國立臺灣師範大學

資訊工程研究所碩士論文

指導教授：黃冠寰 博士

對動態測試產生的同步序列之正規驗證的研究

The Formal Verification of

SYN-Sequences Generated in Dynamic Testing

研究生：李彥佑 撰

中華民國九十八年六月

摘 要

並行程式面臨非確定行為的問題：在相同的輸入下重複執行多次可能產生不同的同步事件序列和執行結果。這使得並行程式難以進行測試。對非確定行為的並行程式進行動態測試的常見方法是將受測的並行程式重複執行許多次。每次執行受測的並行程式會產生一個同步序列（SYN-sequences）。除了執行結果之外，我們通常還必須驗證搜集到的 SYN-sequences 以確保並行程式的行為滿足我們所要求的規格或需求。這篇論文的目標即在於驗證 SYN-sequences。

我們提出了對 SYN-sequence 進行正規驗證的方法。首先，我們研究用以驗證 SYN-sequences 的規格或需求的表達法。根據 SYN-sequence 表達了同步事件間偏序關係的事實，我們採用 temporal logic 來表示規格。接下來的目標就是如何有效率的根據規格來驗證 SYN-sequences。因此，我們發展了三種不同的方法來將 SYN-sequence 轉換成模型，然後就可以利用 temporal logic 做正規驗證。我們也探討了如何用適當的表達法表達規格，以及如何選擇合適的模型。實驗結果驗證了對 SYN-sequence 進行有效率的驗證是可行的。

關鍵字：dynamic testing, reachability testing, formal verification, model checking, SYN-sequence

ABSTRACT

Concurrent programs exhibit nondeterministic behavior in that multiple executions thereof with the same input might produce different sequences of synchronization events and different results. It makes the concurrent programs difficult to test. Dynamic testing of a concurrent program with nondeterministic behavior usually involves multiple executions of the target concurrent program. Each execution of the target concurrent program produces a synchronization sequences or *SYN-sequences*. In addition to the execution results, we usually have to verify the collected SYN-sequences to make sure that the behavior of the concurrent program meets its specification or requirement. In this thesis, we target on how to verify the SYN-sequences.

We propose a scheme for the formal verification of the SYN-sequence. First, we study how to express the specification or requirement which can be used to verify SYN-sequences. According to the fact that a SYN-sequence represents a partial order relationship of synchronization events, we employ the temporal logic to express the specification. Then, we target on how to efficiently verify SYN-sequences according to the specification. As a result, we develop three different methods to transfer the target SYN-sequence to models which can then be verified in temporal logic. We also discussed how to express the specifications in suitable expression types and how to choose an appropriate model. The experimental results show that a systematic verification of SYN-sequence is feasible.

Keyword: dynamic testing, reachability testing, formal verification, model checking, SYN-sequence, temporal logics

Acknowledgment

碩士論文能如期完成，要感謝很多人。首先要感謝我的指導教授黃冠寰博士。唸研究所的這兩年，黃冠寰教授幾乎每個禮拜都親自跟每個碩士班學生 meeting，這篇論文能如期完成，黃冠寰教授的指導和督促功不可沒。除了課業和研究，黃冠寰教授也常常分享人生道理和其他方面的經驗，是個關心學生的好老師。同時要感謝系主任黃文吉教授曾給予的關心與協助。

此外要感謝實驗室的其它成員：博士班的哲生大大常在生活和課業上提供建議和協助，挑燈夜戰時還會跑來跟我聊天；博士班程式功力一流的宇程學長在程式設計上給了很多的建議與協助；學弟登碩在嵌入式系統課程的鼎力相助讓我能如期畢業；同學文賢、光耀、Reki、笠德、招政陪我熬過兩年的碩士班歲月，並給了很多建議與協助；學弟啟洋常常幫忙一些瑣碎的雜事，和笠德、光耀三個人常為實驗室帶來歡樂氣氛；還有學弟柏澄、仲翔。也謝謝大家在口試時的協助。

比較特別的是要感謝健康中心的醫護人員：本部的護士美華很關心我，也常常給我精神上的鼓勵；還有辛苦的麗娟、護士瓊雅、麗珠，藥劑師怡君、分部藥劑師，以及曾經幫我看過診的醫生們。謝謝大家為師大師生的健康所做的努力，能在師大唸書真的是件很棒的事！

最感謝的是我的家人、死黨浩子和蘇主廚。對你們的感謝不是三言兩語可以說得完的。但要特別謝謝你們在畢業前那幾個月，忍受我極端不穩定的情緒，讓我可以無後顧之憂的趕完論文。

TABLE OF CONTENTS

LIST OF TABLES

1. Introduction

Concurrent programming is becoming commonplace in modern computing. A concurrent program contains two or more processes or threads that execute concurrently and work together to perform some task. However, bugs in concurrent programs are difficult to detect. In this chapter, we will first introduce the testing and verification scheme in general; then we will propose the overview of a verification scheme which combines dynamic testing and formal verification. The detail of this scheme will be discussed in the later chapters.

1.1 Nondeterminism of Concurrent Program

Multiple executions of a concurrent program *P* with the same input might produce different results. This so-called *nondeterministic behavior* means that when testing *P* with input *X* which is a sequence of inputs for processes in *P*, a single execution is insufficient to determine the correctness of *P* with *X* [1,2]. Even if *P* has been executed successfully many times with input *X*, it is possible that a future execution of *P* with *X* will produce incorrect results. An execution of a concurrent program exercises a sequence of synchronization events called a *synchronization sequence* (or *SYN-sequence*). Examples of process synchronization include P (acquire) and V (release) primitives applied to a shared semaphore, monitor-entry procedures, sending and receiving of message primitives, and general sharing of memory [3,4]. A concurrent program exhibits nondeterministic behavior because different executions of *P* with the same input *X* might produce distinct SYN-sequences. In this thesis, a *test* of *P* with input *X* refers to a single execution of *P* with input *X* to obtain a SYN-sequence, with checking of the execution result. A *duplicating test* means that different tests of *P* with input *X* produce the same SYN-sequence. *Exhaustive testing* of *P* with input *X* refers to

performing many tests that exercise all feasible SYN-sequences of P with input X. A simple approach to deal with nondeterministic behavior in testing a concurrent program P is to execute P with a fixed input many times in the hope that this will expose faults. This type of testing, called *nondeterministic testing*, is easy to perform, but it is usually very inefficient and has two major problems: (1) some feasible SYN-sequences of P with input X might be executed many times, and (2) some feasible sequences might never be executed [5,6].

Formal verification is another way to verify the correctness of concurrent program. We introduce model checking here among others [7,8]. Model checking involves three parts. The users describe the actual physical system as an *abstract* model $\mathcal{M}$ which is a transition system. The users also have to express the specifications or requirements as formulae ϕ in temporal logic. Then the model checker explore the state space systematically and automatically to decide whether $\mathcal{M}$ satisfies ϕ ($\mathcal{M} \vDash \phi$). If $\mathcal{M}$ does not satisfy ϕ, the model checker will report a counterexample. Unfortunately, models omit lots of real features of a physical system. To use most model checkers, the *users* have to model the system using a specific modeling language. This is labor-intensive and error-prone since one may need to capture all essential states from, e.g. global variables, heap, thread stacks, and register contexts. Moreover, some program behaviors or machine-dependent issues are impossible to be modeled using the specific modeling language of a model checker. For instance, the state of a running program may be stored in the operating system, the hardware, and in the worst case, in a different machine across a network [9,10]. Many researches have focused on abstraction techniques for automatically or semiautomatically producing such models, e.g. Java PathFinder [11] and MAGIC [12,13]. These tools are, however, language-dependent. In this thesis, we proposed an automatic modeling scheme that can transform the SYN-sequence into the

models. Since SYN-sequence can be generated in dynamic testing, no matter what program language it is used, our tool is language-independent.

1.2 Dynamic Testing

Dynamic testing in software engineering refers to examining the physical response of a system to variables that change with time, and is used to test the dynamic behavior of code [14]. Unlike in static analysis or model checking, in dynamic testing the software must actually be compiled and run. This involves working with the software, giving input values, and checking if the output is as expected. Dynamic testing methodologies include unit testing, integration testing, system testing, and acceptance testing. Source-code-level dynamic testing has been considered an important step in the software life cycle (software development process) [15]. Reachability testing is an approach that can perform dynamic testing on concurrent program and exploit different interleaves or SYN-sequences without the static analysis of the target source program [16,17]. Different interleaves of a concurrent program execution usually correspond to different SYN-sequences. The generated SYN-sequences during testing are analyzed to control the subsequent testing to exploit different interleaves so that different SYN-sequences can be produced. Note that only the SYN-sequences would be analyzed rather than the target concurrent program. Give a concurrent program P with an input X, the reachability testing will terminate and every partially-ordered SYN-sequence of P with the input X will be exercised if the feasible SYN-sequences of P with X are finite. However, the reachability testing cannot apply to concurrent program with infinite number of SYN-sequences.

A scheme for performing dynamic testing of concurrent programs with infinite number of SYN-sequences was proposed in [18]. Generally speaking, the reason which causes a concurrent program P to have infinite number of SYN-sequences is because P

is with iterative statements such as busy-waiting loops. A busy-waiting loop is a loop that repeatedly checks where an event occurs, and is usually considered as a single loop containing a small number of statements. This scheme can support the testing of concurrent programs where the synchronization events are located in any type of iterative statement. For simplicity, the synchronization events are model as read and write events on shared memory in this thesis. Modeling all process synchronizations in a parallel program as operations on shared data (or shared memory) [19] was found to be not restrictive, since many communication and synchronization primitives can be reduced to operations on shared data. In particular, message passing can be modeled as communication through a shared port, or mailbox. At least two problems are encountered when applying dynamic testing to concurrent programs that contain busy-waiting loops or iterative statements encounters: (1) the execution of the concurrent program might not stop due to some processes getting stuck in infinite loops or the execution of some processes forming a livelock, and (2) even if no executions of all the processes in a concurrent program get stuck in an infinite loop, the existence of the busy-waiting loop makes the number of feasible SYN-sequences infinite.

The traditional way of handling the first problem in dynamic testing – interrupting the execution of the concurrent program when a predefined maximum execution time is exceeded – has two drawbacks: (1) the size of memory needed to store the event history of the tested program is always prohibitively large since busy-waiting loops usually produce synchronization events continuously when infinite loops or livelocks occur (note that the recorded event history is used to debug the target program when an unexpected execution result is discovered during the testing process), and (2) the cost of the execution after some processes get stuck in infinite loops or form livelocks represents pure overhead. A *dynamic termination decision* scheme has been proposed to

spontaneously stop the execution of tested processes when they get stuck in an infinite loop or form a livelock [20]. Since the halting problem is proven to be undecidable, it is impossible to guarantee that the process will be aware that it is stuck in an infinite loop in any particular situation. Heuristic algorithms were proposed for coping with this.

To address the second problem, the concept of *dynamic effective testing* was introduced [18]. The basic idea is that we do not apply all the possible tests, since the number of feasible SYN-sequences is usually infinite. For a concurrent program P with input X, if the number of possible execution states is finite, we show that the execution of P with X can be represented by a deterministic finite automaton (DFA) F in which the finite sets of states are the possible execution states and the transitions are labeled with synchronization events. We also show that the feasible SYN-sequences of P with X might be infinite even if the execution of P can be represented by a DFA. This makes the exhaustive testing of P with X impossible. Also, it is impossible to traverse all the states in F in a single test of P. The *effective test set* of P with X consists of a finite set of tests ET of P with X. The associated SYN-sequences of ET traverse all the states and transitions in F. The dynamic effective testing of P with input X involves dynamic testing of an effective test set of P with input X. If the execution of concurrent program P with input X causes some of its processes to get stuck in infinite loops or form livelocks, the scheme of dynamic termination decision warns the system that some processes might get stuck in infinite loops when applying dynamic effective testing.

1.3 Verification of SYN-sequence

SYN-sequences are useful data for debugging. However, one may need to verify the *specifications*, i.e. desired properties, manually. There are two major solutions to this: one is to write a procedure for each specification. This is somehow inefficiently. We will discuss this further in 4.2. The other is using formal verification scheme proposed in this

thesis. We translate the SYN-sequence into model in structure of partial order graph or reachability graph. We also express the specifications in temporal logic, according to which structure we use. Then we can use formal verification tools, say NuSMV [21], to explore the model systematically and report whether the specification holds in the SYN-sequence. In case of false, the tool also shows a counterexample. The scheme we proposed is illustrated in Figure 1.1. The structures of our model are discussed in Chapter 3. Chapter 4 tells how to express specifications in our system. The details modeling are illustrated in Chapter 0.

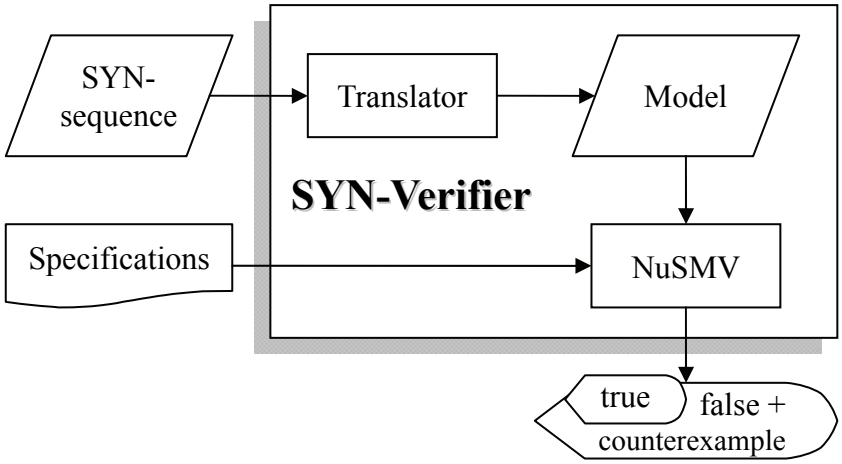


Figure 1.1. The overview of SYN-sequence verifier.

Note that this SYN-sequence verifier can accept a SYN-sequence generated in various dynamic testing schemes. We provide an extendable structure that maintains the essence of a SYN-sequence. Also we provide an API that transforms the SYN-sequence generated in the dynamic effective testing into our SYN-sequence structure.

2. Related Work

We surveyed some papers about dynamic testing and model checking. In this chapter, we abstractly describe these related works and discuss the details of some topics among others.

2.1 Dynamic Testing

One of the schemes for exploiting the nondeterministic behavior of concurrent programs is called a noise maker, which is a tool that forces different legal interleaves for each execution of the test in order to check that the test continues to perform correctly. One proposed scheme transforms a given Java program by inserting calls to a scheduling function at selected points [22]. The scheduling function either does nothing or causes a context switch. Ben-Asher et al. proposed a similar scheme, in which random context switching is performed at accesses to contended shared variables [23]. Edelstein et al. designed a tool to seed the tested program with some primitives at shared memory accesses and events. At run-time, the tool makes random or coverage-based decisions as to whether the seeded primitive is to be executed, which increases the probability of finding concurrent faults [24]. These schemes are less systematic since that they cannot guarantee that all the possible interleaves will be exploited.

Hwang et al. proposed the idea of reachability testing for overcoming the problem of nondeterministic behavior of concurrent programs [17,16]. Assume that S is the SYN-sequence of an execution of P with input X. Reachability testing of P with input X and SYN-sequence S involves the following steps:

1. Use S to derive a set of prefixes of other feasible SYN-sequences of P with input X. Such prefixes (*race variants* of S) are derived by changing the outcome of race

conditions in S. An execution that follows a race variant of S will always exercise a SYN-sequence that is different from S.

2. For each new race variant R derived in step 1, perform a *prefix-based replay* of P with input X and R to execute and collect an additional SYN-sequence for P with input X. The prefix-based replay of P with a race variant R comprises two phases:
   a. Replay phase: control the execution of the program by following the execution order specified in R.
   b. Monitor phase: execute the program without any control and record subsequent synchronization events after replaying R.
3. Repeat steps 1 and 2 for each new SYN-sequence collected in step 2.

Note that R is not a complete SYN-sequence of an execution of P. Thus, after replaying R, we have to record the subsequent synchronization events to obtain a SYN-sequence of an execution of P. This test produces a SYN-sequence. An *entry* protocol and an *exit* protocol must be inserted before and after each synchronization event in the original program, respectively [16,25].

Hwang et al. showed how to perform reachability testing of concurrent programs using read and write operations to shared memory as well as in the semaphore synchronization model [16,25]. Tai described how to apply reachability testing to concurrent programs that employ message passing as a synchronization model [26]. Lei and Tai showed how to derive all race variants for asynchronous message-passing programs for efficient searching and insertion without using prefix-based testing [27]. Carver and Lei presented an algorithm for deriving race variants by constructing a race table [28,29] for a message-passing concurrent program. Hwang et al. described how to apply reachability testing to the exhaustive testing of a client/server database application that exhibits nondeterministic behavior [17]. Carver et al. presented a testing strategy,

called t-way reachability testing, which selectively exercises a subset of SYN-sequences [30]. It has been proven that reachability testing can derive all the possible SYN-sequences of a concurrent program if each execution of that program terminates [17,25]. Such reachability testing terminates when the queue used for storing race variants becomes empty. However, the common assumption in these theoretical proofs is that the concurrent program does not contain busy-waiting loops or iterative statements in which shared objects are repeatedly accessed and updated. In cases where this common assumption does not hold, the reachability testing cannot stop because it continues to exploit some identical program states. The common assumption makes the possible SYN-sequences under a given input finite so that the exhaustive testing is possible.

Sen and Agha presented a testing scheme that uses simultaneous concrete and symbolic executions to consider both inputs and schedules for message-passing distributed programs [31]. In their programming model, the possible inputs and schedules that can be exhibited by the target concurrent program must be finite. Dynamic effective testing can systematically perform dynamic testing on concurrent programs with an infinite number of SYN-sequences [18].

2.2 Model Checking

The first application of temporal logic to program correctness occurred in Pnueli's paper [32]. In this paper, Pnueli, inspired by [33], advocated using future linear temporal logic (LTL) as a logic for the specification of non-terminating programs [34]. In LTL, formulas are constructed from a set $Prop$ of atomic propositions using the usual Boolean connectives as well as the unary temporal connective X (next), and the binary temporal connective U (until). Additional unary temporal connectives F (eventually), and G (always) can be defined in terms of U. All above temporal connectives refer to the

future, in contrast to Kamp's "strict since" operator, which refers to the past [35]. For extensions with past temporal connectives, see [36,37,38]. Pnueli also discussed the satisfiability problem: given a finite-state program P and an LTL formulaϕ, decide if ϕ holds in all traces of P. Pnueli focused on programs, modeled as (labeled) *transition systems* [39]. A transition system $M = (W, W_0, R, V)$ consists of a set W of states that the system can be in, a set $W_0 \subseteq W$ of initial states, a transition relation $R \subseteq W^2$ that indicates the allowable state transitions of the system, and an assignment $V : W \rightarrow 2^{Prop}$ of truth values to the atomic propositions in each state of the system. (A transition system is essentially a Kripke structure [40].) The basic technique for proving the upper bounds of the complexity of LTL reasoning is the tableau technique, which was adapted from dynamic logics [41].

Model checking is first revealed by Edmund Melson Clarke and Ernest Allen Emerson and by J.P. Queille and J. Sifakis [42,43]. The two papers used two different branching-time logics. Clarke and Emerson used CTL (inspired by the branching-time logic UB of [44]), which extends LTL with existential and universal path quantifiers E and A. Queille and Sifakis used a logic introduced by Leslie Lamport [45], which extends propositional logic with the temporal connectives POT (which corresponds to the CTL operator EF) and INEV (which corresponds to the CTL operator AF). Model checking is essentially similar to the satisfiability problem: does a given finite-state program, viewed as a finite transition system, satisfy its given temporal specification. It is an effective verification method for analyzing the correctness of concurrent systems by state-space exploration. Generally speaking, each model checker has a companion modeling language used to describe the concurrent system. Then the state space can be computed automatically from this description. Specifications are represented as logic formulas, usually in temporal logics. Model checkers can systematically explore the

state space of the given system and verify whether the state space satisfies the specifications. If not, it will give a counterexample. Users can then use this information to debug or to modify the system. Clarke and Emerson showed that model checking transition systems of size m with respect to formulas of size n can be done in time polynomial in m and n. This was refined later to $O(mn)$ (even in the presence of fairness constraints, which restrict attention to certain infinite paths in the underlying transition system) [46,47].

It should be noted that the linear complexity of model checking refers to the size of the transition system, rather than the size of the program that gave rise to that system. For sequential circuits, transition-system size is essentially exponential in the size of the description of the circuit (say, in some Hardware Description Language). This is referred to as the "state-explosion problem" [48]. In spite of the state-explosion problem, in the first few years after the publication of the first model-checking papers in 1981, Clarke and his students demonstrated that model checking is a highly successful technique for automated program verification [49,50]. By the late 1980s, symbolic model checking was developed [51,52], and the SMV tool, developed at CMU by McMillan [53], was starting to have an industrial impact [54]. The detailed complexity analysis in [46] inspired a similar detailed analysis of linear time model checking. It was shown in [55] that model checking transition systems of size m with respect to LTL formulas of size n can be done in time $m2^{O(n)}$. (This again was shown using a tableau-based technique.) While the bound here is exponential in n, the argument was that n is typically rather small, and therefore an exponential bound is acceptable.

To the state of the art, a lot of model checker are proposed, e.g. CADP ("Construction and Analysis of Distributed Processes", formerly known as "CAESAR/ALDEBARAN Development Package")) [56], CHESS model checker [57],

ISP (code level verifier for MPI programs) [58], Java Pathfinder (an open source model checker for Java programs) [11], MAGIC [12,13], NuSMV (a new symbolic model checker) [21], SPIN model checker [9,59]… etc. These tools can be found in their website. We use NuSMV [21] as our back end. NuSMV is a reimplementation and extension of SMV. While SMV supports only the CTL specifications, NuSMV supports both CTL and LTL ones among others. Both SMV and early versions of NuSMV support BDD-based model checking only. NuSMV 2 [60] is not the same. It integrated SAT-based bounded model checking which is popular in the industry. Being an open source project, NuSMV 2 was restructured to open up toward the implementations of other model checking algorithms.

3. SYN-sequences

In this chapter, we focus on the presentations and properties of SYN-sequence. We introduce two presentations, i.e. partial order graph and reachability graph. We can transform the SYN-sequence into partial order graph by presenting the events as nodes and adding the dependencies between events as edges. We can transform the SYN-sequence into reachability graph where the edges present the events and nodes present the states by *program counter expansion*. We propose two models of reachability graph according to information kept in a state. With *variable value expansion*, we can arrive at the stateful states which keep the values of all shared variables as well.

3.1 The Read / Write Model

An execution of concurrent program P with input X exercises a sequence of synchronization events or *SYN-sequence*. A synchronization events or *SYN-event* is an operation on a synchronization object (or shared object), e.g. read or write operation on the shared variable. There are different types of SYN-sequence, two of which are *process-based* (or *thread-based*) and *object-based* ones. For an object-based SYN-sequence, each shared object in a concurrent program is associated with a sequence of SYN-events, while the SYN-sequence of a concurrent program is a collection of the SYN-sequences for its synchronization objects [61]. On the other hand, the SYN-events in the process-based SYN-sequence are grouped according to the processes. The SYN-event in the process-based SYN-sequence may also keep track of the program counter. We adopt the process-based SYN-sequence in the thesis.

In the read / write model, the synchronization objects are shared variables and the SYN-events are *read* and *write events* on a shared variable. Each shared variable in P is

assigned a version number which is initialized to zero. Each write event on a shared variable increase the version of the variable by 1 [19]. A read event is denoted as $\mathtt{r}(var, ver)$ where var is the identifier of the shared variable and ver is the version number. Similarly a write event is denoted as $\mathtt{r}(var, ver)$ where ver is the version number *after* this write event is executed. A *ReadWrite-sequence* or *RW-sequence* is a sequence of read and write events operating on shared variables in P. We adopt the process-based RW-sequence in our RW model (read / write model). Hence we associate each process with a RW-sequence and define the RW-sequence of a concurrent program P with input X as the collection of the RW-sequences of all processes in P. Here is an example RW-sequence $\{S[0], S[1]\}$ of a concurrent program consisting of two processes:

$$S[0] = (\mathtt{r}(a,0), \mathtt{r}(a,0), \mathtt{r}(a,1), \mathtt{w}(a,2))$$
$$S[1] = (\mathtt{w}(a,1), \mathtt{r}(b,0), \mathtt{r}(b,0), \mathtt{r}(a,2))$$

We keep additional information, i.e. process and event identifiers, in the events of our RW model. Each event in the RW-sequence of a process is numbered in sequence as its event identifier. The first event in a RW-sequence of a process is usually numbered 0. The execution between different processes can be interleaved. The events within the same process must be executed in sequence according to their events identifiers. We can also keep in each event the program counter of that process. Different events may execute the same code and thus share the same program counter. The order of program counters in a process of SYN-sequence may be different from that of the original code. Nonetheless, we assume the event identifier to be the program counter in this thesis for the sake of simplicity.

If two events in different processes access the same shared variable and at least one of them is the write one, then they are in race condition. The result will depend on the execution order of these two events. By definition, a SYN-sequence does not explicitly record the dependency between processes, but the execution orders of events between

different processes are implicitly enforced by means of variable versions, i.e. all accesses to a variable of the smaller versions must prior to all accesses to the same variable of the larger versions.

3.2 Partial Order Graph of SYN-sequence

For the purpose of verification, we may want to have a data structure that explicitly contains the inter-process dependencies or simply *dependencies*. At the same time, we want our data structure to maintain a minimal number of dependencies. This minimal set of dependencies can be determined by the Dependency Algorithm. This algorithm also gives a formal definition of our dependency relation.

Dependency Algorithm

4. Initialize the arrays `head` and `tail` to be null.
5. Figure out the three-dimensional arrays `head` and `tail` that stores the first and the last event, respectively, of each version number in each process. This is done by, for each event eid in each process pid, if event evn access variable vid of version ver and
   c. if `head`[pid][vid][ver] is null or eid < `head`[pid][vid][ver], then modify `head`[pid][vid][ver] to store eid.
   d. if `tail`[pid][vid][ver] is null or eid > `tail`[pid][vid][ver], then modify `tail`[pid][vid][ver] to store eid.

   Note that the elements of both arrays may be null.
6. For each event `head`[pid][vid][ver], if it is not null and
   a. if the operation is write, then add dependencies from the read events `tail`[pid'][vid][$ver - 1$] for all processes pid' other than pid. If no such read

event and `tail`$[pid][vid][ver-1]$ is also null, then add an dependency from the event `tail`$[pid][vid][ver-1]$ in which the operation is write.

b. if the operation is read, then add an dependency from the write event `head`$[pid][vid][ver]$.

If any events in the dependency are null, then this dependency is ignored.

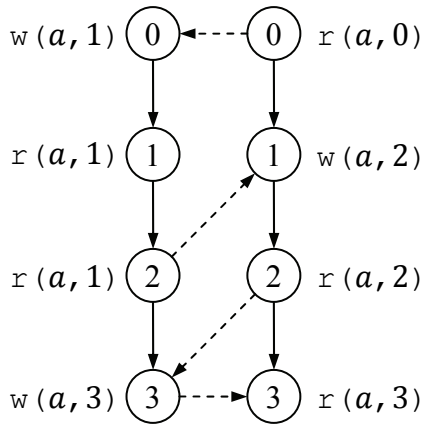


Figure 3.1. An example of partial order graph.

The path on the left simulates process 0, and the other one simulates process 1. The dashed line is the dependency additional to the SYN-sequence.

Figure 3.1 shows an example partial order graph of SYN-sequence. It is obvious that before `w(`a`,1)` of process 0 being executed, all access to variable a with version 0 must be finished. Thus we have a dependency from `r(`a`,0)` of process 1 to `w(`a`,1)` of process 0. This illustrates step 6a in the algorithm.

Now all access to variable a with version 1 can proceed in process 0. After that, process 0 must wait before executing `w(`a`,3)` until all access to variable a of versions less than 2 is completed. In this case, we add a dependency from `r(`a`,2)` of process 1 to `w(`a`,3)`. This also followed the step 6a. Similarly, we have a dependency from `r(`a`,1)` of process 0 to `w(`a`,2)` of process 1. Before executing `r(`a`,3)` of process 1, we must make sure that `w(`a`,3)` was ever executed and thus add the necessary

dependency. This illustrates step 6b in the algorithm. Note that each write to a variable of a specific version will appear in the SYN-sequence exactly once.

What if we have other processes that are not shown and all access to variable a of version 2 in process 1 becomes $\mathtt{r}(a,1)$ as in Figure 3.2? One may thought that we have to add a dependency from the latest $\mathtt{r}(a,1)$ in process 1 to $\mathtt{w}(a,3)$ of process 0. Addition of this dependency is, however, redundant since we are confident by our algorithm that we do have a dependency from $\mathtt{r}(a,1)$ of process 1 to $\mathtt{w}(a,2)$ in certain process, and some dependency may exist till we have one or more dependency to $\mathtt{w}(a,3)$.

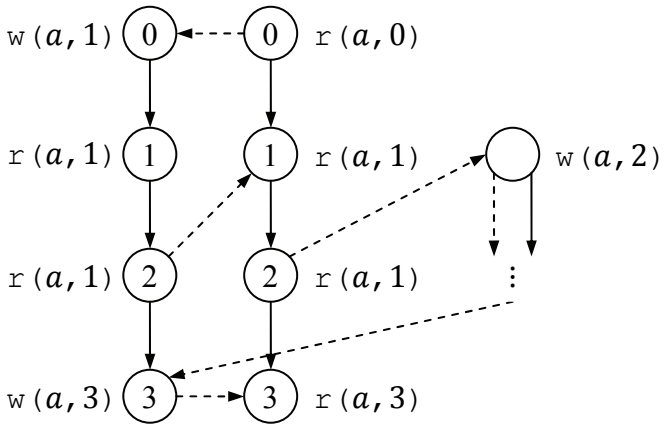


Figure 3.2. Event $\mathtt{w}(a,2)$ occurs in some process other than processes 0 and 1.

Note that all events within the same process must be executed in strict order according to their event identifiers. This is why we restrict the term “dependency” between only events of different processes.

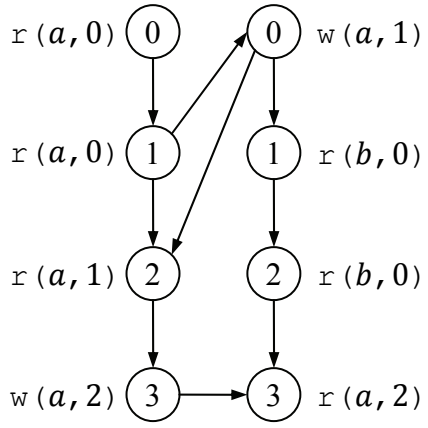


Figure 3.3. The partial order graph of SYN-sequence.

We assume the values of a are 4, 7, 11 respectively when the versions are 0, 1, 2. The value of b is constantly 3.

Now let's see one more example to gain insight into the behavior of the SYN-sequences. Figure 3.3 shows a SYN-sequence in the form of partial order graph. Initially, the first event of process 0 or process 1 may be executed. However, the execution of event 0 in process 1 must be postponed after the execution of event 1 of process 0. Hence the first executed event is event 0 of process 0. After that only the event 1 of process 0 could be executed. Now it's time for event 0 of process 1 to be executed. The three events are executed in a fixed *sequential* order, and then one of event 2 in process 0 and event 1 in process 1 can be nondeterministically chosen for executing.

Some properties exist in SYN-sequence. The first one is happened before relation. If two events must be executed in the fixed sequential order, we say that the prior one must *happen before* the posterior one. The edges on the partial order graph of SYN-sequence present the happened before relation. Event a happens before event b iff there exists a path from a to b on the partial order graph. The second property is parallel relation. If two events can be nondeterministically chosen for executing, we say these events are *parallel*. Events a and b are parallel iff there exists no path between a to b on the partial order graph. This is obvious by definition, since no paths existing between these two events implies no strict execution order for these events. The third property is transition

relation. This property is, however, not explicitly presented in the partial order graph at all. Even there exists a path between two events, one of these events may not happen *directly* after the other. This is illustrated by events 1 and 2 of process 0 in Figure 3.3, since event 2 can be executed only if both event 1 of process 0 and event 0 of process 1 were executed, while event 1 of process 0 always happens before event0 of process 1. On the other hand, if there is no path between two events, one of the events may be executed right after the execution of the other. In Figure 3.3, e.g., once event 2 in process 0 or event 1 in process 1 was executed, the other one can be immediately chosen for executing.

3.3 Reachability Graph

We define a *trace* as a feasible totally-ordered execution sequence of events in the SYN-sequence, as well as an *initial state* that represent a system status before any event being executed and the states that keep track of the system status after the completion of each event. The recorded system status may be the values of all share variables, program counters of all processes, and the properties about executed events… etc. What to be kept in a state is on demand. If all information kept in a state is equal to that in another state, we say these states are *equivalent*. The term *feasible* indicate that the trace must follow all rules on the SYN-sequence, e.g. the dependency relation and the sequential property in each process. We can represent the trace as a path, where a node represents a state and an edge represents an event. Due to the nondeterministic behavior of concurrent programs, it is possible to figure out more than one trace in a SYN-sequence.

A *reachability graph* of a SYN-sequence is a representation of the transition system $\mathcal{M}(S, S_0, T, L)$ where S is the set of all states in all traces of that SYN-sequence, S_0 is the set of all initial states, i.e. the states before the execution of any event, T is the set of transitions between two states, and L is a function that labels each transition with an

event. The reachability graph of a given SYN-sequence comprises all traces from that SYN-sequence, while the events are labeled on edges whereas the identical paths are merged into a single one. Two paths in the reachability graph of a SYN-sequence are *identical* if all states in both paths are equivalent and the sequence of events labeled on both paths is identical. Each path in a reachability graph must be a trace of that SYN-sequence as well. For any state s_i in the reachability graph of SYN-sequence, a *trace to* s_i is a path from any initial state to s_i.

There are two models of reachability graphs according to the records kept in each state. A *stateless* model involves only the program counter expansion while a general or *stateful* one involves variable value expansion as well. The *program counter expansion* is the procedure that enumerates all possible traces from a SYN-sequence. Hence we can transform the partial order graph into reachability graph by program counter expansion.

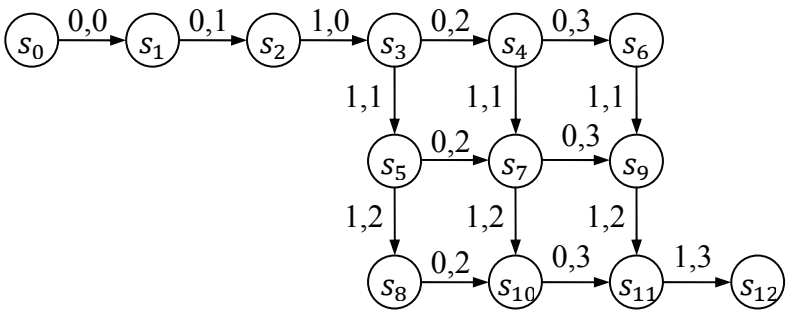


Figure 3.4. The reachability graph of SYN-sequence as in Figure 3.3.

The numbers in each node are the values of variables a and b (in the upper line) and program counters of each process.

To carry out the *variable value expansion*, we need maintain the values of all variables. The properties about all variables are stateful properties. In practice, states in a stateless model keep track of only *stateless* properties, i.e. information about an event. Let's see the difference between stateful and stateless properties. As mentioned, each event may be associated with an operation type, an accessed share variable, a version number, a program counter… etc. Such data are stateless properties. When an event is

executed, the values of all variable are nondeterministic. For instance, see Figure 3.4. When event 1 of process 1 is reading variable b, the value of variable a may be 7 or 11, depending on which state the system is in. In this thesis, the stateful properties kept in stateful model includes version numbers and values of all variables in each state; the stateless properties kept in both stateful and stateless model includes event counters, i.e. identifiers of lastly executed events, and program counters of all processes.[1]

Theorem 1 For any states s_1 and s_2 in the reachability graph of SYN-sequence, if both traces to s_1 and s_2 are comprised of the same set of events, regardless of the execution order, then s_1 and s_2 must be equivalent.

Proof: If any two traces are comprised of the same events in the same execution order, then these two traces are identical by definition. Otherwise, since both traces, say t_1 and t_2, to s_1 and s_2 are comprised of the same events, the maximal event identifier of each process and the maximal version number of each variable in both paths must be the same as well. Since the event with maximum identifier must be executed lastly among those in the same processes, the event counters of all processes kept in s_1 and s_2 must be identical. Furthermore, the event with maximum version must be executed lastly among those on the same variable v, the value of v is identical in s_1 and s_2

This theorem provides a scheme to identify each state other than using all information kept in that state; i.e. we can identify each state by using the set of events executed thus far. In other words, if the traces to states s_1 and s_2 comprise the same

[1] The identifier of lastly executed process is also kept in each state of our implementation of SYN-verifier so that the operation, identifier, version number and value of shared variable lastly accessed are available in each state. This will be discussed in Chapter 1. Even so, we assume the properties of the lastly executed process are absent in the theoretical discuss in this chapter.

events, s_1 and s_2 are said to be identical. In reachability graphs, identical states are merged into one single state.

Theorem 2 For any state in the reachability graph of SYN-sequence, it is possible that there exist more than one trace to this state. In such case, these traces are comprised of the same events while the execution orders are different.

Theorem 2 are informally proved since we have an instance shown in Figure 3.4 We can easily see the *lattice property*, which is the natural consequence of theorems discussed thus far, in all reachability graphs.

Theorem 3 All traces of a SYN-sequence terminate in an identical state.

Proof: Since all events in a SYN-sequence must be executed, no matter what execution order the events are in, all traces arrive at an identical state after the execution of all events in a SYN-sequence.

The reachability graph of the example SYN-sequence shown in 3.1 is shown in Figure 3.4 based on the previous theoretical discussion.

3.4 Comparisons

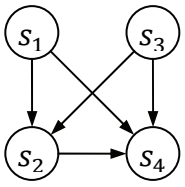


Figure 3.5. A reachability graph

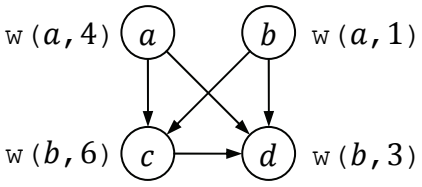


Figure 3.6. A partial order graph of SYN-sequence

Reachability graph is quite different from partial order graph of SYN-sequence. First, the reachability graph involves program counter expansion while partial order graph doesn't. Second, nodes of partial order graph stand for "events" to be done, while those of reachability graph keep the properties of variables and executed events. The state after execution of an event may be different regarding the execution order of the events. For example, we can figure out several traces, e.g. $a \rightarrow b \rightarrow d \rightarrow c$ and $b \rightarrow a \rightarrow d \rightarrow c$

from Figure 3.6. If the initial value of a is 0, we may find that, after executing b, the state of the former trace contains $a = 4$, while that of the later one contains $a = 0$. Third, the edges in partial order graphs of SYN-sequences indicate the dependency relation. Those in reachability graphs are, however, transitions between states. Furthermore, the edges are labeled by an event to designate what happened during the state transition. In reachability graph, one state transits to another along the explicit path, e.g. state s_1 can transit directly to s_2 in Figure 3.5. However, in the partial order graph shown in Figure 3.6, event a happens before c, but never immediately since event d have to be executed after a but before c. In the other way, s_1 never transits to s_3 in the reachability graph, while the events a and b may be executed right after each other in the reachability graph, i.e. both execution sequences $a \rightarrow b$ and $b \rightarrow a$ are permitted.

We may find that the partial order graph encodes the reachability graph, and thus reduced the size of representation for SYN-sequence. This encoded information will help us a lot in Stateless Model Checking on SYN-sequence later. (See 5.2 on page 47.)

Stateful Model vs. Stateless Model

We use similar procedures to construct the general and stateless model of reachability graph. In the case of constructing stateless model, we must preprocess on the SYN-sequence to get the dependencies between processes, using the Dependency Algorithm on page 15. In the case of constructing general model, we use a similar Transition Algorithm to determine the transition relations (see 5.1). Both algorithms have the time complexity of $O(n)$ where n is the number of events in the SYN-sequence. Nevertheless, he size of total state space is different. The stateless model with n processes and n_i events in each process contains only $O(\prod_{i=1}^{n} n_i)$ states[2]. The state space of stateful model is, however, enlarged by $\prod_{i=1}^{k} v_i$ times if each variable has v_i

[2] This is theoretical value. Total states number is $n \times \prod_{i=1}^{n}(n_i + 2)$ in our implementation.

possible values. This increase is quite enormous especially when the write operation occurs frequently, since each write operation will enlarge the range of the written variable by one. Note that the reachable states of both models are isomorphic. The size of state space of partial order model is the smallest. It equals the number of total events. The set of reachable states is identical to the whole state space.

On the other hand, the specifications for reachability graph are more intuitive than those for partial order model. This is because reachability graphs are comprised of traces while a path on partial order graph guarantees only happened before relations. To express specifications for partial order graph, we must have entire insight into the partial order graph. This includes that we have to know all events of the SYN-sequence. This is almost impossible before the execution of concurrent program. We will discuss the specifications in detail in the next chapter.

4. Specifications in Temporal Logics

Generally speaking, *properties* are the description of some behavior of the system, while *specifications* are the desired properties or *requirements* we want to verify. In this chapter, we discuss some frequently used properties or specifications of concurrent programs and their expressions in temporal logics.

4.1 Properties on Program Behavior

The verification of system is the procedure that checks whether some properties exist in the system. Specifications are those properties we require the system to satisfy. We enumerate several types of specifications that are frequently requested by programmer in the concurrent programs, i.e. preconditions, postconditions, invariance, safety, liveness and parallel properties.

Preconditions and postconditions are a method of specifying what a function accomplishes. Frequently a programmer must communicate precisely what a function accomplishes, without any indication of how the function does its work. Preconditions and postconditions are used to specify precisely what a function does. However, a precondition/postcondition specification does not indicate anything about how a function accomplishes its work. This separation between what a function does and how the function works is extremely important – particularly for large programs which are written by a team of programmers. There are many ways to specify the requirements for a function. One of which is using a pair of statements for each function, called the function's precondition and postcondition. The two statements usually work together: The precondition indicates what must be true before the function is called. The postcondition indicates what will be true when the function finishes its work. For instance, we may request our concurrent program to be executed with a pair of nature

number as its input. This is the precondition. If the precondition held, we expect the receiver to get the correct data from some specific sender after the execution. This is an example of postcondition. The precondition and postcondition are not actually part of the program. It is common to place the precondition/postcondition pair in a comment immediately after the function's parameter list. If the execution violates the precondition, then the results are totally unpredictable.

The preconditions and postcoditions specify the requirement of a particular time, i.e. when starting and finishing execution, respectively. Invariance requires the specification to be true as long as the system is executed. We may request that variable a is read only. Of course, we will not expect the requirement hold only when the system starts and finishes. Invariance can be verified by checking the entire system. Once again, it is not a part of the system. (If it were, we do not need to verify it any more.)

Safety and liveness properties are proposed by Lamport in 1977 [62]. Safety properties state that certain undesirable states of the system will never be reached under any circumstances. A well-known example is that no two processes can enter their critical section at the same time. This is somehow like the invariance property. Another type of safety property is known as happened before relation. We may require, e.g., that process 0 always writes variable a before process 1 does in a concurrent system. This type of properties is much different from invariance, and frequently appears in concurrent systems. Liveness properties state that certain events are guaranteed to occur eventually – for instance, that a given thread will eventually wake up instead of sleeping forever. Another example is whenever process 0 writes value 1 to variable a, it will eventually read variable b.

Remind that parallel relation is another frequently used property in concurrent programs. We mentioned it in 3.2.

We classify all types of specification as stateful and stateless. A stateful specification involves properties of individual program, i.e. the values of all shared variable. A stateless one involves only the status of an event, e.g. the operation and variable accessed. The specification that only when the value of variable $turn$ is 1 can process 1 access variable a is a stateful example, while the one that only when process 0 has written 1 to variable $turn$ can process 1 access variable a is stateless. Note the difference between these two examples: the prior one state that when process 1 access variable a, the value of variable $turn$ is 1, while the later one guarantees nothing about the value at the same state. Since stateless specification is a special case of stateful one, we also say the stateful one as general specification.

4.2 Necessity of Applying Temporal Logic

Specifications are expressed in logic for the purpose of automatic verification. Traditionally, propositional or predicate logics are used, such as Boolean expressions and functions which return Boolean value. For example, when the value of x is determined, the expression "x is equal to 4" is a proposition. This is because the truth of this expression is uniquely determined – it is either true or false, but not both. We can use logical connectives such as ∧ (AND), ∨ (OR), ¬ (NOT), → (IMPLY) to express complex specifications. Another example is, when we require our system to satisfy the safety property, say, no two processes are in their critical sections at the same time, we may express this specifications as $\forall p, q[\neg(\mathrm{cs}(p) \wedge \mathrm{cs}(q))$, where the universe is the set of all processes in the system and $\mathrm{cs}(p)$ is the predicate that p is in its critical section. This is known as first order logic and can be interpreted as that for all processes p, q, it is not the case that p is in its critical section and q is in its critical section. It is implicit that we assume this *invariant* specification must be true in *all* states of the system life. To

verify whether our system satisfies this specification, we can code a function to explore the system and returns the answer.

How could we determine whether some properties are true in *some* states? Moreover, how could we express the various liveness and parallel properties systematically? It is obviously that propositional logic is lack of temporal characteristic. The truth of a specification expressed in propositional logic can be determined only in a specific state or over all the time. Hence only preconditions, postconditions and invariance properties can be expressed in propositional logic. Generally speaking, the other three types of properties, i.e. safety, liveness and parallel properties, are difficult to be expressed with conventional logics. It seems fortunate that we can program a procedure to verify a specification; even so, if we have lots of different specification, we must implement each of them separately. This is inefficient, especially when every implementation is quite different. Model checking with temporal specifications provide a general solution to this. We can express our specifications as formulas in temporal logic. This formula can then be interpreted in Kripke structure. Since the system can also be modeled in Kripke structure, we can do the model check automatically.

Another advantage of temporal logic over the predicate logic is that the symbols or operators are definite. Almost all properties of concurrent programs can be expressed in 7 *logic connectives*. The details will be discussed in 4.3. Nevertheless, there exist no such adequate set for predicates to express various type of properties of concurrent programs.

4.3 Linear Temporal Logic and Computation Tree Logic

We give the syntactic definition of linear temporal logic (LTL) and computation tree logic (CTL) here:

```
ltl_expr :: bool_expr                  – Boolean expression
          | ( ltl_expr )
          | ¬ ltl_expr                 – logical not
          | ltl_expr ∧ ltl_expr        – logical and
          | ltl_expr ∨ ltl_expr        – logical or
          | ltl_expr → ltl_expr        – logical implies
          | X ltl_expr                 – next state
          | G ltl_expr                 – globally
          | F ltl_expr                 – finally
          | ltl_expr U ltl_expr        – until
          | ltl_expr R ltl_expr        – releases

ctl_expr :: bool_expr                  – Boolean expression
          | ( ctl_expr )
          | ¬ ctl_expr                 – logical not
          | ctl_expr ∧ ctl_expr        – logical and
          | ctl_expr ∨ ctl_expr        – logical or
          | ctl_expr → ctl_expr        – logical implies
          | EG ctl_expr                – exists globally
          | EX ctl_expr                – exists next state
          | EF ctl_expr                – exists finally
          | AG ctl_expr                – forall globally
          | AX ctl_expr                – forall next state
          | AF ctl_expr                – forall finally
          | E [ ctl_expr U ctl_expr ]  – exists until
          | A [ ctl_expr U ctl_expr ]  – forall until
```

The bold symbols are terminal symbols. These symbols, except the parentheses "(" and ")", are called *logic connectives*. Among these logic connectives, {¬,∨,**X**,**U**} is adequate for LTL since all other connectives can be expressed using connectives in the adequate set. The adequate sets of CTL contains one of {**AX**, **EX**}, one of {**EG**, **AF**, **AU**}, and all of {¬,∨,**EU**}.

The semantics of these temporal connectives are formally defined in [8]. The LTL system interprets the formulas in a linear time sequence. An LTL formula must hold in all execution sequences, while CTL ones can quantify existence or all paths since the formulas are interpret as computation trees. We informally state the semantics of LTL as follows: For any trace $s_0 \to s_1 \to \cdots$,

1. X p is true if p holds in state s_1.
2. F p is true if p holds in *some* state s_i with $i \geq 0$.

3. G p is true if p holds in *all* states s_i with $i \geq 0$.
4. p U q is true if q holds in *some* state s_i with $i \geq 0$ and p holds in *all* states s_j with $0 \leq j < i$.
5. p R q is true if q holds in *all* states s_i with $i \geq 0$ up to and including the state where p also holds. Alternatively, it may be the case that p *never* holds in which case q must hold in *all* states s_i with $i \geq 0$.

We informally state the semantics of CTL as well: For any system represented as a model or a computation tree,

1. EX p is true in a state s if *there exists* a state s' such that a transition goes from s to s' and p is true in s'.
2. AX p is true in a state s if *for all* states s' where there is a transition from s to s', p is true in s'.
3. EF p is true in a state s_0 if *there exists* a trace $s_0 \to s_1 \to \cdots \to s_{n-1} \to s_n$ such that p is true in s_n.
4. AF p is true in a state s_0 if *for all* traces $s_0 \to s_1 \to \cdots \to s_{n-1} \to s_n$ p is true in s_n.
5. EG p is true in a state s_0 if *there exists* a *infinite* trace $s_0 \to s_1 \to \cdots$ such that p is true in *every* s_i.
6. AG p is true in a state s_0 if *for all* infinite traces $s_0 \to s_1 \to \cdots$ p is true in *every* s_i.
7. E[p U q] is true in a state s_0 if *there exists* a trace $s_0 \to s_1 \to \cdots \to s_{n-1} \to s_n$ such that p is true in *every* state from s_0 to s_{n-1} and q is true in state s_n.
8. A[p U q] is true in a state s_0 if *for all* traces $s_0 \to s_1 \to \cdots \to s_{n-1} \to s_n$ p is true in *every* state from s_0 to s_{n-1} and q is true in state s_n.

In SYN-sequence, no two events will be executed simultaneously. We also suppose that each event will be executed exactly within one state for other proposes, e.g. of counting the numbers that an event was executed. For example, we may have

specification such as the execution of event p does not exceed three times. This can then be expressed as $\mathrm{G}\neg\left(p \wedge \mathrm{F}(p \wedge \mathrm{F}(p \wedge \mathrm{F}p))\right)$. Recall that each event can be executed exactly once in SYN-sequences.

To verifying SYN-sequences, one may often want to know whether the execution of event p always happens before event q. We denote this *happened before* relation as $p \prec q$. This relation says that, along all possible execution paths, p must be executed first, and then q will be executed. Note that we assume no two events will be executed at the same time. This can be expressed as $\mathrm{F}(p \wedge \mathrm{XF}q)$ or $\mathrm{G}\neg(q \wedge \mathrm{F}p)$ in temporal logic.

In the case of $\mathrm{F}(p \wedge \mathrm{XF}q)$,

$$
\begin{aligned}
& \pi \vDash \mathrm{F}(p \wedge \mathrm{XF}q) \\
\Leftrightarrow\ & \exists i \geq 1\left[\pi^i \vDash (p \wedge \mathrm{XF}q)\right] \\
\Leftrightarrow\ & \exists i \geq 1\left[\pi^i \vDash p \wedge \pi^i \vDash \mathrm{XF}q\right] \\
\Leftrightarrow\ & \exists i \geq 1\left[\pi^i \vDash p \wedge \pi^{i+1} \vDash \mathrm{F}q\right] \\
\Leftrightarrow\ & \exists i \geq 1\left[\pi^i \vDash p \wedge \exists j \geq i+1\left[\pi^j \vDash q\right]\right]
\end{aligned}
$$

Now we are confident that this formula expresses p always happens before q. In this formula p will be executed at least once. Then q will also be executed at least once. Since p and q cannot be executed simultaneously, this formula can be simplified to $\mathrm{F}(p \wedge \mathrm{F}q)$. The truth table of $p \wedge \mathrm{XF}q$ and $p \wedge \mathrm{F}q$ is given in Figure 4.1.

p	q	$\mathrm{XF}q$	$\mathrm{F}q$	$p \wedge \mathrm{XF}q$	$p \wedge \mathrm{F}q$
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	1	1

Figure 4.1. The truth table of $p \wedge \mathrm{XF}q$ and $p \wedge \mathrm{F}q$.

In the case of $\mathrm{G}\neg(q \wedge \mathrm{F}p)$,

$$
\begin{aligned}
& \pi \vDash \mathrm{G}\neg(q \wedge \mathrm{F}p) \\
\Leftrightarrow\ & \pi \vDash \mathrm{G}(\neg q \vee \mathrm{G}\neg p)] \\
\Leftrightarrow\ & \forall i \geq 1[\pi i \vDash (\neg q \vee G\neg p)]
\end{aligned}
$$

$$\Leftrightarrow \forall i \geq 1[\pi i \vDash \neg q \vee \pi i \vDash G\neg p]$$
$$\Leftrightarrow \forall i \geq 1\left[\pi i \nvDash q \vee \forall j \geq i[\pi j \nvDash p]\right]$$

This tells us that along each trace q will never be executed or p will not be executed from some moment on. In reality, this formula specifies that p will never happen after q. Since we assume that each event will be executed exact once, this formula turns to mean $\forall i \geq 1[\pi^i \nvDash p]$ in the state where q is executed. This asserts that p will never happen *after* q. In all the other states along the path, this formula holds since q is not executed. Note that $\mathrm{G}\neg(q \wedge \mathrm{F}p)$ stands for the happened before relation on the assuming that each event will be executed exact in one state. The truth table of $\neg(q \wedge \mathrm{F}p)$ is shown in Figure 4.2.

$\boldsymbol{q}$	$\mathbf{F}\boldsymbol{p}$	$\neg(\boldsymbol{q} \wedge \mathbf{F}\boldsymbol{p})$
0	0	1
0	1	1
1	0	1
1	1	0

Figure 4.2. The truth table of $\neg(q \wedge \mathrm{F}p)$.

At a glance, we may think that these two formulae are quite different. First, $\mathrm{F}(p \wedge \mathrm{XF}q)$ requires both p and q to be executed at least once, while $\mathrm{G}\neg(q \wedge \mathrm{F}p)$ allows both p and q not to be executed. Since we suppose each event to be executed exactly once, $\mathrm{F}(p \wedge \mathrm{XF}q)$ should be augmented as

$$\pi \vDash \mathrm{G}\neg(q \wedge \mathrm{F}p) \wedge \mathrm{F}p \wedge \mathrm{F}q$$
$$\Leftrightarrow \forall i \geq 1[\pi^i \nvDash q \vee \forall j \geq i[\pi^j \nvDash p]]$$
$$\wedge \exists i \geq 1[\pi^i \vDash p] \wedge \exists i \geq 1[\pi^i \vDash q]$$
$$\Leftrightarrow \exists i \geq 1[\pi^i \vDash p \wedge \exists j \geq i[\pi^j \vDash q]]$$
$$\Leftrightarrow \pi \vDash \mathrm{F}(p \wedge \mathrm{F}q)$$

Hence these two formulae have the same effect.

A fallacious formula for happened before relation is $\mathrm{G}\neg(p \rightarrow \mathrm{F}q)$. Since

$$\mathrm{G}\neg(p \rightarrow \mathrm{F}q)$$
$$\Leftrightarrow \mathrm{G}\neg(\neg p \vee \mathrm{F}q)$$
$$\Leftrightarrow \mathrm{G}(p \wedge \neg \mathrm{F}q)$$
$$\Leftrightarrow \mathrm{G}(p \wedge \mathrm{G}\neg q),$$

this formula is error. If not, p must be true in all states. This contradicts the assumption that each event will be executed in exactly one state.

Another well-known specification in concurrent system is that the execution of events p and q must be in parallel. We denote this *parallel* relation as $p \parallel q$. This can be expressed as $\mathrm{G}(p \vee q \rightarrow \neg(p \prec q \wedge q \prec p))$. We suppose that the happened before relation $\prec$ has higher priority than $\wedge$ has. The truth table of this formula is shown Figure 4.3.

p	q	$\mathbf{F}p$	$\mathbf{F}q$	$\mathbf{A}$ $p \prec q$	$\mathbf{B}$ $p \prec q$	$\neg(\mathbf{A} \wedge \mathbf{B})$	$\mathbf{C}$ $p \vee q$	$\mathbf{C} \rightarrow \neg(\mathbf{A} \wedge \mathbf{B})$
1	0	0	1	1	0	1	1	1
1	0	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	1
0	1	1	0	0	1	1	1	1
0	1	1	1	0	1	1	1	1
1	1	1	0	0	1	1	1	1
1	1	1	1	0	0	1	1	1
0	0	0	0	1	1	0	0	1
0	0	0	1	1	1	0	0	1
0	0	1	0	1	1	0	0	1
0	0	1	1	1	1	0	0	1
1	0	0	0	1	1	0	1	0
1	0	1	0	1	1	0	1	0
0	1	0	0	1	1	0	1	0
0	1	0	1	1	1	0	1	0
1	1	0	0	1	1	0	1	0

Figure 4.3. The truth table for $p \vee q \rightarrow \neg(p \prec q \wedge q \prec p)$.

Sometimes we may want to check some properties involving shared data, e.g. the shared variables x cannot be accessed in events p and q simultaneously. This can be expressed as

$$\mathrm{G}\neg(p(*, x) \wedge q(*, x))$$

where wildcards $*$ stand that we don't care about the operation type of data access.

Let's see one more specification: If there exists an execution loop of events p, q, r, then this loop can iterate at most three times. For simplicity, we first define a *symbol* as following:

$$Loop \stackrel{\text{def}}{=} p \wedge \mathrm{F}(q \wedge \mathrm{F}r)$$

The symbol $Loop$ stands for the execution loop of events p, q, r. In this loop p is executed in this state, and it is supposed that no two events will be executed simultaneously. Then the given specification is

$$\mathrm{G}\neg(Loop \wedge \mathrm{F}(Loop \wedge \mathrm{F}(Loop \wedge \mathrm{F}Loop)))$$

Before closing this section, we list some LTL properties which should hold in the SYN-sequence shown in Figure 3.3:

$$\mathrm{X\ p0.0} \wedge \mathrm{XX\ p0.\ 1} \wedge \mathrm{XXX\ p1.0} \wedge \mathrm{XXXX\ (p0.2} \vee \mathrm{p1.1)}$$
$$\mathrm{XXXX\ p0.2} \rightarrow \mathrm{XXXXX\ (p0.3} \vee \mathrm{p1.1)}$$
$$(\mathrm{XXXX\ p0.2} \wedge \mathrm{XXXXX\ p0.3}) \rightarrow (\mathrm{XXXXXX\ p1.1} \wedge \mathrm{XXXXXXX\ p1.2})$$
$$(\mathrm{XXXX\ p0.2} \wedge \mathrm{XXXXX\ p1.1}) \rightarrow \mathrm{XXXXXX\ (p0.3} \vee \mathrm{p1.2)}$$
$$(\mathrm{XXXX\ p0.2} \wedge \mathrm{XXXXX\ p1.1} \wedge \mathrm{XXXXXX\ p0.3}) \rightarrow \mathrm{XXXXXXX\ p1.2}$$
$$(\mathrm{XXXX\ p0.2} \wedge \mathrm{XXXXX\ p1.1} \wedge \mathrm{XXXXXX\ p1.2}) \rightarrow \mathrm{XXXXXXX\ p0.3}$$
$$\mathrm{XXXX\ p1.1} \rightarrow \mathrm{XXXXX\ (p0.2} \vee \mathrm{p1.2)}$$
$$(\mathrm{XXXX\ p1.1} \wedge \mathrm{XXXXX\ p0.2}) \rightarrow \mathrm{XXXXXX\ (p0.3} \vee \mathrm{p1.2)}$$
$$(\mathrm{XXXX\ p1.1} \wedge \mathrm{XXXXX\ p0.2} \wedge \mathrm{XXXXXX\ p0.3}) \rightarrow \mathrm{XXXXXXX\ p1.2}$$
$$(\mathrm{XXXX\ p1.1} \wedge \mathrm{XXXXX\ p0.2} \wedge \mathrm{XXXXXX\ p1.2}) \rightarrow \mathrm{XXXXXXX\ p0.3}$$
$$(\mathrm{XXXX\ p1.1} \wedge \mathrm{XXXXX\ p1.2}) \rightarrow (\mathrm{XXXXXX\ p0.2} \wedge \mathrm{XXXXXXX\ p0.3})$$
$$\mathrm{XXXXXXXX\ p1.3}$$

where p$i.j$ denotes event j in process i. One might find that this set of properties altogether completely describes the behavior of the given SYN-sequence. We will use this property set to validate the model we constructed later.

4.4 Stateful and Stateless Expressions

We can express our specifications in three types. We can use LTL and CTL in all types. The major different is that specifications in different types can be used in different representations of SYN-sequence, i.e. statful models, stateless models and partial order graphs. General specifications which involve stateful properties, i.e. the value of any variable, can be specified in Type I. We can verify such specifications on the stateful models or the reachability graphs with variable value expansion. Specifications in Type II and III involve only stateless properties, e.g. the operation and variable accessed. We can verify *stateless specifications*, i.e. specifications in Type II on both types of

reachability graph, i.e. stateful and stateless model. Specifications in Type III, however, can be verified only on partial order graph. Note that stateless specification is a special case of general specification. General specifications may contain stateless properties as well.

Here are some example specifications in Type I: For any stateful properties p and q,

1. p happens before q

$$p \prec q \stackrel{\text{def}}{=} \mathrm{F}p \wedge \mathrm{F}q \wedge \mathrm{G}\neg(q \wedge \mathrm{F}p)$$

For example, $\mathrm{F}(a = 1) \wedge \mathrm{F}(a = 2) \wedge \mathrm{G}\neg(a = 2 \wedge \mathrm{F}(a = 1)$

2. p and q are parallel

$$p \parallel q \stackrel{\text{def}}{=} \mathrm{EF}(\mathrm{EX}\, p \wedge \mathrm{EX}\, q)$$

This expression can also be interpreted as p or q may happen before each other. For example, $\mathrm{EF}\big(\mathrm{EX}\, \mathrm{w}(a, 2) \wedge \mathrm{EX}\, \mathrm{r}(b)\big)$

3. p immediately happens before q

$$p\, \mathrm{X}\, q \stackrel{\text{def}}{=} \mathrm{F}(p \wedge \mathrm{X}q)$$

Note that the binary connective X override the unary one. An example specification is $\mathrm{F}(\mathrm{w}(a, 1) \wedge \mathrm{X}\, a = 1)$

4. p may immediately happen before q

$$\mathrm{EF}(p \wedge \mathrm{EX}\, q)$$

For example, $\mathrm{EF}\big(\mathrm{r}(b) \wedge \mathrm{EX}(a = 2)\big)$

Now we will give some example specifications expressed in Type II: For any stateless properties p and q,

1. p happens before q:

$$p \prec q \stackrel{\text{def}}{=} \mathrm{F}q \wedge \mathrm{G}\neg(q \wedge \mathrm{F}p)$$

For example, $\mathrm{F}\, \mathrm{r}(b) \wedge \mathrm{G}\neg\big(\mathrm{r}(b) \wedge \mathrm{F}\, \mathrm{w}(a, 1)\big)$

2. p and q are parallel:

$$p \parallel q \stackrel{\text{def}}{=} \mathrm{EF}(\mathrm{EX}p \wedge \mathrm{EX}q)$$

This formula also expresses p or q may happen before each other. For example, $\mathrm{EF}\big(\mathrm{EX}\,\mathrm{w}(a) \wedge \mathrm{EX}\,\mathrm{r}(b)\big)$

3. p immediately happens before q:

$$p\,\mathrm{X}\,q \stackrel{\text{def}}{=} \mathrm{F}(p \wedge \mathrm{X}q)$$

For example, $\mathrm{F}\big(p0.evnt = 1 \wedge \mathrm{X}(p0.evnt = 2)\big)$

4. p may immediately happen before q:

$$\mathrm{EF}(p \wedge \mathrm{EX}q)$$

For example, $\mathrm{EF}\big(\mathrm{r}(a,1) \wedge \mathrm{EX}\,\mathrm{r}(b)\big)$

Note that the expressions of specifications on partial order graph are quite different from those on reachability graph, since the two graph have entirely different semantics (see Comparisons on page 22 in 3.3 for details). Frequently used specifications are express as the form that can be used on partial order graph is listed below: For any stateless properties p and q,

1. p happens before q:

$$p \prec q \stackrel{\text{def}}{=} \mathrm{EF}p \wedge \mathrm{A}[\mathrm{G}(p \rightarrow \mathrm{XF}q)]$$

Two examples are

$$\mathrm{EF}\,\mathrm{r}(b) \wedge \mathrm{A}\big[\mathrm{G}\big(\mathrm{r}(b) \rightarrow \mathrm{XF}\,\mathrm{r}(a,2)\big)\big]$$

and

$$\mathrm{EF}\,\mathrm{w}(b) \wedge \mathrm{A}\big[G\big(\mathrm{w}(b) \rightarrow \mathrm{XF}\,\mathrm{r}(b)\big)\big]$$

2. p and q are parallel:

$$p \parallel q \stackrel{\text{def}}{=} \mathrm{EF}p \wedge \mathrm{EF}q \wedge \mathrm{A}[\neg(\mathrm{F}p \wedge \mathrm{F}q)]$$

Two examples are

$$\mathrm{EF}\,\mathrm{r}(b) \wedge \mathrm{EF}\,\mathrm{r}(a,1)) \wedge \mathrm{A}[\neg(\mathrm{F}\,\mathrm{r}(b) \wedge \mathrm{F}\,\mathrm{r}(a,1))]$$

and

$$\mathrm{EF\ r}(a,0) \wedge \mathrm{EF\ w}(a,1) \wedge \mathrm{A}\left[\neg\left(\mathrm{F\ r}(a,0) \wedge \mathrm{F\ w}(a,1)\right)\right]$$

3. p immediately happens before q:

$$p\ \mathrm{X}\ q \stackrel{\text{def}}{=} \neg\mathrm{A}[\neg\mathrm{F}(p \wedge \mathrm{X}q)] \wedge \mathrm{A}[\neg\mathrm{G}(p \rightarrow \mathrm{XXF}q)] \wedge \ \ldots$$

The eliminated conjunctions comprise that all nodes in the graph is parallel to those nodes that satisfy p or q. General speaking, the events are dynamically generated *after* testing, and hence this information is not available in advance. Some examples are

$$\neg\mathrm{A}[\neg\mathrm{F}(e_1 \wedge \mathrm{X}e_4)] \wedge \mathrm{A}[\neg\mathrm{G}(e_1 \rightarrow \mathrm{XXF}e_4)] \wedge \ldots$$

$$\neg\mathrm{A}[\neg\mathrm{F}(e_1 \wedge \mathrm{X}e_2)] \wedge \mathrm{A}[\neg\mathrm{G}(e_1 \rightarrow \mathrm{XXF}e_2)] \wedge \ldots$$

and

$$\neg\mathrm{A}[\neg\mathrm{F}(e_3 \wedge \mathrm{X}e_7)] \wedge \mathrm{A}[\neg\mathrm{G}(e_3 \rightarrow \mathrm{XXF}e_7)] \wedge \ldots$$

Generally speaking, Types I and II are easy to use in expressing specifications. To express specifications in Type III, one needs to have sufficient knowledge of partial order graph of SYN-sequence. Even worse, some specifications are not able to be expressed. For example, in the partial order graph that contains isolated component, EFp is not sufficient to express the existence of property p. The SYN-sequence 1 used in the experiments (in Chapter 1) is a good example.

5. Modeling

In this chapter, we illustrate the detail of the automatic translation of SYN-sequence into NuSMV models. The reachability graph, both stateful and stateless models, can be easily described using NSMV language in nature. In particular, we define the partial order model that uses the partial order graph as the skeleton. This model can be described directly using NuSMV language as well. We also discuss how to express the specifications for each model here.

5.1 Verifying General Specifications

We use NuSMV to verify whether the SYN-sequence satisfy the given specifications. Our system can translate the SYN-sequence into the NuSMV model language and run the NuSMV automatically. So far the temporal specifications have to be input manually, and they are parsed by the NuSMV. The details of NuSMV are described in [63]. Figure 5.1 shows the verification scheme.

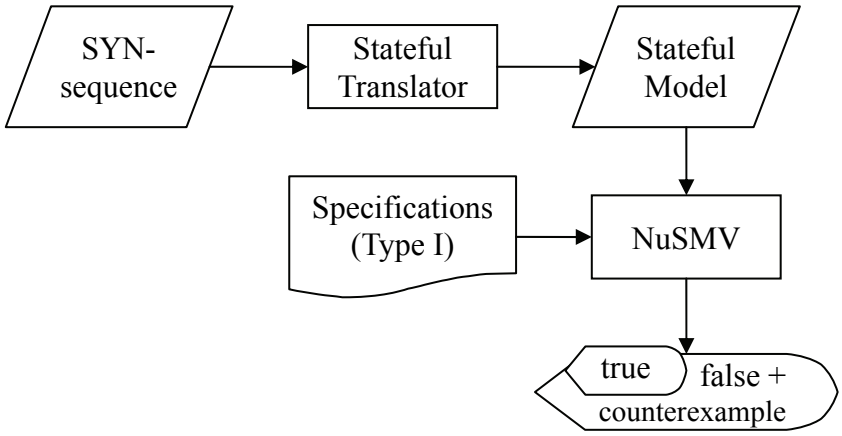


Figure 5.1. The scheme for verification with specification in Type I.

For example, if we have a SYN-sequence given in Figure 3.3 and we want to know whether the value of variable a remains 4 or 7 before variable b being accessed and whether event 1 of process 1 always happens before event 3 of process 0 (see page 18). Let's see how to translate the SYN-sequence given in Figure 3.3. The behavior of each

process is described in separate module. These modules are *process modules*. The NuSMV module that models process 0 is

```
MODULE seq0(prc, aVer, bVer)
    VAR
        evnt : -1..4;
    ASSIGN
        next(prc) := 0;
        init(evnt) := -1;
        next(evnt) :=
            case
                (evnt = -1) & (aVer = 0): evnt+1;
                (evnt = 0) & (aVer = 0): evnt+1;
                (evnt = 1) & (aVer = 1): evnt+1;
                (evnt = 2) & (aVer = 1): evnt+1;
                evnt = 3: evnt + 1;
                1: evnt;
            esac;
        next(aVer) :=
            case
                (evnt = 2) & (aVer = 1): 2;
                1 : aVer;
            esac;
    TRANS
        (next(evnt) = -1 -> next(aVer) <= 0) &
        (next(evnt) = 0 -> next(aVer) <= 0) &
        (next(evnt) = 1 -> next(aVer) <= 1) &
        (next(evnt) = 2 -> next(aVer) <= 1)
```

In this module, we declare a variable `evnt` in `VAR` section to indicate which event of this process is lastly executed before this state. In the right hand side of the colon gives the domain of this state variable. There are two imaginary events, say, -1 and 4 standing for initial and terminated states of the process respectively. Thus before any events being executed, we can leave our process in the state where `evnt = -1`, whereas let our process sunk in the state where `evnt = 4` when all events in the SYN-sequence has been executed. Note that we have only four events so that the last event is numbered 3 here. `ASSIGN` segment comprises the assignments of variables. It is obvious that `init(evnt) := -1;` assigns -1 to the initial value of `evnt` and `next()` represents

the next value of a variable. This will be further constrained in the `TRANS` section and in the `main` module and will be discussed there.

The case analysis uses the keywords `case` and `esac` to enclose it. Each statement in the case analysis sequentially comprises a Boolean expression for the condition, a colon, and the return value, and ends with a semicolon. When evaluating the case analysis, the first return value in the statement where the Boolean expression evaluated to be true is returned to the identifier outside the case analysis. Then this case analysis is finished.

The variables `aVer` and `bVer` keep the version numbers of share variables a and b respectively. Their value is modified only when the event does write operation to that variable and all access to that variable of previous version has been done. For example, version 2 is written into the variable a in event 3 of process 0, hence the next value of `aVer` is changed to `2` when event 2 has been executed and `aVer` is 1. The requirement that all access to that variable of previous version has been done seems not satisfied if we concern only two states; hence we add the TRANS constraints to force all transitions follow the execution behavior of the SYN-sequence. Since all states and transitions must satisfy all constraints in NuSMV, we are confident that the requirement to write a share variable is met. The rule for this requirement is similar to the Dependency Algorithm and will be further discussed later.

We will generalize the transitional behavior of this module now. The events proceed according to the latest version number of variable which will be accessed, e.g. event 2 in process 0 can proceed only if the version number of a is 1 before next state. This is natural since this event is to read variable a of version 1. Event 3 in process 0, however, can proceed only if the version number of a is 1 since it writes a to version 2. This,

along with the transitions of version number of each variable, follows a rule in the following which is similar to Dependency Algorithm.

Transition Algorithm

For each event i in the process being manipulated,

1. if event i writes version ver into variable vid, then the *preversion*, i.e. the version number of vid before this event being executed, must be $ver - 1$. In other words, we have a transition from the state where `evnt` $= j$ with $j = i - 1$ to where `evnt` $= i$ if the version number of vid is $ver - 1$ in the state where `evnt` $= j$. In this case we also modify the version number of vid from $ver - 1$ to ver.
2. if event i reads variable vid of version ver, then the preversion must be ver, i.e. we have a transition from the state where `evnt` $= j$, with $j = i - 1$ to where `evnt` $= i$ if the version number of vid is ver in the state where `evnt` $= j$. Note that the version number remains unchanged after this event.
3. the version number ver of event i must be also less than or equal to all preversions of all other processes, according to the lastly executed event of each process.

Since the lastly executed events in other processes remains unchanged, the value of variable `evnt` keeps the same in each of other process after the transition. The version number ver of variable vid is, however, changed if the event i is a write event on vid. Hence the ver must be less than or equal to the preversion of all other process to reserve the transition in other processes. If the version number ver of variable vid in event i is greater than the perversion of some processes, event i must wait until those processes are executed; otherwise those processes will never proceed anymore. This is the reason for the 3rd constraint, and thus constitutes the `TRANS` section in process modules..

The state variable `prc` indicates which process was running just before that state. Hence after the module representing process 0 is executed, the value of `prc` must be

modified to `0`. Note that prc, aVer, bVer are shared variables in each state and are declared and initialized in the `main` module. The `main` module is the one evaluated by the interpreter. Thus the modules for the processes are also instantiated in the `VAR` section of `main` module. Part of code for `main` module in this example is

```
MODULE main
                        ⋮
    DEFINE
        aVal :=
            case
                aVer = 0 : 4;
                aVer = 1 : 7;
                aVer = 2 : 11;
            esac;
        bVal :=
            case
                bVer = 0 : 3;
            esac;
    VAR
        aVer : 0..2;
        bVer : 0..0;
        prc : 0..1;
        p0 : process seq0(prc, aVer, bVer);
        p1 : process seq1(prc, aVer, bVer);
    ASSIGN
        init(aVer) := 0;
```

The keyword `process` is used to compose the modules asynchronously.

Since this SYN-sequence has two variables *a* and *b*, we ought to declare `a` and `b` in NuSMV input file. However, if we declare both variables for the value and version number for each variable, we will double the state space. Since each version of a variable is mapped into exactly one value, we can use the *define* of NuSMV to assign the values of variables. The identifiers defined in the `define` section are similar to macros. Using define to represent the values of the variables can avoid the enlargement of state space.

Progress Constraints

So far we still have to add some *progress constraints* for both stateful (general) and stateless model. This is done by adding a `TRANS` section in `main` module. Now let's see the `TRANS` section for the given example:

```
TRANS
    (   next(p0.evnt) = p0.evnt
      & next(p1.evnt) = p1.evnt
     -> p0.evnt = 4 & p1.evnt = 4 )
    & (next(p0.evnt) = 4
        -> (p1.evnt = 3 | p1.evnt = 4))
    & (next(p1.evnt) = 4
        -> (p0.evnt = 3 | p0.evnt = 4))
```

NuSMV `TRANS`, as well as `ASSIGN`, is a type of constraint. All transitions must follow all constraints it involved. Thus the state variable `evnt` in each process module follows not only the `ASSIGN` constraints in that process module, but also the `TRANS` constraints in the `main` module. This is natural since we instantiate the process modules in the `main` module. But why should we add this `TRANS` constraint? We find the state variable `evnt` will remain unchanged if all conditions are false in the case analysis. This will lead to all processes idled in a state and is not allowed since we request all events of the SYN-sequence to be executed in exactly one state. The only exception is when all processes are terminated. Thus we add the first clause in the `TRANS` section to constrain that if all processes are idled, they must be all terminated. The remaining clauses further constrain the states from idle until all processes terminate. If these clauses are absent, we may have an execution path such that one process idled in the terminated event and all other processes idled in some non-terminated events.

Stateless Event Properties

Although the NuSMV model we have constructed so far is sufficient for verification, it is not convenient enough for programmer to use. This is because what we have constructed is only the skeleton of SYN-sequence with the pure stateful properties

involved. Our events are represented by process and event identifier. Again we use the define facility of NuSMV to supply the properties on events. Such properties are said to be *stateless* properties. The following DEFINE section is inserted into the `main` module of both general and stateless models:

```
CONSTANTS R, W, r, w, na, a, b;
DEFINE
    evnt :=
        case
            prc = 0 : p0.evnt;
            prc = 1 : p1.evnt;
        esac;
    op :=
        case
            prc = 0 & p0.evnt = 0 : r;
            prc = 0 & p0.evnt = 1 : r;
            prc = 0 & p0.evnt = 2 : r;
            prc = 0 & p0.evnt = 3 : w;
            prc = 1 & p1.evnt = 0 : w;
            prc = 1 & p1.evnt = 1 : r;
            prc = 1 & p1.evnt = 2 : r;
            prc = 1 & p1.evnt = 3 : r;
            1 : -1;
        esac;
    vid :=
        case
            prc = 0 & p0.evnt = 0 : a;
            prc = 0 & p0.evnt = 1 : a;
            prc = 0 & p0.evnt = 2 : a;
            prc = 0 & p0.evnt = 3 : a;
            prc = 1 & p1.evnt = 0 : a;
            prc = 1 & p1.evnt = 1 : b;
            prc = 1 & p1.evnt = 2 : b;
            prc = 1 & p1.evnt = 3 : a;
            1 : -1;
        esac;
    ver :=
        case
            prc = 0 & p0.evnt = 0 : 0;
            prc = 0 & p0.evnt = 1 : 0;
            prc = 0 & p0.evnt = 2 : 1;
            prc = 0 & p0.evnt = 3 : 2;
            prc = 1 & p1.evnt = 0 : 1;
            prc = 1 & p1.evnt = 1 : 0;
            prc = 1 & p1.evnt = 2 : 0;
            prc = 1 & p1.evnt = 3 : 2;
            1 : -1;
        esac;
```

```
val :=
    case
        vid = a & ver = 0 : 4;
        vid = a & ver = 1 : 7;
        vid = a & ver = 2 : 11;
        vid = b & ver = 0 : 3;
        1 : na;
```

The variables `evnt`, `op`, `vid` and `ver` define the information on an event just executed before a specific state. The `val` defines the value just accessed. We declared all constant symbols used in `CONSTANTS` section

Now we are equipped to express our specifications in LTL/CTL. We formulate the specifications that the value of variable a remains 4 or 7 before variable b being accessed and that event 1 of process 1 always happens before event 3 of process 0 as

```
LTLSPEC F((a=4 | a=7)
          & X F(vid=b))
LTLSPEC F(run=1 & p1.evnt=1
          & X F(run=0 & p0.evnt=3))
```

in the `main` module. At last we can use NuSMV to verify whether the SYN-sequence given in Figure 3.3 satisfies our specifications.

Here we use the linear-time temporal logic (LTL) syntax of NuSMV, where $\neg,\wedge,\vee,\rightarrow$ are presented as `!`, `&`, `|`, `->`, respectively. Note that Boolean, CTL and some other specification formats can be used in NuSMV. Since the variables p0.evnt and p1.evnt keep the information that which event is lastly executed in process 0 and 1, respectively, run=0 & p0.evnt=3 can stand for event 3 of process 0 is executed just before that state. These specifications are obviously false in the given SYN-sequence since event 3 of process 0 and event 1 of process 1 are parallel so any one of these events may happen before another one. Running NuSMV returns “false” and gives a counterexample for each of these specifications.

5.2 Stateless Model Checking on SYN-sequence

Traditionally, state space explosion is one of the major problems in model checking. A lot of efforts were proposed to reduce the state space; one of such is the stateless method. The stateless model checking proposed here is somehow different from that proposed in [64]. We rely on the traditional state space exploring technique, but we improve the modeling strategy to arrive at a more compact transition system for the SYN-sequence. This is possible for stateless specifications, i.e. ones that involves only the relations between executions of events, e.g. the value 4 will be read trice from the variable b after writing 7 to variable a, or event 3 of process 1 always happens before event 3 of process 0. The details are discussed in 4.1. In fact, we can regard the node in the partial order graph as a state that keeps the information about the executing event. Since the values of the share variables are nondeterministic when an event is executed, we ignore these values.

With the concept of stateless method and the aid of partial order graph, we can construct a more compact model. The scheme is shown in Figure 5.2. We use the example in Figure 3.3 to illustrate the model construction procedure (see page 18).

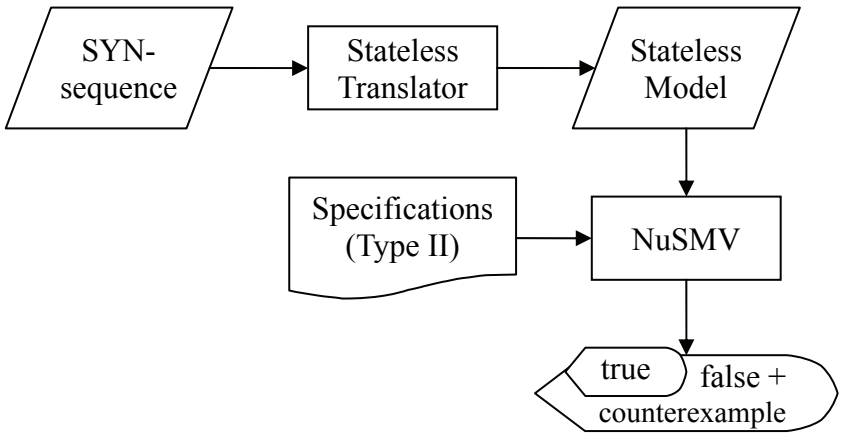


Figure 5.2. The scheme for verification with specification in Type II.

As in section 4.4, we use the variable `run` to indicate which process was just executed, and event -1 and 4 to represent the initial and terminated states respectively. In

the stateless version, the module for each process is quite simple – it simply processes its own events sequentially. The module for process 0 looks as follows:

```
MODULE seq0(run)
    VAR
        evnt : -1..4;
    ASSIGN
        init(evnt) := -1;
        next(run) := 0;
        next(evnt) :=
            case
                evnt = 4: 4;
                1: evnt+1;
            esac;
```

The work of dealing with the dependencies between processes lies on the `main` module. In fact, all the work is done by adding the following NuSMV code in the TRANS section:

```
& (p1.evnt<0 -> next(p0.evnt)<2)
& (p0.evnt<1 -> next(p1.evnt)<0)
& (p0.evnt<3 -> next(p1.evnt)<3)
```

Each case, except for the last default one, expresses a dependency between processes. For instance, the second one says that all transitions to the state where the event of process 0 is less than 2 must transit from the state where the event of process 1 is less than 0. According to this constraint, event 2 of process 0 can be executed only when event 0 of process 1 has been executed. This in turn required that event 1 of process 0 has been executed, according the first case. These constraints correspond the dependencies of this SYN-sequence concisely. The other part of TRANS section is the Progress Constraints used in stateful model (see page 44).

Other information about each event can be encoded in the NuSMV code using the DEFINEs. As mentioned before, all symbols introduced in the `DEFINE` section will not enlarge the state space of the model. The operation of each event, along with the variable and version number accessed in that event, can be determined as long as the event is determined. Hence we add the same Stateless Event Properties as what we used in

stateful models in the `main` module (see page 44). Then the values of share variables accessed in the lastly executed event can be determined according to the variable identifiers and versions. This is done by adding

```
val :=
    case
        vid = a & ver = 0 : 4;
        vid = a & ver = 1 : 7;
        vid = a & ver = 2 : 11;
        vid = b & ver = 0 : 3;
    esac;
```

in the `DEFINE` section.

5.3 Direct Verification on Partial Order Graph

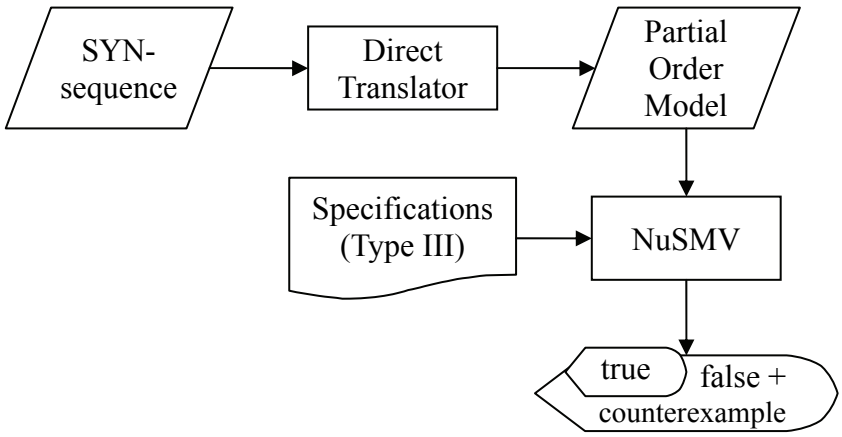


Figure 5.3. The scheme for verification with specification in Type III.

Since we already have the partial order graph of SYN-sequence from the dynamic testing, we may want to verify the specifications directly on the graph. We attempt to model the SYN-sequences directly as the partial order graph without transforming it into state-transition diagram. This special type of model is said to be *partial order model* . Here gives the entire partial model for the example in Figure 3.3:

```
MODULE main
    CONSTANTS R, W, r, w, na, a, b;
    DEFINE
        prc :=
            case
                eid < 4 : 0;
                1 : 1;
```

```
        esac;
    evnt :=
        case
            eid = 0 : 0;
            eid = 1 : 1;
            eid = 2 : 2;
            eid = 3 : 3;
            eid = 4 : 0;
            eid = 5 : 1;
            eid = 6 : 2;
            eid = 7 : 3;
        esac;
                    ⋮
VAR
    eid : 0..7;
ASSIGN
    init(eid) := {0};
    next(eid) :=
        case
            eid = 0 : {1};
            eid = 1 : {2,4};
            eid = 2 : {3};
            eid = 3 : {7};
            eid = 4 : {5,2};
            eid = 5 : {6};
            eid = 6 : {7};
            eid = 7 : {7};
        esac;
```

The remaining of `TRANS` section is similar to the one used in general model (see Stateless Event Properties on page 44) except that we use prc = *i* & evnt = *j* instead of prc = *i* & p*i*.evnt = *j*.

6. Implementation and Experiments

We implemented the SYN-verifier which can translate the SYN-sequence into specific model and call the NuSMV to verify automatically. We designed two SYN-sequence, say SYN-sequences 1 and 2 (or $S1$ and $S2$), for the purpose of experiments. The partial order graph of $S1$ is shown in Figure 6.1, while $S2$ is the previous example as in Figure 3.3.

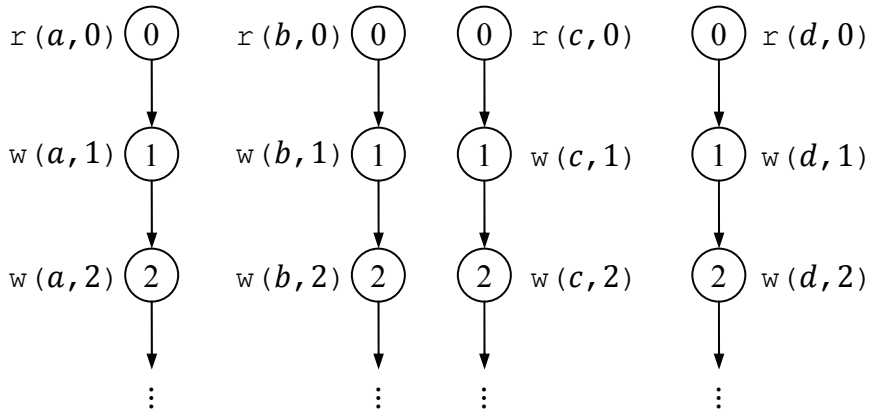


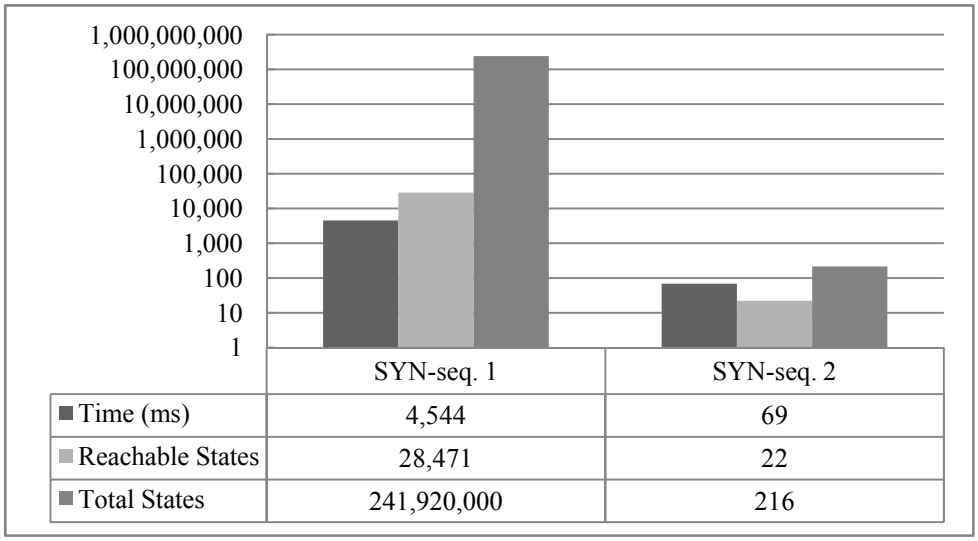
Figure 6.1. The partial order graph of SYN-sequence 1.

The first experiment tests the effective of general specifications which involves both stateful and stateless properties. The specification to be verified is expressed as $\mathrm{F}(\mathrm{w}(a,1) \wedge \mathrm{X}(a = 1 \vee a = 2)$, which means the operation of writing 1 to a immediately happens before certain operation after which the value of a is 1 or 2. The second experiment verifies the stateless specification on both stateful and stateless model and compares the performance. We use the specification $\mathrm{F}(\mathrm{r}(a,0) \wedge \mathrm{X}\,(op = r \vee \mathrm{w}(a,1))$ to express that any read operation on a of value 0 immediately happens before read operation or the operation of writing 1 to a. The third experiment compares all three models. Since only specifications in Type 3 can be verified on partial order model, we express the specification that the operation of writing 2 to a happens before all read operations on b as $\mathrm{G}\neg\big(\mathrm{F}\,\mathrm{w}(a,2) \wedge \mathrm{F}\,\mathrm{r}(b)\big)$. Note that we must make sure that

both operations are existent, since this is unable to be verified on the partial order model of this SYN-sequence. The stateless specification for stateful and stateless models is somehow easier to be expressed in Type II; i.e. $\mathrm{EF}\big(\mathrm{EX}\,\mathrm{w}(a,2) \wedge \mathrm{EX}\,\mathrm{r}(b)\big)$. To show the benefit of partial order model, the last experiment verifies SYN-sequence 1 with the happened before specification. For stateful and stateless models the specification is expressed as

$$\mathrm{F}\,\mathrm{r}(a,0) \wedge \mathrm{F}\,\mathrm{w}(a,1) \wedge \mathrm{G}\neg\big(\mathrm{w}(a,1) \wedge \mathrm{F}\,\mathrm{r}(a,0)\big)$$

in Type II while for partial order model the specification is expressed as $\mathrm{G}\big(\mathrm{r}(a,0) \rightarrow \mathrm{XF}\,\mathrm{w}(a,1)\big)$. Note that the later formula is *smaller* than the former one and is not complete since this model has isolate components (see 4.4).



	SYN-seq. 1	SYN-seq. 2
Time (ms)	4,544	69
Reachable States	28,471	22
Total States	241,920,000	216

Figure 6.2. The result of the first experiment.

The result of experiment 1 is shown in Figure 6.2. The numbers of events of the processes in SYN-sequence 1 are 5, 10, 10, 10, respectively. Since SYN-sequence 1 is highly parallel and involve lots of variable updates, the state space is quite large. However, since the size of the specification is not large, the execution time is feasible.

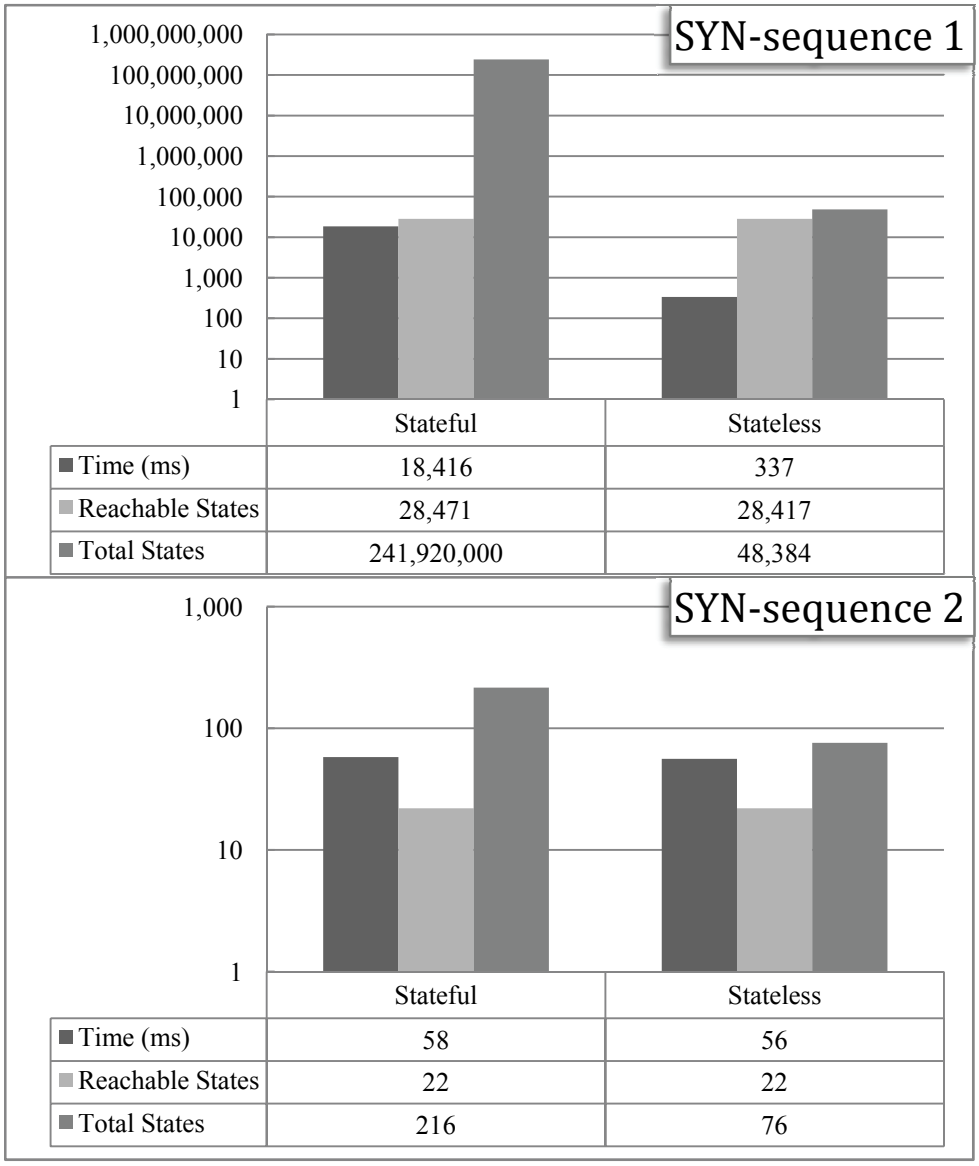


Figure 6.3. The result of the second experiment.

Figure 6.3 gives the result of experiment 2. The numbers of events of the processes in SYN-sequence 1 are also 5, 10, 10, 10, respectively. The stateless specification is in reality much larger than the one in experiment 1, and hence is not so feasible in stateful model of SYN-sequence 1. As we expected, it performance much better on stateless model. Therefore stateless model is the better choice for stateless specifications. Note that when state space size is small such as that of SYN-sequence 2, both models performs well.

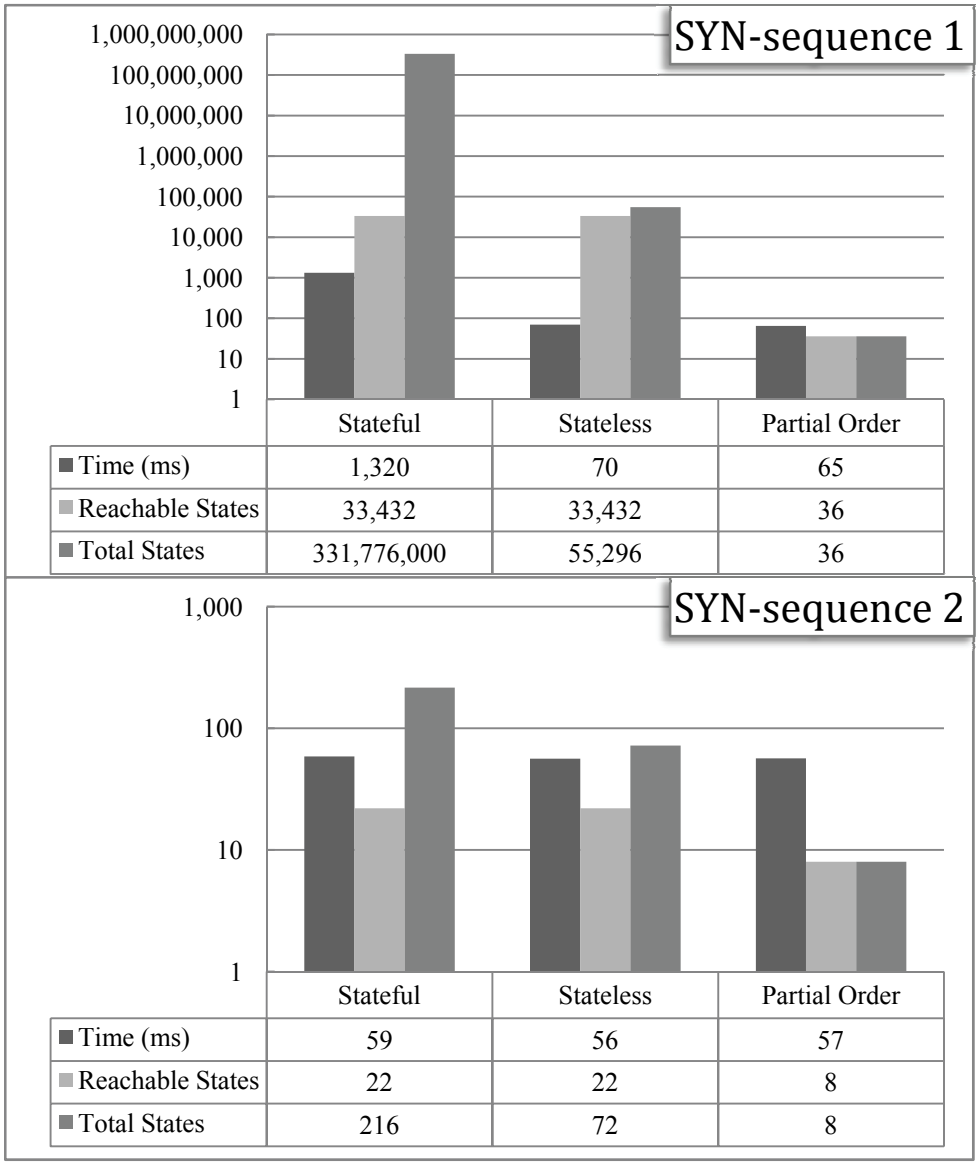
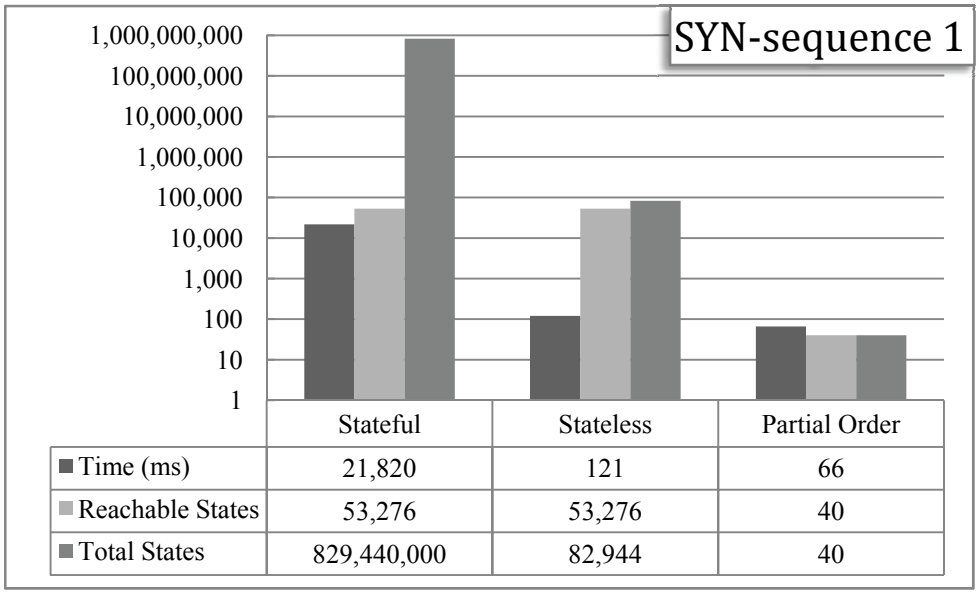


Figure 6.4. The result of the third experiment.

In Figure 6.4 we can see the comparison between all three models. The numbers of events of the processes in SYN-sequence 1 are enlarged to 6, 10, 10, 10, respectively. For both SYN-sequences, the performance of stateless model is much better than that of stateful one. Nevertheless, since the size of specifications in Type III is larger than stateless specifications, the performance of partial order model is not guaranteed to be better than that of stateless model. For instance, verification of SYN-sequence 2 on partial order model may sometimes slower than that on stateless one. However, when state space of a SYN sequence is too large, partial order model may take advantage of

performance. See Figure 6.5 for instance. The number of events of each process in SYN-sequence 1 is enlarged to 10 respectively. The size of specification in Type III is slightly smaller than that of specification in Type II.



	Stateful	Stateless	Partial Order
Time (ms)	21,820	121	66
Reachable States	53,276	53,276	40
Total States	829,440,000	82,944	40

Figure 6.5. The result of the fourth experiment.

7. Conclusions

In this thesis, we propose a scheme to verify the SYN-sequence generated in dynamic testing of concurrent software and provide the theoretic fundations. We first translate the SYN-sequence into the model of reachability graph or partial order graph. Then the model as well as the specifications a feed into the NuSMV model checker. If the model satisfies the specifications, the NuSMV will return true and we are confident that the SYN-sequence also satisfies the specifications. If the model doesn't satisfy the specifications, the NuSMV will report the falsity along with an counterexample. In this case, users can debug or modify the concurrent programs according to this counterexample.

We can apply one out of three different models, i.e. stateful or general model, stateless model and partial order model. It depends on the type of specifications and the problem size. If the specifications are stateful, they can be expressed in Type I and only stateful model can be used. Otherwise, specifications involving stateless properties can expressed in Type II or III. Specifications in Type II, or stateless specifications, can be verified on both stateful and stateless models. Even so, stateless model is better choice since the better performance in large problem size. Specifications in Type III are more difficult to be expressed and can be verified only on partial order model. It request the user to be equipped with entire knowledge to write the correct expression. Unfortunately, when the problem size is too large such as to confront the state space explosion, only the partial order model can be adopted. None the less, since the size of specifications in Type III is usually larger than stateless specifications, the performance of partial order model is not guaranteed to be always better than that of stateless model.

The experimental result shows that a systematic verification of SYN-sequence is feasible.

Reference

REFERENCES

[1] Charles E Mcdowell and David P. Helmold, “Debugging Concurrent Programs,” ACM Computing Surveys, Volume 21, Issue 4, December 1989.

[2] K.C. Tai and Richard H. Carver, “Testing of Distributed Programs,” Chapter 33 in Parallel and Distributed Computing Handbook, editor A. Y. Zomaya, McGraw-Hill, 1996.

[3] Anne Dinning, “A Survey of Synchronization Methods for Parallel Computers,” IEEE Computer, July 1989.

[4] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne, “Operating System Concepts,” John Wiley & Sons, ISBN: 0471417432, 6th edition (June 26, 2001).

[5] D. Helmbold and D. Luckham, “Debugging Ada tasking programs,” IEEE Software, Volume 2, Number 2, 66-74, 1985.

[6] K. C. Tai, “Testing of concurrent software,” Proc. of the 13th Annual International Computer Software and Applications Conference, pp. 62-64, 1989.

[7] Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled, Model Checking, MIT Press, 1999, ISBN 0-262-03270-8.

[8] Logic in Computer Science: Modelling and Reasoning About Systems, Michael Huth and Mark Ryan, Cambridge University Press, 2004.

[9] Holzmann, G.: The model checker SPIN. IEEE Trans. Software Engineering 23(5) (1997) 279–295

[10] Musuvathi, M., Park, D., Chou, A., Engler, D., L. Dill, D.: CMC: A pragmatic approach to model checking real code. In: OSDI 02: Operating Systems Design and Implementation (2002) 75–88

[11] Java PathFinder, http://javapathfinder.sourceforge.net/.

[12] MAGIC website, http://www.cs.cmu.edu/~chaki/magic.

[13] S. Chaki, J. Ouaknine, K. Yorav, and E. M. Clarke, “Automated compositional abstraction refinement for concurrent C programs,” In Proceedings of SoftMC 03. ENTCS 89(3), 2003.

[14] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky, “Validation, Verification, and Testing of Computer Software,” ACM Computing Surveys, Volume 14, Issue 2, June 1982.

[15] Roger S. Pressman, Software Engineering (A practitioner's approach), 5th edition, 2000, Mc Graw-Hill Education, ISBN 978-0071181822.

[16] Gwan-Hwan Hwang, “A Systematic Parallel Testing Method for Concurrent Programs. Master Thesis,” Institute of Computer Science and Information Engineer, National Chiao-Tung University, Taiwan, 1993.

[17] G. H. Hwang, K. C. Tai, and T.L. Huang, “Reachability Testing: An Approach To Testing Concurrent Software,” International Journal of Software Engineering and Knowledge Eng., 5, 4, (Dec. 1995), 493-510.

[18] Gwan-Hwan Hwang and Che-Sheng Lin, “Dynamic Effective Testing: An Approach to Testing Concurrent Programs,” Technical Report, National Taiwan Normal University, 2008. http://www.csie.ntnu.edu.tw/~ghhwang/ DET2008.pdf.

[19] Thomas J. LeBlanc and John M. Mellor-Crummey, “Debugging Parallel Programs with Instant Replay,” IEEE Transactions on Computers, C-36(4), pp. 471-482, April 1987.

[20] Che-Sheng Lin and Gwan-Hwan Hwang, “Dynamic Termination Decision for Concurrent Programs with Busy-Waiting Loops,” Submitted to IEEE International Conference on Software Testing, Verification, and Validation, 2009 for publication.

[21] NuSMV, http://nusmv.irst.itc.it/.

[22] Stoller SD. Testing concurrent Java programs using randomized scheduling. Proceedings of the 2nd Workshop on Runtime Verification (RV) (Electronic Notes in Theoretical Computer Science, vol. 70(4)). Elsevier: Amsterdam, 2002.

[23] Y. Ben-Asher, E. Farchi, Y. Eytani, “Heuristics for finding concurrent bugs,” Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003) PADTAD Workshop, April 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003.

[24] Edelstein O, Farchi E, Nir Y, Ratsaby G, Ur S. Multithreaded Java program test generation. IBM Systems Journal 2002; 41(1):111–125. Available at: http://www.research.ibm.com/journal/sj/411/edelstein.html.

[25] Gwan-Hwan Hwang, Sheng-Jen Chang, and Huey-Der Chu, “Technology for Testing Nondeterministic Client/Server Database Applications,” IEEE Transaction on Software Engineering, Volume 30, Number 1, pp. 59-77, Jan., 2004.

[26] Kuo-Chung Tai, “Reachability Testing of Asynchronous Message-passing Programs,” Proceedings of the second International Workshop on Software Engineering for Parallel and Distributed Systems, 1997.

[27] Yu Lei and Kuo-Chung Tai, “Efficient Reachability Testing of Asynchronous Message-Passing Programs,” Proc. 8th IEEE Intl. Conf. on Engineering for Complex Computer Systems, pp. 35-44, Dec. 2002.

[28] Richard H. Carver and Yu Lei, “A General Model for Reachability Testing of Concurrent Programs,” ICFEM 2004, LNCS 3308, pp. 76-89, 2004.

[29] Yu Lei and Richard H. Carver, "Reachability Testing of Concurrent Programs," IEEE Transaction on Software Engineering, June 2006 (Vol. 32, No. 6), pp. 382-403.

[30] Yu Lei, Richard H. Carver, Raghu Kacker, and David Kung, "A combinatorial testing strategy for concurrent programs," Software Testing, Verification & Reliability ,Volume 17 , Issue 4 (December 2007) pages 207-225.

[31] Koushik Sen, Gul Agha, "Automated Systematic Testing of Open Distributed Programs," Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, pp. 339-356.

[32] Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. on Foundations of Computer Science. (1977) 46–57

[33] N. Rescher, A.U.: Temporal Logic. Springer (1971)

[34] Goldblatt, R.: Logic of time and computation. Technical report, CSLI Lecture Notes, no.7, Stanford University (1987)

[35] Kamp, J.: Tense Logic and the Theory of Order. PhD thesis, UCLA (1968)

[36] Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Logics of Programs. Volume 193 of Lecture Notes in Computer Science., Springer (1985) 196–218

[37] Markey, N.: Temporal logic with past is exponentially more succinct. EATCS Bulletin 79 (2003) 122–128

[38] Vardi, M.: A temporal fixpoint calculus. In: Proc. 15th ACM Symp. on Principles of Programming Languages. (1988) 250–259

[39] Keller, R.: Formal verification of parallel programs. Communications of the ACM 19 (1976) 371–384

[40] Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press (2002)

[41] Pratt, V.: A near-optimal method for reasoning about action. Journal of Computer and Systems Science 20(2) (1980) 231–254

[42] Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proc. Workshop on Logic of Programs. Volume 131 of Lecture Notes in Computer Science., Springer (1981) 52–71

[43] Queille, J., Sifakis, J.: Specification and verification of concurrent systems in Cesar. In: Proc. 9th ACM Symp. on Principles of Programming Languages. Volume 137 of Lecture Notes in Computer Science., Springer (1982) 337–351

[44] Ben-Ari, M., Manna, Z., Pnueli, A.: The logic of nexttime. In: Proc. 8th ACM Symp. on Principles of Programming Languages. (1981) 164–176

[45] Lamport, L.: “Sometimes” is sometimes “not never” - on the temporal logic of programs. In: Proc. 7th ACM Symp. on Principles of Programming Languages. (1980) 174–185

[46] Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In: Proc. 10th ACM Symp. on Principles of Programming Languages. (1983) 117–126

[47] Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languagues and Systems 8(2) (1986) 244–263

[48] Clarke, E., Grumberg, O.: Avoiding the state explosion problem in temporal logic model-checking algorithms. In: Proc. 16th ACM Symp. on Principles of Distributed Computing. (1987) 294–303

[49] Browne, M., Clarke, E., Dill, D., Mishra, B.: Automatic verification of sequential circuits using temporal logic. IEEE Transactions on Computing C-35 (1986) 1035–1044

[50] Clarke, E., Mishra, B.: Hierarchical verification of asynchronous circuits using temporal logic. Theoretical Computer Science 38 (1985) 269–291

[51] Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. In: Proc. 5th IEEE Symp. on Logic in Computer Science. (1990) 428–439

[52] Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. Information and Computation 98(2) (1992) 142–170

[53] McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers (1993)

[54] Clarke, E.: The birth of model checking. In: 25 Years of Model Checking. Volume 5000 of Lecture Notes in Computer Science., Springer (2008) 1–26

[55] Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Proc. 12th ACM Symp. on Principles of Programming Languages. (1985) 97–107

[56] CADP, http://www.inrialpes.fr/vasy/cadp/

[57] CHESS, http://research.microsoft.com/en-us/projects/CHESS/.

[58] ISP (In-situ Partial Order), http://www.cs.utah.edu/formal_verification/ISP-release/

[59] Spin – Formal Verification, http://spinroot.com/spin/whatispin.html

[60] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic

Model Checking. In: Computer Aided Verification. Volume 2404 of Lecture Notes in Computer Science., Springer (2002) 241–268

[61] H.Carver, R., Tai, K.-C.: Modern multithreading. John Wiley & Sons (2006)

[62] Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng, SE.3, 7 (1977) 125–143.

[63] Cavada, R., Cimatti, A., A. Jochim, C., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: NuSMV 2.4 User Manual. ITC-irst (2005)

[64] Godefroid, P.: Model Checking for Programming Languages using VeriSoft. In: Proc. 24th ACM Symp. on Principles of Programming Languages. (1997) 174–186