```

Using this approach only the first transition (*transmitted*) would fire during the simulation (the second one would always be tested when it was not enabled). A similar implementation was defined for the *achannel* process (the one acknowledging the reception of a message) and in that case both transitions would fire. To avoid this situation, we defined a variable (*tran*) that changed its value randomly and was part of the guard for each transition (this is the variable in `test_to_pick_transition`), resulting in the modified code shown below:

Monitor Type	PrT Net to Promela	Communication Monitor - PrT Net Promela Process
Online	Multiple Processes	Global Variables Channels
	Single Process	Global Variables Channels
Offline	Multiple Processes	*printf* Promela function
	Single Process	*printf* Promela function

Table 2. Summary of the approaches used to monitor PrT nets using Promela+.

```
proctype mchannel() {
  byte tran=0;
  do
    ::atomic{guard_transmitted && tran==0 -> fire_transmitted}
    ::atomic{guard_corrupted && tran==1 -> fire_corrupted}
    ::else -> tran=(tran+1)%2
  od
}
```

6.2 Monitor Code in Promela

Effects of Monitor Code: It's important to study how the monitoring code affects the execution of the Promela program. For offline monitoring there is no effect, since the only thing we do is to pipe the events out to a file. Whereas, for online monitoring, besides logging events, the monitor process added to the Spin process set, affecting the execution sequence of the Promela code. Spin includes the monitor process in the choice it makes at every step on which process to execute. Since Spin guarantees execution fairness, the only effect the selection of the monitor process has, is to postpone the election of some other process (and the possibility of firing one of its transitions).

The monitor code introduced after each transition has no effect on the way Spin selects process and transitions since that code is enclosed within the "atomic" construct. See outline of code below:

```
::atomic {test_transition -> fire_transition; monitor_code}
```

Alternative Monitoring Approaches: We studied two approaches to evaluating test coverage criteria for PrT nets, online and offline. In terms of online evaluation, we implemented two different procedures for the communication between the PrT net processes in Promela and the monitor process. The approaches include one that usses channels and the other that uses global variables. In Table 2 we summarize the monitoring approaches from the perspectives of: (1) the translation from PrT nets to Promela, and (2) the communication between the monitor and the PrT net processes. We applied these approaches to the five-dining philosopher problem only.

For the communication using global variables, we define two arrays variables, one for the abstract states and the other to log the transitions covered. It is possible to automatically generate the transition coverage code in Promela, whereas the abstract state coverage code needs to be tailored to the specifics of the application.

7 Related Work

In this section we compare our work to other structural testing techniques that use the specification, program, or a combination (specification and program) to generate test information. There are several program based testing techniques for concurrent programs that use reachability analysis to create graphs form the source code. These techniques try to minimize the effects of the state explosion. Taylor et al. [14] describe a testing approach that extends structural testing criteria for sequential programs to concurrent programs, and propose a hierarchy of supporting structural testing techniques. These criteria are defined in terms of the features of a concurrency

graph. Koppol et al. [10] use annotated labeled transition systems (ALTSs) to select test sequences for concurrent programs. ALTSs reduce the impact of the state explosion problem by performing incremental reachability analysis. ALTSs are similar to concurrency graphs and the criteria by Taylor et al. can be applied to ALTSs. Koppol et al. define additional test criteria that focus on synchronization events. Unlike the approaches in [14] and [10], our approach uses a higher level of abstraction thereby removing some implementation details and the dependence of certain language features.

Several researchers have used model checking to analyze and/or verify Petri nets. Gannod and Gupta [4] describe a tool that supports the use of the Spin model checker to analyze and verify Petri nets constructed using the DOME tool. The tool by Gannod and Gupta focuses on integrating a modeling environment (DOME) with an analysis environment (Spin). Gannod and Gupta do not consider the analysis of PrT nets in their work. Grahlmann and Pohl [6] integrate the Spin verifier into the PEP tool (Programming Environment based on Petri nets). Several examples were presented in [6] highlighting the advantages of the integration between Spin and PEP, the major one being the speedup of using Spin based analysis versus prefix based analysis. We use an approach similar to the ones presented by Grahlmann and Pohl when converting PrT nets into Promela [6]. None of the above approaches that use Spin to analyze and/or verify Petri nets consider analyzing the net with respect to test coverage criteria for a given initial marking. Using test coverage criteria during analysis provides a measure of the adequacy of the initial marking used in the verification process. In addition our approach uses a monitor written in Promela to record and/or evaluate net events against the test criteria during the analysis process.

In our work we do not use the model checking facility in Spin, however, several researchers have investigated using model checking to support the testing of software. Ammann et al. [1] explores the role of model checkers in software testing by investigating how the powerful computation engines in model checkers are used to generate and evaluate test sets for a variety of test coverage criteria. The model Ammann et al. use is an FSM with constraints over states and execution represented as temporal logic constraints. The main contribution by Ammann et al. is using the model checker SMV to generate test sets using specification-based mutation analysis. Other test criteria described in [1] include: uncorrelated full predicate coverage, transition pair, coverage and branch coverage. Gargantini and Heitmeyer [5] use a model checker to produce counterexamples that are then used to generate test sequences. The model checker generates these counter examples by verifying negated premises (trap properties) taken from the specification. They show how a model checker can be used to automatically generate test cases to satisfy certain structural coverage criteria. Their approach claims to generate test cases from any development artifact that can be represented as an FSM. The structural coverage criteria by Rayadurgam and Heimdahl [12] are defined in terms of the transition relation of the FSM, each transition is thought of as a triple (pre-state, post-state, guard). A guard is a condition that must be satisfied for a change from a pre-state to a post-state. We do not focus on generating test cases but on evaluating the structural coverage criteria for a PrT net given a test case (initial marking). We also evaluate if the test criteria are satisfied using the simulation facility in the Spin model checker.

Zhu and He [16] formally define the test coverage criteria of PrT nets. Our work extends their work by providing a practical implementation for measuring the test coverage criteria associated with PrT nets. We use the model checker Spin to simulate the execution of the PrT net, log the net events generated and evaluate the coverage criteria.

8 Summary and Future Work

In this paper we presented a unique approach of applying the simulation capability of the model checker Spin for evaluating the test coverage adequacy of PrT nets, which provides an effective and practical technique to analyze high level Petri net specifications for finite and infinite state systems. Furthermore, our results on high level Petri nets can be easily adapted to other formal specification methods in which the corresponding coverage criteria can be defined. We also described several optional approaches to translating PrT net models into Promela code and performing the evaluation of test coverage criteria.

Our future work includes (1) extending our approach to handle other test coverage criteria used in flow-oriented testing ans specification-oriented testing (2) how to associate specific system properties and various testing coverage criteria, (3) how to use coverage information to select test cases, and (4) how to use the design level coverage information to derive specification-based test cases at the implementation level.

Appendix I: A Promela Program for Dinning Philosophers

```
#define N 5

typedef tokenE{
  byte    ph;
  byte    ch1;
  byte    ch2
}; /*type used in Eating predicate*/

byte    thinking[N], chopstick[N]; tokenE   eating[N];

byte trans[2];
/* Monitor transitions - trans[0]:pickup, trans[1]:putdown */
byte states[3];
/* Monitor states - states[0]:all philosophers thinking,
                    states[1]:1 philosopher eating, states[2]:
                    2 philosophers eating*/

inline transUp(cnt, right, left) {
  eating[cnt].ph = thinking[right];
  eating[cnt].ch1 = chopstick[right];
  eating[cnt].ch2 = chopstick[left];
  thinking[right] = 0;
  chopstick[right] = 0;
  chopstick[left] = 0;
}

inline transDown(idx) {
  thinking[idx] = idx+1;
  chopstick[idx] = idx+1;
  chopstick[(idx+1)%N] = ((idx+1) % N) + 1;
  eating[idx].ph = 0;
  eating[idx].ch1 = 0;
  eating[idx].ch2 = 0;
}

/*Transitions covered*/
inline set_trans_covered(idx){
  trans[idx]=1;
}
/*This keeps track of which state is being covered.*/
inline set_states_covered(){
  byte cnt;
  cnt = 0;
  byte idx;
  idx = 0;/*No philosopher eating*/
  do
    :: (cnt<N) ->
      if
        ::( eating[cnt].ph != 0 ) -> idx++
        ::else -> skip
      fi;
      cnt++
    ::else -> break
  od;
  states[idx] = 1
  /*we assume that 0<=idx<3, otherwise it's an specification
     error*/
}

proctype DP() {
  byte count = 0;
  byte tran = 0;
  do
    /*Pickup*/
    /*guard_pickup -> exec_pickup*/
    ::atomic { (tran ==0 && thinking[count] > 0
          && chopstick[count] > 0
                && chopstick[(count+1)%N] > 0)
              -> transUp(count, count, (count+1)%N);
                 set_trans_covered(0);
                 set_states_covered() }
    /*Putdown*/
    /*guard_putdown -> exec_putdown*/
    ::atomic { (tran ==1 && eating[count].ph > 0)
              -> transDown(count);
                 set_trans_covered(1);
                 set_states_covered() }
                 /*states[0] is set to 1 when none is eating*/
```

```
79      ::else -> atomic{ /*Select a transition and a substitution*/
80                        do
81                          ::count=(count+1)%N
82                          ::count>0 -> count=count-1
83                          ::tran=(tran+1)%2
84                          ::tran >0 -> tran=tran-1
85                          ::break
86                        od };
87    od
88  }
89
90  proctype monitor(byte monitor_type){
91   /*monitor_type==0 : transitions and places
92      monitor_type==1 : transitions only
93      monitor_type==2 : places only*/
94    if
95      :: monitor_type==0 ->
96        do
97          :: atomic{(states[0]==0 || states[1]==0 || states[2]==0)
98             -> skip }
99          :: atomic{(trans[0]==0 || trans[1]==0) -> skip }
100         :: else -> break
101       od;
102     :: monitor_type==1 ->
103       do
104         :: atomic{(trans[0]==0 || trans[1]==0) -> skip }
105         :: else -> break
106       od;
107     :: monitor_type==2 ->
108       do
109         :: atomic{(states[0]==0 || states[1]==0 || states[2]==0)
110            -> skip }
111         :: else -> break
112       od;
113   fi;
114   assert(false) /*Stop the simulation*/
115 }
116
117 init {
118   byte count = 0;
119   /* Initial Marking */
120   atomic {
121     do
122       ::(count < N) -> thinking[count] = count + 1;
123                        chopstick[count] = count + 1;
124                        count++
125       ::else -> break
126     od
127   }
128   count = 0;
129   atomic{
130     do
131       ::(count < N) -> eating[count].ph = 0;
132                        eating[count].ch1 = 0;
133                        eating[count].ch2 = 0;
134                        count++
135       ::else -> break
136     od
137   }
138   /* End initial marking */
139
140   /*Monitoring initilization*/
141   trans[0]=0;
142   trans[1]=0;
143   states[0]=0; /*This should be 1, but eventually the net
144                  will have to reach the initial marking*/
145   states[1]=0;
146   states[2]=0;
147
148   /* Execute net */
149   atomic{
150     run DP();
151     run monitor(0)/*0 means keep track of both criteria: state
152                    and transition.*/
153   }
154 }
155
156   }
```

Appendix II: A Promela Program for ABP

```
1
2  #define  N  8
3
4  mtype = {EmptyP, corrupted, one, zero}
5          /*EmptyP->4,corrupted->3,one->2,zero->1*/
6
7  typedef frameType {
8    mtype          ack;
9    byte     msg
10 };
11
12 typedef dataType {
13   byte     token;
14   byte     msg[N]
15 };
16
17 dataType     accept, deliver;
18 frameType    dataOut, dataIn, dataBuf;
19 mtype        ackIn, ackOut, ackBuf;
20
21 /*Monitoring code*/
22 #define N_TRANS 8
23 #define N_STATES 3
24 byte trans[N_TRANS];
25 /* transitions */
26 /* trans[0]: sendData, trans[1]:resendData
27    trans[2]: transmitted, trans[3]: corrupted
28    trans[4]: acorrupted, trans[5]: atransmitted
29    trans[6]: resendAck, trans[7]: deliverData */
30
31 byte states[N_STATES];
32 /* states */
33 /* states[0]: ready to send,
34    states[1]: sending,
35    states[2]: received*/
36
37 /*Transitions covered*/
38 inline set_trans_covered(idx){
39   trans[idx]=1
40 }
41 /*This keeps track of which state is being covered.*/
42 inline set_states_covered(){
43   /*Check for the three states*/
44   if
45     ::accept.token==N->states[0]=1
46     ::else->skip
47   fi;
48   if
49     ::(accept.token!=N || deliver.token!=N )->states[1]=1
50     ::else->skip
51   fi;
52   if
53     ::deliver.token==N->states[2]=1
54     ::else->skip
55   fi
56 }
57
58 proctype sender() {
59   byte tran=0;
60   byte  i = 0;
61   do
62     /*transition: sendData*/
63     ::atomic{(accept.token > 0) && (ackIn == one ||
64               ackIn == zero) &&
65                 /*(dataBuf.ack == zero || dataBuf.ack == one) &&*/
66                 (dataBuf.ack == ackIn) && tran==0
67              -> dataOut.ack = (1 - (ackIn-1)+1); /*3-ackIn*/
68                 /* We have ackIn-1 because ackIn at this point
69                    can be  2 (meaning one) or 1 (meaning zero) */
70                 i = 0;
71                 do
72                   ::(i < N) ->
73                     if
74                       :: accept.msg[i] > 0 ->
75                            dataOut.msg = accept.msg[i];
76                            dataBuf.msg = accept.msg[i];
77                            dataBuf.ack = dataOut.ack;
78                            accept.token--;
```

```
79                              accept.msg[i] = 0;
80                              ackIn = EmptyP;
81                              break
82                          :: else -> i++
83                          fi
84                        ::(i == N) -> break
85                    od;
86                    set_trans_covered(0); set_states_covered() }
87      /*transition: resendData*/
88      ::atomic{ (ackIn == corrupted) &&
89                (dataBuf.ack == one || dataBuf.ack == zero) &&
90                (dataBuf.msg > 0) && tran==1
91              -> dataOut.ack = dataBuf.ack;
92                 dataOut.msg = dataBuf.msg;
93                 ackIn = EmptyP;
94                 set_trans_covered(1); set_states_covered() }
95      ::else -> tran=(tran+1)%2
96    od
97  }
98
99  proctype mchannel() {
100   byte tran=0;
101   do
102     /*transition: transmitted*/
103     ::atomic{ ( dataOut.ack == zero || dataOut.ack == one )&&
104               (dataOut.msg > 0) && tran==0
105             -> dataIn.ack = dataOut.ack;
106                dataIn.msg = dataOut.msg;
107                dataOut.ack = EmptyP; dataOut.msg = 0;
108                set_trans_covered(2); set_states_covered() }
109     /*transition: corrupted*/
110     ::atomic{ ( dataOut.ack == zero || dataOut.ack == one ) &&
111               (dataOut.msg > 0) && tran==1
112             -> ackIn = corrupted;
113                dataOut.ack = EmptyP; dataOut.msg = 0;
114                set_trans_covered(3); set_states_covered() }
115     ::else -> tran=(tran+1)%2
116   od
117 }
118
119 proctype achannel() {
120   byte tran=0;
121   do
122     /*transition: atransmitted*/
123     ::atomic{ (ackOut == zero)||(ackOut == one) && tran==0
124             -> ackIn = ackOut; ackOut = EmptyP;
125                set_trans_covered(5); set_states_covered() }
126     /*transition: acorrupted*/
127     ::atomic{ (ackOut == zero)||(ackOut == one) && tran==1
128             -> dataIn.ack = corrupted;
129                dataIn.msg = 0; ackOut = EmptyP;
130                set_trans_covered(4); set_states_covered() }
131     ::else -> tran=(tran+1)%2
132   od
133 }
134
135 proctype receiver() {
136   byte tran=0;
137   byte k = 0;
138   do
139     /*transition: resendAck*/
140     ::atomic{ (dataIn.ack == corrupted) &&
141               (ackBuf == zero || ackBuf == one) && tran==0
142             -> ackOut = ackBuf; dataIn.ack = EmptyP;
143                set_trans_covered(6); set_states_covered() }
144     /*transition: deliverData*/
145     ::atomic{ (dataIn.ack == zero || dataIn.ack == one) &&
146               (dataIn.ack == 3-ackBuf/*1 - ackBuf*/) &&
147               (dataIn.msg > 0)  && tran==1
148             -> ackBuf = dataIn.ack; ackOut = dataIn.ack;
149                k = 0;
150                do
151                  ::(k < N ) ->
152                    if
153                      ::(deliver.msg[k] == 0) ->
154                        deliver.token++;
155                        deliver.msg[k]= dataIn.msg;
156                        break
157                      ::else -> k++
```

```
158                     fi
159                   ::else -> break
160                 od;
161                 set_trans_covered(7); set_states_covered() }
162       ::else -> tran=(tran+1)%2
163     od
164   }
165
166   /*The monitor process*/
167   proctype monitor(byte monitor_type){
168   /*monitor_type==0 : transitions and places
169     monitor_type==1 : transitions only
170     monitor_type==2 : places only*/
171     if
172       :: monitor_type==0 ->
173         do
174           :: atomic{(states[0]==0 || states[1]==0 ||
175              states[2]==0) ->  skip }
176           :: atomic{(trans[0]==0 || trans[1]==0 ||
177                      trans[2]==0 || trans[3]==0 ||
178                      trans[4]==0 || trans[5]==0 ||
179                      trans[6]==0 || trans[7]==0 ) -> skip }
180           :: else -> break
181         od;
182        :: monitor_type==1 ->
183         do
184           :: atomic{(trans[0]==0 || trans[1]==0 ||
185                      trans[2]==0 || trans[3]==0 ||
186                      trans[4]==0 || trans[5]==0 ||
187                      trans[6]==0 || trans[7]==0 ) -> skip }
188           :: else -> break
189         od;
190        :: monitor_type==2 ->
191         do
192           :: atomic{(states[0]==0 || states[1]==0 || states[2]==0)
193                     ->  skip }
194           :: else -> break
195         od;
196     fi;
197     assert(false) /*Stop the simulation*/
198   }
199
200   init {
201     byte count = 0;
202     atomic {
203       ackIn = one; dataBuf.ack = one; dataBuf.msg = one;
204       ackBuf = one; dataIn.ack = EmptyP; dataIn.msg = one;
205       ackOut = EmptyP;
206       accept.token = N; deliver.token = 0;
207       do
208         ::(count < N) ->
209            accept.msg[count] = 128 + count;
210            deliver.msg[count] = 0; count++
211         ::else -> break
212       od;
213     }
214
215     /*Monitoring initilization*/
216     trans[0]=0; /*sendData*/
217     trans[1]=0; /*resendData*/
218     trans[2]=0; /*transmitted*/
219     trans[3]=0; /*corrupted*/
220     trans[4]=0; /*acorrupted*/
221     trans[5]=0; /*atransmitted*/
222     trans[6]=0; /*resendAck*/
223     trans[7]=0; /*deliverData*/
224     states[0]=1; /*initial state is already covered*/
225     states[1]=0;
226     states[2]=0;
227
228     atomic{
229       run sender();
230       run mchannel();
231       run achannel();
232       run receiver();
233       run monitor(0)
234     }
235   }
```

Acknowledgments: This research was supported in part by the National Science Foundation of the USA under grant HRD-0317692, and by the National Aeronautics and Space Administration of the USA under grant NAG2-1440.

References

[1] P. Ammann, P. E. Black, and W. Ding. Model checkers in software testing. Technical Report NIST-IR 6777, National Institute of Standards and Technology, 2002.

[2] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.

[3] J. Ding. *A Methodology for Formally Modeling and Analyzing Software Architecture of Mobile Agent Systems*. PhD thesis, Florida International University, 2004.

[4] G. C. Gannod and S. Gupta. An automated tool for analyzing petri nets using spin. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 404–407. IEEE, November 2001.

[5] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 146–162. ACM, Oct. 1999.

[6] B. Grahlmann and C. Pohl. Profiting from spin in PEP. In *Proceedings of the 4th International SPIN Workshop (SPIN '98)*, Nov. 1998.

[7] X. He and T. Murata. *High-Level Petri Nets - Extensions, Analysis, and Applications*. Electrical Engineering Handbook (ed. Wai-Kai Chen). Elsevier Academic Press, 2005.

[8] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng. Formally analyzing software architectural specifications using sam. *Journal of Systems and Software*, 71(1-2):11–29, 2004.

[9] G. J. Holzmann. *The Spin Model Checker: Primer and reference manual*. Addison-Wesley, Boston, MA., 2003.

[10] P. V. Koppol, R. H. Carver, and K. C. Tai. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering*, 28(6):607–623, 2002.

[11] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys (CSUR)*, 21(4):593 – 622, 1989.

[12] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.

[13] Spin. On-The-Fly, LTL Model Checking with SPIN , Feb. 2006. `http://spinroot.com/`.

[14] R. Taylor, D. Levine, and C. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206 – 215, 1992.

[15] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit testing coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[16] H. Zhu and X. He. A methodology of testing high-level petri nets. *Information and Software Technology*, 44:473–489, 2002.