2005

Comparison of two approaches for test case generations from EFSMs.

Yongdong Tan
University of Windsor

Comparison of two approaches for test case generations from EFSMs

by

Yongdong Tan

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2005

Your file Votre référence
ISBN: 0-494-09778-7
Our file Notre référence
ISBN: 0-494-09778-7

Abstract

Testing is one of the vital steps in software development process. To convey testing, test cases need to be generated to check whether an implementation conforms to the design specification. Design specifications are usually expressed as Extended Finite State Machines (EFSMs) and test cases are actually a path from the initial state to a specific state on that EFSM. One of the most difficult issues of test case generation for EFSMs comes from the fact that infeasible paths exist on EFSMs. Two approaches have been developed in earlier 90s' to generate feasible paths from EFSMs: one is to develop algorithm to search EFSMs directly to generate feasible paths, and the other is to expand EFSMs into Finite State Machines (FSMs), followed by applying FSM techniques to generate feasible paths. Model checking method was proposed recently as a new approach for test case generation. It has some advantages over previous methods such as efficiency on number of states explored. However, by nature, it also has some disadvantages such as time inefficiency. Here we present a comparison between the model checking method and the previous expansion method from pragmatic aspect by running experiments. To carry on this comparison, we implemented a classical expansion algorithm, defined the translation from EFSMs to Promela models, and used SPIN model checker in the model checking approach. We have run sufficient number of test case generation experiments, compared the two approaches on their time consumptions, numbers of states explored, performance changes when EFSMs' sizes increase etc. By this comparison, we can see the tradeoff between time consumptions and the number of states explored in the two approaches and observe their performance changes while EFSMs change. Finally, we show the existence of the trade-off between state efficiency and time efficiency of the two approaches, the impact of domain size of variable value, the native drawbacks of the expansion algorithm and the performance improvement by tuning Premela models.

Keywords: Test Case Generation, Extended Finite State Machines, Model Checking, Feasible Path

Acknowledgement

First of all, I would like to thank my supervisor, Dr. Jessica Chen, for her invaluable guidance and advices, for her enthusiastic encouragement and her great patience to me. Without her help, the work presented here would not have been possible.

Next, I would like to thank my committee members, Dr Ezeife, Dr. Hu and Dr. Jaekel, for spending their precious time to read this thesis and putting on their comments, suggestions on the thesis work.

My special thanks go to Mrs. Hanmei Cui, Ms. Lihua Duan, Mr. Xiaoshan Zhao and other members of our research group, for their help.

Finally, I also would like to thank my parents, my wife for their support, understanding and patience, and my daughter, Alice, for giving me endless happiness.

Table of Content

List of Figures

List of Tables

1 Introduction and Motivation

Testing is one of the vital steps in software development process. Given a software application, it is necessary to measure its quality – Have all the requirements been satisfied under all possible circumstances? If not, how well has the application satisfied the requirements? Is it acceptable?

If the application is determined to be not acceptable, it will need revision – the first step is to identify the errors: where are the errors and how did they occur?

Such questions can only be answered through testing. For testing, basically, we run a set of test cases: Each test case is a set of tuples of inputs, execution preconditions and expected outcomes developed for a particular objective, such as to verify compliance with a specific requirement. The application is executed with the specified input and preconditions and its outcomes are observed.

Ideally, combining together, the test cases should be able to cover all requirements, all possible behavior of the application, or it might be desirable to use as few test cases as possible to cover some critical parts.

As we can see, it is highly unlikely that such criteria will be met if the test cases were chosen randomly. Picking test cases manually is possible, but it will not be very efficient and can likely be affected by human errors. Thus, it is highly desirable that the test cases be generated automatically.

To generate test cases automatically, the application must be specified formally: The assumptions about the world in which the application will operate, the requirements that the application is to achieve and the design to meet those requirements must all be expressed using formal notations. A formal specification can be understood and analyzed by computers and the test cases can be generated based on it.

One of the most commonly used set of notations for formal specification is the Extended Finite State Machine (EFSM).

An EFSM is formally represented as a 6-tuple <S, s_0, I, O, T, V> Where

1. S is a non empty set of states,
2. s_0 is the initial state,
3. I is a finite set of input symbols,

4. O is a finite set of output symbols,

5. T is a finite set of transitions,

6. V is the finite set of variables.

Each element of T is a 5-tuple t=(source_state, dest_state, input, predicate, action).

Here "source_state" and "dest_state" are the states in S representing the starting state and the ending state of t, respectively. The "input" is either an input interaction from I or empty: it will trigger the specific transition. For example in protocol specification, an input often represents an incoming message from the message channel. Not all EFSMs have input and output messages. The "predicate", also called guard, is a Boolean expression of the variables in V. Only when the "predicate" is satisfied, the transition is enabled.

Figure 1.1 presents a simple example of EFSM. This EFSM can also be expressed by a transition table shown in Table 1.1. In Figure 1.1, the nodes represent states, and the arcs are transitions between states. Transitions may have guards (preconditions) expressed by an "if" statement. Every transition leads the system to evolve from one state to another.

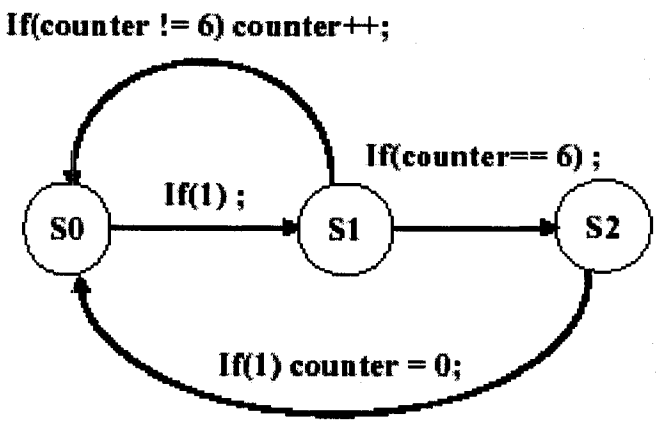

Figure 1.1 An Extended Finite State Machine

transitions	Starting state	Ending state	Guard	Action
t0	s0	s1	True	Null
t1	s1	s0	counter!=6	counter++
t2	s1	s2	counter==6	Null
t3	s2	s0	True	counter=0

Table 1.1 Transitions table for EFSM in Figure 1.1

Intuitively, for an EFSM specification, the coverage criterion for test case generation is all transition coverage, requiring every transition be covered at least once, which means the test cases should cover all the edges in the EFSM. But it is not easy to generate such test cases automatically. The difficulty stems from the fact that, in general, an EFSM model contains infeasible paths.

Figure 1.2 shows a part of an EFSM. We can see that the system can not evolve along the dashed line although there are transitions connecting state nodes s_j and s_k. The guard "x=1" blocks the way. While in Figure 1.3, the system can evolve along the solid line. It is a feasible path.

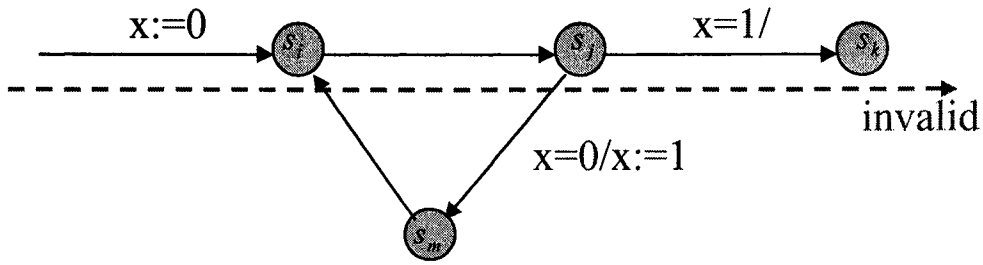

Figure 1.2 An infeasible path

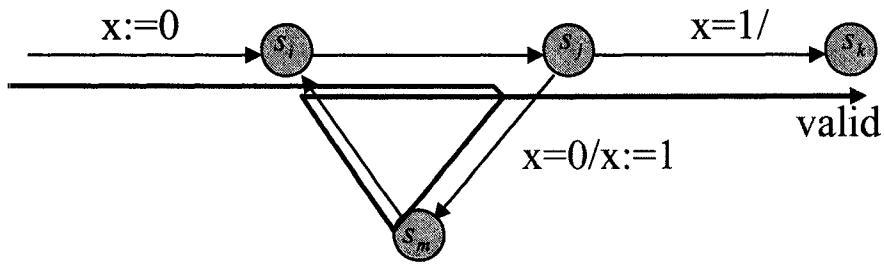

Figure 1.3 A feasible path

By nature, the infeasible path problem is due to the existence of the so-called context variables. A variable of an EFSM is called a context variable if there exists a path from the initial state such that the variable is used in either an assignment or output statement but without a prior value assigned to it [LCM94].

The EFSM example in Figure 1.4 is from [CZ93], which is a real protocol from industry. It has 36 transitions, 20 states and five self-loops. In this Figure, the variable "number" "counter" are context variables. This is because they did not have values in when the system is in initial state, but they are assigned values when the system evolves into some reachable states.

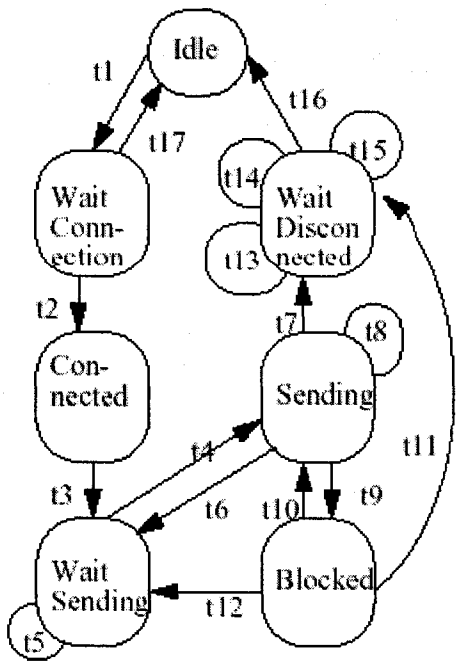

Figure 1.4 Example of an EFSM specified protocol.

The corresponding transitions table is shown below in Table 1.2.

	Input	Guard	Output	Action
t1	U.sendrequest		L.cr	
t2	L.cc		U.sendconfirm	

t3	U.data_req(sdu,n,b)			number:=0;
				counter:=0;
				no_of_segment:=n;
				blockbound:=b;

t4	L.tokengive		L.dt(sdu[number])	start timer number:=number +1;
t5	L.resume			
t6		expire_timer	L.tokenrelease	blockbound:=b;
t7	l.ack()	number==no_of_ segment	U.monitor_complete(co unter) token_release L.disrequest	
t8	L.ack()	number<no_of_s egment not expire_timer	L.dt(sdu[number])	number:=number +1
t9	L.block	not expire_timer		counter:=counter

				+1
t10	L.resume	not expire_timer and counter<=blockbound		
t11		counter>blockbound	!L.token_realease !U.monitor_incomplete(number) !U.dis_request	
t12		expire_timer counter<=blockbound	L.token_release	
t13	L.resume			
t14	L.block			
t15	L.ack			
t16	L.dis_request		U.disindication	
t17	L.dis_request		U.disindication	

Table 1.2 The transition table of EFSM in Figure 1.4

Using the method in [CZ93], the author can generate test cases by using static loop analysis and symbolic evaluation techniques to determine how many times the self loop should be repeated so that test cases become executable. But when applying this method onto the EFSM in Figure 1.4, the result, in Table 1.3, shows that among the generated test sequences, only some are feasible (executable). More than half of the test sequences have to be discarded.

Path	Discarded	Reason why path is discarded
1,2,3,4,9,10,6,4,7	no	-
1,2,3,4,9,10,6,4,8,7	yes	predicate in t7 become (3=2)
1,2,3,4,9,10,6,5,4,7	no	-
1,2,3,4,9,10,6,5,4,8,7	yes	predicate in t7 become (3=2)
1,2,3,4,9,10,7	yes	will be equivalent to the first path after solving the executability
1,2,3,4,9,10,8,6,4,7	yes	predicate in t7 become (3=2)
1,2,3,4,9,10,8,6,5,4,7	yes	predicate in t7 become (3=2)
1,2,3,4,9,10,8,7	no	-
1,2,3,4,9,12,4,7	no	-
1,2,3,4,9,12,4,8,7	yes	predicate in t7 become (3=2)

1,2,3,4,9,12,5,4,7	no	-
1,2,3,4,9,12,5,4,8,7	Yes	predicate in t7 become (3=2)

Table 1.3 Generated test cases including feasible and infeasible paths

An infeasible path can result from many reasons, but the key reason is that the values of the context variables are nondeterministic. For example, if a state has two out-going transitions, and the preconditions of both transitions are true, then which transition will be chosen to go will be totally undetermined. This will obviously lead to the resulting values undetermined. How can we make sure the generated sequences are feasible? Some researchers [SBC97, MP92, CZ93, LHHT97, HB94, HLJ95] tried to develop algorithms to search the graph or the EFSM models directly, with some heuristic techniques, to find feasible path. More details are given in section 2.1. Other researchers [PTB85, PT87, KS90, BFH90, LY92] tried to expand EFSM into Finite State Machines (FSM). FSM is similar to EFSM, but it has no variables and no guards on the transitions, so all paths are feasible. Through this expansion, we can apply various FSM tools and techniques to generate test cases. D. Lee and M. Yannakakis's expansion algorithm is such a classical algorithm [LY92]. Details are given in section 2.2.

In recent years, researchers proposed another approach for generating test cases covering all edges in EFSM specification – using model checkers [ABM98], [CSE96], [EFM97], [GH99], [HLSU02], [RH01].

The idea of this approach is to take advantage of the counter-example generation capability of model-checkers for constructing test cases. The basic capability of model checkers is to check whether a model satisfies a specific property (which can be expressed by e.g. Linear Temporal Logic (LTL). Details of LTL are given in section 2.3.1). To generate test cases using model checkers, we can claim that a state (a node in EFSM graph) or a transition (an arc in EFSM graph) is unreachable. The claim can be expressed as a LTL property. Then a model checker is used to check against it. If the state (or transition) is reachable, then the model checker should find the property unsatisfiable, and give a counter-example by error tracing to prove that the state (or transition) is reachable. The counter-example is actually a feasible path to the state (or transition). More details about model checker are given in section 2.3 and 2.4.

Having the three test case generation approaches, we want to know which is better. As the

first method often require some kinds of heuristics, while the other two can be applied to general test case generations, we have more special interest on which is better between the expansion method and model checking method.

As a new proposed method for generating feasible test sequence, the model checking method has many advantages: first, of course, it guarantees that the test sequence is feasible; second, finding violations of the properties is relatively easy because it only needs to find a counter example against the property, but does not need to explore all states; third, it can take the advantage of many efficiency improvement techniques that have been integrated into the model checkers.

However, there is a conceivable issue: during each run, the model checker can only find one trace – which is one test case, guaranteed to cover only one edge in the EFSM. So if the EFSM has n edges, in the worst case we need to run the model check n times to ensure complete coverage. It is inefficient when compared with expansion methods which are intended to derive all feasible paths in one run.

But as a trade-off, the state space of model checking method is reduced. This is because a model checker does not need to keep and explore all states, but can check the model on-the-fly, which means exploring state only as needed. Further, model checker can leverage some performance improvement method, like partial order reduction, binary decision diagrams (BDD), efficient memory management etc, which can help us to gain a lot of performance improvement.

Thus it is hard to say which approach is definitely better. It is conceivable that there is a trade off between the two methods: model checking is more memory efficient but worse on time efficiency: it gains less number of states to be explored, at cost of more time consumed to generate the corresponding test suit.

In this thesis, we compare the two methods from empirical study. The major concerns of test generation are time consumption and space efficiency. We run a set of EFSM examples with a linear increase in sizes, observe a) the trend of their time consumption while their sizes increase, b) the state number explored, and its relation to the time consumption. Since we want to do comparison, every test generation will be run under both approaches. The time consumption and state number explored are the two objects we will compare, and they will be recorded.

Through the comparison, we answer the question which approach has what advantages over the other. We figure out how worse or better one approach is over the other, and on what kind of EFSMs, one approach has overall advantages over the other. We demonstrate the above trade off quantitatively through empirical study. We show how the consumed time varies as the size of EFSMs increase on both methods. We also show how the number of state explored varies as the size of EFSMs increase on both methods. By analyzing the result, we make further observation and investigate on, for example, whether there exist a size range within which one method is better than the other, etc.

Another way to compare the two approaches is via theoretical analysis. But the theoretical analysis is only applicable on expansion method, not on model checking method. This is because almost all model checkers have integrated many techniques for efficiency such as partial order deduction which makes the resulting time consumption unpredictable.

To the best of our knowledge, a direct experimental comparison of the performance of these two basic approaches has never been made. The contribution of this work is to perform such a comparison. For the expanding EFSM approach, we use D. Lee and M. Yannakakis's algorithm [LY92]. For the model checking method, we use SPIN model checker.

2 The three EFSM test generation approaches

In this chapter, we briefly review the directly searching EFSM methods, and give a detailed introduction of the other two approaches: expansion approach and model checking approach.

2.1 Search EFSM directly

In this section, we briefly review those works on generating test case via searching EFSM directly.

Sarikaya et al. used the functional program testing approach to generate test sequences from the EFSMs without considering feasibility of tests in advance [SBC97].

Miller and Paul [MP92] introduced a method to generate tests from EFSM models under the assumption that the variables used in an implementation under test (IUT) are accessible by a tester (i.e., the IUT is a white box). Such an assumption may not be applicable to many implementations.

Chanson and Zhu [CZ93] studied a test generation method using the constraint satisfaction problem technique from the artificial intelligence field. The feasibility of the tests is checked only after they are constructed. Furthermore, some of the assumptions for the EFSM model (e.g., the presence of the influencing self-loops) may not hold for general EFSM models.

Li et al. [LHHT97] introduced a method for EFSM state verification. He defined and used the Extended-UIO (E-UIO) sequences, each of which, if exists, contains predicates with feasible conditions for each of the outgoing transitions. Therefore, it may be argued that the generation of E-UIO sequences is equivalent to generating feasible test sequences. However, in general, a state may not have an E-UIO sequence for each of its outgoing transitions, which limits the applicability of this method.

For a restricted class of LOTOS expressions, called P-LOTOS, Higashino and Bochmann proposed a test case derivation method [HB94]. A tree, called the extended labeled transition system (ELTS), which can be an infinite tree for the general case, is defined to represent the possible event sequences of a P-LOTOS expression. After all infeasible paths are deleted from the ELTS by using linear programming, the resulting tree is used to derive test cases. The applicability of this method is restricted to tree-like structures. For a general EFSM model, the equivalent tree structure may be exponentially large.

Chung-Ming [HLJ95] overcomes this problem by executing the EFSM to find all possible executable paths. The problem with this method is that the test cases generated do not cover control flow. Also, the method can not deal with large EFSMs.

All these methods mentioned above made sound contributions toward test generations from the EFSMs. Inclusion of infeasible paths in the test sequences may be inevitable since the underlying models are EFSMs. Therefore, without a proper analysis of the interdependencies among the variables used in the actions and conditions of the EFSMs, considerable effort may be wasted on test generation since the infeasible portions will have to be discarded later.

2.2 D. Lee and M. Yannakakis's Expansion Algorithm

In this section, we introduce expansion approach and D. Lee and M. Yannakakis's expansion algorithm [LY92], and give a detailed explanation on how it works and its efficiency.

The expansion approach is to transform an EFSM into an equivalent FSM which can present the same behaviors. Many researchers have worked on this kind of equivalent transformation. The first significant result related to the algorithmic solution of the equivalence problem is in [Hop71], where Hopcroft presents an algorithm for the minimization of the number of states in a given finite state automaton. The problem is equivalent to that of determining the coarsest stable partition of a set with respect to a finite set of functions. A variant of this problem is studied in [PTB85], where it is shown how to solve it in linear time. Finally, in [PT87] Paige and Tarjan solved the problem for the general case (which is the same as computing equivalence) in which the stability requirement is relative to a relation E (on a set N) with an algorithm whose complexity is $O(|E| \log|N|)$. In [KS90] Kannellakis and Smolka noticed that the algorithm by Paige and Tarjan [PT87] can be used to determine the maximum bisimulation over a graph G = <N, E>. In [BFH90] Bouajjani, Fernandez, and Halbwachs proposed an algorithm for the relational coarsest partition problem tailored for the context of the so-called on-the-fly Model Checking. The algorithm stabilizes only the reachable blocks with respect to all blocks at each iteration. In [LY92] Lee and Yannakakis improved this method by using only reachable blocks to stabilize the reachable blocks.

We choose D. Lee and M. Yannakakis's expansion algorithm [LY92] in our experiment

rather than others. This is because it has improved the previous methods, it is more efficient and it is regarded as a classical algorithm that was widely referenced.

In this algorithm, an EFSM is expressed as a tuple $\Theta = (Q, \pi, I, T)$ consisting of

(1) A set of *configurations* Q. A configuration is a state the system could be in;

(2) A partition π of Q. It defines how configurations are divided into blocks. It is a description of what values the variables could take when the system is in a specific state;

(3) A finite set I of actions (or inputs); and (4) a set T of transition relations on Q corresponding to the actions, i.e., for each action $a \in I$, there is a relation $R_a \subseteq Q \cdot Q$. The transition system is deterministic if the transition relation for every action is a function, which means each state has no more than one out-going transition feasible at any time, otherwise it is nondeterministic.

Algorithm Input: An EFSM expressed as (Q, π, I, T) with an initially marked block < $B0$, $p0$ >.

Algorithm Output: The minimal reachable graph (R, ρ, I, T).

Figure 2.1 is an EFSM representing Active Monitor Protocol.

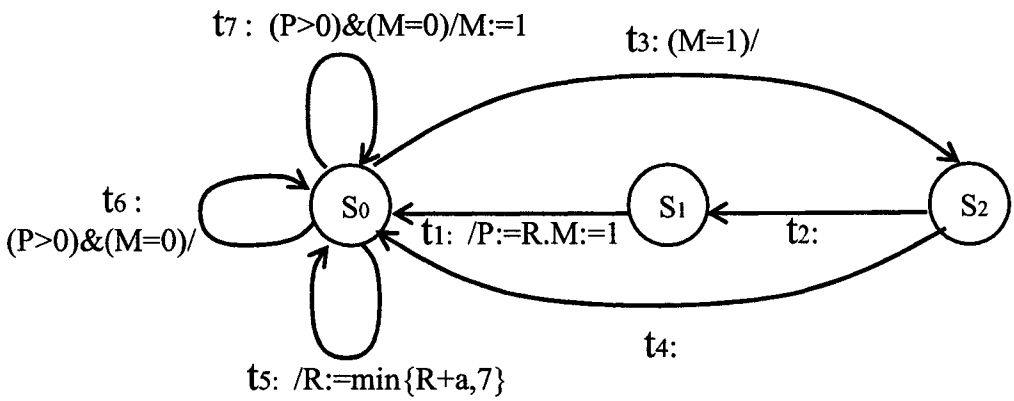

Figure 2.1 An EFSM representing Active Monitor Protocol

In the example in Figure 2.1, the input Q, π, I, T are as follows

Q: { (S,P,M,R)|S∈ {S0,S1,S2 }; P∈ {0-7}; M∈ {0,1}; R∈ {0-7} }, Totally there are 3x8x2x8=384 configurations (states).

π: contains 3 blocks: S0: { (S,P,M,R)|S∈ {S0}; P∈ {0-7}; M∈ {0,1}; R∈ {0-7} },

S1: { (S,P,M,R)|S∈ {S1}; P∈ {0-7}; M∈ {0,1}; R∈ {0-7} },

S2: { (S,P,M,R)|S∈ {S2}; P∈ {0-7}; M∈ {0,1}; R∈ {0-7} },

I and T describe the transitions between states:

transition	from	To	guard	Action
t1	S1	S0		P:=R M:=0
t2	S2	S1		
t3	S0	S2	M==1	
t4	S0	S2		
t5	S0	S0		R:=min{R+a, 7}
t6	S0	S0	P==0 & M==0	
t7	S0	S0	P>0 & M==0	M:=1

Table 2.1 Transitions table for EFSM in Figure 2.1

B0: initial state the system will be in is S0

P0: the initial values of all variables: P=0, M=0, R=0

The output of the algorithm is a minimal reachable graph and a semi-stable transition (R, ρ, I, T).

R is the reduced configuration set, and ρ is a new partition generated by splitting the original partition.

We say that an arc, B → C with label a, of the graph is stable if every configuration of B has an a-arc to some configuration of C; otherwise, arc a is unstable, which means some configuration of B can not transfer to any configuration of C via arc a. The transition system is stable if all arcs of its quotient graph are stable. An important property is that, every unstable transition system has a unique coarsest stable refinement, and that refinement is precisely the reduced transition system.

We can obtain the reduced transition system by splitting unstable arcs straightforwardly until there are no unstable arcs.

The two obvious ways for constructing the reachable minimal graph are: (1) forward search to compute all the configurations that are reachable from p0, and then minimize the derived FSM; (2) first minimize the given EFSM, and then compute the part that is reachable from the block of the initial configuration.

Both of these methods can be arbitrarily bad. In general, the reachable minimal graph can be arbitrarily smaller than both the reduced system and the number of reachable configurations, which are the minimal amount of work to be done using the two obvious methods, respectively. Furthermore, the reachable minimal graph can be finite while the other two can be infinite. Thus an intermediate method that explores the graph and splits

blocks simultaneously is necessary. This method should combine the forward inference of reachability information with the backward inference of inequivalence information. The key point is to split the unstable arc at an appropriate time. D. Lee and M. Yannakakis's algorithm is such a method that it keeps track of some configurations reached from the initial one, and prefers to search forward than split, but it does not search unless it knows for sure we are accessing inequivalent configurations. Instead of split unstable arc and block immediately, it maintains a queue to keep all unstable blocks, splits them until the current round of search traversed all reachable blocks. It does not split blocks unless it knows they are reachable, and it gives every reachable block a fair chance to split. During the execution of the algorithm, when it reaches some blocks, it will pick one reachable configuration pB from that block, and use it to do further determination of whether the following blocks are reachable.

It maintains one stack and one queue. The stack keeps all reachable states that have been reached. These blocks will be checked whether they are stable or not. The queue keeps all blocks that are unstable and will be split.

Starting from the initial configuration in the initial block, the algorithm does a depth-first search. It marks every unmarked block it reached and checks whether this block is stable or not. If it is not stable, it will put it into the queue, and then continue the search until there is no unmarked blocks. Then it begins to split blocks in the queue. When a block is split, a new block will be generated, and the old block will shrink. The edges on the original blocks need to be checked whether they are still available on the old block, and whether they are applicable on the new block. The new block will be put into stack for the next round of search. When the block is split, all blocks connected to it will be checked again to see whether they need to be split.

To carry out the expanding, the following *Basic Operations* on blocks in π are needed: (i) The intersection of two blocks C∩B; (ii) The inverse of a block B: $a^{-1}(B)$, actually we only need the combination of (i) and (ii): $C \cap a^{-1}(B)$; (iii) The difference of two blocks: C-C' and (iv) Test for emptiness. Assume for now that each operation takes time c.

During the whole process, configurations in different blocks are not equivalent. Let N be the number of blocks in the reachable reduced system (R, ρ, I, T). At any moment of the execution of Algorithm 3.1, there are three classes of blocks: (i) marked block, which

contains one or more reachable blocks in ρ; (ii) unmarked block that contains one or more reachable blocks in ρ; (iii) unmarked block that is disjoint from R. The size of the union of Class (i) and (ii) blocks is no more than N.

The time complexity is $O(ckN^2)$ where N is the number of blocks in the resulting FSM, k is the number of actions, and each block operation takes time c.

2.3 SPIN model checker and Promela

For the model checking approach, we choose SPIN/Promela model checker.

SPIN is a widely used model checker, especially for communication protocols. It is free to get and easy to use. It is well maintained by Bell Lab. Promela stands for Process Meta Language, which is a model description language coupled with SPIN.

2.3.1 Introduction to SPIN

SPIN is a popular open-source software tool for the formal verification of distributed software systems. The tool was developed at Bell Labs in the original UNIX group of the Computing Sciences Research Center, starting in 1980. It supports a high level language, called PROMELA, to specify systems descriptions. It has been used to trace logical design errors in distributed systems design, such as operating systems, communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. The tool checks the logical consistency of a specification. It reports on e.g. deadlocks, unspecified receptions, flags incompleteness and race conditions about the relative speeds of processes. SPIN provides direct support for the use of embedded C code as part of model specifications. This makes it possible to directly verify implementation level software specifications, using SPIN as a driver and as a logic engine to verify high level temporal properties.

SPIN works on-the-fly, which means that it avoids the need to pre-construct a global state graph, or Kripke structure, as a prerequisite for the verification of system properties.

SPIN can be used in three basic modes:

- as a simulator, allowing for rapid prototyping with a random, guided, or interactive simulations
- as an exhaustive verifier, capable of rigorously proving the validity of user specified correctness requirements (using partial order reduction theory to optimize the search)

- as proof approximation system that can validate even very large system models with maximal coverage of the state space.

To generate test case, we need to use the second mode.

2.3.2 SPIN working as a model checker

Model checking is one kind of formal verification, and it relies on building a finite model of a hardware or software system and checking that the model satisfies the desired properties. In order to perform model checking, a formal abstract model has been established in advance. The desired correctness properties are expressed in a concise and unambiguous way. A series of model checking techniques will be applied to perform exhaustive state analysis in order to search the desired properties in verification models.

SPIN accepts a verification model and correctness requirement, and generates a C code model checker. After compiling and executing the model checker, the final results are reported. The correctness requirements can be expressed in 3 aspects: assertions, state labels and never claims. Never claims are used to describe the temporal properties of a Promela model and it can also be expressed in LTL expressions. SPIN embeds an LTL converter, which translates LTL formula into never claims in Promela.

For verification, the Promela and LTL correctness claims are translated into a C model checker. After this model checker is compiled, an executable verifier is generated. When this verifier is executed, it performs on-the-fly modeling checking according to model checking algorithms provided in SPIN. If the verification model does not satisfy the correctness requirements, some counter examples are created.

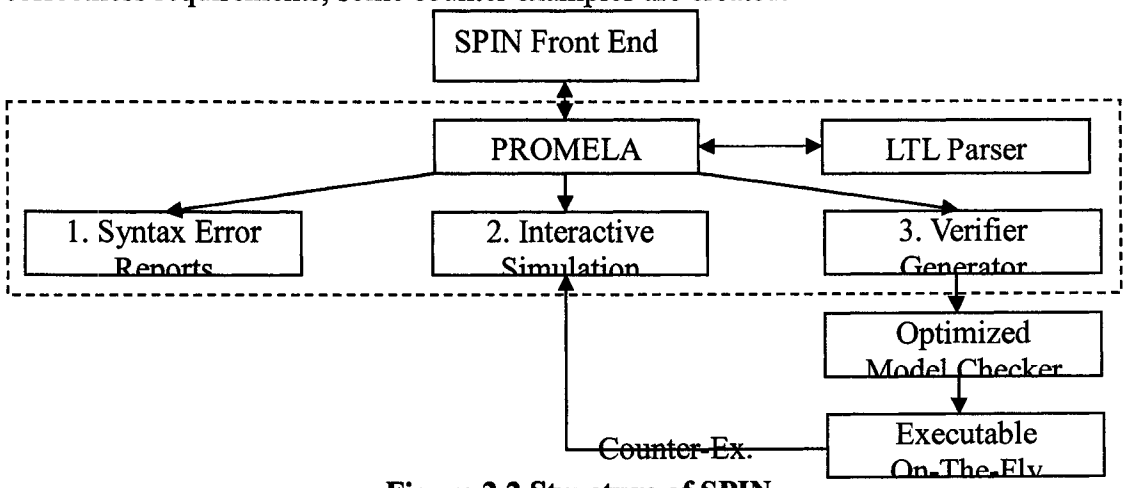

Figure 2.2 Structure of SPIN

SPIN can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic

Correctness properties can be specified as system or process invariants (using assertions), as LTL requirements, as formal Büchi Automata, or more broadly as general omega-regular properties in the syntax of never claims.

The tool also supports both exhaustive and partial proof techniques, based on either depth-first or breadth-first search. To optimize the verification runs, the tool exploits efficient partial order reduction techniques, and (optionally) BDD-like storage techniques. This feature is very important for test case generation and is applied in this thesis.

2.3.3 Promela Introduction

Promela is a verification modeling language. It is for making abstractions of (distributed) software systems that suppress details unrelated to process interaction. The system's behavior is modeled in Promela and verified by SPIN.

Promela programs consist 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 and channels, and global variables define the environment in which the processes run.

In Promela there is no difference between conditions and statements: even isolated Boolean conditions can be used as statements. The execution of every statement is conditional on its executability. Statements are either executable or blocked. The executability is the basic means of synchronization. A process can wait for an event to happen by waiting for a statement to become executable. For instance, instead of writing a busy wait loop:

```
while (a != b)
        skip    /* wait for a= =b */
```

one can achieve the same effect in Promela with the statement

```
(a = = b)
```

A condition can only be executed (passed) when it holds. If the condition does not hold, the execution blocks until it does.

2.4 Using model checker to generate test cases

We introduce test case generation via model checking in this section. Before that, we need

to introduce Linear Temporal Logic first.

2.4.1 Temporal Logic and Linear Temporal Logic (LTL)

In year 1977, Pnueli proposed temporal logic as a very convenient formal language to state, and reason about, the behavioral properties of parallel programs and more generally reactive systems [Pnu77, Pnu81]. Correctness of these systems typically involves reasoning upon related events at different moments of a system execution [OL82].

In defining a system of temporal logic, there are two possible views regarding the underlying nature of time. One is that the course of time is linear: at each moment there is only one possible future moment. The other is that time has a branching, tree-like nature: at each moment, time may split into alternate courses representing different possible futures. In linear time logics, temporal modalities are provided for describing events along a single time line. In contrast, in branching time logic, the modalities reflect the branching nature of time by allowing quantification over possible futures. A major distinction between them is reflected in the classes of time frames: linear orderings or trees.

Regarding a linear sequence of states: $s_0 \rightarrow s_1 \rightarrow ..... \rightarrow s_t \rightarrow s_{t+1} \rightarrow .....$

LTL provides the following temporal operators (p and q represent logic statements):

- "**Finally**"(or 'future"): Fp is true means (p holds in a future point
- "**Globally**" (or "always"): Gp is true means p always holds from now on
- "**Next**": Xp is true means p holds in the next time point
- "**Until**": pUq is true means q will hold in a future point, and p hold from now to that point

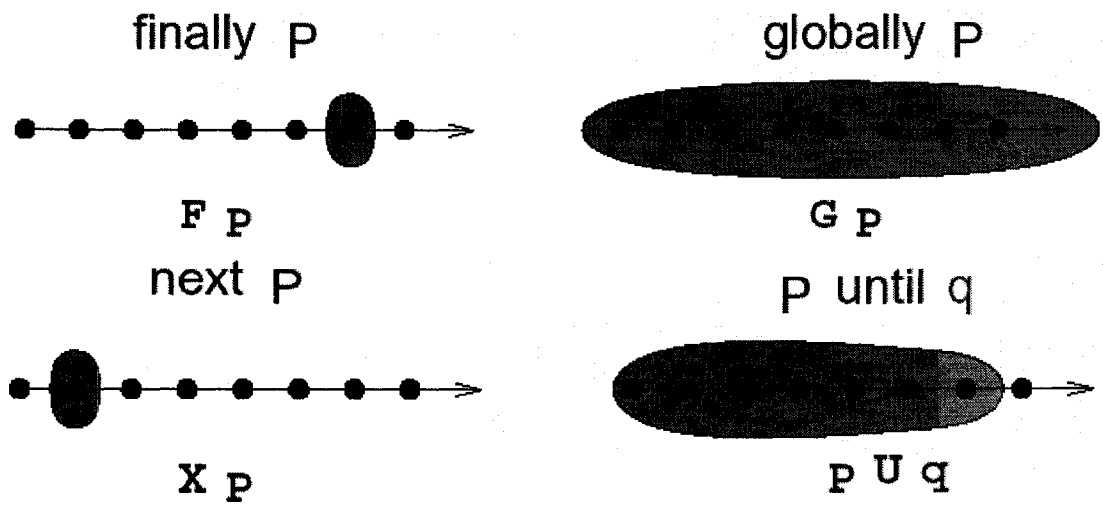

Figure 2.3 Four types of LTL operations

2.4.2 Using model checker to generate test cases

Generally, a model checker (shown in Figure. 2.4) takes a model (of a finite state system) and a specification written as a temporal formula as the input, checks whether the model satisfies the formula. The algorithm returns "true" if the model satisfies this specification; otherwise it returns "false" and provides a counterexample demonstrating why the model does not satisfy the formula. The counterexample feature is vital to the testing & debugging of the system.

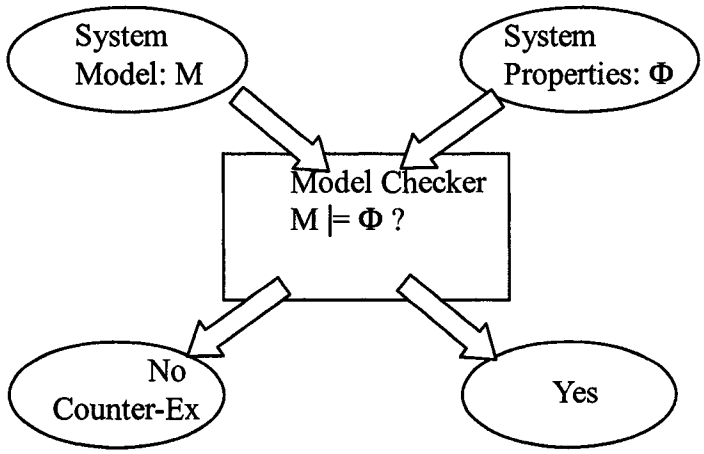

Figure.2.4 Model Checker framework

The model checker was developed to check the correctness of a design by checking whether it satisfies the given properties, while it can be used to do test generation.

Model checking techniques have been proposed as a method for test sequences generation

from formal models in many papers [ABM98], [CSE96], [EFM97], [GH99], [HLSU02], [RH01]. These proposed approaches leverage the witness (or counter-example) generation capability of model-checkers for constructing test cases. Test criteria are expressed as temporal properties. Witness traces generated for these properties are instantiated to create complete test sequences satisfying the criteria. It is well-known that one of the issues that often hinders model-checking is the state-space explosion problem. As the size of the state space to be explored increases, model-checking might become too time-consuming or infeasible. But, in the context of test generation, we are only interested in finding counter-examples against given properties so that counter-examples can be instantiated to test sequences. Generally, finding violations of the properties is relatively easy and that the counter-examples can be constructed easily even for quite large models. Given a finite state transition system, a model checker will exhaustively explore the reachable state space searching for violations of the given LTL properties. Should a property violation be detected, the model checker will produce a counter-example illustrating how this violation can take place. In short, a counter-example is a sequence of transitions that will bring the finite state model from its initial state to a state where the violation occurs.

A model checker can be used to find test cases by formulating a test criterion as a verification condition for the model checker. For example, we may want to test a transition (guarded with condition C) between states A and B in the formal model. We can formulate a property stating that the transition sequence must take the model to state A; in state A, C must be true, and the next state must be B. This property is expressible in the logics that can be used in common model checkers, for example, LTL. We can now challenge the model checker to find a way of getting to such a state by negating the property (saying that we assert that there is no such sequence) and start verification. The model checker will now search for a counterexample demonstrating that this negated property is satisfiable; such a counterexample constitutes a test case that will go through the transition we want. By repeating this process for each transition in the formal model, we use the model checker to automatically derive test sequences that will cover all transitions of the model.

This approach can be used to generate tests for a wide variety of coverage criteria, such as all state variables have taken values, and all decisions in the model have been evaluated to both true and false.

This test generation process is outlined in the following Figure 2.4. The figure takes the simple EFSM in Figure 1.1 as an example. It claims a property that counter= =6 would never happen as an LTL statement, !([](counter= =6)). Then, the SPIN (details are given in the following section) model checker runs to check whether the property holds or not, and find that when the system is in state s_2, the property does not hold. So the model checker gives an error trace, starting from initial state s_0 to ending state s_2, to prove the property does not hold. This error trace is exactly what we want. It can be used as a test sequence from s_0 to s_2.

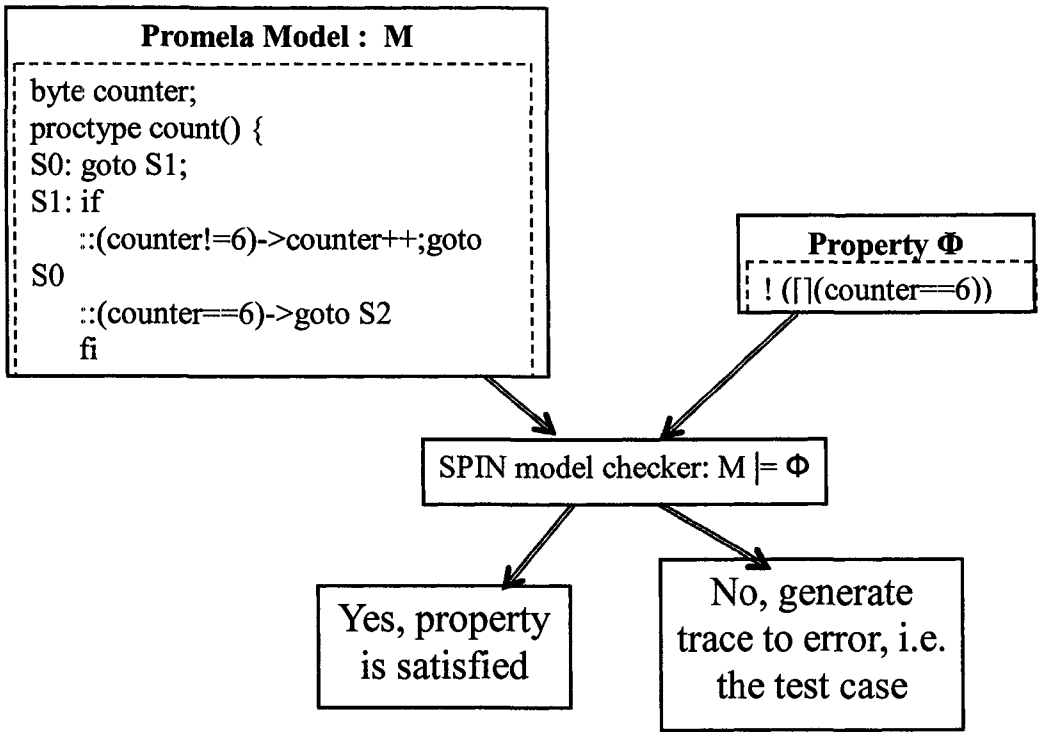

Figure 2.5 Framework of test case generation via model checking

2.5 Coverage criteria

When we perform software testing, we want to know on what degree the testing can demonstrate the absence of errors in a program? In other words, is the testing adequate? How much detail or rigor should be applied to the testing? How large and diverse should the test be?

Coverage criterion is the measurement.

A coverage criterion is an assumption about how defects are distributed in the program in relation to the program model. The stronger the assumptions are made, the smaller the size

of test set is.

Among many types of coverage criteria, we use two most common ones:

All State coverage: Requires every state to be covered at least once.

All Transition coverage: Requires every transition be covered at least once.

For the EFSM in Figure 1.1, the state overage can be expressed by the following set of LTLs: <>**S*n***, n=0,1,2, which means "eventually, it can reach Sn", and transition **S*i*→S*j*** can be expressed by the following set of LTLs: <>**(S*i*** && **xS*j*)**, i,j=0,1,2, which means "eventually, it will hold that Si reached and the next state is Sj". "<>", stands for "Finally" operator of LTL (see LTL operators in Figure 2.3). && is logic "and" operator.

To make the comparison, we make the two methods generate test suites that meet the two coverage criteria.

3 Comparison of the Two Approaches

The purpose of the experiment is to compare the two test generation approaches: model checking approach and expansion approach.

To compare the two approaches, we need to do the followings:

1. Define the comparison content and comparison standard.
2. Define a set of translation rules that can translate a general EFSM into a Promela model, against which SPIN model checker can run model checking to generate feasible paths we want.
3. Implement D. Lee and M. Yannakakis's algorithm [LY92] which can expand a general EFSM into a FSM and further derive all feasible paths.
4. Obtain sets of EFSM examples so that we can run the two approaches against them.

As we mentioned in section 1, the content of comparison are time consumption and number of states explored. The implementations of the two approaches are discussed in section 4. In this section, we introduce how to obtain sets of EFSM examples for the experiments and how to carry the experiment.

3.1 Generic EFSMs generation from real protocols

To make the experiment convincing, the set of EFSMs used in our experiments should meet the following requirements:

1. They are from well-known protocols,
2. They should cover as many types of EFSM graphs' characteristics as possible, such as cycles and self-loops,
3. Their sizes should have a linear equitable increase.

Running sufficient number of EFSMs from well-known protocols is very necessary to reach a convincing result, but it is not enough. First, because we want to compare the time consumption in the two methods and their performances when EFSMs' sizes increase, the set of EFSMs should have a linear equally increasing sizes, by which we can observe the increase of the consumed time as EFSM's size increase. As we mentioned before, a major difficulty of expanding an EFSM into a FSM is from self-loops and cycles that exist in EFSMs, so we must have our examples contain such characteristics. For the same reason, the EFSMs must contain other characteristics that common EFSMs have, such as branches and guards.

However these requirements are not easy to meet all together. Most examples only meet one or two requirements. For example, EFSMs in Figure 1.4 and 2.1 are small in size, while others may not contain self-loops or cycles.

The most difficult thing is to have a set of EFSMs with linear increasing sizes. The size of an EFSM refers to its number of states, number of transitions, number of variables and domain sizes of variables' values. The more states, we have the larger the EFSM will be. The more transitions, we have the more complex the EFSM will be, and generally, more transitions lead to more states be generated when we expand the EFSM to FSM. We browsed a wide range of available EFSMs. Some are small and some are large, but we could not collect a set of EFSM examples with linear increasing sizes.

The solution to this problem is to implement an EFSM generation mechanism to derive EFSMs with specified size and characteristics from the selected protocols, by which we can make them meet the above requirements.

To derive an EFSM from a select protocol means we already have an EFSM for that protocol, but the EFSM may not meet our expectation, so we modify it by adding nodes, transitions or expanding domain size of variable values. The additional nodes and transitions should be constructed via duplicating original nodes and transitions. By duplicating some nodes, transitions and expanding variable value range, we enlarge an EFSM to a specific size. In the mean while, we can add self-loops and cycles as we need.

For example, we can enlarge the EFSM in Figure 2.1 into the following EFSMs, in which all original transitions are kept and transplant into the additional states. The generated EFSM has a larger size as we expected.

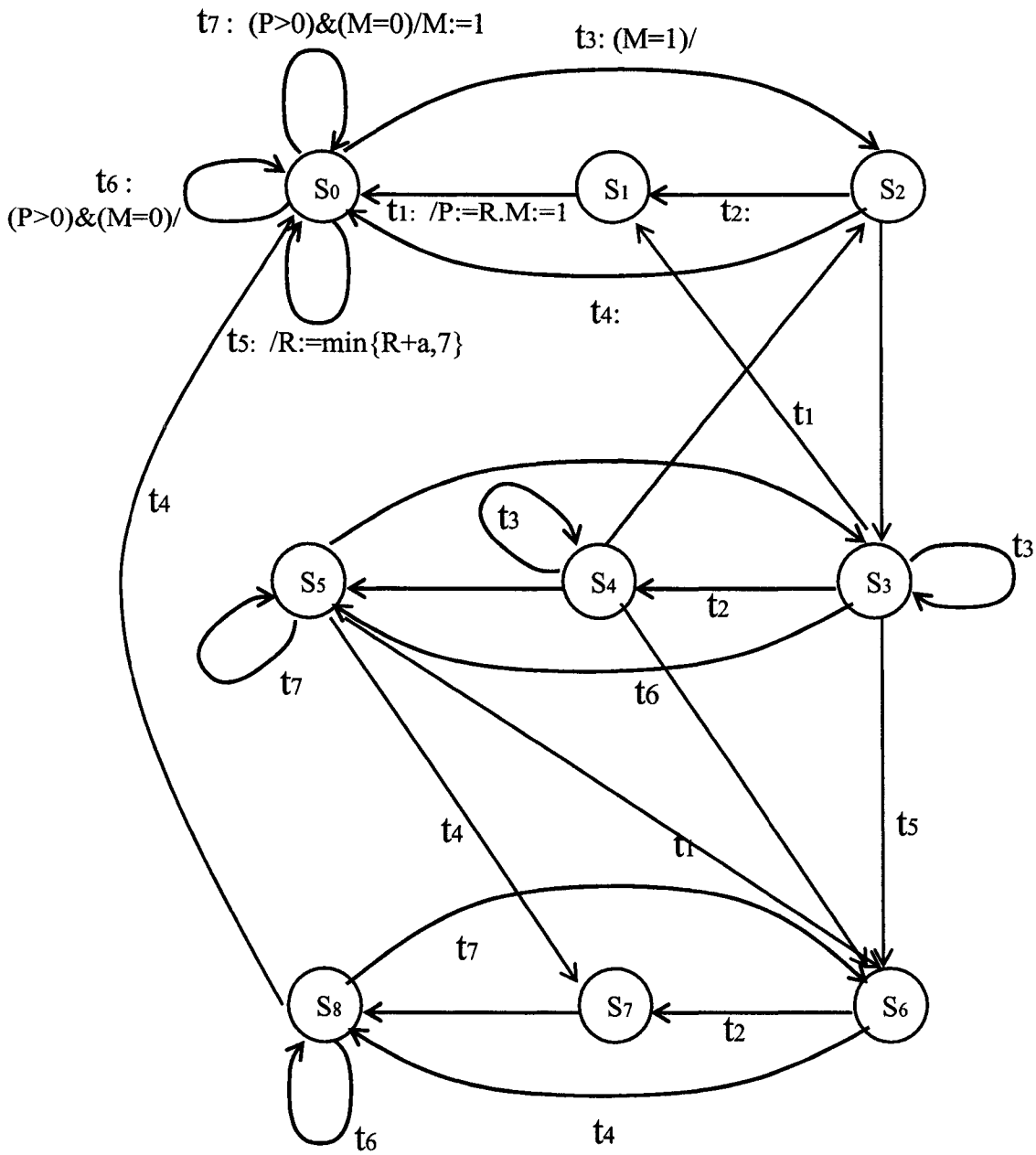

Figure 3.1 An generated EFSM with a specified size

It is reasonable to do this derivation. Although the derived EFSM does not represent the original protocols any more, it has no impact on our experiment.

In this thesis, we select three real communication protocols: Active Monitor Protocol, which is part of the token ring protocol of ANSI/IEEE Standard 802.5 [ANSI2], the INitiator-RESponder protocol [Hog91] and a simplified class 2 transport protocol [RTD96]. Having the original EFSMs corresponding to the protocols, we enlarge the

EFSMs via the approach mention above. We make a set of EFSMs with 10, 20, 30, 40, 50, 60 states and add 4 transitions to every additional state. We have chosen number 10, 20, 30, 40, 50, 60 only because we want the EFSMs have linear increasing sizes. We can also make them 15, 30, 45, 60, 75, ... etc. For the additional parts of states, we want most of them to be reachable. Otherwise the additional states are meaningless for us. It is obvious the more transitions we add, the less number of states will be unreachable. By observation, we found it is good enough if we add 4 more transitions for every addition states.

Having the sets of EFSMs ready, we run ten times for each experiment to eliminate possible interference. For the result, we eliminate obvious aberrancies, then calculate the average value of others. As we have 3 sets of EFSMs, every set contains 7 EFSMs, and we need to run them by two approaches, we totally run experiments10x3x7x2 times.

4 Implementation of two approaches

4.1 Implement the expansion approaches

The implementation is composed of two parts. The first part is a direct implementation of the algorithm [LY92], which expands an EFSM into a FSM by splitting states and eliminating predicates. The second part is an implementation to find the shortest path between the initial state (the root node) and destination transitions (arcs) or states (nodes) of a generated FSM. We developed a segment of code to do this job. The generated path is the test case for the original EFSM.

We use Microsoft C# to do the coding.

4.1.1 Implement the algorithm

The implementation is straightforward. The code is in Appendix A.

4.1.2 Implement the path generation

The code is the shadow part of Appendix A

This part was not discussed in [LY92]. We developed a segment code to implement it. The input is the generated FSM. We can regard it as a direct graph with an initial node. The output is a set of paths staring from initial node to all other reachable nodes and arcs. To make sure the generated path is the shortest path from the initial node to a specific node or arc, we browse the graph in this way: We start from the initial node, then at each iteration, we check all the nodes and arcs directly reached from the nodes we checked in the previous iteration. If a node or arc has been visited, we skip it. If it has not been visited, we record the current path for output. We repeat it until there is no unvisited reachable nodes

This part of work only takes less than 1% time consumption of the whole process. See section 5.4.

4.2 Translating EFSM into Promela model

In this section, we discuss how to translate an EFSM model into a Promela model and introduce some specific issues. We use the example EFSM in Figure 2.1 to demonstrate the translation. Then we discuss how the verification works on the translated Promela mode.

An EFSM is composed of states, transitions, variables, inputs and outputs. We discuss them one by one.

- **One EFSM model → One Promela process**

One Expanded Finite State Machine will be translated into one Promela process. Although processes in Promela can be recursive, accept parameters and exchange values with other processes by the use of global variables and message channels, we only use it in the simplest way. We declare a process to represent an EFSM. A declaration of a process starts with the keyword "proctype" followed by the name of the process. Each process has a body in which variables are declared and statements are specified. The body of a process is marked with "{" and "}".

```
active proctype ActiveMonitor
{ ......
}
```

- **EFSM Variables → Promela variables**

Promela provide the following 4 types of variable types

Typename	Typical Range
bit or bool	0..1
byte	0..255
short	$-2^{15}-1..\ 2^{15}-1$
int	$-2^{31}-1..\ 2^{31}-1$

Integer is the most important data type. Promela pre-defined types: bit, byte, short and int are all integers, but their value ranges are different. As the real and float numbers will lead to infinite size of the data set, they are not included into Promela. For the same reason, we only consider integers in our EFSM models

The range of a variable in an EFSM may not exactly match any one of the four provided data type, for example, the variables R and P of EFSM in Figure 2.1 range from 0 to 7. Among all the types that cover this range, we can choose one with the smallest data range. So we define P and R as byte type. As the actual range depends on the operations carried on that variable, this enlargement will not lead to the increasing of state space. As the variable M can only be 0 or 1, we can assign bit data type directly as the followings.

```
byte P, R;
byte a=1;
bit M;
```

```
active proctype ActiveMonitor
{ ......
}
```

- **EFSM states → Promela blocks with labels**

The process body is composed of several blocks, each corresponding to one state of the EFSM. The block is labeled with the state# for identification, and contains transitions description. Although a Promela program, like all other structure programs, is sequentially executed, we use "goto state#" statement to enforce it to switch between state blocks. Thus the Promela model will behave exactly same as a EFSM model.

```
byte P, R;
byte a=1;
bit M;
active proctype ActiveMonitor
{
state0:
        ......
state1:
        ......
state3:
        ......
}
```

- **EFSM Transitions → Promela "if ... goto ..." statements**

The transition between two states are translated into Promela "if" and "goto" statements. As we mentioned above, one EFSM state corresponds to one block in Promela program. For example, the block labeled with state0 corresponds to the S0 in the EFSM graph. The block is actually a segment quoted by "if... fi". Promela provides double colon "::" operator together with arrow operator "→" to represent "if condition is satisfied, then action, otherwise hold". The condition here can be used to represent the guard of an EFSM transition. The actions followed correspond to the computation jobs the transition will complete. In one "if ... fi" block, there can be multi "::" operators. These features are dedicated to multi transitions from one state. For example, in Figure 2.1, there are five

outgoing transitions from S0. Correspondingly there are five "**::** (condition) → action1; action2; …" in that block. Each corresponds to one transition. In an EFSM, if there are more than one guard that can be satisfied, it will randomly choose one transition to go through. Promela use multi "::" operation in "if … fi" statement to handle it.

```
state0:
    tempR=R+a;
    if
    :: (true)->
        if
        :: (tempR<7)->R=temp; goto state0
        :: (tempR>=7)->R=7; goto state0
        fi
    :: (P==0&&M==0)-> goto state0
    :: (P>0&&M==0)->M=1; goto state0
    :: (M==1)-> goto state2
    :: (true)->goto state2
    fi
    fi;
```

Coverage criteria→Promela's *never* claim

As we stated in section 2, to use model checker to generate a feasible path, we need to give a property that claims the EFSM can not reach a specific state or edge, then run model checker against this property to generate a counterexample, which actually contains the path we want. For example, if we want to test the edge from S1(state==1 && M==1 && R==5 && P==5) to S0(state==0 && M==0 && R==5 && P==5), we need to define the following two statement:

```
#define p (M==1 && R==5 && P==5)
#define q (M==0 && R==5 && P==5)
```

Then, we claim the following LTL property:

```
!(p&&(Xq))
```

It means: it will never happen that p holds and q holds right after that. X is temporal logic

operator "next". SPIN will translate the property into a never claim as:

```
never {    /* (<>p) */
T0_init:
    if
    :: ((p)) -> goto accept_all
    :: (1) -> goto T0_init
    fi;
accept_all:
    skip
}
```

It is actually a monitor automata. Whenever the system goes into a new state, this automata will check whether p holds. If so, the automata will check whether q holds at the immediate state followed. If so, the model checker stops and the counterexample including the path from initial state to current state is provided as output.

Having all the above elements, now we give the complete Promela specification derived from the EFSM model in Figure 2.1..

```
#define p (M= =1 && R= =5 && P= =5)
#define q (M= =0 && R= =5 && P= =5)
byte P, R;
byte a=1;
bit M;
byte temp;
active proctype ActiveMonitor()
{
M=0;
P=0;
R=0;
state0:
    R=R+a;
    if
    :: (true)->
```

```
        if
        :: (R<7)->R=temp; goto state0
        :: (R>=7)->R=7; goto state0
        fi
    :: (P==0&&M==0)-> goto state0
    :: (P>0&&M==0)->M=1; goto state0
    :: (M==1)-> goto state2
    :: (true)->goto state2
    fi;
state1:
    P=R;
    M=0;
    goto state0;
state2:
    goto state1;
}
never {    /* (<>p) */
T0_init:
    if
    :: ((p)) -> goto accept_all
    :: (1) -> goto T0_init
    fi;
accept_all:
    skip
}
```

5 Result analysis and Conclusion

5.1 Tuning Promela model can improve performance

Through the experiment, we found that, the time consumption of test case generation via model checking can be reduced by tuning the Promela model.

This is due to the SPIN model checker deals with non-determinism. When a state has two or more feasible outgoing transitions, it will be non-deterministic which transition will be first chosen and traversed. See the example in Figure 5.0. Transitions 1 and 5 are always true, and transitions 2, 3, 4 have guards. At anytime, the state could have two (1 and 5) or three (1, 5 and one of 2, 3, 4) outgoing feasible transitions available. SPIN always traverses the first feasible one, then the second, etc. In this case, it always traverses transition 1, while 5 is always the last transition to traverse. Thus, if the faulty state occurs on the path following the last transition, the worst case happened, and the time consumption will be maximum. So when facing non-determinism, SPIN follows a fixed sequence to traverse rather than randomly traverses. We can not say this is a drawback. On the contrary, sometimes it is even better than randomly traverse. This is because with a fixed sequence traverse, we have a chance to tune the Promela model to get a better performance. For example, if we have some heuristics such as where the faulty state could locate, we could tune the Promela model to move that specific transition forward. With the random traverse, the average performance cannot be improved, and the worst case still could happen, while we have no way to control it.

```
State n:
    tempR=R+a;
    if
    :: (true)→ ..... transition 1
        if
        :: (tempR<7)→R=temp; goto state0
        :: (tempR>=7)→R=7; goto state0
        fi
    :: (P= =0&&M= =0)→ goto state0 ..... transition 2
    :: (P>0&&M= =0)→M=1; goto state0 ..... transition 3
    :: (M= =1)→ goto state2 ..... transition 4
```

```
:: (true)→goto state2 .....transition 5
fi;
```

Figure 5.0 A state has multi non-deterministic out-going transitions

Table 5.1 is a comparison of time consumption of test generation via SPIN model checker between a well tuned Promela model and an untuned one. The domain size of variables is 9800. The time consumption for untuned Promela model is 40-50 times more than that of that tuned one.

Number of EFSM states	Test case generation time consumption(second)	
	well tuned Promela model	not tuned Promela model
3	0.09	5.65
10	0.19	9.12
20	0.32	13.32
30	0.71	23.9
40	0.94	33.14
50	1.25	41.13
60	1.63	52.21

Table 5.1 Comparison: Time consumption of test generation via SPIN model checker for well tuned Promela model and not tuned Promela model

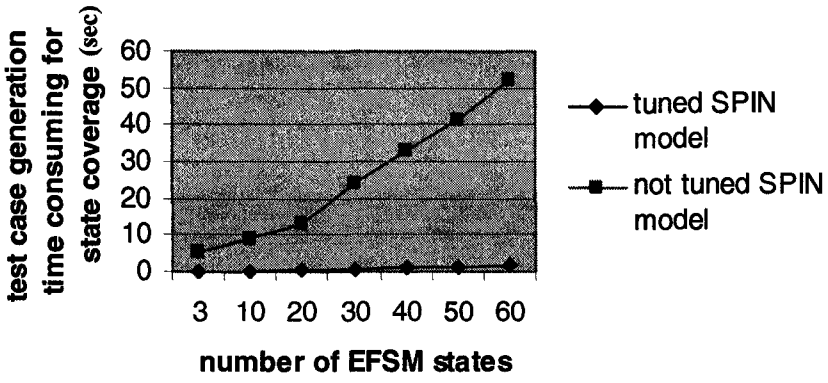

Figure 5.1 Comparison: Time consumption of test generation via SPIN model checker for well tuned Promela model and not tuned Promela model

5.2 Impact of the domain size of the variables values

During the experiment, we found that with expansion method in [LY92], the domain size of the variables values has a bigger impact on the time consumption. As domain size

increases, of course, the time consumption of both methods will increase, but that of expansion method increases much faster than that of SPIN model checking method. Figure 5.2, 5.3 and 5.4 show the impact of domain size of variable value on time consumption. The three figures are the results of three experiments with all other conditions the same except for the domain size of variable value. The domain sizes of the variables' values in the three experiments are 128, 3200 and 9800 respectively. The time consumptions are shown in Table 5.2, 5.3 and 5.4. We can see in Figure 5.2 when domain size is 128, the time consumption of expansion method is roughly 1/20 ~ 1/3 of that of SPIN model checking method.

number of EFSM states	Test case generation time consumption (second)	
	Expansion method	SPIN method
3	0.02	0.43
10	0.06	0.62
20	0.11	0.9
30	0.23	1.48
40	0.56	2.31
50	1.17	3.07
60	1.74	4.24

Table 5.2 Time consumptions of the two approaches when domain size is 128

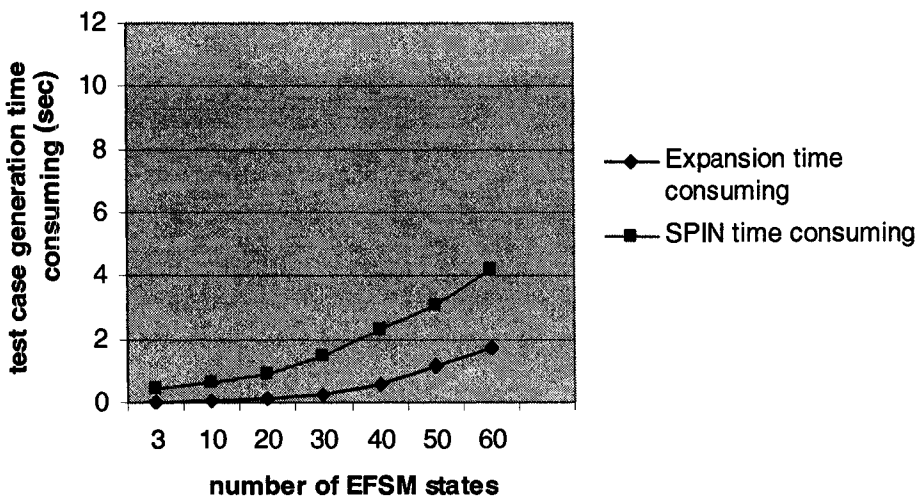

Figure 5.2 Time consumptions of the two approaches when domain size is 128

When the domain size increases to 3200 (see the Table 5.3 and Figure 5.3) the time consumption of both method increase, but that of expansion method increases faster and is approximately equal to that of SPIN model checking method.

number of EFSM states	Test case generation time consumption (second)	
	Expansion method	SPIN method
3	0.11	0.17
10	0.56	0.45
20	1.18	0.94
30	1.9	1.81
40	2.81	2.71
50	3.94	3.63
60	5.61	5.35

Table 5.3 Time consumptions of the two approaches when domain size is 3200

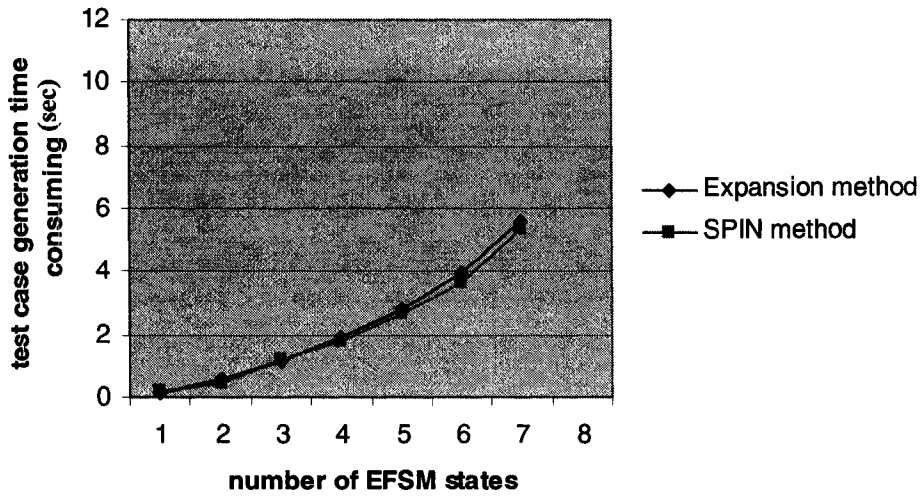

Figure 5.3 Time consumptions of the two approaches when domain size is 3200

When the domain size increased to 9800 (see Table 5.4 and Figure 5.4), the time consumption of expansion method is approximately 1.5~2 time of that of SPIN method.

number of EFSM states	Test case generation time consumption (second)	
	Expansion method	SPIN method
3	0.26	0.23
10	0.78	0.5
20	2.09	1
30	4.21	1.96
40	5.77	3.02
50	8.22	4.24
60	10.83	5.61

Table 5.4 Time consumptions of the two approaches when domain size is 9800

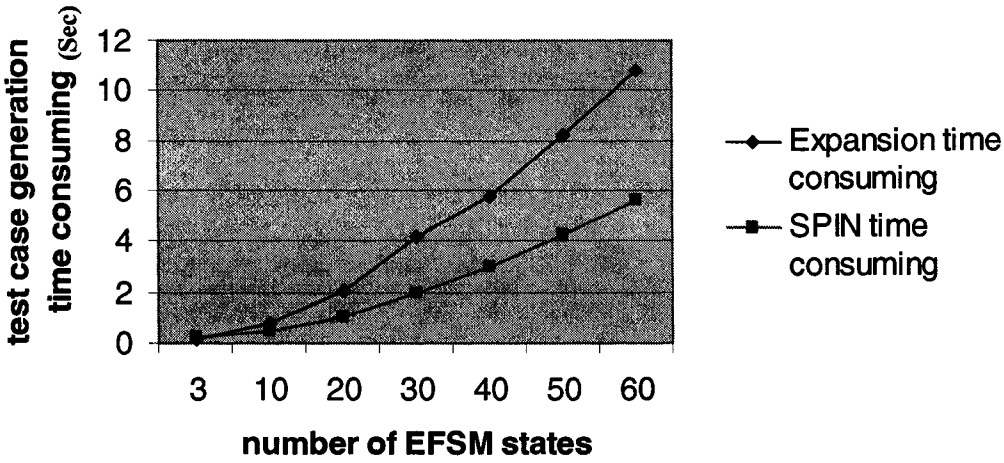

Figure 5.4 Time consumptions of the two approaches when domain size is 9800

The reason of this phenomenon is that for expansion method, the program needs to keep all values of all variables for every state in memory, and does calculations on them when a transition occurs. While for SPIN model checker, SPIN traverses states with only particular values that the variables are having.

domain size of variables values	Test case generation time consumption (second)	
	Expansion method	SPIN method
128	1.74	4.24
3200	5.61	5.35
9800	10.83	5.61

Table 5.5 The time consumptions increase as the domain size of variables values increases

So increasing domain size of variables values does not have much impact on SPIN model checking. We compare the time consumption of the two methods as the domain size

increases in Table 5.5 and Figure 5.5. The curves show the time consumption of SPIN model checking method almost stays unchanged while that of expansion method increases a lot.

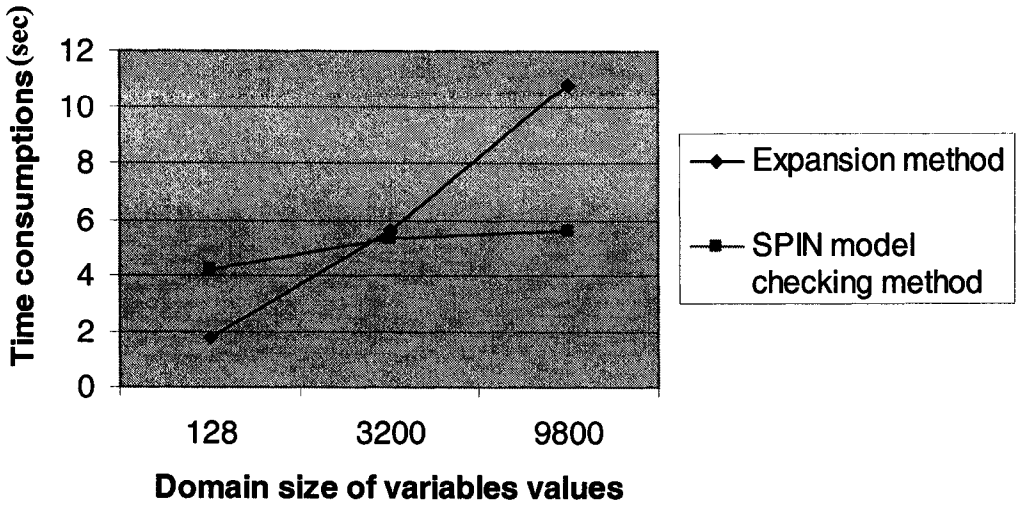

Figure 5.5 The time consumptions increase as the domain size of variables values increases

5.3 Time consumption and state space efficiency tradeoff

As we mentioned before, every run of SPIN model checker can only generate one test case. So if we want to cover all the *n* states of an EFSM which has *n* states, we need to run SPIN model checking *n* times, which is less efficient than expansion method, while on every run of SPIN model checker, it checks less number of states than expansion method. So actually there is a tradeoff here between time efficiency and state space efficiency.

The experiment shows that the tradeoff does exist. Table 5.6, 5.7 and Figure 5.6, 5.7 show the tradeoff.

In Figure 5.6, the curve on the top represents the time consumption of SPIN model checking method for generating both states and transitions coverage test cases. It is the sum of the two dashed curves below. The solid curve equipped with triangles represents the time consumption of expansion methods for generating the same test cases. It shows that the time consumption of SPIN method is always more than that of expansion method, which is reasonable and conform to what we expected.

number of EFSM states	Time consumption (second) of SPIN method test case generation for			Time consumption (second) of expansion method test case generation time for state and transition coverage
	Transition coverage	state coverage	total	
3	0.26	0.17	0.43	0.02
10	0.4	0.22	0.62	0.06
20	0.6	0.3	0.9	0.11
30	1.02	0.46	1.48	0.23
40	1.62	0.69	2.31	0.56
50	2.04	1.03	3.07	1.17
60	3.01	1.23	4.24	1.74

Table 5.6 Comparison of time consumptions

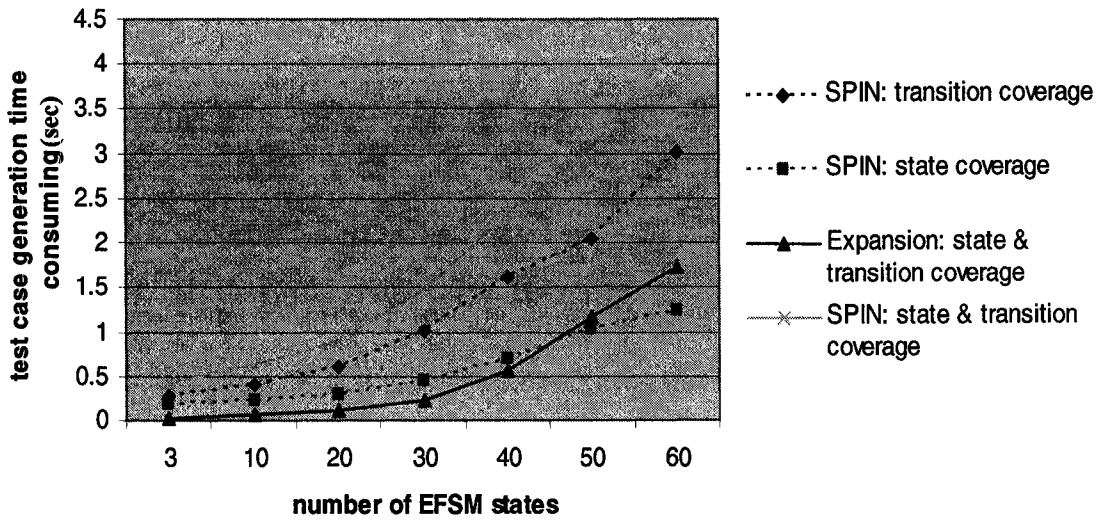

Figure 5.6 Comparison of time consumptions

Table 5.7 is composed of two sets of numbers. The numbers in the first column are the number of states in the original EFSM. The numbers in the second column are the number of the states in the derived FSM. As we mentioned before, the derived FSM contains two parts: the reachable states and unreachable states. We regard the stetes number of FSM as the number of states that expansion method needs to explore. The numbers in the third and fourth columns are the numbers of states that SPIN model checker needs to explore to generate a test case for a specific state and transition respectively. For states and transitions coverage, we need to generate one test case for every state and transition, so actually we have many test cases generated. Here we use the maximum number, the number when

worst case happens, as the number of states explored for SPIN method generating the test case, and to compare with the expansion method.

In Figure 5.7, the curve on the top represents the number of derived FSM states, the other two curves represent the number of states explored for SPIN method. It shows that the number of expansion method is larger the number of SPIN method. The result is reasonable because expansion method needs to explore all states to derive FSM, while SPIN model checker does not need to do so.

number of EFSM states	number of states expansion method needs to explore	worse case number of states explored when using SPIN to generate a test for	
		a transition	a state
3	8	6	5
10	20	8	7
20	41	13	12
30	68	18	17
40	80	23	22
50	101	28	27
60	121	33	32

Table 5.7 Comparison of number of states explored

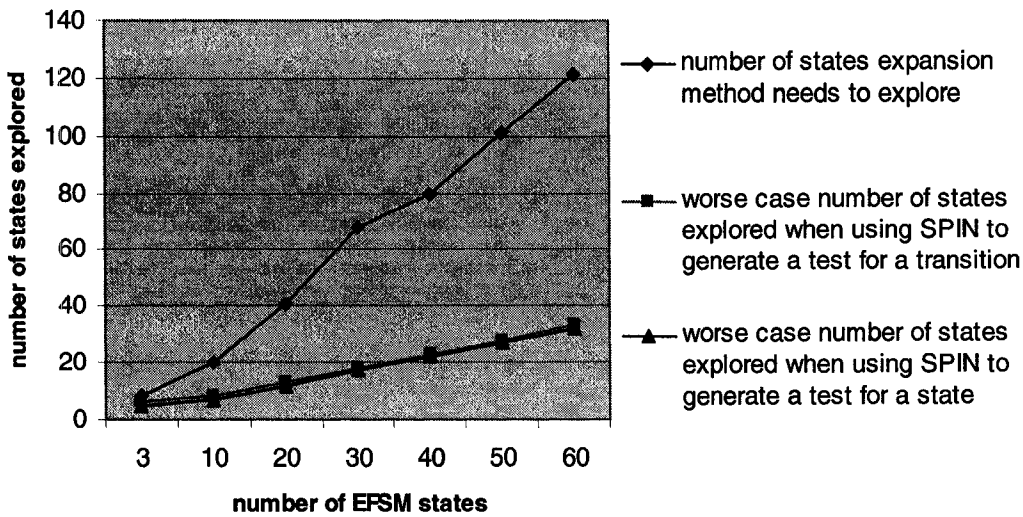

Figure 5.7 Comparison of number of states explored

Comparing Figure 5.6 and 5.7, we can see there is obviously a tradeoff between time efficiency and state space efficiency. The expansion method keeps all derived states in the memory, so it is more efficient on time consumption at the cost of state space (memory)

inefficiency. On the contrary, the SPIN model checking method does not need to explore all states, and stops when it reaches the state that is against the property, so it is more state space efficient at the cost of time efficiency.

5.4 Expansion method will be more efficient in the presence of more coverage criteria

Another conclusion is that expansion method will be more efficient if more coverage criteria present. Of course, the test case generation for the additional criteria should not be harder than that of state or transition coverage.

The expansion process includes two steps: expanding EFSM into FSM first, then generate needed paths from the FSM. Figure 5.8 shows how much time needed for the two steps. The higher one is time consumed on expanding step. We can see that generating paths only consume a very small part of time. The rate is 0.015:1.73 ≈ 8.6:1000. If we have more coverage criteria, and these coverage criteria are similar to state or transition coverage criteria regarding their time complexity of path finding, the expansion method could be more efficient, especially when compared with SPIN method. For example, path coverage criterion [ZHM96] is such a similar criterion.

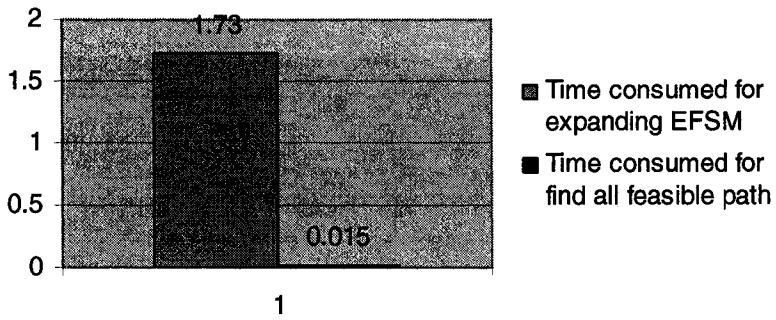

Figure 5.8 Time consumption of two steps of expansion method

5.5 The expansion algorithm has two native drawbacks:

5.5.1 The reverse transition need to be coded individually

As we mentioned before, both methods need EFSMs to be modeled and the modeling is straightforward in both methods. However, the algorithm [LY92] additionally needs reverse transitions to be implemented, in another word, coded into the program. As we mentioned before, the algorithm assumes having reverse functions available, which means

given transition T: A→ B, and all variables' values when the system is in state B, we can calculate and obtain all the variable values when the system is in state A. But unfortunately, the reverse function, expressed as C^{-1}, is not available directly. As we know, the function C could be a "one to one" or "many to one" mapping from variable set A to variable set B. So in the reversed direction, "one to many" mapping could exist, which leads to difficulty of coding and more time consumption. Further, to code the reverse function manually prevents the algorithm to be generic.

5.5.2 The expansion method could waste time on unreachable states

As we know, expanding EFSM will generate two parts of states, the reachable part and unreachable part. When we further generate paths, the unreachable states will not be reached, so they will be excluded automatically without any efforts. D. Lee and M. Yannakakis did not implement a mechanism to eliminate the generation of unreachable states in this algorithm, which means it could waste some time to generate unreachable states. This part of job is meaningless. As the algorithm can not determine a state is reachable or not before it is generated, it is hard to eliminate the waste in advance. Fortunately, the algorithm will not go further to traverse and expand from an unreachable state, which means it is not a serious waste.

While a model checker does not have such a problem because it will never reach unreachable states.

6 Future Work

The future work can be considered in two aspects: comparing the symbolic version of those expansion algorithms with model checking method and applying other kind of comparison on the two approaches.

In this thesis, we only considered explicitly expressed EFSM models. The states can also be symbolically represented: for example we can represent states using a Binary Decision Diagram (BDD) [BCM+92]. There are also some algorithms developed to do the equivalent expansion on symbolically expressed models (cf. [FV99], [HL95], [CS01]). It is possible to do the similar comparison on that.

In this thesis, we focus our comparison on time consumption and number of states explored regarding all transitions coverage and all states coverage. The comparison could be done on other approaches. For example, we can consider other coverage criteria.

Bibliography

[AB99] P. E. Ammann and P. E. Black, "A specification-based coverage metric to evaluate test sets", In *Proc. of the Fourth IEEE International Symposium on High-Assurance Systems Engineering, IEEE Computer Society*, Nov. 1999.

[ABM98] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications", In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods* (ICFEM'98), pages 46–54. IEEE

[AD97] Larry Apfelbaum and John Doyle "Model-Based Testing", In *Proceedings of Software Quality Week* 1997

[ANSI2] *International standard ISO/IEC 8802-5, ANSI/IEEE std 802.5*, 1992. [BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond", In *Information and Computation*, 98(2):142-170, June 1992

[BEI83] B. Beizer, "Software Testing Techniques", In *New York: Van Nostrand Reinhold Company*, 1983.

[BFH90] A. Bouajjani, J. C. Fernandez, and N. Halbwachs, "Minimal model generation", In *Proc. of Int. Conference on Computer Aided Verification* (CAV'90), volume 531 of Lecture Notes in Computer Science, pages 197–203. Springer-Verlag, Berlin, 1990.

[CS01] R. Cleaveland and O. Sokolsky, "Handbook of Process Algebra", chapter Equivalence and Preorder Checking for Finite-State Systems, pages 391–424, North-Holland, 2001.

[CZ93] S. Chanson and J. Zhu, "A Unified Approach to Protocol Test Sequence Generation," In *Proc. IEEE INFOCOM,* pp. 1d.1.1-1d.1.9, 1993.

[CK96] K.T. Cheng and A.S. Krishnakumar, "Automated Generation of Functional Vectors Using the Extended Finite State Machine Model," In *ACM Trans. Design Automation*, vol. 1, no. 1, pp. 57-79, Jan. 1996.

[CSE96] J. Callahan, F. Schneider, and S. Easterbrook, "Specification based testing using model checking", In *Proc. of the SPIN Workshop*, August 1996.

[DU00] A. Duale and U. Uyar, "Generation of Feasible Test Sequences for EFSM Models," In *Proc. IFIP Int'l Conf. Testing of Communicating Systems (TestCom)*, pp. 91-109, Sept. 2000.

[EFM97] A. Engels, L. M. G. Feijs, and S. Mauw, "Test generation for intelligent networks using model checking", In *Proceedings of TACAS'97*, LNCS 1217, pages 384–398. Springer, 1997.

[FV99] K. Fisler and M. Y. Vardi, "Bisimulation and model checking", In L. Pierre and T. Kropf, editors, *Proc. of Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of Lecture Notes in Computer Science, pages 338–341. Springer-Verlag, Berlin, 1999.

[GH99] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications", In *Software Engineering Notes*, 24(6):146–162, November 1999.

[HB94] T. Higashino and G. Bochmann, "Automatic Analysis and Test Case Derivation for a Restricted Class of LOTOS Expressions with Data Parameters", In *IEEE Trans. Software Eng.*, vol. 20, no. 1, pp. 29-42, Jan. 1994.

[HCLSU03] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural, "Data flow testing as model checking", In *Proceedings of 2003 International Confernece on Software Engineering*, Portland, Oregon, May 2003.

[HL95] M. Hennessy and H. Lin, "Symbolic bisimulations", In *Theoretical Computer Science*, 138(2):353–389, 1995.

[HLJ95] Chung-Ming Huang, Yuan-Chuen Lin, and Ming-Yuhe Jang, "An Executable Protocol Test Sequence Generation Method for EFSM-Specified Protocols", In *IFIP Transactions C: Communication Systems - Protocol Test Systems*, pp. 20-35, 1995.

[HLSU02] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '02), Grenoble, France, April 2002.

[Hog91] D. Hogrefe, "OSI formal specification case study: The INRES protocol and service" tech. rep., IAM 91-012, University of Berne, Institute of Computer Science and Applied Mathematics, 1991

[Hop71] J. E. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton", In Kohavi and Paz, editors, *Theory of Machines and Computations*, pages 189–196, Academic Press, 1971.

[KS90] P. C. Kannellakis and S. A. Smolka, "CCS expressions, finite state processes, and three problems of equivalence", In *Information and Computation*, 86(1):43–68, 1990.

[LY92] D. Lee and M. Yannakakis, "Online minimization of transition systems", In *Proc. of 24th ACM Symposium on Theory of Computing* (STOC'92), pages 264–274, ACM Press, 1992.

[LCM94] Liang-Seng Koh, Chang-Jia Wang and Ming T. Liu, "A Functional Model for Test Sequence Generation", In *Computers and Communications*, 1994 IEEE 13th Annual International Phoenix Conference on 12-15 Apr 1994 Page(s):336

[LY94] D. Lee and M. Yannakakis, "Testing Finite-State Machines: State Identification and Verification", In *IEEE Trans. Computers*, vol 43, no. 3, pp. 306-320, Mar. 1994.

[LHHT97] X. Li, T. Higashino, M. Higuchi, and K. Taniguchi, "Automatic Generation of Extended UIO Sequences for Communication Protocols in an EFSM Model", In *Proc. Seventh Int'l Workshop Protocol Test Systems*, pp. 225-240, Nov. 1997.

[MP92] R. Miller and S. Paul, "Generating Conformance Test Sequences for Combined Control Flow and Data Flow of Communication Protocols", In *Proc. 12th Int'l Symp. Protocol Specification, Testing, and Verification*, pp. 12-27, 1992.

[PBY96] A.F. Petrenko, G. v Bochmann, and M.Y. Yao, "On Fault Coverage of Tests for Finite State Specifications", In *Computer Networks and ISDN Systems*, vol. 29, no. 1, 1996.

[PT87] R. Paige and R. E. Tarja, "Three partition refinement algorithms", In *SIAM Journal on Computing*, 16(6):973–989, 1987.

[PTB85] R. Paige, R. E. Tarjan, and R. Bonic, "A linear time solution to the single function coarsest partition problem", In *Theoretical Computer Science*, 40:67–84, 1985.

[RTD96] T. Ramalingom, Krishnaiyan Thulasiraman and Anindya Das, "Context Independent Unique Sequences Generation for Protocol Testing", In *INFOCOM* 1996: 1141-1148

[RH01]S. Rayadurgam and M. P. Heimdahl, "Coverage based testcase 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.

[SBC97] B. Sarikaya, G. Bochmann, and E. Cerny, "A Test Design Methodology for

Protocol Testing", In *IEEE Trans. Software Eng.*, vol. 13, no. 5, pp. 518-531, May 1987.

[SHO83] M.L. Shooman, "Software Engineering", New York: McGraw-Hill Book Company, 1983.

[U92] H. Ural, "Formal Methods for Test Sequence Generation", In *IEEE Trans. Comm.*, vol. 39, no. 4, pp. 514-523, 1992.

[U89] H. Ural, "A Test Derivation Method for Protocol Conformance Testing", In *Proc. Sixth Int'l Conf. Protocol Specification, Testing, and Verification*, pp. 347-358, 1989.

[UY91] H. Ural and B. Yang, "A Test Sequence Selection Method for Protocol Testing", In *IEEE Trans. Comm.*, vol. 39, no. 4, pp. 514-523, Apr. 1991.

Appendix A

```
using System;
using System.Collections;
namespace OLMA
{
        public class partition
        {
                public point partpoint;
                public int[,,] ps;
                public ArrayList es;
                public int pi;
                public int psi;
                public bool marked=false;
                public partition(int[,,] pointset, ArrayList edgeset, int pindex, int
psubindex)
                {
                        ps=pointset;
                        es=edgeset;
                        pi=pindex;
                        psi=psubindex;
                }
                public partition()
                {
                        ps=OLMA.initPS(1);
                }
        }
        public class OLMA
        {
                public static int nov=2;
                public static int nop=99;
                public static t1 action1=new t1(0,0,0);        //
                public static t2 action2=new t2(0,0,0);        //
                public static t3 action3=new t3(1,0,0);        //              (M==1)
                public static t4 action4=new t4(0,0,0);        //
                public static t5 action5=new t5(0,0,0);        //
                public static t6 action6=new t6(0,0,0);        //      (P=0)&(M=0) /
                public static t7 action7=new t7(0,0,1);        //      (P>0)&(M==0)/M:=1
                public static Transition[] actionset=new Transition[7]
{action1,action2,action3,action4,action5,action6,action7};

                public static ArrayList initP(int[,,] ps, int[][][] pesd)
                {
                        ArrayList partitions=new ArrayList();
                        for ( int i=0; i<pesd.GetLength(0); i++ )
                        {
```

```
                        ArrayList es=new ArrayList();
                        for ( int j=0; j<pesd[i][0].Length; j++ )
                        {
                                Edge e=new
Edge(pesd[i][0][j]-1,pesd[i][1][j],pesd[i][1][j]);  // j actionset index, pesd[i][1][j]
                                es.Add(e);
                        }
                        partition p=new partition(ps, es, i, i); // int[,,] pointset,
                        partitions.Add(p);
                }
                ((partition)partitions[0]).partpoint=new point(0,0,0);
                ((partition)partitions[1]).partpoint=new point(0,0,0);
                ((partition)partitions[2]).partpoint=new point(0,0,0);
                return partitions;
        }
        public static int[,,] initPS(int val) //val could be 1 or 0
        {
                int[,,] ps=new int[2,8,8];
                for ( int a=0; a<ps.GetLength(0); a++ )
                        for ( int b=0; b<ps.GetLength(1); b++ )
                                for ( int c=0; c<ps.GetLength(2); c++ )
                                {
                                        ps[a,b,c]=val;
                                }
                return ps;
        }

        public static int[,,] initPS()
        {
                int[,,] ps=new int[2,8,8];
                return ps;
        }

        public static void Main()
        {

                DateTime begintime = DateTime.Now;
                int[][][] pesd=new int[60][][];//partition edge set desc, describ what
edges a partition has,
                                                                                // for
ex:s0 has edge 3,4,5,6,7

                pesd[0]=new int[2][];
                pesd[0][0]=new int[] {3,4,5,6,7};
                pesd[0][1]=new int[] {2,2,6,0,0};
```

```
                    pesd[1]=new int[2][];
                    pesd[1][0]=new int[] {1};
                    pesd[1][1]=new int[] {0};

                    pesd[2]=new int[2][];
                    pesd[2][0]=new int[] {2};
                    pesd[2][1]=new int[] {1};

                    int[,,] ps=initPS(1);
                    // initialization part >>
                    ArrayList partitions=initP(ps,pesd); //new ArrayList();//keep
blockpt, initially it has some blocks, then new blocks will be put into
                    Stack myStack = new Stack();              //stack: keep
blockponit, blocks to search from
                    Queue myQ = new Queue();                  //queue: keep
blockponit, unstable block to be split in a FIFO order
                    for ( int i=0; i<partitions.Count; i++)
                    {
                        partition temp=(partition)partitions[i];
                        Console.WriteLine( "partition + {0:G} {1} ", temp.pi,
temp.psi);
                    }
                    mark((partition)partitions[0]);
        //mark a blockpoint
                    myStack.Push(partitions[0]);
                    Console.WriteLine( "search: Stack + {0:G}
{1:G}",((partition)partitions[0]).pi,((partition)partitions[0]).psi);
                    ArrayList edges=new ArrayList();
                    search:
                        while(myStack.Count!=0)
                        {
                            partition B=(partition)myStack.Pop();
                            if (B.psi==30)
                                B.psi=30;
                            Console.WriteLine( "search:  Stack - {0:G}
{1:G}",B.pi, B.psi);
                            point tempendp=new point(); int[,,]
tempps=OLMA.initPS(0);
                            foreach ( Edge e in B.es ) //tempes1)
                            {
                                ArrayList blocksap = new ArrayList();
//keep all partitions that a(B) reach;
                                int[,,] D=OLMA.initPS(); //new int[2,8,8];
                                point pc=(e.t).action(B.partpoint);
                                partition
Cbp=(partition)partitions[e.EndBlockSubIndex];
```

```
                                        if (pc.M !=-1 && Cbp.ps[pc.M, pc.P,
pc.R]==1 )
                                                // && a(p) intersect C is not empty
                                        {
                                                if ( !(e.t).rev_ints_eq_bg(B.ps,Cbp.ps)
&& !myQ.Contains(B))
                                                {
                                                        myQ.Enqueue(B);
                                                        Console.WriteLine( "search:
        myQ + {0:G} {1:G} ",B.pi, B.psi);
                                                }
                                                if ( !Cbp.marked)
                                                {
                                                        point
pointC=(e.t).action(B.partpoint);
                                                        Cbp.partpoint=pointC;
                                                        mark(Cbp);
                                                        myStack.Push(Cbp);
                                                        Console.WriteLine( "search:
Stack + {0:G} {1:G}",Cbp.pi, Cbp.psi);
                                                }
                                        }
                                        else
                                        {
                                                e.tobedeleted=true;
                                        }
                                }
                                for ( int i=(B.es).Count-1; i>=0; i--) // Edge e in B.es )
//tempes1)
                                {
                                        if ( ((Edge)(B.es)[i]).tobedeleted==true )
                                                (B.es).RemoveAt(i);
                                }
                        }

                        //split:
                        while(myQ.Count != 0)
                        {
                                partition B=(partition)myQ.Dequeue();
                                if (B.pi==10)
                                        B.pi=10;
                                Console.WriteLine( "split :          myQ - {0:G} {1:G}
",B.pi, B.psi) ; //(Bp.bpblock).blockindex,(Bp.bpblock).blocksubindex);
                                partition B1=new partition();
                                B1.ps=(int[,,])(B.ps).Clone();
```

```
                        point b1p=findp(B1.ps); B1.pi=B.pi; B1.psi=B.psi;
B1.partpoint=b1p; B1.marked=B.marked; B1.es=B.es;
                        if ( b1p.M==-1 ) continue;
                        ArrayList tempes=esclone(B.es);
                        foreach ( Edge e in tempes)   // compute B' := { q<-B':
blocks(a(q))=blocks(a(p)) };
                        {
                                point endp=(e.t).action(B.partpoint);
                                if ( endp.M!=-1 &&
intersect(endp,((partition)partitions[e.EndBlockSubIndex]).ps) )

        B1.ps=(e.t).buildB1(B1.ps,((partition)(partitions[e.EndBlockSubIndex])).ps);
                        }
                        partition B2=new partition();
                        B2.ps=blockminus(B,B1);                 //B" := B-B'
                        point b2p=findp(B2.ps);
                        if ( b2p.M==-1 ) continue;
                        int B2subindex=partitions.Count;        // get current
next partition index
                        B2.pi=B.pi; B2.psi=B2subindex; B2.marked=false;
B2.partpoint=b2p; B2.es=esclone(B.es);
                        partitions.Add(B2);
                        Console.WriteLine( "split :
        partition + {0:G} {1} ",B2.pi, B2.psi);
                        B.ps=(int[,,])(B1.ps).Clone();
        // B=B'
                        B.partpoint=findp(B.ps);

                        ArrayList tempedges=new ArrayList();
                        int pnn=partitions.Count;
                        for (int pn=0; pn<pnn; pn++) //foreach ( partition part in
partitions )
                        {
                                partition Cpc=new partition();
                                if ( ((partition)partitions[pn]).marked==true )
                                        Cpc=(partition)partitions[pn];
                                if ( Cpc.marked==false)
                                        continue;
                                int nE=((ArrayList)Cpc.es).Count;
                                for ( int i=nE-1; i>=0; i--) //Edge Cpce in Cpc.es)
                                {
                                        if
(((Edge)Cpc.es[i]).EndBlockSubIndex!=B.psi)
                                                continue;
                                        point pc=Cpc.partpoint;
```

```
                                        int[,,]
pc2b=(((Edge)Cpc.es[i]).t).action_fwd(Cpc.ps);
                                        int[,,] C2B=intersect(pc2b,B.ps);
                                        point C2Bp=findp(C2B);
                                        int[,,] tempps=OLMA.initPS(0); point
tempendp=new point();
                                        if ( C2Bp.M!=-1 ) // if ( a(q)^B =0 ) delete
edge <C,q> -> <B,p>
                                        {
                                                if ( !myQ.Contains(Cpc)
&& !(((Edge)Cpc.es[i]).t).rev_ints_eq_bg(Cpc.ps,B.ps) )
                                                {
                                                        myQ.Enqueue(Cpc);
        //
                                                        Console.WriteLine( "split :
        myQ + {0:G} {1:G} ",Cpc.pi,Cpc.psi);
                                                }
                                        }
                                        else
                                        {
                                                /*
                tempps=intersect( ((Edge)Cpc.es[i]).t.action_fwd(B.ps), B.ps );

                if (Cpc.pi==2 && ((Edge)Cpc.es[i]).EndBlockSubIndex==1 )

                        tempendp.M=-1;

                tempendp=findp(tempps);

                if ( tempendp.M==-1 )

                {

                        if (Cpc.pi==2 && ((Edge)Cpc.es[i]).EndBlockSubIndex==1 )

                                tempendp.M=-1;*/
                                                Console.WriteLine( "split :
        Edge - to {0:G} {1:G} {2:G}",
((Edge)Cpc.es[i]).ActionIndex,((Edge)Cpc.es[i]).EndBlockIndex,((Edge)Cpc.es[i]).EndB
lockSubIndex);
                                                ((Edge)Cpc.es[i]).tobedeleted=true;
                                                //the edge from Cpc to B after splitted
is not valid any more, (Cpc.es).Remove(((Edge)Cpc.es[i]));
                                                //}
                                        }
```

```
                                        int[,,]
pc2b2=(((Edge)Cpc.es[i]).t).action_fwd(Cpc.ps);
                                        int[,,] C2B2=intersect(pc2b2,B2.ps);
                                        point C2B2p=findp(C2B2);
                                        if ( C2B2p.M!=-1 )  // the edge from Cpc to
B2 need to be added
                                        {
                                                if (!B2.marked )
                                                {
                                                        B2.partpoint=findp(B2.ps);
                                                        mark(B2);
                                                        myStack.Push(B2);
                                                        Console.WriteLine( "split :
        Stack + {0:G} {1:G}",B2.pi, B2.psi);
                                                }
                                                if (Cpc.pi==2 &&
((Edge)Cpc.es[i]).EndBlockSubIndex==1 )
                                                        Cpc.pi=2;
                                                Edge newedge=new
Edge(((Edge)Cpc.es[i]).ActionIndex,B2.pi,B2.psi);
                                                (Cpc.es).Add(newedge);
                                                Console.WriteLine( "split :
        Edge + to {0:G} {1:G} {2:G}", newedge.ActionIndex,
newedge.EndBlockIndex,newedge.EndBlockSubIndex);
                                                if (!myQ.Contains(Cpc))
                                                {
                                                        if
( !(((Edge)Cpc.es[i]).t).rev_ints_eq_bg(Cpc.ps,B2.ps) )
                                                        {
                                                                myQ.Enqueue(Cpc);
        //
        Console.WriteLine( "split :          myQ + {0:G} {1:G} ",Cpc.pi,Cpc.psi);
                                                        }
                                                }

                                        }
                                        if ( ((Edge)Cpc.es[i]).tobedeleted==true )
                                                (Cpc.es).Remove(((Edge)Cpc.es[i]));
                                }
                        }
                        int edgesCount=(B.es).Count ; //edges.Count;
                        for( int m=edgesCount-1; m>=0; m--)
                        {
                                int[,,]
tempendps=((Edge)((B.es)[m])).t.action_fwd(B.ps);
```

```
                                        int[,,]
tempps=intersect( ((partition)partitions[((Edge)((B.es)[m])).EndBlockSubIndex]).ps,
tempendps);
                                        point tempendp=findp(tempps);
                                        if ( tempendp.M==-1 )
                                        {
                                                Console.WriteLine( "split :
        Edge - to {0:G} {1:G} {2:G}",
((Edge)((B.es)[m])).ActionIndex,((Edge)((B.es)[m])).EndBlockIndex,((Edge)((B.es)[m])
).EndBlockSubIndex);
                                                B.es.RemoveAt(m);
                                        }
                                }
                        }
                        if (myStack.Count != 0) goto search;
                        int b =13;
                        DateTime b4genpath = DateTime.Now;
                        System.TimeSpan diff1 = b4genpath.Subtract(begintime);
                        string difftostring1=diff1.ToString();
                        Console.WriteLine( "Time B4 genpath: {0:G}", difftostring1);
                        genpath(partitions);
                        DateTime endtime = DateTime.Now;
                        System.TimeSpan diff = endtime.Subtract(begintime);
                        string difftostring=diff.ToString();
                        Console.WriteLine( "Total Time: {0:G}", difftostring);
                        DateTime timegenpath = DateTime.Now;
                        System.TimeSpan diff2 = timegenpath.Subtract(b4genpath);
                        string difftostring2=diff2.ToString();
                        Console.WriteLine( "Time for genpath: {0:G}", difftostring2);
                        int tempa =13;
                }

                public static void genpath(ArrayList partitions)
                {
                        ArrayList sfp=new ArrayList();
                        ArrayList efp=new ArrayList();
                        // marked means visited, marked true means the blcok(state) has
been visited
                        for (int i=0; i<partitions.Count; i++)
                        {
                                ((partition)partitions[i]).marked=false;
                                ArrayList spath=new ArrayList();
                                sfp.Add(spath);
                                // tobedeleted means visited,tobedeleted true means the edge
has been visited
                                for (int j=0; j<((partition)partitions[i]).es.Count; j++)
```

```
                        {

        ((Edge)((((partition)partitions[i]).es)[j])).tobedeleted=false;
                                ArrayList epath=new ArrayList();
                        }
                }
                Queue stateQ = new Queue(); // contains a state ( partition psi) just
found as
                //unvisited one, and it will be checked in the next round
                // assume partion[0] contains the initial state, ( need to be checked )

                stateQ.Enqueue(0);
                //int offset=0;
                while (stateQ.Count!=0)
                {
                        int i=(int)stateQ.Dequeue();
                        partition cS=(partition)partitions[i];   // cS current State
being checked
                        for (int j=0; j<((((partition)partitions[i]).es).Count; j++)
                        {
                                Edge cE=(Edge)((((partition)partitions[i]).es)[j]);
        // cE curren Edge being checked
                                if ( cE.tobedeleted==false )
                                {
                                        cE.tobedeleted=true;
                                        stateQ.Enqueue(cE.EndBlockSubIndex);
                                        //generate edge path
                                        ArrayList
path=(ArrayList)((ArrayList)sfp[i]).Clone();
                                        if
( ((partition)partitions[cE.EndBlockSubIndex]).marked==false ) // unvisited
                                        {

        ((partition)partitions[cE.EndBlockSubIndex]).marked=true;
                                                //generate state path

        sfp.RemoveAt(cE.EndBlockSubIndex);

        sfp.Insert(cE.EndBlockSubIndex,path);

        stateQ.Enqueue(cE.EndBlockSubIndex);
                                        }
                                        path.Add(cE.EndBlockSubIndex);
                                        efp.Add(path);
                                }
                        }
```

```
        }
        for (int i=0; i<efp.Count; i++)
        {
                int last=((ArrayList)efp[i]).Count-1;
                int k=(int)((ArrayList)efp[i])[last];
                int j;
                if ( last >= 2)
                        j=(int)((ArrayList)efp[i])[last-1];
                else
                        j=0;
                Console.WriteLine( "{0:G} {1:G} {2:G}",i, j, k);
        }
        int tempabc=111;
}

public static point findp(int[,,] ps)
{
        point p=new point();
        p.M=-1;
        p.P=-1;
        p.R=-1;
        for (int i=0;i<ps.GetLength(0);i++)
                for (int j=0; j<ps.GetLength(1);j++)
                        for (int k=0;k<ps.GetLength(2);k++)
                        {
                                if ( ps[i,j,k]==1 )
                                {
                                        p.M=i; p.P=j; p.R=k;
                                        return p;
                                }
                        }
        return p;
}
public static int[,,] intersect(int[,,] A, int[,,] B)
{
        int[,,] AB=OLMA.initPS();
        for (int i=0;i<AB.GetLength(0);i++)
                for (int j=0; j<AB.GetLength(1);j++)
                        for (int k=0;k<AB.GetLength(2);k++)
                        {
                                if ( A[i,j,k]==1 && B[i,j,k]==1 )
                                        AB[i,j,k]=1;
                        }
        return AB;
}
```

```
public static bool intersect(point p, int[,,] C)
{
        if ( p.M!=-1 &&  C[p.M,p.P,p.R] == 1 )
                return true;
        else
                return false;
}

public static ArrayList esclone(ArrayList edgeset)
{
        ArrayList newes=new ArrayList();
        for (int i=0; i<edgeset.Count; i++)
        {
                Edge ne=new Edge(((Edge)edgeset[i]).ActionIndex,
((Edge)edgeset[i]).EndBlockIndex, ((Edge)edgeset[i]).EndBlockSubIndex);
                newes.Add(ne);
        }
        return newes;
}
public static int[,,] blockminus(partition C, partition B)
{
        int[,,] ps=(int[,,])(C.ps).Clone();
        for (int i=0;i<ps.GetLength(0);i++)
                for (int j=0; j<ps.GetLength(1);j++)
                        for (int k=0;k<ps.GetLength(2);k++)
                        {
                                if ( (B.ps)[i,j,k]==1 )
                                        ps[i,j,k]=0;
                        }
        return ps;
}               // >>>>>>>>>>>
public static int[,,] psminus(int[,,] C, int[,,] B)
{
        int[,,] A=OLMA.initPS();
        A=(int[,,])C.Clone();
        for (int i=0;i<A.GetLength(0);i++)
                for (int j=0; j<A.GetLength(1);j++)
                        for (int k=0;k<A.GetLength(2);k++)
                        {
                                if ( B[i,j,k]==1 )
                                        A[i,j,k]=0;
                        }
        return A;
}               // >>>>>>>>>>>
public static void mark(partition bp)
```

```
            {
                bp.marked=true;
            }
        }

        public class point
        {
            public int M, P, R;

            public point(int Pm, int Pp, int Pr)
            {
                M = Pm;
                P = Pp;
                R = Pr;
            }
            public point()
            {
                M = -1;
                P = -1;
                R = -1;
            }
        }

        public class  Edge
        {
            public int StartBlockIndex;
            public int EndBlockIndex;
            public int StartBlockSubIndex;
            public int EndBlockSubIndex;
            public int ActionIndex;
            public Transition t;
            public bool tobedeleted;
            public int p1;
            public int p2;
            public int p3;
            public Edge(int ai, int ebi, int ebsi) //, int parameter1, int parameter2, int
parameter3)
            {
                EndBlockIndex=ebi;
                EndBlockSubIndex=ebsi;
                t=OLMA.actionset[ai];
                ActionIndex=ai;
                tobedeleted=false;
            }
        }
```

```
public abstract class Transition
{
        public Transition()
        {
                // Code to initialize the class goes here.
        }

        public int StartBlockIndex;
        public int EndBlockIndex;
        public int StartBlockSubIndex;
        public int EndBlockSubIndex;
        abstract public point action(point begin); //because return point could be
null, use object instead of point
        abstract public int[,,] action_fwd(int[,,] beginps);
        abstract public bool rev_ints_eq_bg(int[,,]beginps,int[,,] endps);
        abstract public bool rev_ints_eq_null(int[,,]beginps,int[,,] endps);
        public int[,,] buildB1( int[,,] beginps, int[,,]endps)
        {
                int[,,] B1ps=OLMA.initPS();
                for ( int i=0; i<B1ps.GetLength(0); i++)
                        for (int j=0; j<B1ps.GetLength(1); j++)
                                for (int k=0; k<B1ps.GetLength(2); k++)
                                        if ( beginps[i,j,k]==1 && (action(new
point(i,j,k))).M!=-1 && endps[(action(new point(i,j,k))).M,(action(new
point(i,j,k))).P,(action(new point(i,j,k))).R]==1)
                                                B1ps[i,j,k]=1;
                return B1ps;
        }
}

public class t1:Transition
{
        public int p1;
        public int p2;
        public int p3;
        public t1(int par1, int par2, int par3)
        {
                p1=par1; p2=par2; p3=par3;
        }
        override public point action(point begin)
        {
                point end=new point();
                end.M=p1; // p1=0;
                end.P=begin.R;
                end.R=begin.R;
                return end;
```

```
        }
        override public int[,,] action_fwd(int[,,] beginps)
        {
            int[,,] endps=OLMA.initPS();
            for ( int i=0; i<beginps.GetLength(0); i++)
                for (int j=0; j<beginps.GetLength(1); j++)
                    for (int k=0; k<beginps.GetLength(2); k++)
                        if ( beginps[i,j,k]==1 )
                            endps[p1,k,k]=1; // p1=0
            return endps;
        }
        override public bool rev_ints_eq_bg(int[,,]beginps,int[,,] endps)
        {
            for (int i=0; i<beginps.GetLength(0); i++)
                for (int j=0; j<beginps.GetLength(1); j++)
                    for (int k=0; k<beginps.GetLength(2); k++)
                    {
                        if ( ( beginps[i,j,k]==1 ) &&
endps[p1,k,k]!=1)
                            return false;
                        if ( ( beginps[i,j,k]==0 ) &&
endps[p1,k,k]==1)
                            return false;
                    }
            return true;
        }
        override public bool rev_ints_eq_null(int[,,]beginps,int[,,] endps)
        {
            for (int i=0; i<beginps.GetLength(0); i++)
                for (int j=0; j<beginps.GetLength(1); j++)
                    for (int k=0; k<beginps.GetLength(2); k++)
                        if ( beginps[i,j,k]==1 && endps[p1,k,k]==1)
                            return false;
            return true;
        }
    }

    public class t2:Transition
    {
        public int p1;
        public int p2;
        public int p3;
        public t2(int par1, int par2, int par3)
        {
            p1=par1; p2=par2; p3=par3;
        }
```

```
        override public point action(point begin)
        {
                point end=new point();
                end.M=begin.M;
                end.P=begin.P;
                end.R=begin.R;
                return end;
        }
        override public int[,,] action_fwd(int[,,] beginps)
        {
                int[,,] endps=(int[,,])beginps.Clone();
                return endps;
        }
        override public bool rev_ints_eq_bg(int[,,]beginps,int[,,] endps)
        {
                for (int i=0; i<beginps.GetLength(0); i++)
                        for (int j=0; j<beginps.GetLength(1); j++)
                                for (int k=0; k<beginps.GetLength(2); k++)
                                {
                                        if ( beginps[i,j,k]==1 && endps[i,j,k]!=1)
                                                return false;
                                        if ( beginps[i,j,k]!=1 && endps[i,j,k]==1)
                                                return false;
                                }
                return true;
        }
        override public bool rev_ints_eq_null(int[,,]beginps,int[,,] endps)
        {
                for (int i=0; i<beginps.GetLength(0); i++)
                        for (int j=0; j<beginps.GetLength(1); j++)
                                for (int k=0; k<beginps.GetLength(2); k++)
                                        if ( beginps[i,j,k]==1 && endps[i,j,k]==1)
                                                return false;
                return true;
        }
}

public class t3:Transition
{
        public int p1;
        public int p2;
        public int p3;
        public t3(int par1, int par2, int par3)
        {
                p1=par1; p2=par2; p3=par3;
        }
```

```
        override public point action(point begin)
        {
                point end=new point();
                if (begin.M==p1)  //M=1, p1=1
                {
                        end.M=begin.M;
                        end.P=begin.P;
                        end.R=begin.R;
                }
                return end;
        }

        override public int[,,] action_fwd(int[,,] beginps)
        {
                int[,,] endps=OLMA.initPS();
                for (int j=0; j<beginps.GetLength(1);j++)
                        for (int k=0;k<beginps.GetLength(2);k++)
                                endps[p1,j,k]=beginps[p1,j,k];
                return endps;
        }
        override public bool rev_ints_eq_bg(int[,,]beginps,int[,,] endps)
        {
                        for (int i=0; i<beginps.GetLength(0); i++)
                                for (int j=0; j<beginps.GetLength(1); j++)
                                        for (int k=0; k<beginps.GetLength(2); k++)
                                        {
                                                if ( i!=p1 && beginps[i,j,k]==1 )
                                                        return false;
                                                if ( beginps[p1,j,k]==1 &&
endps[p1,j,k]!=1)
                                                        return false;
                                        }
                        return true;
        }
        override public bool rev_ints_eq_null(int[,,]beginps,int[,,] endps)
        {
                for (int j=0; j<beginps.GetLength(1); j++)
                        for (int k=0; k<beginps.GetLength(2); k++)
                                if ( beginps[p1,j,k]==1 && endps[p1,j,k]==1)
                                        return false;
                return true;
        }
}

public class t4:Transition
{
```

```
        public int p1;
        public int p2;
        public int p3;
        public t4(int par1, int par2, int par3)
        {
                p1=par1; p2=par2; p3=par3;
        }
        override public point action(point begin)
        {
                point end=new point();
                end.M=begin.M;
                end.P=begin.P;
                end.R=begin.R;
                return end;
        }
        override public int[,,] action_fwd(int[,,] beginps)
        {
                int[,,] endps=(int[,,])beginps.Clone();
                return endps;
        }
        override public bool rev_ints_eq_bg(int[,,]beginps,int[,,] endps)
        {
                for (int i=0; i<beginps.GetLength(0); i++)
                        for (int j=0; j<beginps.GetLength(1); j++)
                                for (int k=0; k<beginps.GetLength(2); k++)
                                {
                                        if ( beginps[i,j,k]==1 && endps[i,j,k]!=1)
                                                return false;
                                        if ( beginps[i,j,k]!=1 && endps[i,j,k]==1)
                                                return false;
                                }
                return true;
        }
        override public bool rev_ints_eq_null(int[,,]beginps,int[,,] endps)
        {
                for (int i=0; i<beginps.GetLength(0); i++)
                        for (int j=0; j<beginps.GetLength(1); j++)
                                for (int k=0; k<beginps.GetLength(2); k++)
                                        if ( beginps[i,j,k]==1 && endps[i,j,k]==1)
                                                return false;
                return true;
        }
}

public class t5:Transition
{
```

```
public int p1;
public int p2;
public int p3;
public t5(int par1, int par2, int par3)
{
	p1=par1; p2=par2; p3=par3;
}
override public point action(point begin)
{
	point end=new point();
	end.M=begin.M;
	end.P=begin.P;
	end.R=Math.Min(begin.R+1,7); // ??????????? 7 need to be
replaced
	return end;
}
override public int[,,] action_fwd(int[,,] beginps)
{
	int[,,] endps=OLMA.initPS();
	for (int i=0; i<beginps.GetLength(0); i++)
		for (int j=0; j<beginps.GetLength(1); j++)
			for (int k=0; k<beginps.GetLength(2); k++)
				if ( beginps[i,j,k]==1 )
				{
					int
r=Math.Min(k+1,beginps.GetLength(2)-1); // ????????? 7 need to be ..
					endps[i,j,r]=1;
				}
	return endps;
}
override public bool rev_ints_eq_bg(int[,,]beginps,int[,,] endps)
{
	for (int i=0; i<beginps.GetLength(0); i++)
		for (int j=0; j<beginps.GetLength(1); j++)
			for (int k=0; k<beginps.GetLength(2)-1; k++)
			{
				if ( k<(beginps.GetLength(2)-1) &&
beginps[i,j,k]==1 && endps[i,j,k+1]!=1)
					return false;
				if ( beginps[i,j,beginps.GetLength(2)-2]!=1
&& beginps[i,j,beginps.GetLength(2)-1]==1 && endps[i,j,beginps.GetLength(2)]!=1 )
					return false;
			}
	return true;
}
override public bool rev_ints_eq_null(int[,,]beginps,int[,,] endps)
```

```
            {
                for (int i=0; i<beginps.GetLength(0); i++)
                    for (int j=0; j<beginps.GetLength(1); j++)
                        for (int k=0; k<beginps.GetLength(2); k++)
                        {
                            if ( k<(beginps.GetLength(2)-1) &&
beginps[i,j,k]==1 && endps[i,j,k+1]==1)
                                return false;
                            if ( beginps[i,j,beginps.GetLength(2)-2]!=1
&& beginps[i,j,beginps.GetLength(2)-1]==1 && endps[i,j,beginps.GetLength(2)]==1 )
                                return false;
                        }
                return true;
            }
        }

        public class t6:Transition
        {
            public int p1;
            public int p2;
            public int p3;
            public t6(int par1, int par2, int par3)
            {
                p1=par1; p2=par2; p3=par3;
            }
            override public point action(point begin)
            {
                point end=new point();
                if (begin.P==p2 && begin.M==p1)  //p1=0 M=0 ;;;   p2=0 P=0
                {
                    end.M=begin.M;
                    end.P=begin.P;
                    end.R=begin.R;
                }
                return end;
            }
            override public int[,,] action_fwd(int[,,] beginps)
            {
                int[,,] endps=OLMA.initPS();
                for ( int k=0; k<beginps.GetLength(2); k++ )
                    if ( beginps[p1,p2,k]==1 )
                        endps[p1,p2,k]=1;
                return endps;
            }
            override public bool rev_ints_eq_bg(int[,,]beginps,int[,,] endps)
            {
```

```
                for (int k=0; k<beginps.GetLength(2); k++)
                        if ( beginps[p1,p2,k]==1 && endps[p1,p2,k]!=1)
                                return false;
                for (int i=0; i<beginps.GetLength(0); i++)
                        for (int j=0; j<beginps.GetLength(1); j++)
                                for (int k=0; k<beginps.GetLength(2); k++)
                                        if ( ( i!=p1 || j!=p2 ) && beginps[i,j,k]==1 )
                                                return false;
                return true;
        }
        override public bool rev_ints_eq_null(int[,,]beginps,int[,,] endps)
        {
                for (int k=0; k<beginps.GetLength(2); k++)
                        if ( beginps[p1,p2,k]==1 && endps[p1,p2,k]==1)
                                return false;
                return true;
        }
}

public class t7:Transition
{
        public int p1;
        public int p2;
        public int p3;
        public t7(int par1, int par2, int par3)
        {
                p1=par1; p2=par2; p3=par3;
        }
        override public point action(point begin)
        {
                point end=new point();
               if (begin.P>p1 && begin.M==p2) // P>0: p1=0,  M=0: p2=0,   M>1:
p3=1
                {
                        end.M=p3;
                        end.P=begin.P;
                        end.R=begin.R;
                }
                return end;
        }
        override public int[,,] action_fwd(int[,,] beginps)
        {
                int[,,] endps=OLMA.initPS();
                for ( int j=0; j<beginps.GetLength(1); j++)
                        for ( int k=0; k<beginps.GetLength(2); k++)
                                if ( j>p1 && beginps[0,j,k]==1 )
```

```
                                        endps[p3,j,k]=1;
                return endps;
            }
            override public bool rev_ints_eq_bg(int[,,]beginps,int[,,] endps)
            {
                for (int i=0; i<beginps.GetLength(0); i++)
                    for (int j=0; j<beginps.GetLength(1); j++)
                        for (int k=0; k<beginps.GetLength(2); k++)
                        {
                            if ( j>p1 && i==p2 && beginps[i,j,k]==1
&& endps[p3,j,k]!=1 )
                                return false;
                            if ( (j<=p1 || i!=p2) && beginps[i,j,k]==1 )
                                return false;
                        }
                return true;
            }
            override public bool rev_ints_eq_null(int[,,]beginps,int[,,] endps)
            {
                for (int i=0; i<beginps.GetLength(0); i++)
                    for (int j=0; j<beginps.GetLength(1); j++)
                        for (int k=0; k<beginps.GetLength(2); k++)
                            if ( j>p1 && beginps[i,j,k]==1 )
                                return false;
                return true;
            }
        }

}
```

Vita Auctoris

Yongdong Tan was born in 1972 in Beijing, China. He graduated from Zhejiang University, Hangzhou, China, 1995, where he received a Bachelor's degree in Electronic Engineering. He is currently a Master's candidate in the School of Computer Science at the University of Windsor and expects to graduate in summer, 2005.