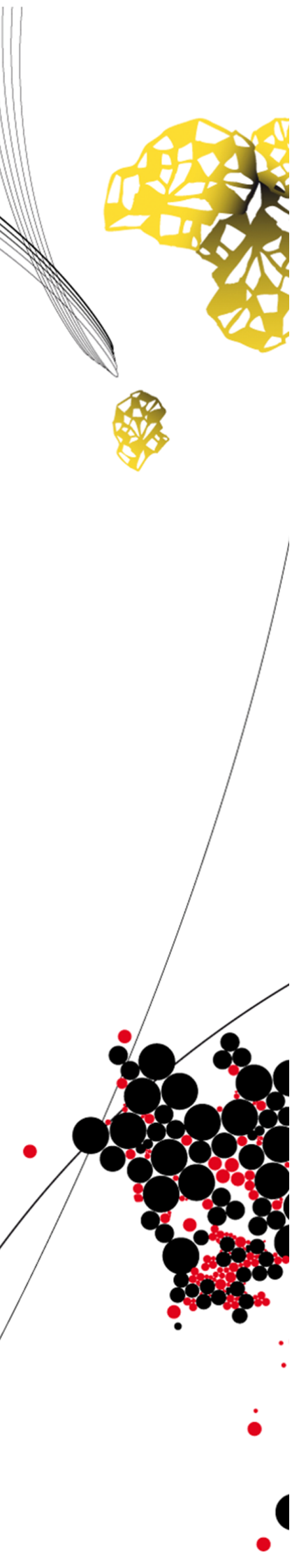


UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

ADA Software Model Checking

Ramesh Krishnamurthy

M.Sc. Thesis
April 2021

Supervisors:
prof. dr. Marieke Huisman
dr. Raúl Monti
dr. Marco Gerards
dr. Jacques Verriet
dr. Yonghui Li

Formal Methods and Tools
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
7500 AE Enschede
The Netherlands

Abstract

The increase in software systems' complexity in recent years requires rigorous validation of these systems to improve their reliability. In particular, failures in industrial software may affect production and revenue. Therefore, improved verification techniques that ensure a high degree of confidence in the software are essential.

This thesis aims at improving the reliability of software systems whose implementation is based on the Ada programming language. More specifically, model checking is applied to verify Ada programs, with a particular focus on concurrency problems. The work is meant as an experience report for future researchers to automatically generate a formal model (e.g. a Promela model) to verify industrial software.

Model checking is an advanced technique that can provide a greater level of assurance about system correctness. However, applying this technique to software systems faces challenges like model generation and state space explosion; developing techniques to overcome these problems is ongoing research.

A toolset is presented in this thesis to address the problems of (1) creating formal models from Ada programs, (2) developing abstraction techniques to reduce the state space, (3) generating traceability from model to code, and (4) proving the presence/absence of concurrency errors in the modeled programs.

Firstly, the manual construction of models that specify the behavior of industrial software systems is a time-consuming, complex, and error-prone process due to the complexity of these systems. To overcome this problem, the toolset automatically generates models from Ada programs. Secondly, the generated model should undergo a complete verification without memory or timeout issues. The toolset includes an algorithm to address this scalability issue. Thirdly, any violation detected by the model checker should be traceable back to the source code. An algorithm that automatically achieves model to code traceability is built into the tool set to address this challenge. Finally, the generated model with reduced state space should indicate concurrency error(s) in the model after verification. A combination of compositional verification and the model checker's built-in proof approximation techniques are used to conclude the presence or absence of such errors.

The toolset is evaluated on one specific real-life software system. The toolset is successful in automatically generating a scalable model of the software system. The algorithm developed to address state explosion could reduce the state space by 75% compared to the original model. Along with model generation, the tool can generate a mapping from model to code automatically. Finally, breaking up the model into groups of related tasks (divide and conquer) and proof approximation techniques provides a tractable verification result.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1 INTRODUCTION

Concurrency is the interleaved or parallel execution of two or more independent, interacting programs over the same period. A concurrent system is implemented using processes and/or threads. To achieve a program's objective, threads must communicate and coordinate with other threads. For a thread to communicate at the right time, it should synchronize with another thread, thereby arriving together at an agreed-upon control point. The interaction of parallel threads is complex and is difficult to verify mainly due to the following reasons:

- multiple threads can interact in unexpected ways and a task can communicate across several sequences of instructions executing asynchronously
- the threads in a concurrent system block or wait for an external event

These concurrent programming challenges make the detection of deadlock, livelock, and race conditions a challenge [Bri+10]. A concurrent program P is said to have a deadlock if P can reach a state such that some process in P is blocked in this state and remains blocked forever [Tai94]. A concurrent program P is said to have a race condition when two or more processes attempt to modify a shared data at the same time without mutual exclusion. The checks for deadlock freedom and race condition freedom are safety properties. A concurrent program P is said to have a livelock if P can reach a state such that after entering this state, some process never becomes deadlocked or terminated, but does not make progress [Tai94]. The check for livelock freedom is a liveness property.

The majority of the programming languages have built-in concepts like locks and semaphores to avoid concurrency problems like deadlock and race conditions. Although these techniques are beneficial while writing a concurrent program, it does not mean that concurrency problems cannot occur. To deduce that a given concurrent program behaves as expected, the classical approach is to do testing. Tests can be written with different inputs, but it is not possible to try every possible input. It is also not possible to test every possible thread interleaving on every possible test input. Therefore, testing can detect errors, but it cannot confirm the absence of errors. A better verification technique is needed to verify every possible execution of the software to find errors that cannot be found through testing. Model checking is one such technique.

Model checking is a formal verification technique that verifies every possible execution sequence. Generally, a model checker does not verify the original system but a model of the system. The requirements of the system are specified as properties of the model during verification. Formal verification of such a model then means verifying the original system.

The work presented in this thesis uses model checking to verify the absence of deadlock and absence of livelock in a pick-and-place machine's software written in the Ada programming language. Nexperia's Equipment & Automation Technologies (E&A) department develops pick-and-place machines for the production of semiconductor products, such as transistors, diodes, and ICs. The Bright project is a co-operation between Nexperia and ESI (TNO), applying model-based systems engineering for Nexperia's pick-and-place machines as the industrial carrying case. One of the topics being addressed in the Bright project is modeling and analysis of the

software of the pick-and-place machines. The software has more than one million lines of code. As in a typical concurrent software, the control and co-ordination of the pick-and-place machines' software's parallel tasks are realized using signals and locks. A large number of synchronizations between these parallel tasks result in complex code-level parallelism. This makes it difficult to verify and validate all possible control flows of the code for concurrency problems.

1.1 Motivation

Model checking a real-life concurrent program has been a challenge mainly due to state space explosion problem. Even if this problem is overcome, there is another challenge of establishing a complete traceability from the model back to the program. This traceability is essential to trace detected violations in the model back to the program.

Quasar [Eva+03b], Ada Translating Toolset [DPC98] and ATOS [FMP12] are the earlier methods that use model checking to verify concurrent Ada programs. We will discuss each of these approaches and their limitations in Chapter 4.

Besides simple examples, there is no evidence of using any of these works to verify a real-life concurrent Ada software. This means that as Ada programs' complexity increases, it may not be guaranteed to obtain: (a) a scalable model that does not face state explosion problems and (b) a model that faithfully represents the Ada program. Furthermore, the earlier works do not consider the need for traceability between the input Ada program and the output model. This consequently makes it very hard to map error traces from the model back to the Ada program.

1.2 Problem Statement

Is it possible to automatically verify a real-life concurrent Ada software using model checking by a software developer with little or no knowledge of model checking?

Here, the mentioned real-life Ada software is of a large size, say, over one million code lines and with many synchronizations.

Based on the above problem statement, the following are the goals of this work.

1. Automatically generate a formal model of a software system implemented in Ada
2. Automatically apply state space reduction techniques on the generated model
3. Automatically apply model checking to the generated model to prove the presence/absence of concurrency problems in the code's control and co-ordination flow
4. Automatically generate traceability from model to code
5. Evaluate the level of model checking knowledge required for a software developer to verify his/her software using our work.

1.3 An Overview of the Approach

The approach presented in this thesis is based on a toolset that automatically translates Ada programs to formal models. Besides generating formal models, the toolset controls state explosion and generates automatic traceability from code to model. In summary, the generated model can be fed directly to the model checking tool (without any user intervention) to complete

verification. In the case of property violation, the generated traceability is used to trace the violation back to the source code.

Figure 1.1 shows the overall approach used. In stage-1, the Extraction code (not a part of this thesis) is used to parse the source code and retrieve the information needed using an Abstract Syntax Tree (AST) of the source code. The retrieved information is represented in an XML file format. This XML contains only parts of the code that potentially lead to concurrency problems, along with the control flow leading to these parts. The parts of the code that potentially lead to concurrency problems include the signals and locks that implement the software's parallel tasks' control and co-ordination.

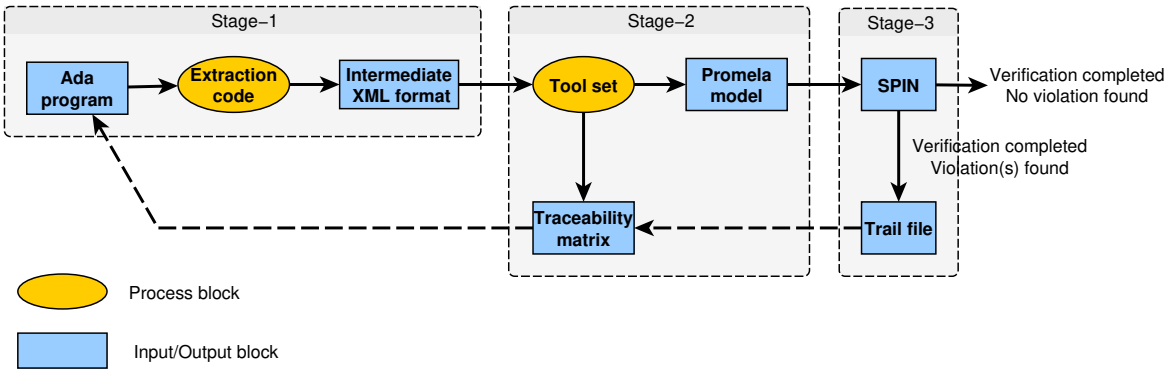


Figure 1.1: Approach

In stage-2, the XML with the extracted information is used as input to the toolset to generate a formal model. The toolset generates a output formal model in Promela, the SPIN model checker's modeling language. SPIN is a model checking tool [Hol04] designed to verify the correctness of formal models of concurrent systems. The toolset retains Ada code constructs' behavior and the code control flow in the generated model. This ensures that the model retains the properties of the system to be verified. Furthermore, the toolset has a built-in algorithm that reduces the state space of the generated model such that the model can complete verification without state space problems.

In stage-3, the generated Promela model is input to the SPIN model checker for verification. If the verification returns any violation, the model's path leading to the violation is part of a trail file generated by the model checker. A mapping is required between the model and the code to trace this path back to the source code. To achieve this, the toolset has a mechanism to automatically generate a traceability matrix that maps the model's code lines and transitions to the source code lines and control flow, respectively.

The most important contributions in this work are:

1. A model generation mechanism to automatically generate a Promela model from the intermediate XML
2. A method to automatically apply state space reduction techniques on the generated model
3. A method to automatically generate traceability from model to code
4. Continuous feedback and validation to generate the intermediate XML

1.4 Thesis Overview

The rest of this thesis is organized as follows. Chapter 2 introduces the Ada programming language and its concurrency constructs. Chapter 3 provides the necessary background on model checking, the Promela language, and the SPIN model checker. Chapter 4 describes the related work regarding the formal verification of Ada programs and discusses each work's limitations.

Chapter 5 illustrates the translation of different Ada constructs into corresponding Promela constructs. The background on the Ada language in Chapter 2 and the background on Promela language in Chapter 3 serves as a reference for Chapter 5.

Chapter 6 describes the toolset's logic, explaining how the information in the XML is used to generate a Promela model. This chapter then describes the algorithm used to reduce the state space of the generated model. Finally, this chapter describes the algorithm used to generate the model to code traceability. Chapter 5 and Chapter 6 correspond to stage-2 of Figure 1.1.

Chapter 7 explains the verification of the generated model, the test sets, and the outcome of verification supported by metrics on state space, execution time, and the verification results. Chapter 7 corresponds to stage-3 of Figure 1.1. Chapter 8 gives conclusions and provide suggestions for future work.

2 ADA AND CONCURRENCY IN ADA

This chapter introduces the Ada programming language, its concurrency features, and the problems in Ada concurrency.

The chapter starts with an overview of Ada's origin and evolution, followed by a brief mention of its language features. Next, the concurrency constructs supported by Ada are discussed in detail with supporting examples. Finally, we discuss different concurrency problems that can occur in Ada with supporting examples.

2.1 History and Standardization of Ada

The Ada programming language was developed at the request of the United States Department of Defense. The aim was to use Ada as the universal programming language for military systems. The programming language quickly expanded to safety-critical systems in general. Ada's use in critical systems development increased gradually due to its careful and safe design and the existence of clear guidelines for building such a system [Bar08].

The first standard of the Ada language was published in 1983 and therefore denominated as Ada 83. Ada's next version appeared in 1995, called Ada 95, bringing many enhancements like object-oriented programming, synchronization, etc. Subsequently, Ada 2005 provided synchronized interfaces. The latest is Ada 2012, which supports contract-based programming. A detailed comparison between different Ada standards can be found at [Ada20a].

2.2 Language Features

Ada is a structured programming language that makes extensive use of the control flow constructs like if/then/else (selection constructs) and while/for (repetition constructs) to improve the program's clarity. Ada has strict typing rules (i.e., strongly typed) that helps to catch errors and exceptions during compile time. The language supports modular programming by separating programs into independent packages such that each package contains what is necessary to execute a specific functionality. Ada supports run-time checking to analyze and report defects during execution like exceptions, memory leaks, null pointers, etc. The support for object-oriented programming was added in Ada 95.

In the context of this work, we will mainly focus on Ada's concurrency features. These features include the support for task, synchronous message passing, non-deterministic select statements, and protected types. We will discuss these in detail in the subsequent sections. For exhaustive information on Ada language features, we refer to [Ada20b].

2.3 Concurrency in Ada

In this section, we will look into the different concurrency constructs supported by Ada. We first start with the fundamental concurrent unit in Ada, *task*. We then proceed to communication between Ada tasks using synchronization by explaining the rendezvous and selective rendezvous constructs. Finally, we will discuss the protected types that support mutual exclusion. Each construct is illustrated with supporting examples.

2.3.1 Task

Ada supports task-based concurrency. Ada tasks represent different threads of control, which execute independently and concurrently. An Ada task may contain several interaction points, which allow communication with other tasks. Beyond these interaction points, tasks can interact with other tasks in many ways (e.g., through shared variables).

Listing 2.1: An Ada task

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Main is
3    task T;
4    task body T is
5    begin
6      Put_Line ("Inside task T");
7    end T;
8  begin
9    Put_Line ("Inside main");
10 end Main;
```

Listing 2.1 shows a simple Ada task `T`, declared using the keyword `task` as in line-3. The implementation of a task is specified within a `task body` block as in line-4. `Ada.Text_IO` at line-1 is a unit of the Ada's predefined language environment; `Text_IO` is the package used for simple Input/Output (I/O) in text format.

When the procedure `Main` starts at line-8, task `T` starts automatically. When the program is executed, the print messages at both lines, line-9 and line-6, are called. The procedure `Main` can be seen as the main task which terminates after its subtask `T` finishes. The main task's waiting process provides synchronization between the main task and its subtask `T`. This is the simplest form of synchronization in Ada.

2.3.2 Rendezvous

Communication between two Ada tasks can be achieved through an entry call. A called task executes the entry call, which is suspended until this call is completed. A calling task accepts this entry call using an `accept` statement. This accept statement represents the interaction point of a task. In summary, the entry call and the accept statement execute as a pair. The Ada rendezvous synchronization takes place using this pair.

Ada rendezvous's basic principle is that the first task to reach the "rendezvous point" will wait for the other task to make the communication. In other words, when a task calls, the entry call is suspended until the communication is finished.

Listing 2.2 shows an example of Ada rendezvous. The entry `Entry1` declared at line-4 is the external interface to the task `Called`. The task's body has an accept statement to receive `Entry1` as in line-8. The main procedure starts at line-11, and its body has the interaction point between

the task `Called` and the task `Main`, i.e., `Called.Entry1` at line-13. To understand the control flow between the main task and its subtask, print statements are added at line-9, line-12, and line-14.

Listing 2.2: Ada rendezvous

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Main is
3    task Called is
4      entry Entry1;
5    end Called;
6    task body Called is
7    begin
8      accept Entry1;
9      Put_Line ("Inside called task");
10   end Called;
11 begin
12   Put_Line ("Inside main");
13   Called.Entry1;
14   Put_Line ("After entry call");
15 end Main;
```

When the main task starts at line-11, the task `Called` is also started (at line-7), and the main task is suspended. When the control reaches the accept statement at line-8, the task `Called` is suspended, and the main task is awakened to call the entry `Entry1` at line-13. So far, the message at line-12 is printed, and the entry at line-13 is called. Since the task `Called` is suspended on its accept statement for this entry, the call succeeds. After this rendezvous, the main task continues its execution to print the message at line-14. Before the main task ends, it checks for any pending statements of the subtask to be executed. Therefore, the main task is suspended to print the message at line-9 of the subtask. After that, the subtask terminates at line-10, and subsequently, the main task terminates at line 15.

2.3.3 Selective Rendezvous

The Ada accept statement will wait to receive a single entry call. If more than one task needs to *service* a waiting accept statement, then this is implemented using the Ada `select` statement. The Ada selective accept statement allows for the non-deterministic selection of one of the multiple alternatives. These alternatives can be *accept*, *terminate*, *delay* or a combination of them.

Listing 2.3 shows an example of Ada's selective accept. Either a call to `Entry1` at line-9 or a call to `Entry2` at line-13 will be accepted, based on what call arrives first. One of these calls must arrive before the expiry of the delay interval `Duration1` as in line-17. If neither of the entries arrives before the expiry of `Duration1`, then the task `Synchro` is suspended.

2.3.4 Protected types

The protected type in Ada is a structured mechanism that provides mutually exclusive access to shared data. The protected types declare an entry call as a procedure-like operation. An entry call has a queue of tasks waiting for the protected type instance. There is an additional *barrier condition* that must be true for the entry call to succeed. When the barrier condition is satisfied by a task, we say the entry's barrier is closed and other tasks calling this entry will be queued.

Listing 2.3: Ada selective accept

```
task Synchro is
  entry Entry1;
  entry Entry2;
end Synchro;

task body Synchro is
begin
  select
    accept Entry1 do
     -- do something
    end Entry1;
  or
    accept Entry2 do
     -- do something
    end Entry2;
  or
    delay Duration1;
  end select;
end Synchro;
```

Listing 2.4: Ada protected type

```
protected type Semaphore is
  entry Acquire;
  procedure Release;
private
  Acquired : boolean := False;
end Semaphore;

protected body Semaphore is
  entry Acquire when not Acquired is
  begin
    Acquired := True;
  end Acquire;
  procedure Release is
  begin
    Acquired := False;
  end Release;
end Semaphore;

-- some code for task declaration
declare
  S:Semaphore;
begin
  S.Acquire; -- Wait infinitely for the lock to be free
  ...        -- Critical section
  S.Release; -- Release the lock
end;
```

The barrier condition ensures that other tasks cannot access the protected type until the current task using the protected type releases it, thereby preventing a possible race condition.

Listing 2.4 shows the protected type called `Semaphore` with the procedure-like entry call `Acquire`. The boolean variable `Acquired` is the barrier condition that must be true for the entry call to succeed. The entry `Acquire` has a queue of tasks waiting for the semaphore. When a task seizes the semaphore (line-23), `Acquired` is set to true as in line-11. At this point, no other task can access the entry; any other task calling the entry will be queued. After the current task finishes executing the critical section (line-24), it releases the semaphore (line-25). Consequently, `Acquired` is set to False (line-15), which means a waiting task in the queue can now access the entry.

2.4 Problems in Ada Concurrency

In the previous Section 2.3, we discussed Ada's different concurrency constructs. In this section, we will see the problems that can occur while using these concurrency constructs.

The rest of this section is organized as follows. We first discuss how deadlock occurs in Ada programming, along with different deadlock classes supported by examples. We then see how livelock occurs in Ada programming by using an example. Finally, we discuss the third concurrency problem, race condition in Ada program, with an example.

2.4.1 Deadlock in Ada

Deadlock in Ada occurs if there exists a circular dependence between tasks that are in a dead state. A dead state refers to the state in which a task cannot terminate normally. A *normal* termination refers to using, for example, an *abort* statement within the program. An abnormal termination refers to using an external interrupt, such as an operating system command, to terminate the task forcefully.

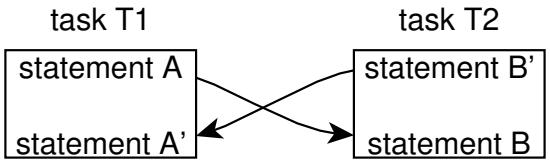


Figure 2.1: A typical deadlock scenario in Ada

A typical example of Ada deadlock is shown in Figure 2.1. Statement A of a task T1 waits for statement B in another task T2 to complete its execution. Statement B in T2 can complete its execution only after statement B'. But, statement B' can execute only if the statement A' (that comes after statement A) in T1 completes its execution. This circular dependence blocks both the tasks.

Deadlock in Ada can occur in several ways. In the following section, we will look into some scenarios [Lev89] that cause a deadlock in the Ada program. The study in [CAU88] and [Che90] provide more details on different deadlock classes of Ada.

1. *Circular-entry-calling deadlock*: This deadlock occurs when there is a closed loop of tasks, such that each task has issued an entry call to the next task in this loop.

Listing 2.5: untimed unconditional task T1

```
1 task body T1 is
2 begin
3  T2.A;
4  accept B;
5 end;
```

Listing 2.6: untimed unconditional task T2

```
1 task body T2 is
2 begin
3  T1.B;
4  accept A;
5 end;
```

In Listing 2.5 task T1 waits at line-3 for entry A from task T2 in Listing 2.6 to continue execution. On the other side, task T2 waits for entry B, as in line-3, from task T1 to continue execution. This circular-entry-calling leads to a deadlock.

2. *Dependence blocking deadlock*: This class of deadlock is a variant of circular-wait deadlock in which an attempted rendezvous prevents the termination of the sub-program, which may indirectly prevent the acceptance of the rendezvous.

Listing 2.7: Dependence blocking task T1

```
1 task body T1 is
2 begin
3  A;
4  accept B;
5 end;
```

Listing 2.8: Dependence blocking task T2

```
1 procedure A is
2  task T2;
3  task body T2 is
4  begin
5   T1.B;
6  end;
7 begin
8  null;
9 end A;
```

In Listing 2.7, task T1 calls procedure A at line-3. In Listing 2.8, Procedure A calls another task T2 (nested call) at line-2. When T2 encounters entry B at line-5, this entry cannot be accepted by T1 since procedure A has not yet terminated. In other words, the entry B at T2 prevents the termination of procedure A, thereby blocking the acceptance of the entry by T1.

3. *Call-wait-deadlock*: This class of deadlock occurs when a task accepts an entry call that is sent by another task, and the other task is calling a different entry of the first task.

Listing 2.9: call-wait-deadlock task T1

```
1 task body T1 is
2 begin
3  accept A;-- no other task calls A
4  accept B;
5 end;
```

Listing 2.10: call-wait-deadlock task T2

```
1 task body T2 is
2 begin
3  T1.B;
4  T1.A;
5 end;
```

In Listing 2.9, task T1 is accepting entry A at line-3 which can be sent only by task T2 as in line-4 of Listing 2.10. To send entry A, T2 should first complete a rendezvous for entry B as in line-3. But, T1 cannot accept entry B before it accepts entry A, leading to a deadlock.

4. *Acceptance deadlock*: In this deadlock class, the task is to accept an entry call whose caller is blocked in a deadlock.

Listing 2.11: acceptance deadlock Task T1

```
1 task body T1 is
2 begin
3  accept A;
4  T2.B;
5 end;
```

Listing 2.12: acceptance deadlock Task T2

```
1 task body T2 is
2 begin
3  accept B;-- no other task calls A or B
4  T1.A;
5 end;
```

In Listing 2.12, when task T2 is accepting the entry call B at line-3, its caller task T1 is blocked to accept A as in line-3 of Listing 2.11.

2.4.2 Livelock in Ada

Livelock refers to two or more processes executing continuously but making no progress towards a final goal. Here the final goal can be normal process termination, statements that are waiting to be executed, etc. Therefore, unlike deadlock, a livelocked system still does some work which may not be useful [HSH05].

Ada tasks in the dead state may continue to execute, perhaps in an infinite loop. However, such tasks cannot progress beyond some set of statements before their termination, leading to a livelock.

We will now discuss an example as to how livelock occurs in Ada. Listing 2.13 shows two tasks, `Send` and `Receive`, defined within the procedure `Main`. The task `Send` uses Ada's *selective accept* construct to either accept the entry E as in line-12 or exit the task with the terminate statement at line-14. This Ada construct is executed repeatedly because of the loop statement at line-10. The other task, `Receive`, repeatedly sends the entry call E as in line-22.

When the procedure `Main` starts execution at line-25, the other two tasks also starts executing in parallel, and the `Main` task is suspended due to the call `Send.E` at line-26. At a parallel instance, the `Receive` task is also suspended due to the same call at line-22. When the control chooses the accept statement at line-12, the task `Send` is suspended. Now, both `Main` and `Receive` (which are in a suspended state) are awakened to send the entry `E` from line-26 and line-22 respectively. Assuming first-in-first-out as the default scheduling, the `Main` procedure succeeds in completing a rendezvous with the task `Send`. The control now resumes to procedure `Main` which checks for any pending statements of the subtask to be executed before terminating. By this time, the task `Send` has made another request for the entry as in line-12 in the second iteration of the loop. The task `Receive` has also sent the entry in parallel, as in line-22 in the second iteration of the loop.

Under a scenario where the `terminate` statement at line-14 is never chosen, the tasks `Send` and `Receive` synchronizes in an infinite loop. In other words, the procedure `Main`, which is waiting for its subtasks to finish any pending execution, will wait forever. Therefore procedure `Main` cannot terminate normally, causing a livelock. Figure 2.2 shows the control-flow graph (CFG) of the two tasks, `Send` and `Receive`. At the bottom of this figure, we can see the livelock scenario showing the infinite synchronization between the tasks `Send` and `Receive`, causing the procedure `Main` to wait forever to terminate.

2.4.3 Race Condition in Ada

When multiple processes of a concurrent program attempt to modify shared data simultaneously without mutual exclusion, it leads to a problem called race condition.

Listing 2.13: Ada tasking livelock

```
1  procedure Main is
2   task Send is
3    entry E;
4   end Send;
5   task Receive is
6    end Receive;
7
8   task body Send is
9    begin
10    loop
11     select
12      accept E;
13     or
14      terminate;
15     end select;
16    end loop;
17   end Send;
18
19  task body Receive is
20   begin
21    loop
22     Send.E;
23    end loop;
24   end Receive;
25 begin
26   Send.E;
27 end Main;
```

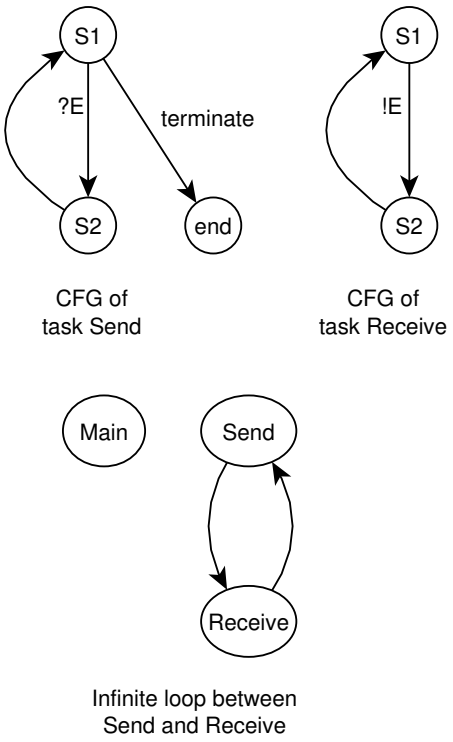


Figure 2.2: CFG and communication for the Ada tasking livelock

Listing 2.14: Example for race condition in Ada

```
1 i1 : Integer := 1000; -- shared variable
2 procedure Step_Up is
3 begin
4   i1 := i1 + 1;
5 end Step_Up;
```

Let us consider Listing 2.14 to understand how race condition can occur in Ada. The procedure `Step_Up` increments integer `i1` by one. Here `i1` is a shared variable. Assume a scenario where two Ada tasks call this procedure simultaneously. Each task reads the initial value of `i1` as 1000, adds one to the value and stores 1001 to the variable `i1`. But the problem here is that the procedure `Step_Up` has been executed twice, but the value stored in `i1` is incremented only by one from the initial value. In other words, two tasks performed overlapping writes to the shared variable `i1` without mutual exclusion leading to a race condition.

2.5 Chapter Summary

This provided an insight into the Ada programming language, the concurrency constructs supported by it, and the different problems that can occur while using these concurrency constructs. The next chapter provides the necessary background for this work concerning model checking.

3 BACKGROUND

This chapter provides the necessary background to follow the subsequent chapters. The chapter starts with an introduction to Non-deterministic finite automata (NDFA). In the following sections of this chapter and later chapters, we will extensively use the NDFA.

The chapter then proceeds to introduce model checking and explain the challenges in model checking. Next, the SPIN model checker is introduced. As a subsection within SPIN, the modeling language of SPIN (Promela) is introduced, followed by a discussion on the language features of Promela with examples. Using the knowledge obtained about Promela's language features, we will then create a model of two synchronizing processes as a carrying Promela example.

We then mention some features of the SPIN's verifier and simulator, which we will use in the subsequent chapters. Finally, we will discuss how concurrency problems can be detected using SPIN, supported by three examples. The first example shows how SPIN detects a deadlock, the second example shows how SPIN detects livelock, and the third example shows how a race condition can be detected in the model using SPIN.

3.1 Non-deterministic Finite Automata

A non-deterministic Finite Automaton (NDFA) [HMU01] is a finite state machine in which, for a particular input, the machine can transition to any states in the machine. In other words, in NDFA, the transition to a specific state cannot be determined.

An NDFA is formally defined as a quintuple $< S, s_0, L, F, T >$ where:

- S is the finite set of states
- $s_0 \in S$ is the initial state
- L is a finite set of input symbols
- $F \subseteq S$ is the set of final (or accepting) states
- T: $S \times L \rightarrow P(S)$ is the transition function, where $P(S)$ is the power set of S

With reference to L, "symbols" can be, for example, a condition, a message, etc., required to make a transition from one state to another. The transition function T is the power set of S (or 2^S) because in NDFA, it is possible to transition to any combination of S states from a state.

NDFAs are useful to model concurrent systems as it helps in the abstraction of a system to eliminate details that are not related to process interaction. In the context of this work, the SPIN model checker [Hol04] uses NDFA as the basis for defining transition systems.

3.2 Model Checking

Model checking is a formal verification technique suitable for analyzing the functional properties of a software system. Model checking-based methods are among the most successful formal methods, at least in the industrial context [VDW12]. This is due to their automated nature, which can contribute to significantly reducing the time and cost.

Given a model and property, the model checking technique exhaustively explores the model's states to prove its correctness with respect to the specified property. When a property is not valid for a given model, an error trace is produced containing the path in the model that violates the property. The three main phases of model checking are modeling, specification and verification [Cla+18a].

The first phase is the modeling phase, which involves translating a software system to a mathematical model, generally a state transition system. The success of model checking depends on how close the model's behavior is to the behavior of the code [BK08].

After modeling, the second phase is to specify the properties to be verified by the model checker in the model. Generally, a temporal logic formalism such as Linear Temporal Logic (LTL) [HR04] and Computational Tree Logic (CTL) [HR04] are used for property specification.

The third phase is the verification phase, which is performed automatically by the model checker to check the specified properties. An error trace, in case of property violation, is produced in this phase. As the number of state variables in the system increases, the size of the system state space grows exponentially [Cla+18b]. This is called the state explosion problem, which is the most common challenge during the verification phase. Several research techniques have been developed to effectively use model checking without encountering this obstacle. Choosing the right technique is not easy as it varies from one application to another and requires user proficiency [Bošo1].

Applying model checking to real-life software systems requires a solution to the challenges explained in the following sections.

3.2.1 Model Generation

Programming languages have more features than modeling languages; the semantics of a programming language like Ada, C, and C++ is far more complicated than the semantics of a modeling language. Therefore, it is important to map the program primitives into similar model checker primitives such that the model simulates the program's behavior. Furthermore, manually creating a model of the software is time-consuming and prone to human errors. Therefore, the model should be generated automatically.

3.2.2 State Space Explosion

Due to industrial software systems' size, the corresponding formal models will have many states. The model's state space limits the model checker's time and memory to produce useful results. Therefore, rigorous abstraction techniques need to be considered such that the model preserves only those details required for the property to be verified [CGL94].

3.2.3 Property Specification

Generally, temporal logic is used to specify properties. Two challenges are noteworthy to mention regarding property specification. First, a good understanding of the system to express the

system's properties in temporal logic. Second, mapping the system's properties to the properties of the model is not direct. This is because the design of specification languages is meant to state the model's properties rather than the system's properties. This leads to semantic gaps between the model's specification language and the system's source code language.

Explicitly specifying properties may not always be required depending on the verified property and the model checking tool. For example, the SPIN model checker checks for deadlock in the model without an explicit property specification.

3.2.4 Traceability

If a specified property does not hold, the model checker reports a counterexample indicating the model's path leading to the property violation. Typically, the formal model of a large software system can generate traces of varying sizes ranging from short traces to very long traces. Manually tracing violations back to the source code is a tedious process. It is, therefore, essential to automatically generate traceability from model to code.

3.3 The SPIN Model Checker

SPIN [Hol04] is one of the leading model checkers used to verify concurrent and distributed software systems, developed by Gerard J Holzmann and others at Bell Labs in 1980.

SPIN targets both software verification and hardware verification. It is designed to address scalability problems and to support the verification of even large problem sizes. The modeling language of SPIN is called Process Meta Language or Promela. The models written in Promela are validated in SPIN for violations like deadlocks, livelocks, and assertion violations. Properties to be verified in the model are expressed in LTL.

The present work is focused on detecting *concurrency problems* in the Ada software system. Due to its size, modeling such a system can quickly lead to scalability problems. This means the chosen model checker should address both the concurrency aspect and the scalability problems. Following are some major reasons to choose the SPIN model checker for this work:

1. SPIN has an implicit notion of synchronization and supports both synchronous and asynchronous message passing.
2. A concurrent system usually has several processes doing some computation and also communicating with each other. Modeling such a system in SPIN does not consider the internal computations of the processes; rather, SPIN's focus is to verify whether the processes communicate correctly. This helps in creating tractable models.
3. SPIN's modeling language, Promela supports the abstraction of distributed systems to eliminate details that are not related to process interaction.
4. SPIN offers several options to control state explosion problem and to minimize the verification time. The options include: partial order reduction [HP95], state compression [Hol97] and bitstate hashing [Hol88].

3.3.1 Promela Modeling Language

Promela is the verification modeling language of the SPIN model checker. Promela supports the modeling of distributed systems; it allows to dynamically create concurrent processes. Processes communicate with each other using message channels. It is possible to define message

channels as synchronous (i.e., rendezvous channels) or asynchronous (i.e., buffered channels). Files written in Promela have a `.pml` extension.

The three types of objects supported by Promela are *processes*, *message channels* and *variables*. Processes are global objects. Message channels and variables can be either global or local objects declared within a process. Processes are used to specify behavior. For example, if a task T1 in a distributed system communicates synchronously with another task T2, the *synchronous communication* of T1 is the behavior modeled in the corresponding Promela process P1. Channels and global variables, on the other hand, define the environment in which the processes run.

Promela has a C-like syntax. The tokens like identifiers, keywords, constants, and operators in Promela are identical to that of the C language. The rest of this section focuses mainly on the language features of Promela that will be used in the upcoming chapters. For additional details on the Promela language, refer [Hol80].

3.3.1.1 Comments

Comments in Promela starts with `/*` and ends with `*/`, similar to the C language; however, Promela does not allow nested comments.

3.3.1.2 Separators

Promela supports two statement separators: an arrow (->) and the semicolon (;). Unlike in the C language, the semicolon in Promela is not a statement terminator but a statement separator. In other words, not placing a semicolon after the last statement is not a syntax error in Promela.

The arrow separator is conventionally used to indicate the relation between the two statements. For example, an arrow is placed between a condition and the resulting execution sequence.

3.3.1.3 mtype

An mtype or message type is used to declare symbolic constants (similar to symbolic constants in the C language). In other words, an mtype allows using symbolic names for constant values. Promela allows one global definition of mtype with a maximum declaration of 256 symbolic constants. An mtype variable is of size 8 bits.

Messages that are passed through channels are modeled as mtypes. For example, if `ack`, `err` represent acknowledgement and error messages respectively sent over a channel, then these messages are modeled using the mtype declaration: `mtype = { ack, err }`.

3.3.1.4 Message channels

The transfer of data from one Promela process to another is modeled using message channels. The keyword `chan` is used to declare channels, either locally or globally (similar to integer variables in the C language).

When initializing a channel, the channel capacity can be set as a constant. A channel with capacity zero is called a synchronous or rendezvous channel. A rendezvous channel passes messages through a synchronous handshake between the sender and the receiver processes

without storing any message. A channel with a capacity larger than zero is called an asynchronous or buffered channel. A buffered channel can store messages using the given number of slots specified as a constant.

A rendezvous channel is modeled as: `chan a = [0] of {mtype}`
This initializes a rendezvous channel `a` of capacity zero for a message containing one message field of type mtype.

A buffered channel is modeled as: `chan b = [2] of {short}`
This initializes a buffered channel `b` which can store up to two messages, each containing one message field of type short.

The number of messages stored in a buffered channel can be determined using Promela's predefined function `len`. The function takes a channel name as an argument and returns the number of messages that are currently present in the channel.
An example usage is: `len (b) > 0 -> /* do something */` where `b` is a buffered channel.

3.3.1.5 Message passing

Using the mtype and channel declaration, a message can be sent over the channel from one Promela process to another. Channels, by default, pass messages in first-in-first-out order. A single channel may also be used for bidirectional communication.

An example statement to receive a message is: `chan?ack`
Here, the channel `chan` is used to receive ("?") the message `ack`.

An example statement to send a message is: `chan!ack`
Here, the channel `chan` is used to send ("!") the message `ack`.

3.3.1.6 Processes

The keyword `proctype` is used to define the behavior of a Promela process. A proctype definition only declares the behavior of the process but does not execute it. The keyword `init` is used to explicitly declare a process to be executed from the initial system state. If more than one processes are active, then each of the processes is instantiated using the `run` keyword within `init`.

Listing 3.1 defines two processes, P1 and P2. P2 sends the `ack` message using the rendezvous channel C, as in line-8. P1 accepts this message as in line-5. The two processes are instantiated using the `run` keyword, and the `init` keyword is used to state both processes are active in the initial system state, as in line-10.

Listing 3.1: Processes in Promela

```
1  mtype = { ack };
2  chan C = [0] of {mtype};/* rendezvous channel
3                             from P2 to P1 */
4  proctype P1(){
5  C?ack
6  }
7  proctype P2(){
8  C!ack
9  }
10 init{run P1();run P2()}
```

3.3.1.7 atomic

Promela's `atomic` keyword helps to: (a) define a fragment of the model to be executed as one indivisible unit, non-interleaved with other processes, and (b) start a set of processes such that none of the processes can start execution until all the processes have been initialized.

By restricting the amount of interleaving allowed in a concurrent process, the atomic sequences reduce the complexity of the verification model. For example, by enclosing local variables of a process within an atomic sequence, the manipulations of these variables through interleaving is restricted.

Listing 3.2 shows an example where two variables x and y are swapped without being interrupted (interleaved) by any other statements.

Listing 3.2: Promela's atomic construct to restrict interleaving

```
1 atomic {
2   temp = y;
3   y = x;
4   x = temp
5 }
```

Listing 3.3 shows an example where a series of processes, T1, T2, and T3, cannot start until all of them have been initialized. This helps avoid a scenario where only T1 and T2 keep executing without T3 being initialized.

Listing 3.3: Promela's atomic construct to start a series of processes

```
1 atomic {
2   run T1();
3   run T2();
4   run T3()
5   }
6 }
```

3.3.1.8 Control flow construct - case selection

Case selection is one of the three Promela's control flow constructs. The other two constructs are the repetition and unconditional jump, discussed in the subsequent sections.

The selection structure allows to non-deterministically choose one sequence among multiple execution sequences. A particular sequence will be selected if its *guard* is true. The guards and the corresponding execution sequences are listed using double-colon (::). The guards may or may not be mutually exclusive; one of the corresponding sequences will be selected non-deterministically if more than one guard is executable.

Listing 3.4: Case selection with guards

```
1 if
2 :: (x == 1) -> C!ack
3 :: (x == 2) -> C!err
4 fi
```

Listing 3.5: Case selection without guards

```
1 if
2 :: C!ack
3 :: C!err
4 fi
```

In Listing 3.4, depending on the guard (x equals one or x equals two), one of the messages (ack or err) is selected to be sent over channel C. If neither of the guards is executable, the process is blocked until one of them can be selected.

If the guards are removed, then only the execution sequences are listed as shown in Listing 3.5. Here, one of the messages (ack or err) is selected non-deterministically to be sent over channel C. If both the messages cannot be accepted by any receiving processes, then the process is blocked until one of the messages can be received.

To avoid the process blocking, Promela supports two pseudo-statements, namely `else` and `timeout`.

Promela's `else` statement is used as the last option sequence among a list of sequences in a selection. In Listing 3.6, if both guards (x equals one and x equals two) are not executable, then the `skip` statement following the `else` is executed, thereby avoiding the process being blocked. The `skip` is a dummy statement used to jump at the end of the program.

Listing 3.6: Promela's else statement

```
1 if
2 :: (x == 1) -> C!ack
3 :: (x == 2) -> C!err
4 :: else -> skip
5 fi
```

The `timeout` is a predefined, global, read-only, boolean variable used to abort a process if none of its non-deterministic choices are executable. In other words, the `timeout` is used as a guard in case selection to escape from a system hang state. The `timeout` is particularly useful when modeling delay time interval to send/receive a message. The `timeout` cannot specify precise delay time intervals (e.g., 2ms, 3us etc.).

Listing 3.7: Promela's timeout

```
1 if
2 :: C!ack
3 :: C!err
4 :: timeout
5 fi
```

Listing 3.7 shows the usage of `timeout`. If neither of the messages (ack, err) are accepted by another process, then the `timeout` at line-4 is chosen to abort the process without getting into a block state.

3.3.1.9 Control flow construct - repetition/loop

Promela's repetition construct is the logical extension of the case selection construct. The `do` keyword is used to repeat the execution of a structure. The statement `break` is used to exit the repetition structure.

Listing 3.8: Promela's repetition construct

```
1 do
2 :: (x == 1) -> C!ack
3 :: (x == 2) -> C!err
4 :: else -> break
5 od
```

Listing 3.8 shows the usage of the repetition construct. One of the options is chosen non-deterministically. That is, the message `ack` is sent if the guard at line-2 is true, or the message `err` is sent if the guard at line-3 is true. After choosing one of these options, the execution of the structure is repeated. If neither of the guards is true, the structure is exited with a break statement as in line-4.

3.3.1.10 Control flow construct - unconditional jump

Promela's `goto` statement provides an unconditional jump to a labeled statement. The `goto` statement transfers control from its immediate preceding statement to a label at its immediate succeeding statement. A label in Promela is an identifier followed by a colon (:). It is possible to label any Promela statement.

Listing 3.9: Promela's goto statement

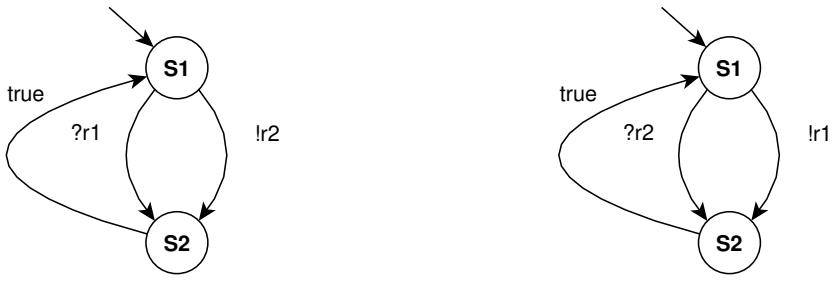
```
1 if
2 :: (x == 1) -> goto L1
3 :: (x == 2) -> goto L2
4 fi
5 L1: ...
6 L2: ...
```

Listing 3.9 shows that any one of the labels, L1 or L2, will be reached based on the guard condition.

3.3.2 An Example SPIN model

This section will use the Promela's language features learned so far to model a rendezvous synchronization between two processes as an example.

Figure 3.1 shows two synchronizing processes, P1 and P2. S1 is the initial state of P1. From this state, the process can go to state S2 either by receiving the message r1 (i.e., ?r1) or by sending the message r2 (i.e., !r2). From S2, the process can come back to S1. This execution sequence repeats forever. On the other side, again, S1 is the initial state of P2. From this state, the process can go to S2 either by receiving the message r2 (i.e., ?r2) or by sending the message r1 (i.e., !r1). From S2, the process can come back to S1. This execution sequence repeats forever. In summary, two rendezvous synchronizations are possible between P1 and P2, of which one of them is chosen. After that, this sequence repeats forever.



(a) Process P1 (b) Process P2

Figure 3.1: Synchronization between processes P1 and P2

Listing 3.10 shows the Promela model of the two synchronizing processes, P1 and P2. The synchronization messages, r1, and r2, are declared as symbolic constants or mtype. Two rendezvous channels of type mtype are declared for communication between P1 and P2, as in line-2 and line-3. As a convention, a channel is named such that the process sending a message occurs first in its name. For example, P1 sends the message r2 to P2; therefore, the naming convention `P1_to_P2`. The two processes are defined as two *proctypes*. In P1, the channel `P1_to_P2` is used to send the message r2, and the channel `P2_to_P1` is used to receive the message r1 from P2. The repetition of the execution sequence described for Figure 3.1 is modeled using Promela's repetition construct `do` in both P1 and P2. The `init` at line-16 ensures that both processes are active in the initial system state, the `atomic` at line-17 ensures that the execution starts only after both processes are initialized. The `run` at line-18 and line-19 ensures that both processes are instantiated.

Listing 3.10: Promela model `P1.pml` of two synchronizing processes

```
1  mtype = {r1, r2};
2  chan P1_to_P2 = [0] of {mtype}; /* rendezvous channel */
3  chan P2_to_P1 = [0] of {mtype}; /* rendezvous channel */
4  proctype P1(){
5  do
6  :: P1_to_P2!r2 -> skip;
7  :: P2_to_P1?r1 -> skip
8  od
9  }
10 proctype P2(){
11 do
12 :: P2_to_P1!r1 -> skip;
13 :: P1_to_P2?r2 -> skip
14 od
15 }
16 init {
17   atomic{
18      run P1();
19      run P2()
20   }
21 }
```

The above Promela model is input to SPIN for verification. By default, SPIN checks for safety properties (i.e., check for deadlock) in the input model. Listing 3.11 shows SPIN's verification result. Line-4 with a plus sign indicates that SPIN's default partial order reduction algorithm is used (more on this in Chapter 7, Section 7.1.3.1.1). Lines-9, 10, and 11 respectively indicate that system state description required 28 bytes of memory per state, there were 6 transitions from the initial state, and no errors were found in this search. Line-12 indicates that a total of 6 unique system states were stored in the state space (with each state represented by a vector of 28 bytes). Line-13 shows that the search returned to a previously visited state in the search tree during state exploration in 4 cases. Line-14 shows that ten transitions were explored in total. Line-15 indicates that one of the ten transitions was part of an atomic sequence. The key takeaway from this verification log is the confirmation from SPIN that there was no deadlock found in the model, as indicated in line-11 (zero errors).

After no errors were found in verification, the next step is to check whether the model behaves as expected. In other words, we need to confirm that both processes are synchronizing. This

is done by using SPIN's simulator output, as shown in Figure 3.2. It is seen that the model's `init` section is first considered to instantiate both processes. Next, there are arrows going from process P1 to P2 and the other way. For example, the first arrow from P1 to P2 indicates that P2 has accepted the message r2 sent by P1 over channel 1. Note that SPIN's simulator assigns identification(ID) to channels and processes. Here, channel `P1_to_P2` is assigned the ID 1 and channel `P2_to_P1` is assigned the ID 2. Similarly, process P1 is assigned the ID 1, and process P2 is assigned the ID 2. The output shows that all the non-deterministic choices as in lines-6,7,12 and 13 of Listing 3.10 were considered during the simulation. This confirms that the model behaves as expected.

As a side note, SPIN allows the user to select the type of property to be verified, like safety property or liveness property, in the input model before starting verification. The above Promela model is input to SPIN by enabling the check for safety property to see if there is any deadlock in the model.

Listing 3.11: SPIN's verifier output

```
1  Verification result:
2
3  (Spin Version 6.5.1 -- ...)
4    + Partial Order Reduction
5
6  Full statespace search for:
7     ....
8
9  State-vector 28 byte,
10 depth reached 6,
11 errors: 0
12   6 states, stored
13   4 states, matched
14  10 transitions (= stored+matched)
15   1 atomic steps
16 ...
```

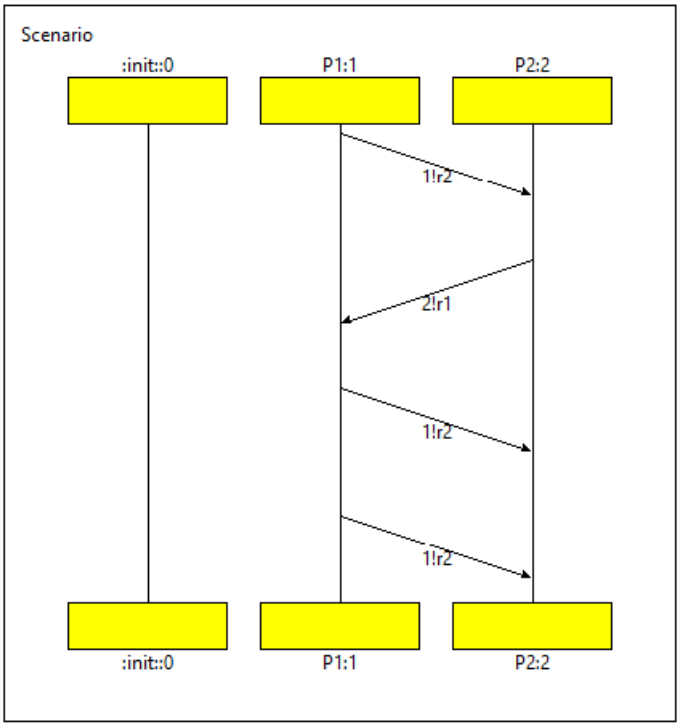


Figure 3.2: SPIN's simulator output

3.3.3 Other Features of SPIN

In the previous Section 3.3.2, we saw the usage of SPIN's verifier and simulator. The example used the SPIN's default verifier to verify a simple model. However, as the model's size increases, we need to use other verifier modes to overcome state space problems. Also, the example used the SPIN's default random simulation mode. However, if SPIN reports a property violation, we need other simulation modes to identify the path leading to this violation. In this section, we will mention the different verifier and simulator modes supported by SPIN. More details on the advanced features of SPIN and their application will be discussed in Chapter 7.

- *Verifier modes*: Besides partial order reduction, the verifier can perform an optimized search using techniques like state compression [Hol97]. Very large system models can also be validated with maximum coverage of the state space using abstraction technique

like bitstate hashing (supertrace) [Hol88]. SPIN supports an additional swarm verification mode [HJG10]. Swarm verification is a modern form of swarm computing that uses a large number of available compute cores to enhance parallelism and search depth.

- *Simulator modes*: SPIN's default random simulation mode helps to get a quick overview of the model's behavior for initial debugging. However, assume that the verifier outputs a counterexample reporting a violation. Using this counterexample as input, SPIN's guided simulation mode (SPIN's second simulation mode) can be used to trace the path in the model leading to this violation. The third, interactive simulation mode, helps to understand the model's behavior in a chosen path. The interactive simulation prompts the user at every execution step that requires a non-deterministic choice to be made.

3.3.4 Detecting Concurrency Problems Using SPIN

So far, we have seen how to verify a model using SPIN. This section will discuss how SPIN detects concurrency problems, if any, in the input Promela model. The detection of three concurrency problems, deadlock, livelock, and race condition by SPIN, will be discussed with examples.

3.3.4.1 Deadlock Detection in SPIN

To understand deadlock detection by SPIN, we consider the modified version of Listing 3.10. Two lines of process P2 are modified such that the sending of message r1 (!r1) at line-12 is now replaced with receiving of message r1 (?r1), and the receiving of message r2 (?r2) is now replaced with sending of message r2 (!r2). This modified version is shown in Listing 3.12. P2 cannot receive the message r2 sent by P1, and no process is ready to serve the waiting message r1 in P1 and P2. During execution, the model *as a system cannot evolve*, leading to a deadlock. Let us now see how SPIN detects this problem.

Listing 3.12: Deadlock.pml

```
1  /* mtype declarations */
2  /* channel declarations */
3  ...
4  proctype P1(){
5  do
6  :: P1_to_P2!r2 -> skip;
7  :: P2_to_P1?r1 -> skip
8  od
9  }
10 proctype P2(){
11 do
12 :: P2_to_P1?r1 -> skip;
13 :: P1_to_P2!r2 -> skip
14 od
15 }
16 init {
17   atomic{
18      run P1();
19      run P2()
20   }
21 }
```

Listing 3.13: SPIN's verifier output for deadlock example

```
1  Verification result:
2  pan:1: invalid end state (at depth 1)
3  pan: wrote Deadlock.pml.trail
4
5  (Spin Version 6.5.1 -- ...)
6  Warning: Search not completed
7    + Partial Order Reduction
8
9  Full statespace search for:
10   ...
11
12 State-vector 28 byte,
13 depth reached 3,
14 errors: 1
15   2 states, stored
16   0 states, matched
17   2 transitions (= stored+matched)
18   1 atomic steps
19   ...
```

The modified model is input to the SPIN verifier, enabled with a default check for safety properties. The verifier output is shown in Listing 3.13. Line-2 and line-3 respectively say there is an

invalid end state in the model, and the path leading to this invalid end state (i.e., the error trace) is written into *Deadlock.pml.trail*. The extension `.trail` indicates the trail file containing the path to the error. Line-14 indicates an error in the model (this is unlike the zero errors reported for Listing 3.10).

Listing 3.14: SPIN's counterexample for the model with deadlock

```
1  ...
2  Starting P1 with pid 1
3    1:proc  0 (:init::1) Deadlock.pml:18 (state 1) [(run P1())]
4  Starting P2 with pid 2
5    2:proc  0 (:init::1) Deadlock.pml:19 (state 2) [(run P2())]
6  spin: trail ends after 2 steps
7  #processes: 3
8    2:proc  2 (P2:1) Deadlock.pml:11 (state 5)
9    2:proc  1 (P1:1) Deadlock.pml:5 (state 5)
10   2:proc  0 (:init::1) Deadlock.pml:21 (state 4)
11 3 processes created
12 Exit-Status 0
```

The generated trail file is input to the SPIN's simulator in guided simulation mode to identify the path in the model leading to the error. Listing 3.14 shows the output from the guided simulation. As before, both the processes are instantiated first, as in line-3 and line-5. The meaning of line-6 is that the simulation could not proceed any further after instantiating both the processes. Lines-8, 9, and 10 provide the error path. After process instantiation, the simulator checked if it can take any one of the choices available within the `do` primitive of process P1 at line-11 of Listing 3.12 to complete synchronization using any one of the choices available within the `do` primitive of process P2 at line-5. However, the simulator could not succeed. Therefore, it inevitably reached the end line-21 of the model (as shown in line-10).

In this section, we discussed how SPIN detects deadlock in a model. In the next section, we will discuss how SPIN detects a livelock.

3.3.4.2 Livelock Detection in SPIN

SPIN detects livelock in a model as *non-progress cycles*. An error trace to a non-progress cycle indicates a possible infinite loop in the model. We will now discuss an example as to how SPIN detects such a non-progress cycle.

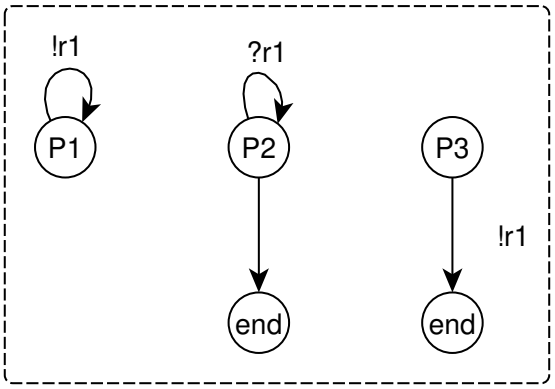


Figure 3.3: Concurrent system with three processes

Figure 3.3 shows a concurrent system with three processes. Process P1 sends message r1 repetitively, in a loop. Process P2 can either accept the message r1 repetitively or can exit the system. Process P3 reaches its end state after sending message r1.

Listing 3.15 shows the Promela model of the system. Process P1 repetitively sending the message r1 is modeled using the Promela's `do` construct as in line-5. Note that the channel `ch` is a buffered channel with a channel capacity of 1. The use of a buffered channel is to model the behavior of process P2 such that if there is a message waiting in the channel queue, P2 can accept it, else exit the system. Therefore, line-10 is a check for a waiting message; if true, accept the message as in line-12; if false, exit the system as in line-14. Process P3 sends the message r1 as in line-18 and exits the system.

Listing 3.15: Livelock.pml

```
1  mtype = {r1};
2  chan ch = [1] of {mtype};
3  proctype P1(){
4  do
5  :: ch!r1
6  od
7  }
8  proctype P2(){
9  if
10 :: len(ch) > 0 ->
11    do
12    ::ch?r1
13    od
14 :: timeout
15 fi
16 }
17 proctype P3(){
18 ch!r1
19 }
20 init {
21   atomic{
22      run P1();
23      run P2();
24      run P3()
25   }
26 }
```

Listing 3.16: SPIN's verifier output for livelock example

```
1  Verification result:
2  pan:1: non-progress cycle (at depth 18)
3  pan: wrote Livelock.pml.trail
4
5  (Spin Version 6.5.1 -- ...)
6  Warning: Search not completed
7
8  Full statespace search for:
9   ...
10  non-progress cycles
   + (fairness enabled)
11  invalid end states
   - (disabled by never claim)
12
13 State-vector 36 byte,
14 depth reached 25,
15 errors: 1
16  6 states, stored (13 visited)
17  3 states, matched
18  16 transitions (= visited+matched)
19  2 atomic steps
20 ...
```

The Promela model is input to the SPIN verifier by enabling the check for liveness properties. Within liveness properties, SPIN supports an option to check for non-progress cycles, enabled for this verification. The verifier output is shown in Listing 3.16. Line-2 and Line-3 respectively indicate a non-progress cycle in the model, and the path leading to this cycle (i.e., the error trace) is written into `Livelock.trail`. The extension `.trail` indicate the trail file containing the path to the error. Line-10 confirms that the check for the non-progress cycle has been enabled for this verification. The plus mark followed by *fairness enabled* indicate that *weak fairness* is enabled for this verification.

Weak fairness guarantees that each process will be eventually scheduled; i.e., the verifier will repeatedly switch between the three processes as desired. If weak fairness is not enabled, it leads to an undesired scenario where only P1 and P2 are scheduled repeatedly, leaving out P3.

The generated trail file is input to the SPIN's simulator in guided simulation mode to identify the

Listing 3.17: SPIN's counterexample for the model with livelock

```
1  ...
2  Starting P1 with pid 2
3   2: proc 0 (:init::1) Livelock.pml:22 (state 1) [(run P1())]
4  Starting P2 with pid 3
5   3: proc  0 (:init::1) Livelock.pml:23 (state 2) [(run P2())]
6  Starting P3 with pid 4
7   4: proc  0 (:init::1) Livelock.pml:24 (state 3) [(run P3())]
8   6:     proc  3 (P3:1) Livelock.pml:18 (state -) [values: 1!r1]
9   6:     proc  3 (P3:1) Livelock.pml:18 (state 1) [ch!r1]
10                queue 1 (ch): [r1]
11  8: proc 3 terminates
12  10:    proc  2 (P2:1) Livelock.pml:10 (state 1) [((len(ch)>0))]
13  12:    proc  2 (P2:1) Livelock.pml:12 (state -) [values: 1?r1]
14  12:    proc  2 (P2:1) Livelock.pml:12 (state 2) [ch?r1]
15                queue 1 (ch):
16  14:    proc  1 (P1:1) Livelock.pml:5 (state -) [values: 1!r1]
17  14:    proc  1 (P1:1) Livelock.pml:5 (state 1) [ch!r1]
18                queue 1 (ch): [r1]
19  ...
20 <<<<<START OF CYCLE>>>>>
21  20:    proc  2 (P2:1) Livelock.pml:12 (state -)        [values: 1?r1]
22  20:    proc  2 (P2:1) Livelock.pml:12 (state 2)        [ch?r1]
23                queue 1 (ch):
24  22:    proc  1 (P1:1) Livelock.pml:5 (state -) [values: 1!r1]
25  22:    proc  1 (P1:1) Livelock.pml:5 (state 1) [ch!r1]
26                queue 1 (ch): [r1]
27  ...
28 spin: trail ends after 26 steps
29 #processes: 3
30                queue 1 (ch): [r1]
31  26:    proc  2 (P2:1) Livelock.pml:11 (state 3)
32  26:    proc  1 (P1:1) Livelock.pml:4 (state 2)
33  26:    proc  0 (:init::1) Livelock.pml:26 (state 5)
34 4 processes created
35 Exit-Status 0
```

path in the model leading to the error. Listing 3.17 shows the output from the guided simulation mode. After instantiating the three processes (line-2 to line-7), the simulator considered line-18 of P3 in the model to send the message r1, which is stored in the channel's queue, as shown in line-10. After this, P3 terminates as in line-11. The simulator then considers line-10 of P2 in the model. Since there is message r1 waiting in the queue sent by P3, line-12 of the model is executed instead of line-14. After accepting r1, the queue is now empty, as shown in line-15. Next, the simulator considers process P1. P1 sends the message r1 as shown in line-16, and the queue again has message r1 waiting to be accepted as indicated in line-18. The message will be accepted by P2 in the next cycle (not shown).

The generated trail file is input to the SPIN's simulator in guided simulation mode to identify the path in the model leading to the error. Listing 3.17 shows the output from the guided simulation mode. After instantiating the three processes (line-2 to line-7), the simulator considered line-18 of P3 in the model to send the message r1, which is stored in the channel's queue, as shown in line-10. After this, P3 terminates as in line-11. The simulator then considers line-10 of P2 in the model. Since there is message r1 waiting in the queue sent by P3, line-12 of the model is executed instead of line-14. After accepting r1, the queue is now empty, as shown in line-15.

Next, the simulator considers process P1. P1 sends the message r1 as shown in line-16, and the queue again has message r1 waiting to be accepted as indicated in line-18. The message will be accepted by P2 in the next cycle (not shown).

From line-20 of the counterexample, a non-progress cycle starts. This is a cycle, and hence it repeats forever. Line-21 to Line-26 shows that P1 and P2 synchronize using message r1 in an infinite loop. A potential livelock in the model is indicated by line-31 to line-33. These lines state that the model could not make any progress besides accepting the message r1 within the `do` construct at line-11 of the model and sending the message r1 within the `do` construct at line-5 of the model.

3.3.4.3 Race Condition Detection in SPIN

We will now see an example as how to find race condition in a Promela model using SPIN. Listing 3.18 shows two processes `add` and `remove` that uses a shared variable `value`. The `add` process initializes variable `new_val` to zero, increments the variable by one and assigns the updated variable to the shared variable `value`. The second process, `remove`, initializes variable `new_val` to zero, decrements the variable by one and assigns the updated variable to the shared variable `value`.

Listing 3.18: Race_condition.pml

```
1  /* Variable value is shared
2    between two processes */
3  show int value = 0;
4  active proctype add(){
5    show int new_val;
6    new_val = value;
7    new_val++;
8    value = new_val;
9  }
10 active proctype remove(){
11   show int new_val;
12   new_val = value;
13   new_val--;
14   value = new_val;
15 }
```

The simulation of the model using SPIN shows that the shared variable `value` is set to three different values, 0, 1 and -1; the keyword `show` helps monitoring the values set to `value` and `new_val`. In other words, after one process updates the shared variable, the other process can update it based on the original un-updated value resulting in a race condition. Note that we have only used SPIN's simulation results to detect the race condition; this is unlike the examples to detect deadlock and livelock where we used counterexample traces. In other words, interpretation of SPIN's simulation results helps to detect important violations in the model. In the upcoming chapters, we will see why model simulation is a mandatory step and not optional during verification.

3.4 Chapter Summary

We started this chapter with an introduction to NDFA, followed by introducing model checking and its challenges. Next, the SPIN model checker is presented, followed by a concise language

reference to the Promela modeling language. To apply the knowledge obtained on the Promela language, a two-process concurrent system is modeled as an example. In this example, we saw how to interpret the verifier and simulator log messages from SPIN. We then mentioned some additional features of SPIN to address the state explosion problem and identify the error trace. These features are just introduced here but will be used extensively in the subsequent chapters. Finally, we looked into three concrete examples that explained how SPIN detects concurrency problems like deadlock, livelock and race condition.

Until now, we gained insight into Ada, concurrency problems in Ada, model checking, and detecting concurrency problems using model checking. The next chapter is a study on previous work related to the formal verification of Ada programs. We will look into the different model checking approaches used to detect concurrency problems in Ada programs, along with their limitations.

4 STATE OF THE ART IN THE VERIFICATION OF ADA PROGRAMS

In this chapter, we will discuss the related work with respect to the verification of Ada programs. The chapter is divided into two major sections based on the formal verification technique used.

In the first section, we will look into the contract-based (or annotation-based) verification of Ada programs using SPARK and RavenSPARK. In the second section, we will discuss the application of model checking to verify Ada programs using three tools: Quasar, Ada Translating Toolset, and ATOS. At the end of each work, we will look into the limitations in their approach. We then proceed to list the lessons learned from each of these earlier works. The chapter is concluded with its summary and an introduction to the next chapter.

4.1 Contract Based Verification of Ada Programs

4.1.1 SPARK

The programming language SPARK [Bar03] is a restricted subset of the Ada language, with added annotations for the specification of the desired behavior of programs resembling the principles of Design by Contract [Mey92]. These annotations are embedded in programs as comments, enabling SPARK and Ada programs to share the same compiler. In other words, a valid SPARK program is also a valid Ada program. SPARK provides different tools for various purposes, such as a static analyzer or an automated theorem prover.

4.1.2 RavenSPARK

The Ravenscar Profile [BDV04] [AD03], which is a subset of the Ada tasking model, aims to make the formal verification of concurrent and real-time Ada programs. The Ravenscar profile imposes language restrictions on the Ada run-time system using pragma `Restrictions` [BDV04].

The SPARK language supported only the verification of the sequential aspects of the Ada language. The Ravenscar profile, on the other hand, supported only the concurrent aspects of the Ada language. Combining the Ravenscar profile features with SPARK will then help construct concurrent programs with high reliability. This gave rise to another subset called the RavenSPARK [SPA08].

The RavenSPARK is an extension of SPARK compliant with the Ravenscar profile. This subset of the Ravenscar profile includes another set of language restrictions (further to the pragma `Restrictions` in Ravenscar profile) and additional annotations. These additional features in RavenSPARK supported the detection of unexpected exceptions and exceptions due to concur-

rency, which was not supported by the Ravenscar profile. For more details on these additional pragma `Restrictions` and annotations, see [SPA08].

4.1.3 Limitations of SPARK and RavenSPARK

The SPARK programming language and its toolset are important landmarks for formal verification of Ada programs. However, the SPARK Ravenscar profile supports limited annotations to verify concurrent properties [HM03]. Most importantly, a Ravenscar model assumes that all intertask communication and synchronization of a concurrent system are deterministic. This severely limits the run-time behavior of the program by ignoring the non-determinism.

Lundqvist *et al.* [LA99] adapts the Ravenscar profile to a form that is suitable for input to the UPPAAL model-checking tool. An UPPAAL [LPY97] model is created to verify the concurrency feature *delay until*. This feature is used to implement periodic tasks, allowing a task to suspend itself until a fixed point in the future. The work concludes that the complexity level is unknown (i.e., the approach may not scale well) when this adaption of the Ravenscar profile is applied to large systems with several concurrent tasks.

4.2 Model Checking Ada Programs

4.2.1 Quasar

The Quasar model checking tool [Eva+03b] supports the verification of concurrent Ada programs. The tool receives Ada programs as input and converts them into colored Petri Nets or CPNs (see [Jen13]). The translation of an Ada program to a CPN is done in two phases:

1. Each construct, like the statement, expression, declaration, etc., of the Ada program, is translated into predefined CPN patterns. For more details on these predefined patterns, see Figure 1 in [Eva+03a]. In other words, when the complete Ada program is translated, it results in producing small sets of CPN components.
2. The CPN components produced in step-1 are combined using two operations: substitution and merging. The result of these two operations produces CPNs, which is used as input for model checking.

We will now discuss how QUASAR constructs CPNs for the client-server Ada program shown in Listing 4.1, as an example. Shown in Figure 4.1 and Figure 4.2 are the two-phase translation performed by Quasar for the Ada program. The $\prec$ and ∞ operators indicate substitution and merging, respectively.

While translating a loop statement, the statement within the loop is considered as an abstract transition and is ignored. Starting from the left side of Figure 4.1, with respect to Client side, the abstract transition `loop ... end loop;` net is substituted with the entry call `server.e` net to obtain the translated CPNs on the top-right side of Figure 4.1. A similar substitution is applied for the Server side shown at the bottom of Figure 4.1. So far, we have applied two successive substitutions to obtain two independent CPNs.

Next, the two independent CPNs are merged (∞ operator) around the `E.RETURN` and `E.CALL` places to obtain the CPNs corresponding to the two loop statements. This merging operation is shown in Figure 4.2. Note that naming the places helps to merge dependent parts of the program. In this example, the Ada entry call and the accept statement depend on the same entry `Service`.

The validation of models extracted using Quasar is performed using High LEvel Net Analyzer

(HELENA) [Eva05]. Quasar provides four templates that correspond to the specification of the most common LTL properties, namely, state accessibility, bounded wait, critical section, and stable property. The instantiation of these templates is done with a specific task state, variable, or entry queue using Quasar's graphical interface. For more details on these templates, refer to [Eva+03b].

Listing 4.1: A simple client-server concurrent program [Eva+03b]

```
1  task body Client is
2  begin
3     loop
4        Server.Service
5     end loop;
6  end Client;
7  task body Server is
8  begin
9    loop
10     accept Service;
11   end loop;
12 end Server;
```

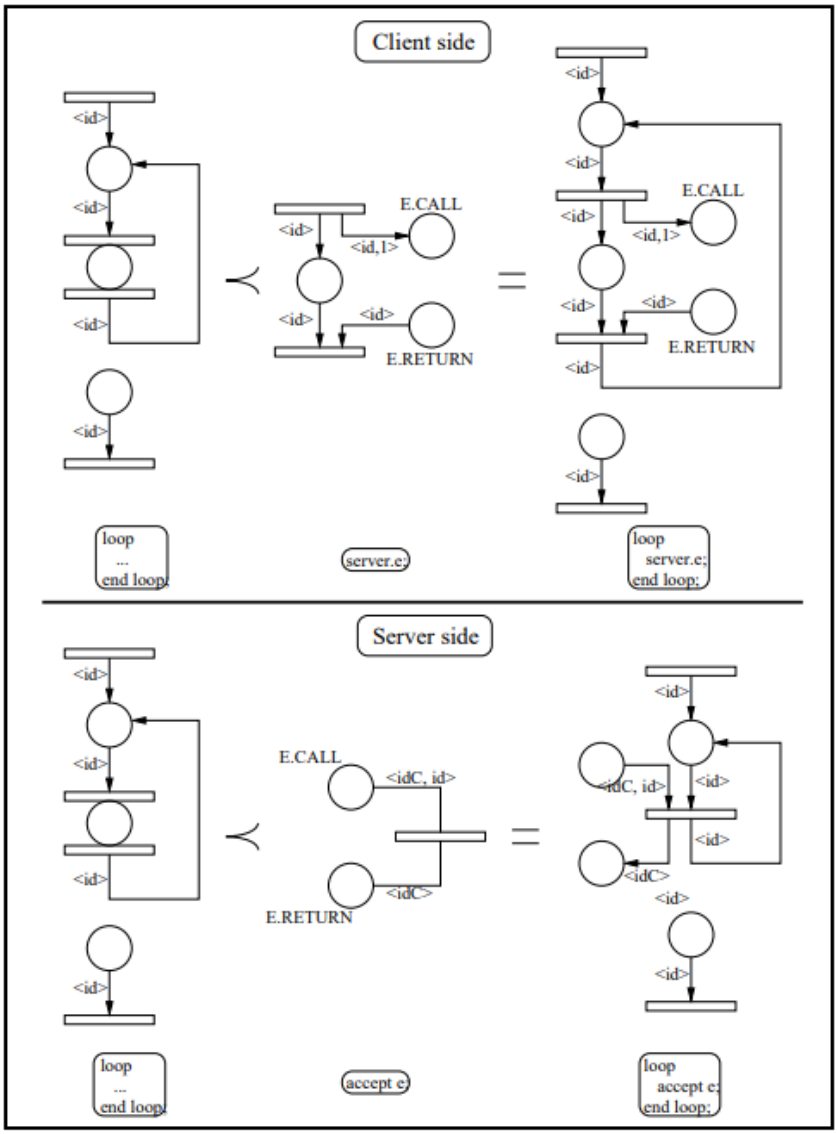


Figure 4.1: First phase of Quasar translation process [Eva+03b]

4.2.2 Limitations of Quasar

The approach followed in Quasar has the following limitations:

1. *Combinatorial explosion*: Quasar has been tested only on a static set of Ada's concurrency constructs. However, to analyze concurrency problems in a real-life Ada program, it is required to preserve all the task interleaving from the Ada program to the translated CPNs. Preserving all such task interleaving involves several substitution and merging of different CPNs patterns. This leads to the state explosion problem; in the context of Petri Nets, this is called 'combinatorial' state explosion. In summary, it is not guaranteed to generate a scalable model of a real-life Ada program using Quasar.

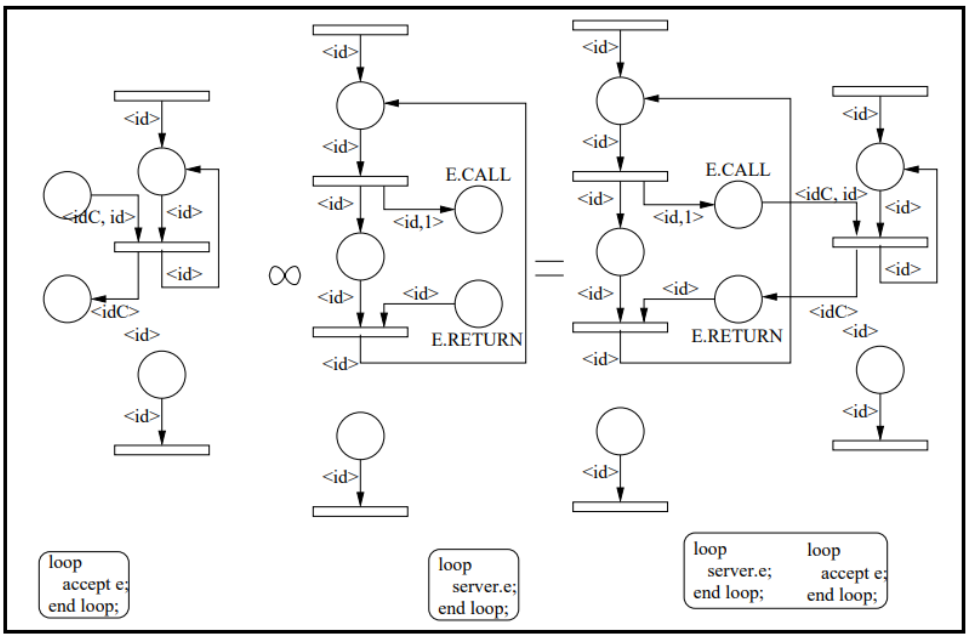


Figure 4.2: Second phase of Quasar translation process [Eva+03b]

2. *Traceability*: There is no evidence as to how Quasar can support to trace violations detected by HELENA back to the Ada program.
3. *Dynamic task creation*: Quasar does not support dynamic task creation. The tool does not create a behavior similar to Ada's concurrent task creation in the generated model.

4.2.3 Ada Translating Toolset

The Ada Translating Toolset [DPC98] supports automatic verification of Ada programs by converting Ada programs as input to SPIN and NuSMV.

As a first step, the toolset applies program abstraction techniques to convert the original Ada program to a finite state machine model to apply model checking. In the second step, this FSM is then converted to S-Expression Design Language (SEDL) [Cor93] using IRIS-Ada toolset [Arc97]. SEDL helps represent a concurrent system in a compact notation that can be easily converted to a finite-state automaton. In the third step, the properties are defined using the INCA query language [Sie02]. In the last step, with the concurrent system represented in SEDL and the properties specified in the INCA query language, the INCA tool generates a model with the properties to be verified. This generated model is then input to SPIN or NuSMV to complete the verification. The translation process is shown in Figure 4.3.

4.2.4 Limitations of Ada Translating Toolset

The approach followed in Ada Translating Toolset has the following limitations:

1. The translation from Ada to SEDL already comes with several bugs: Ada task header syntax errors, procedure syntax errors, and improper modeling of Ada's *accept* bodies [DPC98]. After the translation to SEDL, all these bugs should be hand fixed before the next step.
2. The approach uses an Ada toolset, IRIS. The IRIS toolset supports three actions: compiler front-end processing, generating control-flow graphs for the Ada tasks and procedures, and finally generating the SEDL. To evaluate the effectiveness of these important steps,

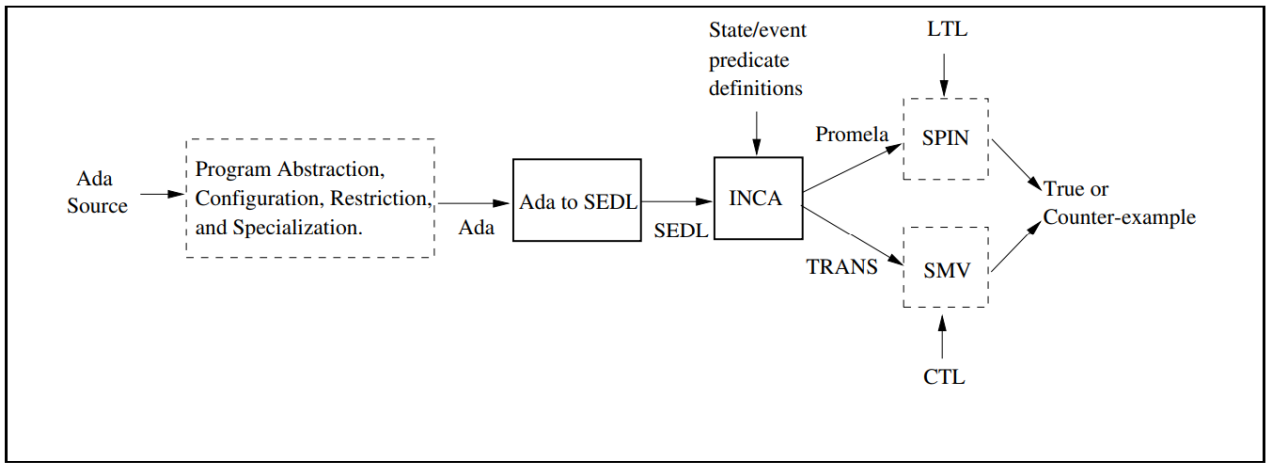


Figure 4.3: Ada translation toolset [DPC98]

it is essential to do hands-on experiments on IRIS. However, IRIS and other Ada toolsets developed as part of the Arcadia project are no longer available to use, and any support to use them cease to exist [Arc97].

3. Besides simple examples, there is no evidence of using the Ada Translating Toolset for a real-life concurrent Ada software. This means as the complexity of the Ada program increases, it may not be guaranteed to obtain: (a) a scalable model that does not face state explosion problems and (b) a model that faithfully represents the input Ada program.
4. The approach does not consider the need for traceability between the input Ada program and the output model. This makes it hard to map error traces from the model back to the Ada program.

4.2.5 ATOS

The ATOS tool [FMP12] receives Ada programs as input and generates a Promela model of these programs and the properties to be verified against this model. ATOS is implemented in Ada.

The tool extracts properties based on the annotations in the Ada program (SPARK annotations) and based on the automatic inference of properties from the Ada program. The annotation language provided by ATOS for property specification is similar to Ada SPARK. The inferred properties refer to the properties extracted from the Ada program without user inference or annotations. For example, checking whether the range of a variable with a numeric type is not violated is an inferred property. The properties extracted based on annotation and the automatic inference can either be in the form of LTL formulas or assertions embedded in the generated models. ATOS uses the SPIN model checker to check the correctness of the generated models against the extracted properties.

4.2.6 Limitations of ATOS

1. *State explosion*: ATOS has been used for some case studies. However, the tool does not address state space explosion problems when translating a real-life Ada software to Promela.
2. *Traceability*: The model of a real-life Ada software can produce very long error traces during model checking. ATOS does not support to trace such error traces back to the

source code.

3. *Ada delay*: ATOS does not support the translation of Ada's delay time intervals to Promela. This severely limits the translation of Ada concurrency constructs associated with delay, like Ada's *selective rendezvous* (discussed in Section 2.3.3) and Ada's *event handling* (more on this in Section 5.1.4).

4.2.7 Discussion

This section highlights some key takeaways from the earlier work on using model checking to verify Ada programs.

The Quasar tool focuses on model checking a particular language feature of Ada - *concurrency*, to minimize the combinatorial explosion problem. The tool proceeds to automatically extract the *concurrent parts* of the Ada program to report issues like deadlock and starvation. This approach highlights the need for abstraction; to create a scalable model of the program, it is necessary to extract relevant parts of the program with respect to a given property.

The Ada Translating Toolset automatically generates a state transition model that approximates the run-time behavior of the Ada software system. Each Ada task is modeled as an automaton whose states represent the control points of the task that are relevant to the property checked. This approach provides a direction to represent Ada's task as a finite state automaton.

The work on ATOS describes the importance of preserving the behavior of each Ada construct in the translated Promela construct. From this, the need to keep the language formalisms intact when translating a program to a formal model is realized. In other words, the behavior of the program constructs should not be lost during translation from code to formal model.

4.3 Chapter Summary

This chapter described two formal verification techniques used in earlier work to verify Ada programs - contract-based verification and applying model checking. Under contract-based verification, we discussed the features and limitations of SPARK and RavenSPARK. We discussed the approach and limitations of three tools under model checking: Quasar, Ada Translating Toolset, and ATOS. Finally, we looked into the lessons learned from each of these model checking approaches.

Besides some case studies, there is no evidence of using any of these model checking approaches to verify a real-life concurrent Ada software. This means, with the increase in complexity of the Ada program, it is not guaranteed to obtain a scalable model that faithfully represents the input Ada program. Furthermore, these model checking approaches do not describe how to trace detected violations by the model checker back to the source code.

From the next chapter onwards, we will discuss as our toolset automatically generates a formal model of a real-life Ada program by overcoming earlier work limitations like scalability and traceability. Before proceeding to the automatic generation of the model, we will first discuss, in the next Chapter 5, the translation of different Ada constructs to Promela constructs with a justification for the semantic equivalence between the two language formalism.

5 BEHAVIOR MAPPING FROM ADA TO PROMELA

The purpose of modeling a software system is to explain and predict the system's behavior. If the model created does not allow this to do, it is simply not a good model. In other words, it is important that the extracted model indeed simulates the behavior of the software system.

The developed toolset aims to extract an accurate model of the system. It is capable of translating a subset of the Ada language into the Promela language. The toolset handles the following Ada primitives:

1. Ada concurrency primitives - task, entry call, rendezvous, event, selective accept, delay
2. Ada conditional primitives - if/else, or else/and then, case, loop.

The translation performed by the toolset tries to map the Ada primitives into similar Promela primitives. Nevertheless, when an Ada primitive does not have a correspondent similar primitive in Promela, the toolset tries to convert these so that the semantics of the Ada primitive is respected.

Several small experiments are conducted by observing the behavior of different Ada primitives (during run-time) versus the behavior of their correspondent Promela primitives. These observations helped to build the required logic for the toolset after confirming the equivalence in the two behaviors.

Throughout this chapter, the above-stated experiments are illustrated. Each of the subsequent sections is organized as follows. First, the Ada construct is illustrated with an example and a control-flow graph. Second, the corresponding Promela translation of the Ada construct is illustrated. Finally, each section is concluded by making a comment on the behavioral equivalence (similarities and differences) between the Ada construct and the translated Promela construct.

5.1 Modeling Ada Concurrency Primitives in Promela

5.1.1 Tasks

Ada tasks represent different threads of control, which execute independently and concurrently. An Ada task may contain several interaction points, which allow communication with other tasks. Beyond these interaction points, tasks can interact with or affect tasks in many ways (e.g., through shared variables).

Just like Ada tasks, processes in Promela exhibit parallel activities. The Promela primitive `proctype` is used to define processes correspondent to Ada tasks.

Each Promela process (corresponding to Ada task) has an identification number to identify it uniquely. This ID is used while the process is making an entry call (see Section 5.1.2). Promela processes receive messages from other processes using channels.

Listing 5.1: A simple Ada task

```
1 task body SimpleTask is
2 begin
3   -- Statements
4 end SimpleTask
```

Listing 5.2: Promela process of the Ada task

```
1 proctype SimpleTask {
2      /*Statements*/
3 }
```

In summary, Ada tasks correspond to Promela proctypes; synchronization between Ada tasks correspond to synchronization between Promela proctypes using channels.

Listing 5.1 shows an Ada task called `SimpleTask` and Listing 5.2 shows the correspondent Promela `proctype` of the Ada task.

5.1.2 Entry Call and Accept

Communication between two Ada tasks is achieved through *entry call*. The called task executes the entry call, which is suspended until this call completes. The calling task accepts the entry call by executing an `accept` statement; the accept statement indicates the interaction point of a task. In other words, the entry call and the `accept` statement execute as a pair.

The Ada entry call is translated to Promela by sending a message to the corresponding `accept` statement channel. The sender task has to wait for the reply of the requested task, which occurs at the end of the correspondent `accept` statement execution.

As mentioned earlier in Section 5.1.1, Promela processes (correspondent to Ada tasks) use channels to communicate with other processes. The Ada `accept` statement is translated to Promela as an execution point where a task waiting for a signal over a channel will eventually receive the signal from another task.

The communication between Ada tasks using entry call and accept, versus, communication between Promela processes using channels is best explained using the Ada rendezvous communication. The next Section 5.1.3 explains this translation with an example.

5.1.3 Rendezvous

The basic principle of rendezvous is that the first party to reach the “rendezvous point” must wait for the other party to make the communication. So when a task calls, an entry is suspended until the communication finishes.

Listing 5.3 shows two Ada tasks T1 and T2, where T2 sends an entry call `Start` which is accepted by T1. This is basic rendezvous communication. Figure 5.1 shows the control flow of Ada rendezvous. The rendezvous is indicated by γ. E is the state of entry call (this corresponds to line-7 of Listing 5.3), A is the state of acceptance (this corresponds to line-3 of Listing 5.3). R and R’ indicate the two successful states of synchronization. By the definition of Ada rendezvous, note that each edge of the CFG is an atomic action. The red-colored edges 1,2,3,

Listing 5.3: Ada rendezvous

```
1 task body T1 is
2 begin
3   accept Start;
4 end T1;
5 task body T2 is
6 begin
7   T1.Start;
8 end T2;
```

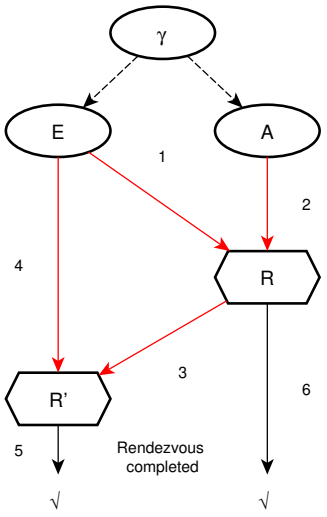


Figure 5.1: CFG of Ada rendezvous

Listing 5.4: Promela model of Ada rendezvous

```
1  mtype = {Start};
2  chan T2_to_T1 = [0] of {mtype};/* rendezvous channel
3                                    from T2 to T1 */
4  proctype T1(){
5   T2_to_T1?Start -> skip
6  }
7  proctype T2(){
8   T2_to_T1!Start -> skip
9  }
10 init{atomic{run T1();run T2()}}
```

and 4 involve in synchronization (the purpose of using a different color is to distinguish synchronizing edges from the non-synchronizing ones). E and A execute concurrently. When E reaches R, it waits for A to reach R (until A reaches R, E is suspended). When A reaches R, E wakes up from its suspended state and reaches R', thereby completing the rendezvous.

Listing 5.4 is the corresponding Promela model for the two Ada tasks. An mtype declaration in Promela allows for the introduction of symbolic names for synchronizing signals; the signal `Start` at line- 1 is a symbolic name. The two processes that correspond to two Ada tasks synchronize over the channel *T2_to_T1*. The sending side of the signal (entry call in Ada) is indicated with an exclamation mark (!), as in see line-8, and the receiving side (accept call in Ada) is indicated with a question mark (?), as in line-5. The two processes are instantiated to execute as parallel tasks, as in line-10. This model is simulated in SPIN, and the following is observed. From the initial state, when T2() sends the `Start` signal, it is *blocked* on this state (same as entry call suspension in Ada) until T1() receives this signal at its initial state. In other words, synchronization between T2() and T1() is an atomic operation. This confirms that the behavior of this Promela model is the same as the Ada rendezvous.

5.1.4 Event

Ada supports user-defined types and procedures for event handling. An event is similar to an interrupt. This section will discuss a user-defined event handling mechanism used in the Ada code under consideration.

An Ada event coordination, unlike rendezvous, may not be an atomic operation. A task that sends an event need not wait for another task to receive it, and a task that is waiting for an event cannot continue execution until it receives the expected event. Ada events may or may not be associated with a delay. An event with delay is generally used to prevent starvation; the task waits for a pre-defined time before it continues without receiving the event.

Listing 5.5: `event.adb`

```
1  task body T1 is
2   begin
3    if WaitForEvent(Wait, 2) then
4       -- accept an entry
5    end if;
6  end T1;
7  task body T2 is
8   begin
9    SetEvent (T1.Wait);
10 end T2;
```

Listing 5.6: `event.ads`

```
1  type Event_Operation is
2     (WaitForEvent, SetEvent);
3  type Event is private;
4  function WaitForEvent (E:Event;
5            Time:Duration)
6    return boolean;
7  procedure SetEvent (E:Event);
8  Wait:Event;
```

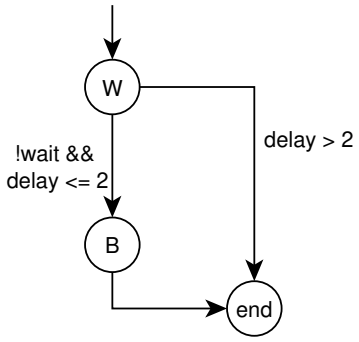


Figure 5.2: CFG of Ada event

Listing 5.5 shows the Ada file `event.adb` with two tasks T1 and T2. The definitions for WaitForEvent and SetEvent are shown in the specification file `event.ads` in Listing 5.6. WaitForEvent takes an event and time duration as input parameters, as in line-4 and line-5, and returns a True or False. SetEvent has only an event as input parameter without time duration, as in line-7.

In Listing 5.5, Task T1 is waiting for the event `Wait`, as in line-3. T1 accepts this event as long as it arrives within a delay of 2ms. Task T2 sends the event `Wait` to serve the waiting task T1, as in line-9. T1 is suspended if it does not receive the event from T2 within 2ms.

Figure 5.2 shows the control flow of the Ada event. W is the wait state, and B indicates the body of the waiting task. Upon receiving the event `wait` within 2ms, there will be a transition from wait state W to continue executing statements in the body B of the waiting task. If the event does not

Listing 5.7: Translation of Ada event to Promela

```
1  mtype = { wait };
2  chan C1 = [1] of { mtype }; /* buffered channel */
3  proctype T1(){
4  initial:
5  if
6  :: len (C1) > 0 -> C1?wait-> goto next
7  :: timeout
8  fi
9  next: /*accept an entry*/
10 }
11 proctype T2(){
12 initial: C1!wait -> skip
13 }
```

arrive within 2ms, there will be a transition from the wait state to the end state corresponding to task suspension.

In Listing 5.5, unlike rendezvous, T2 need not pause its execution until T1 accepts the event but can continue. In other words, an Ada event with delay is asynchronous communication, whereas a rendezvous is a synchronous communication. This means, in Promela, a buffered channel is required for asynchronous communication as opposed to the unbuffered channel (i.e., rendezvous channel) that we used in the previous Section 5.1.3.

The Promela translation in Listing 5.7 uses a buffered channel C1 to account for the waiting time in Ada. In buffered channels, the event is added to a channel maintaining *fifo* (first in, first out) order. The `len C1 > 0` in line-6 is to check if the event `wait` is waiting in the queue. If that is true, then T1 can go from state `initial` at line-4 to state `next` at line-9. If there is no event waiting in the queue from process T2, then T1 is exited as indicated by Promela's `timeout` primitive at line-7.

SPIN's verification algorithm does not make assumptions about the relative passing of time. Therefore it is not possible to specify a timeout interval, like 2ms, corresponding to the wait time of T1 in `event.adb`. Instead, Promela's `timeout` is used to withdraw the waiting event if there is no event in the queue sent by T2. In Section 5.1.6, we will see why this semantic difference between Ada and Promela is not a problem for the model's behavior.

5.1.5 Selective Accept Without Delay

The Ada selective accept statement allows for the non-deterministic selection of one of the multiple alternatives if their conditions are valid. These alternatives can be accept, terminate, or delay. If none of the conditions to select an alternative is valid, the task executing the statement is suspended. Here conditions refer to- (a) if/else statements that need to be true to choose an accept statement, or (b) entry call from a sender's task that is waiting to be accepted.

In Listing 5.8, one of the entry calls (entry1 or entry2 or ... entryN) will be accepted depending on whichever arrives first. The control flow of selective accept in Figure 5.3 shows A as the accept state. The entry calls with an exclamation mark indicate the required entry call from a sender's task to go from state A to state end.

The Promela translation for Ada selective accept is shown in Listing 5.9. Similar to Ada, one of

the multiple entries will be accepted if there is a caller task ready to send one of these entries. This selection of "one among the several entries" is represented with a non-deterministic selection of entries using the Promela keyword `if`. The Promela keyword `skip` at line-2 and line-4 exits the process after accepting an entry (correspondent to `end select` at line-6 of Listing 5.8).

A Promela model is simulated with this translation, and the following is observed. Depending on the first entry call received from a sender's task, one of the entries is accepted, and the process is exited. This confirms that the behavioral equivalence between both the languages for this concurrency construct.

Listing 5.8: Ada selective accept statement

```
1 select
2   accept entry1;
3 or
4   ...
5   accept entryN;
6 end select;
```

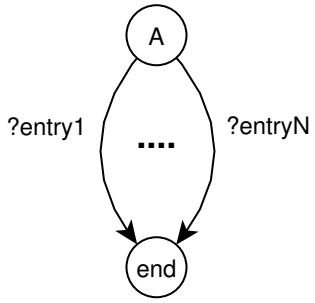


Figure 5.3: CFG of Ada selective accept

Listing 5.9: Translation of Ada selective accept statement to Promela

```
1 if
2 :: channel?entry1 -> skip
3 ...
4 :: channel?entryN -> skip
5 fi
```

5.1.6 Selective Accept With Delay

Ada's Selective accept with delay is used to decide, as a design choice, the duration until an entry call shall be waiting in a queue to be accepted; and the duration until which the accept alternatives will be accessible, waiting for the arrival of an entry call.

Listing 5.10: Ada `selective accept` with delay

```
1 select
2   accept entry1;
3 or
4   accept entry2;
5 or
6   delay 3.0;
7 end select;
```

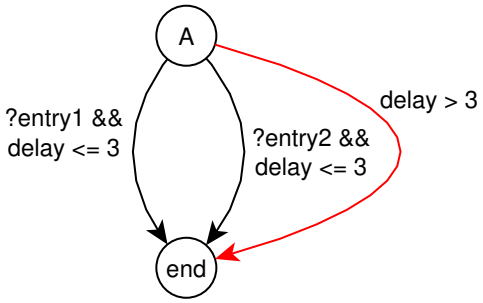


Figure 5.4: CFG of Ada selective accept with delay

In Listing 5.10, based on a first-come, first-serve basis, either `entry1` or `entry2` will be accepted, provided either one of them arrives before the time limit of 3ms. Both the entries will

be withdrawn if no caller task is ready to send either of the entries within 3ms. The control flow of the selective accept with delay in Figure 5.4 shows A as the accept state. The entry calls, entry1, and entry2, prefixed with an exclamation mark, indicate the required entry call from a sending task within the delay time interval to go from state A to state end. If no caller is ready to send any entries within the delay time interval, both the entries are withdrawn; this is indicated by the transition in red from state A to state end.

Listing 5.11: Translation of Ada selective accept with delay to Promela

```
1 if
2 :: len (channel) > 0 ->
3     if
4     :: channel?entry1 -> skip
5     :: channel?entry2 -> skip
6     fi
7  :: timeout
8  fi
```

The Promela translation for Ada selective accept is shown in Listing 5.11. The delay time interval associated with Ada's delay statement accounts for a buffered channel in Promela. The check for channel length in line-2 and the non-deterministic selection in lines 3 to 6 can be stated as follows. "If there are multiple messages from a sender's task waiting in the queue to be accepted, then one of these messages will be accepted depending on the entry which arrives first. After accepting one of these entries, the process is exited." As stated earlier, SPIN's verification algorithm does not allow assumptions about the relative passing of time. Therefore, it is not possible to specify a timeout interval (like 3ms) as in Ada. Instead, Promela's `timeout` is used, as in line-7, to withdraw both the entries if there is no entry in the queue waiting to be accepted.

A Promela model is simulated with this translation, and the following is observed. If there are more than one entry calls in the queue waiting to be accepted, then one of these entries will be accepted depending on the entry that arrived first. After accepting this entry, the process is exited. However, if there are no entries in the queue waiting to be accepted, then the `timeout` primitive is executed (thereby suspending the process), and the process is exited.

We now discuss the semantic difference between Ada's selective accept with delay and its Promela translation, followed by mentioning its consequence. The only difference observed between the behavior of the Ada construct, and its corresponding Promela translation is the precision in waiting time before suspending the process. In Ada, the task is suspended after waiting for a precise time interval of 3ms. But in Promela, the process is exited if there are no entries in the queue waiting to be accepted.

Note that one of the main reasons, as a design choice, to use Ada's selective accept with delay is to avoid a task to wait infinitely to receive an entry. That is, by using a definite time interval, a task waits only for a specific time interval before it is suspended, thereby avoiding an infinite wait time. This necessary behavior is preserved in our Promela translation; the `timeout` primitive avoids a possible system hang state when there are no entries in the queue waiting to be accepted. Therefore, we conclude that this semantic difference between Ada and Promela does not affect to preserve the required behavior in the model.

5.2 Modeling Ada Control-Flow Constructs in Promela

Conditional statements are common in most programming languages, including Ada. These statements are also present in Promela. The toolset can translate the following Ada conditional statements to Promela: if/else, case, loop, or else/and then and goto.

The translation of Ada conditional statements to Promela preserves only the control flow leading to the concurrency constructs discussed in the previous Section 5.1. The details like conditional variables, their type, control-flow leading to Ada statements that do not involve concurrency, etc., are abstracted to support the generation of a tractable formal model.

In summary, while preserving the control flow (leading to concurrency constructs) from Ada to corresponding state transitions in Promela, all the state transitions are made non-deterministic, without preserving details related to conditions. This is the abstraction principle used for translating Ada conditional constructs to Promela.

A specific translation rule is used for each of the supported Ada conditional statements. There are some common statements between Ada and Promela. However, there are other translation rules which require some creativity in mapping their semantics to Promela. The rest of this section explaining these translation rules is organized as follows. First, each Ada conditional construct is illustrated with an example and a control-flow graph. Second, the corresponding Promela translation of the Ada construct is illustrated. Finally, each section is concluded by making a comment on the behavioral equivalence (similarities and differences) between the Ada construct and the Promela translation.

5.2.1 if/elsif/else

The `if` statement for conditional execution is common in most programming languages, including Ada. Promela also supports this statement; however, its semantics is slightly different from the common `if` statement semantics.

Listing 5.12 shows Ada's if/else structure. Each of the entries at line-2, line-4, and line-6 can be reached in a mutually exclusive manner; a unique condition should be true to accept a specific entry. Figure 5.5 shows the CFG of the Ada if/else structure. The entry calls prefixed with an exclamation mark at each edge indicates the required entry call from a sender's task to make a specific transition. For example, if condition2 is true and entry2 is waiting to be accepted, the end state is reached after accepting the entry2 call. After accepting one of the three entries, the end state is reached.

Listing 5.13 shows the Promela translation of the Ada if/else statement. The translation shows non-deterministic transitions to all three accept states using the Promela selection statement `if`. After accepting one of the three entries, the process is exited. Figure 5.6 shows the NDFA of the translated Promela model. The automaton can reach any one of the states, S1, S2, S3, based on the entry call received before reaching its end state from the initial state.

Several questions arise by looking into this Promela translation, like: (a) what happened to the conditions in the Ada program, (b) why is the nested if/else structure present in Ada not seen in the Promela translation, and (c) how is the violation found in the model traced back to the Ada program (with such a non-deterministic selection in the translated Promela).

To answer these questions, let us consider the *exact* Promela translation of the same Ada program as shown in Listing 5.14. This translation additionally considers all the conditions from the Ada program. With this translation comes an advantage that the model is deterministic; a specific entry call is accepted based on a specific condition. Such a model has minimum states to explore. On the other hand, a major disadvantage of this model is the need to keep track of the

Listing 5.12: Ada conditional statement if/elsif/else

```
1 if condition1 then
2   accept entry1;
3 elsif condition2 then
4   accept entry2;
5 else
6   accept entry3;
7 end if;
```

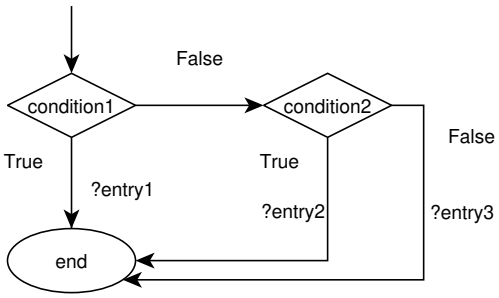


Figure 5.5: CFG of Ada if/elsif/else

Listing 5.13: Translation of Ada conditional if/else statement to non-deterministic transitions in Promela

```
1 if
2 :: channel?entry1 -> skip
3 :: channel?entry2 -> skip
4 :: channel?entry3 -> skip
5 fi
```

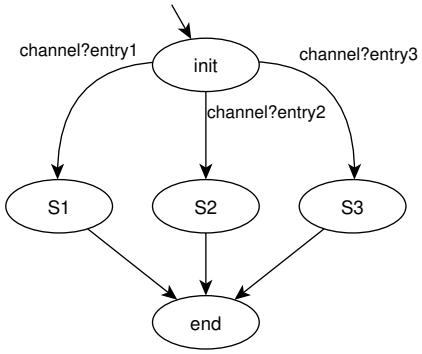


Figure 5.6: NDFA of the translated Promela model

conditional variables. During the run-time of the program, each condition may be dynamically set to true or false due to operation(s) performed in different task(s). Including these dynamic variable updates in the model increases the model complexity substantially. Therefore, to overcome this challenge, all conditions are replaced with non-deterministic choices for executing a concurrency construct (i.e., accepting an entry in this example) as in Listing 5.13. This decision helps to - (1) avoid keeping track of dynamic update of conditional variables, (2) *collapse* the Ada program's nested if/else structure to a simple set of choices in Promela.

Listing 5.14: Translation of Ada conditional if/else statement to Promela including conditions

```
1 if
2 :: condition1 -> channel?entry1
3 :: else ->
4    if
5    :: condition2 -> channel?entry2
6    :: else -> channel?entry3
7    fi
8 fi
```

Three side effects are important to discuss because of the above decision:

1. The first side effect is, of course, an increased number of choices leading to an increased number of states to explore. In essence, this decision is a trade-off between increased state space and reduced model complexity. As a solution to the state space problem, the

toolset is built-in with efficient state space reduction techniques, which will be discussed in Chapter 6.

2. The second side effect is the over-approximation in the model; the model has transitions that may not actually happen in the Ada program. For example, during run-time, it may not be possible, say, for condition2 in the Ada program to be set to true by any operation(s) performed in different task(s). However, this behavior is part of the Promela model as in line-3 of Listing 5.13. As a consequence of this additional behavior, if a violation is found in the model related to condition2, it may be a spurious violation that need to be removed.
3. The third side effect is the challenge to trace a specific error trace from the model back to the source code. If a violation is found while taking any one of the non-deterministic choices, this violation corresponds to a specific path in the program control flow. For example, assume that the model checker detects a violation while accepting entry2 in line-3 of Listing 5.13. Without a traceability matrix, it is hard to trace this choice precisely to the Ada program's control flow leading to line-4 of Listing 5.12. As a solution to this problem, the toolset is built-in with an algorithm that generates a traceability matrix to map the model to code. More details on this traceability matrix are discussed in Chapter 6.

5.2.2 Case

The Ada case statement selects one alternative from a list of possible choices for the value of an expression. This list must contain all the possible results of this expression, and the conditions defined in each alternative must be mutually exclusive.

Listing 5.15: Ada `case` statement

```
1 case expression is
2    when value1 =>
3         accept entry1;
4    ...
5    when valueN =>
6         accept entryN;
7 end case;
```

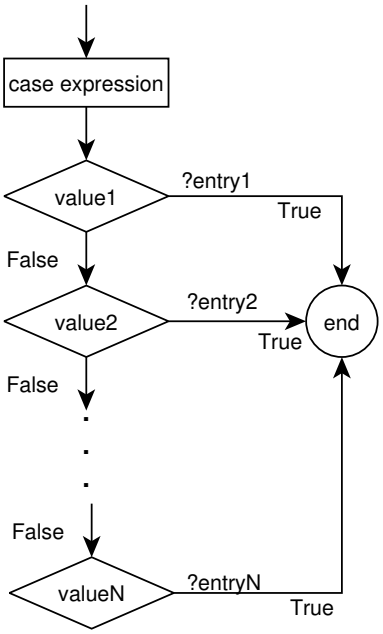


Figure 5.7: CFG of Ada case statement

In Listing 5.15, one of the entry calls (entry1 or entry2 or ... or entryN) will be accepted depending on whether `expression` is `value1` or `value2` or ... or `valueN`, respectively. The control flow of the Ada case statement in Figure 5.8 shows that one among the multiple values of the expression

is chosen to accept a specific entry from a sender's task (indicated with entry names prefixed with an exclamation mark) before reaching the end state.

Listing 5.16 shows the Promela translation of the Ada case statement. Any one of the entries (entry1 or entry2 or ... or entryN) is accepted non-deterministically before exiting the process. Replacing values of an expression in Ada with non-deterministic choices in Promela is similar to replacing conditions with non-deterministic choices as discussed in Section 5.2.1.

Listing 5.16: Translation of Ada case statement to non-deterministic transitions in Promela

```
1 if
2 :: channel?entry1 -> skip
3 ...
4 :: channel?entryN -> skip
5 fi
```

5.2.3 Loop Statement

Ada loop statements allow the repetitive execution of statements. Each repetition is called an iteration, and its execution may or may not depend on a certain condition.

In Listing 5.17, entry1 is accepted repetitively from a sender's task without any condition. The CFG of Ada loop statement is shown in Figure 5.8 with A as the accept state, which is reached repetitively by receiving entry1 from a sender's task.

Listing 5.18 shows the Promela translation of the Ada loop statement. Promela's `do` construct allows repeated execution to accept entry1 from a sender's process. This translation is simulated in SPIN to observe that the entry1 is accepted repeatedly from a sender's process. This confirms the behavioral equivalence between the Ada loop statement and the Promela translation.

Listing 5.17: Ada `loop` statement

```
1 loop
2   accept entry1;
3 end loop;
```

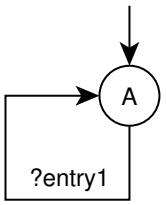


Figure 5.8: CFG of Ada case statement

Listing 5.18: Translation of Ada loop statement to Promela

```
1 do
2 :: channel?entry1
3 od
```

5.2.4 or else/and then

In Ada `and then` and `or else` are called 'short-circuit' forms corresponding to `and` and `or` operators. Short circuit operators are used to make conditional evaluation of boolean expressions.

A common usage of such operators is to avoid an exception that may be caused by the order of evaluation of expressions. An example showing the usefulness of a short-circuit operator is shown below.

Example:
Instead of, `if B /= 0 and (A/B /= 5) then`
the short-circuit form, `if B /= 0 and then (A/B /= 5) then`
helps to avoid a possible divide-by-zero Ada constraint error. In this short-circuit form, the latter condition, *A divided by B not equal to five*, will be evaluated only if the former condition, *A not equals zero*, is true. In other words, the short-circuit form `and then` guarantees the order of evaluation, the absence of which may raise an exception.

The Ada program `short_circuit.adb` shown in Listing 5.19 uses the short-circuit operators. Without the use of functions that accept entry calls, the corresponding Promela translation of this Ada program becomes too simple which leads to missing the correct program control flow. The task T1 accepts entry4 at line-5 from a sender's task if function `Fun1` and function `Fun2` returns true, or if function `Fun3` returns true. If these three functions return false, then entry5 at line-7 is accepted. Listing 5.20 shows the specification file for `short_circuit.adb`. Functions `Fun1`, `Fun2` and `Fun3` accepts entry1, entry2 and entry3 respectively from a sender's task before returning a boolean to task T1.

Listing 5.19: `short_circuit.adb`

```
1  task body T1 is
2   begin
3    if (Fun1 and then Fun2)
4       or else Fun3 then
5         accept entry4;
6    else
7         accept entry5;
8    end if;
9  end T1;
```

Listing 5.20: `short_circuit.ads`

```
1  function Fun1 return boolean is
2  begin
3   accept entry1;
4    -- some code to return boolean
5  end Fun1;
6
7  function Fun2 return boolean is
8  begin
9   accept entry2;
10  -- some code to return boolean
11 end Fun2;
12
13 function Fun3 return boolean is
14 begin
15  accept entry3;
16   -- some code to return boolean
17 end Fun3;
```

Figure 5.9 shows the control flow of Ada and then/or else statements. Note that `Fun2`'s decision block is reached only if `Fun1` returns true. The return from `Fun3` is checked when either `Fun1` returns false or when `Fun2` returns false after `Fun1` returns true. If `Fun1` returns true and subsequently if `Fun2` also returns true, then the end state is reached after accepting entry1, entry2 and entry4 from a sender's task. On reaching Fun3, if it returns true, then the end state is reached after accepting entry3 and entry4 from a sender's task. On reaching Fun3, if it returns false, then the end state is reached after accepting entry5 from a sender's task.

Listing 5.21 shows the Promela translation of the Ada and then/or else statements. The three different possibilities to accept entry1 or accept entry3 or accept entry5 are indicated by the non-deterministic choices at lines-5, 6 and 7 respectively. The entry2 is accepted only after accepting entry1 as in lines-5 and 9. The entry4 is accepted after accepting entry1 and entry2 as in lines-5, 9 and 10, or after accepting entry3 as in lines-6 and 10. This order of accepting entries is the same as in the Ada program, confirming the behavioral equivalence between the Ada program

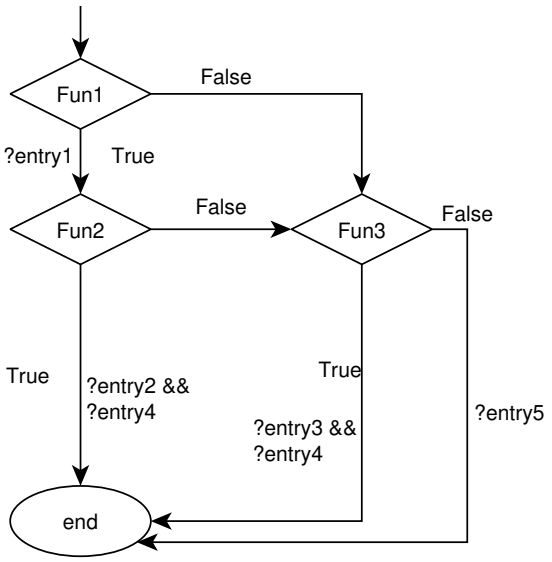


Figure 5.9: CFG of Ada or else/and then

Listing 5.21: Translation of Ada `or else/and then` to non-deterministic transitions in Promela

```
1  mtype = { entry1, entry2, entry3, entry4, entry5 };
2  chan C1 = [0] of { mtype }; /* rendezvous channel */
3  proctype T1(){
4  if
5  :: C1?entry1 -> goto accept_entry2
6  :: C1?entry3 -> goto accept_entry4
7  :: C1?entry5 -> skip
8  fi
9  accept_entry2: C1?entry2 -> goto accept_entry4
10 accept_entry4: C1?entry4 -> skip
11 }
```

and the Promela translation. It is important to note that despite the non-determinism in the Promela translation, the control flow from the Ada program is preserved.

5.2.5 Goto

The `goto` statement corresponds to a jump from the point where it is being executed to another point in the program, with the destination point identified through a label. The `goto` mechanism exists in Ada and Promela with the same semantics, and the syntax is the same in both. Therefore, the translation is direct.

5.3 Chapter Summary

This chapter described different Ada concurrency and conditional constructs and their corresponding Promela constructs.

In general, an Ada software is developed using one or more Ada constructs that we discussed. For example, in a real-life Ada software, it is obvious to see multiple nested levels if/else leading to different concurrency constructs. This means our toolset needs to be built in with algorithms that can handle this complex Ada code structure and translate them into an equivalent Promela model. In the next chapter, we will discuss these algorithms supported by examples.

6 MODEL GENERATION

In Chapter 1, Figure 1.1 showed different stages of the overall approach. This chapter focuses on Stage-2 of this approach, describing how the toolset generates the Promela model and the traceability matrix from the Promela model to the Ada code.

The chapter is organized as follows. Section 6.1 starts with a brief overview of the extraction code that extracts concurrency constructs from the Ada program and represents the extracted information in an XML file format. The extraction code is not a part of this work. However, the validation of the extraction code is a part of this work. The validation ensures that the extracted XML data faithfully represents the concurrency constructs from the Ada code and the control flow.

We proceed to Section 6.2 to understand the generated XML file structure. We will use an example Ada program and illustrate the corresponding (abstract) XML representation of this Ada program. Section 6.3 describes how the extracted XML information is translated to a Promela model. For this translation, we will extensively use the knowledge gained from Chapter 5 on behavior mapping from Ada to Promela. Section 6.4 illustrates how the toolset's translator automates the translation of the XML data to a Promela model. Section 6.5 summarizes the steps until model generation and provides the state space metrics of the generated model. The model generated by the translator cannot be verified by SPIN due to state space explosion. We then proceed to Section 6.6 which describes the technique used to avoid the state space explosion. Section 6.7 illustrates the technique used to generate the traceability matrix from the generated model back to the Ada code. Section 6.8 presents the algorithms developed for the toolset. Finally, Section 6.9 describes the validation of the toolset.

6.1 Extraction Code

This section will briefly describe the extraction code and the Ada library used for the extraction process.

The extraction code is written using the Ada language. It parses Ada source code and retrieves the information needed using the details in an Abstract Syntax Tree (AST). The AST is a tree representing the source code. Each node of the tree represents an operator, and the children of the node represent its operands.

The AST is used as the starting point to – (1) extract the concurrency constructs like signals and locks from the source code and (2) extract the code's control flow leading to these constructs.

6.1.1 Libadalang Library

Libadalang [Ada20c] is a library used for parsing and semantic analysis of Ada code. The library can be integrated into other IDEs and static analyzers. Libadalang supports syntactic analysis

of Ada code with an error recovery feature. The *error recovery* refers to producing a precise syntax tree when the source code is correct and producing a best effort syntax tree when the source code is incorrect. With this feature, the library can even be used on an evolving Ada code.

Besides syntactic analysis, the Libadalang library also supports the semantic analysis of Ada code. The semantic analysis includes *resolution of references* (indicating what a reference corresponds to), *resolution of types* (indicate the type of an expression), and *cross-references* (finding the references of entities). For more details on how to use Libadalang to parse an Ada code and explore the abstract syntax tree of the parsed code, see its user manual [Lib18]. The library is available as open-source at [Ada20c].

The concurrency constructs extracted using the extraction code are output to an XML file. In the next section, we will discuss the structure of this XML file.

6.2 The Intermediate XML Structure

This section describes the structure of the XML file generated using the extraction code. We will use an example Ada program and show the extracted XML file structure for this Ada program.

Listing 6.1: `T1_T2.adb` with conditional if/else construct

```
1  task body T1 is
2   b1: boolean;
3   i1: Integer;
4  begin
5    if i1 = 1 then
6       accept entry1;
7    elsif i1 = 2 then
8      if b1 = true then
9         accept entry2;
10     end if;
11   else
12      accept entry3;
13   end if;
14   accept entry4;
15 end T1;
16 task body T2 is
17  i2: Integer;
18 begin
19   if i2 = 1 then
20      T1.entry1;
21   elsif i2 = 2 then
22      T1.entry2;
23   else
24      T1.entry3;
25   end if;
26   T1.entry4;
27 end T2;
```

Listing 6.2: `T1_T2.ads`

```
1 task type T1_T2 is
2   entry entry1;
3   entry entry2;
4   entry entry3;
5   entry entry4;
6 end T1_T2;
```

The Ada file `T1_T2.adb` in Listing 6.1 shows two Ada tasks, T1 and T2. The tasks synchronize using rendezvous communication. Listing 6.2 is the specification file of the Ada file `T1_T2.adb`. The Ada specification file is similar to a header file in C/C++ containing function definitions, type definitions, packages, etc.

In Listing 6.1, task T1 has a boolean variable `b1` and an integer variable `i1`. We assume that the values set to these variables result from some operation performed in tasks other than T1 and T2. Task T1 has a nested if/else structure with unique condition(s) to reach each of the accept statements. An additional `if` nesting is introduced at line-8 to understand how this (nested) control-flow is preserved in the generated model (discussed in the subsequent paragraphs).

T2 has an integer variable `i2` whose value is set by some operation performed in a different task, other than T1 and T2. The task has a nested if/else structure, simpler than that of T1, with unique conditions to reach each entry call.

The entry calls involving rendezvous between T1 and T2 are declared in the specification file `T1_T2.ads`. Assume that the task type `T1_T2` is defined within a package declaration, and the corresponding package body is placed in the *.adb file.

```
1  <task_sync>
2  <task name="T1" location="TaskBody T1_T2.adb:1">
3    <if location="IfStmt T1_T2.adb:5">
4      <then>
5        <accept target="entry1" targetLocation="DefiningName T1_T2.ads:2"/>
6      </then>
7    <elsif location="IfStmt T1_T2.adb:7">
8      <then>
9        <if location="IfStmt T1_T2.adb:8">
10         <then>
11           <accept target="entry2" targetLocation="DefiningName T1_T2.ads:3"/>
12         </then>
13       </if>
14     </then>
15   </elsif>
16   <else location="IfStmt T1_T2.adb:11">
17     <accept target="entry3" targetLocation="DefiningName T1_T2.ads:4"/>
18   </else>
19   </if>
20    <accept target="entry4" targetLocation="DefiningName T1_T2.ads:5"/>
21 </task>
22 <task name="T2" location="TaskBody T1_T2.adb:16">
23   <if location="IfStmt T1_T2.adb:18">
24     <then>
25       <rendezvous target="entry1" targetLocation="DefiningName T1_T2.ads:2"/>
26     </then>
27   <elsif location="IfStmt T1_T2.adb:20">
28     <then>
29       <rendezvous target="entry2" targetLocation="DefiningName T1_T2.ads:3"/>
30     </then>
31   </elsif>
32   <else location="IfStmt T1_T2.adb:22">
33     <rendezvous target="entry3" targetLocation="DefiningName T1_T2.ads:4"/>
34   </else>
35   </if>
36   <rendezvous target="entry4" targetLocation="DefiningName T1_T2.ads:5"/>
37 </task>
38 </task_sync>
```

Figure 6.1: `T1_T2.xml` showing the concurrency constructs extracted from `T1_T2.adb`

We will now see how the Ada file `T1_T2.adb` looks when its concurrency constructs and the control flow are extracted into an XML file. Figure 6.1 shows the extracted task synchronizations (`<task_sync>`) - accept and entry calls from `T1_T2.adb`. The XML contains the location of if/else conditions in the code. For example, line-3 of the XML shows that the first `if` condition is at line-5 of the code. As mentioned in Section 6.1, our abstraction decision is to extract only the code's concurrency constructs along with the code's control-flow; we do not take into account the code's conditional variables and their values. Therefore, details from the *.adb file, like, the boolean and integer variables, are not part of this XML.

Table 6.1 lists some of the tags in the XML and their meaning. Besides the tags `<accept>` and `<rendezvous>`, the `<WaitForEvent>` tag is used to identify a waiting Ada event (see Section 5.1.4); the `<SetEvent>` tag is used to identify the sending of Ada event. The `<select>` tag identifies the Ada selective accept statement (see Section 5.1.5). The `<delay>` tag along with the `<select>` tag identifies the Ada selective accept statement with delay (see Section 5.1.6). The tags that represent the Ada conditional statements have tag names matching the original Ada code. For instance, the Ada `if` statement is indicated by the XML tag `<if>`, the Ada `elsif` statement is indicated by the XML tag `<elsif>` and the Ada `case` statement is indicated by the XML tag `<case>` etc. For simplicity, these tags are not listed in Table 6.1.

XML tag name	**Meaning**
`<accept>`	Ada accept statement
`<rendezvous>`	Ada entry call
`<SetEvent>`	Ada user-defined type that sets an event
`<WaitForEvent>`	Ada user-defined type that waits for an event
`<select>`	Ada select statement
`<delay>`	Ada delay statement

Table 6.1: Some XML tags and their meaning

Back to Figure 6.1, the accept statements from the Ada file are identified by the `<accept>` XML tag and the entry calls are identified by the `<rendezvous>` XML tag. Note that it may be possible to have entry calls with identical names from other tasks; for example, `entry1` at line-25 may also be an entry call from another task, say T3, accepted by a task T4. It is therefore important to distinguish the entry calls that are specific to a task. This entry call distinction is achieved using the `targetLocation` XML attribute.

Suppose the targetLocation of an entry call (i.e., `<rendezvous>` XML tag) of a task is the same as the targetLocation of an accept statement (i.e., `<accept>` XML tag) of another task. In that case, there can be a rendezvous communication between these tasks at this control point in the code. For example, line-5 and line-25 show that the target and targetLocation attributes of the entry call of T1 are the same as the target and targetLocation attributes of the accept statement of T2 (highlighted in red). Therefore, a rendezvous is possible between these two tasks at this control point when T2 makes the entry call `entry1`. Note that the targetLocation is nothing but the entry call `entry1` at line-2 of the specification file `T1_T2.ads`.

Rendezvous communication can also take place in the following situations:

1. Entry calls from multiple tasks can service an *accept* of a task if the target and targetLocation of each of these entry calls are the same as the target and targetLocation of the accept statement.
2. An entry call can occur multiple times within the same task to service an *accept* of another task. The condition of identical target and targetLocation must hold for this.

These multi-task synchronization scenarios will be discussed in detail in Section 6.8.

6.3 Translating the Intermediate XML to a Promela Model

We have seen how the concurrency constructs and the control flow extracted from the Ada program are represented in an XML file. Next, we discuss how this extracted data is translated to an NDFA in Promela. For this translation, we will use the knowledge on behavior mapping from Ada constructs to Promela constructs discussed in Chapter 5.

As an example, we will see how to generate the Promela model in Listing 6.3 from the XML in Figure 6.1. The following elements are required for model generation.

1. The model should have synchronization messages corresponding to the Ada entry calls.
2. The model should have channels that support synchronization between Promela processes; this corresponds to synchronization between Ada tasks.
3. The model should have synchronization points. The locations in the model having a synchronization correspond to the program's location at which this synchronization can occur.
4. The model should have meaningful state names such that these names correspond to the control point in the program where synchronization can occur.
5. The model should have state transitions that respects the program's control-flow.

We will now discuss, one by one, how these elements should be generated for the Promela model from the XML. In other words, this section describes the requirements for the model generation (*what needs to be done*). The next section on the toolset's translator describes the implementation of these requirements (*how it is done*).

1. Define synchronization messages:
   Using the XML in Figure 6.1, we have seen how a rendezvous is possible between two tasks with the same target and targetLocation attributes. With this understanding, a *task pair* is defined as in Definition 6.1.

   Definition 6.1. If T1 and T2 represent two Ada tasks in the XML file, a 2-tuple (T1, T2) is called a task pair if any one of the following is true:

   - T1 has a `<accept>` tag and T2 has a `<rendezvous>` tag or vice versa, AND both these tags are bound to the same entry call, OR
   - T1 has a `<WaitForEvent>` tag, and T2 has a `<SetEvent>` tag or vice versa, AND both these tags are bound to the same entry call.

   Informally, 'bound to the same entry call' refers to having the same `target` and `targetLocation` attributes.

   With this definition, an algorithm should be developed to define entry calls of task pairs from the XML as *synchronization messages* in Promela. The extracted synchronization messages (or sync messages) will be in the Promela's `mtype` list. Lines-1,2,3 of Listing 6.3 define the sync messages. The naming convention of each `mtype` is such that it starts with the name of the receiving task. For example, the entry call `entry1` is defined as `T1_entry1`. This convention helps to avoid duplicate `mtype` names if there is more than one task sending an entry call of the same name.

2. Define channels:
   After identifying the paired tasks using Definition 6.1, a synchronization channel (or sync channel) must be defined for communication between the paired tasks. By convention,

a sync channel is named such that the task sending the message (!) appears first in its name. For example, the channel definition at line-4 of Listing 6.3 has the name `T2_to_T1`, where T2 is the task sending the message. This convention helps identify a channel uniquely if the same task sends entry calls to more than one receiving task.

In this example, there is a rendezvous communication between T2 and T1. Therefore, the channel capacity is set to zero as in line-4. But, the Promela processes for Ada events (see Section 5.1.4) or Ada selective accept with delay (see Section 5.1.6) use asynchronous communication. In these cases, the toolset should automatically define buffered channels with a default channel capacity of two.

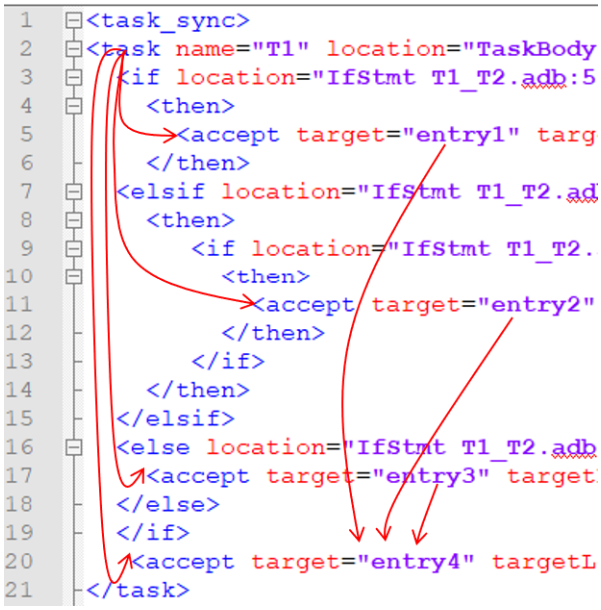


Figure 6.2: Marking non-deterministic transitions in the XML

Listing 6.3: Promela model `T1_T2.pml` of `T1_T2.adb`

```
1  mtype =
2  {T1_entry1,T1_entry2,
3  T1_entry3,T1_entry4};
4  chan T2_to_T1 = [0] of {mtype};
5  proctype T1(){
6  initial:
7  if
8  :: T2_to_T1?T1_entry1 ->
9     goto T1_accept_entry4
10 :: T2_to_T1?T1_entry2 ->
11    goto T1_accept_entry4
12 :: T2_to_T1?T1_entry3 ->
13    goto T1_accept_entry4
14 :: T2_to_T1?T1_entry4 -> skip
15 fi;
16 T1_accept_entry4:
17 T2_to_T1?T1_entry4 -> skip
18 }
19 proctype T2(){
20 initial:
21 if
22 :: T2_to_T1!T1_entry1 ->
23    goto T2_rendezvous_entry4
24 :: T2_to_T1!T1_entry2 ->
25    goto T2_rendezvous_entry4
26 :: T2_to_T1!T1_entry3 ->
27    goto T2_rendezvous_entry4
28 fi;
29 T2_rendezvous_entry4:
30 T2_to_T1!T1_entry4 -> skip
31 }
32 init {atomic {T1(); T2()}}
```

3. Define synchronization points:
   After defining the synchronization messages (mtype) and the channels, the synchronization points (or sync points) that use these elements should be defined at the *appropriate* locations in the model. Here the term *appropriate* location refers to the location in the model that corresponds to the Ada program's control point at which synchronization can happen.

   The sync points for T1 are lines-8,10,12,14 and 17 of Listing 6.3. Line-8, for example, indicates the possibility to receive the message `T1_entry1` using the channel `T2_to_T1`

from T2. The right side of the arrow separator (see Section 3.3.1.2) indicates the state transition to `T1_accept_entry4` as a result of the synchronization. Note that the sync points in Promela respect the Ada program's control points at which the two tasks can synchronize.

4. Define states:
   The control points in the program containing concurrency constructs should be translated to states of the model. For example, line-20 of Figure 6.1 is a control point in the program with the `<accept>` construct.

   A meaningful state name should be chosen to identify the program's corresponding control point in the model. For example, from the state name `T1_accept_entry4` on line-9 of Listing 6.3, we can infer that this state corresponds to the program's control point at which task T1 accepts entry4 — line-14 of Listing 6.1.

5. Define transitions:
   The *control flow* in the program containing concurrency constructs should be translated to state transitions in the model. To understand how the transitions are defined, we will use a marked version of the XML in Figure 6.1, with arrows indicating transitions. This marked version is shown in Figure 6.2.

   Line-2 of Figure 6.2 is the starting point of task T1 that corresponds to the initial state at line-6 of the Promela model in Listing 6.3. The red arrows in the figure indicate that it is possible to have the following transitions from line-2:

   (a) go from line-2 to line-5 when the `if` condition at line-3 is true

   (b) go from line-2 to line-11 when the `if` condition at line-3 is false and the `elsif` condition at line-7 is true

   (c) go from line-2 to line-17 when the `if` condition at line-3 is false and the `elsif` condition at line-7 is also false

   (d) go from line-2 to line-20 when the `if` condition at line-3 is false, the `elsif` condition at line-7 is true, but the `if` condition within `elsif` at line-9 is false.

   These four control flow paths in the program correspond to line 7 to line 15 in Listing 6.3. At most, one of the three entries - T1_entry1, T1_entry2, T1_entry3 is accepted by task T1, as in line 8 to line 13, before reaching the entry4's accept state at line-16. Line-14 indicates that it is possible to bypass these three entries and directly accept T1_entry4. The process is exited after accepting T1_entry4 from T2 as in line-14 and line-17.

   An algorithm that automates the above translation should not allow transitions that cannot happen in the program's control flow. For example, it is incorrect to have a transition in the model from entry1's accept state (that corresponds to `if` block of the Ada code) to entry2's accept state (that corresponds to the `elsif` block of the Ada code). Retaining such invalid transitions in the model, besides increasing the state space, will lead to having spurious error traces if the model checker detects violations along these invalid transitions (this is over-approximation).

   On the other hand, the algorithm should allow all possible transitions in the program's control flow. For example, when the condition of the inner `if` at line-9 of Figure 6.2 is false, even though if the `elsif` at line-7 is true, the control should exit the main `if` block to reach entry4's accept state at line-20. This means there must be a transition in the model from entry1's accept state to entry4's accept state. Failing to include such *valid* transitions leads to the model having less behavior than the code. Consequently, this increases the chance of missing a real problem in the code (this is under-approximation).

In summary, the required algorithm for the control flow translation should be able to:(a) retain valid transitions that are possible to occur in the code's control flow, and (b) exclude the invalid transitions that are not possible to occur in the code's control flow.

This section listed the different elements required to generate a Promela model from the XML. The next section describes how the model is generated automatically using the toolset's translator.

6.4 The Toolset's Translator

The previous section shows how the XML represents the Ada program's extracted concurrency constructs. We then discussed how the extracted information in the XML is to be translated into different elements (like sync messages, channels, states, and transitions) to generate a Promela model of the Ada program.

In this section, we will discuss how the toolset's translator automates the above-described steps. The translator is coded in the Python programming language using an object-oriented approach.

The rest of this section is organized as follows. We start with the class diagrams of the translator. The class diagrams show the different classes used to parse the XML data and generate a Promela model from the parsed XML. We then present an XML to Promela translation scheme. This scheme uses an XML as input to show how the XML data is parsed to generate a Promela model.

6.4.1 Class Diagrams of the Translator

We will now present the translator's class diagrams. The translation of the XML to Promela is a two-step process. The first step is the parsing of the XML data. The parsing is implemented using the class `XmlParser`. The class diagram of `XmlParser` is shown in Figure 6.3. The second step is generating the Promela model from the parsed XML data. The model generation is implemented using the class `PromelaWriter`. The class diagram of `PromelaWriter` is shown in Figure 6.4. The class diagrams show the complete list of class instances and the generalization relationship (shown by the arrows) between them. The generalization relationship shows the relationship between the parent class and its child classes (inheritance).

Note that a class name starting with an "X" corresponds to an operation performed on the XML data. For example, `XModule` and `XTask` represents the XML module and the task in the XML (<task> tag) respectively. Similarly, a class name starting with a "P" corresponds to an operation performed for generating the Promela model. For example, `PModule` and `PProcess` represents the Promela module (corresponding to `XModule`) and the Promela process (corresponding to `XTask`) respectively. The meaning of some of the class names shown in the class diagrams is described in Table 6.2.

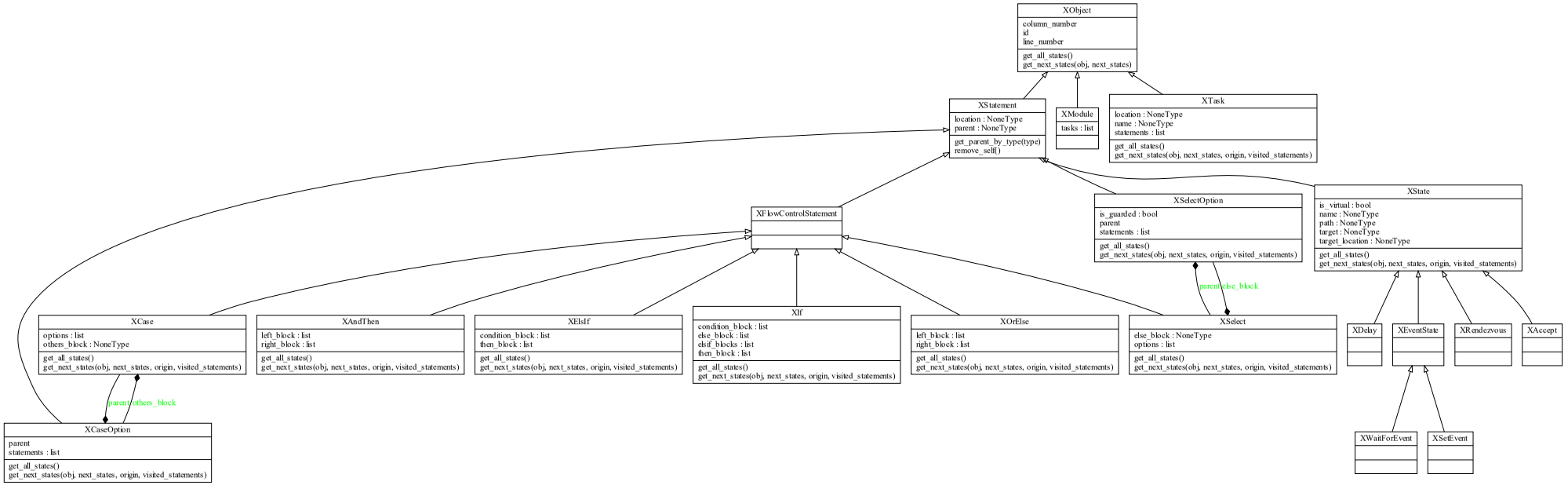


Figure 6.3: Class diagram of XmlParser

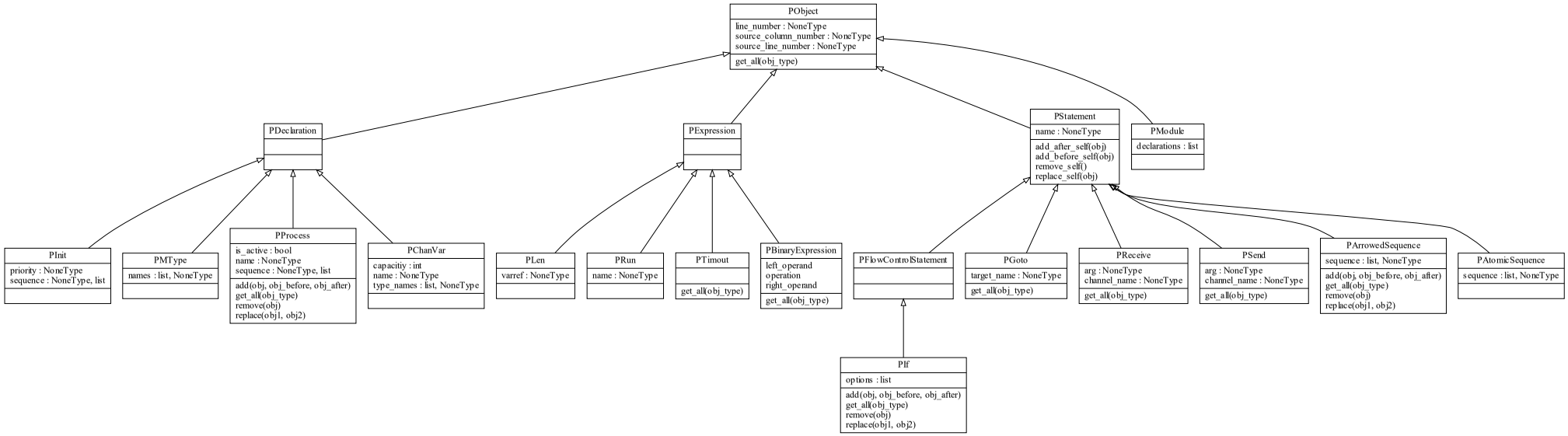


Figure 6.4: Class diagram of PromelaWriter

Class	Meaning
XModule	**XModule** represents the input XML in the form of a statement tree in order to make it easier to perform further analysis and data transformation
XTask	**XTask** is a container of statements corresponding to the "task" tag in XML. This instance further has list of instances of the classes **Xif**, **Xaccept** etc.
XState	**XState** represents the XML tags <accept>, <rendezvous>, <WaitForEvent> etc., that translates to states in the model; these are state tags
PModule	**PModule** represents the Promela code in the form of a statement tree
PChanVar	**PChanVar** represents Promela's channel declaration
PProcess	**PProcess** represents the Promela's "proctype" corresponding to each of the XML's "task" tag represented by **XTask**

Table 6.2: Some classes used in the class diagrams and their meaning

6.4.2 XML to Promela Translation- An Example

In the previous section, we have seen the class diagrams of the translator. This section will illustrate how the different classes are used to generate the Promela model from the XML.

Let us consider the XML in Figure 6.1 as the input to the translator. Figure 6.5 shows the scheme describing the translation of this XML to the Promela model, `T1_T2.pml` shown in Listing 6.3.

The scheme uses the following semantics. Container blocks are used to represent classes, the input file, and the output file. Dotted lines with arrows indicate the class dependencies. The dependency, more specifically the usage dependency, indicates that the source element uses (or depends on) the target element. The arrow's tail is at the source element, and the head of the arrow is at the target element. The solid lines without arrows indicate the class associations. The class association, more specifically the binary association, refers to the link between the class instances required to communicate with each other. In other words, the association represents the general relationship between instances of the classes.

Starting from the top of this scheme, the `XmlToPromelaTranslator` is the main class of the toolset's translator. Towards the left of this main class are the steps that describe the parsing of the XML data, and towards the right of the main class are the steps that describe the generation of a Promela model from the parsed XML.

The class `XmlToPromelaTranslator` has four major functions, namely, `translate()`, `_translate_module()`, `_translate_task()` and `_translate_state()`. Note that the class names are preceded by colon (:). The class `XmlParser` has `parse()` as its major function, and the class `PromelaWriter` has `write()` as its major function. The functions used in the translator and their purpose is described in Table 6.3.

On a high-level view, the `translate()` function of the main class converts the instances of the `XModule` to instances of `PModule`, and this `PModule` instance is written into an output Promela (*.pml) file.

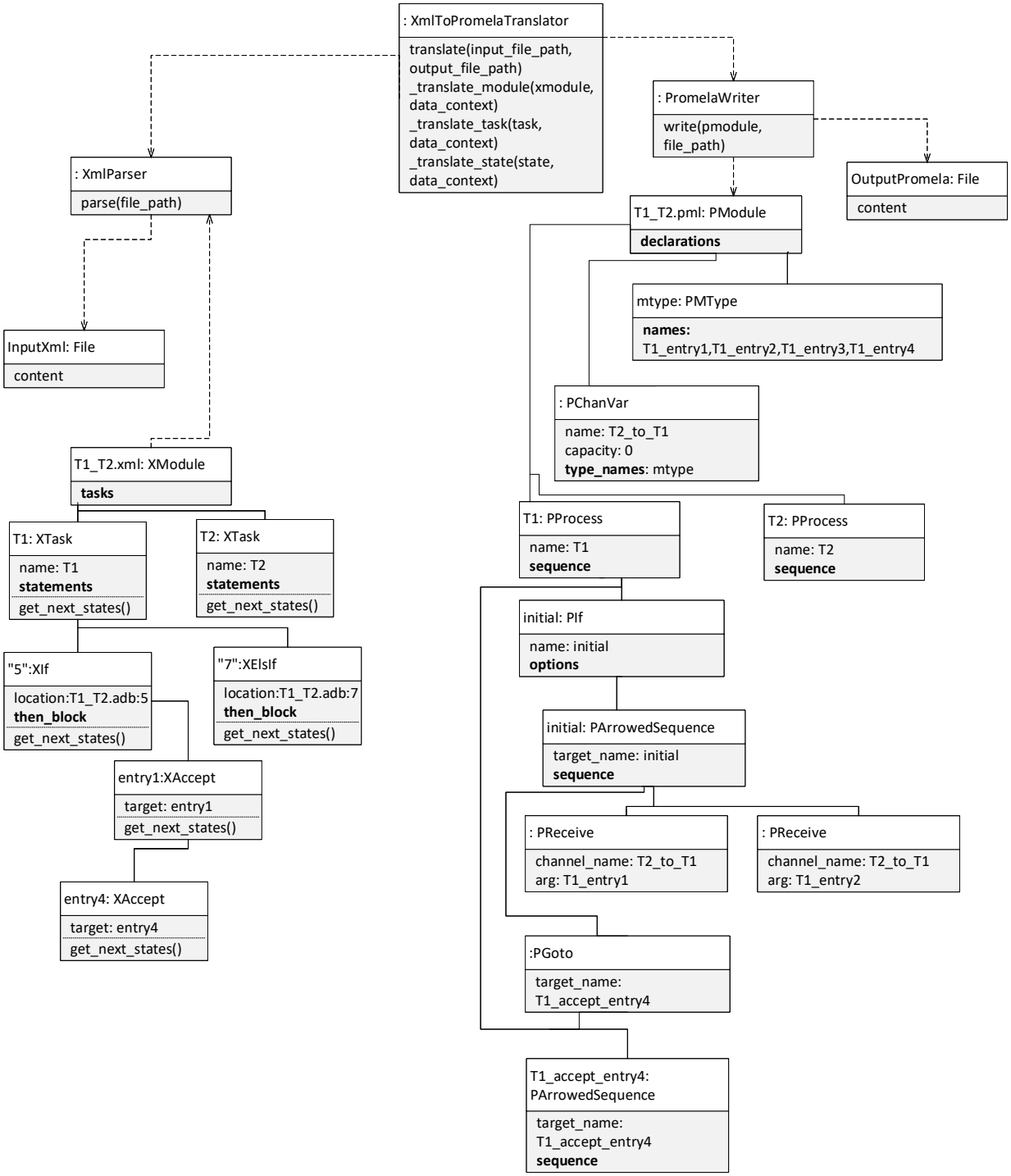


Figure 6.5: The toolset's translator scheme showing a section of the translation from T1_T2.xml to T1_T2.pml

6.4.2.1 Parsing the XML Data - the Left Side of the Translation Scheme

Having all the details about the classes and their primary functions, we now go back to Figure 6.5 to understand the translation scheme. We start with the left side of the main class.

The class `XmlParser` parses the input XML file to convert it into a tree structure; we call this tree an `XModule` tree. Remember the XML in Figure 6.1 contained different tags like `<task>`, `<if>`, `<accept>`, `<rendezvous>` etc. All these tags together form the `XObject` (or XML's object), and

each of these individual tags turns into an instance of the corresponding `XObject` class, like, `XTask`, `Xif`, `XAccept`, `XRendezvous`, etc. Each of these instances can be parsed independently using dedicated functions like `_parse_task`, `_parse_if`, etc.

Translator function name	Purpose
translate(input_file_path, output_file_path)	Serves three major purposes: 1. Parse XML file to create **XModule** instance 2. Convert **XModule** instance to **PModule** instance 3. Write **PModule** instance to a *.pml output file
_translate_module(xmodule, data_context)	Convert a list of **XTask** instances to a list of **PProcess** instances. Besides this, other operations performed are: 1. Add synchronization 2. Add channel 3. Convolution of tasks (reduce statements quantity to focus only on synchronized statements)
_translate_task(task, data_context)	1. Find the next state transitions for each of the task states (task states are nothing but instances of **XState** classes like **XAccept**, **XSetEvent** etc.) 2. After that, the sequence of states is converted into a sequence of Promela statements ("if" or "goto" statements). 3. The result is a **PProcess** instance that is initialized with this sequence.
_translate_state(state, data_context)	Using the task state (i.e., instances of **XState** like **XAccept**, **XSetEvent** etc.) and a list of the next state transitions, *_translate_state* function creates: 1. Promela "goto" statement if next states list has only one statement 2. Promela "if" statement if next states list has more than one statement. This is nothing but a set of non-deterministic transitions in Promela
parse(file_path)	Parse XML file to **XModule** tree. Each node of the XML tree turns into an instance of the corresponding **XObject** class like **XTask**, **XIf**, **XAccept**, **XRendezvous** etc. by calling the appropriate methods like **_parse_if**, **_parse_orelse**, **_parse_select** etc.
write(pmodule, file_path)	For each instance that inherits the **PObject** class, the corresponding method for converting the instance to Promela text is called recursively. The resulting text is saved to a *.pml file

Table 6.3: Toolset's translator functions and their purpose

So far, the input `T1_T2.xml` is converted into an `XModule` tree as indicated by the third container from the top in Figure 6.5.

The `XModule` tree has two tasks, T1 and T2, as indicated by the two `XTask` containers. Each of the `XTask` instances then looks for conditional tags like `<if>`, `<elsif>`, `<case>` etc., within each task. Furthermore, each of the `XTask` instances then recursively looks for next state transitions using the function `get_next_states()`.

The idea of the next state transition (`get_next_states()`) search is to recursively traverse a portion of the XML's statement tree. More details on this recursive traversal are provided in

Section 6.8 under Algorithm 4.

The search for conditional tags like <if>, <else>, etc., and the call to next transitions retrieves all the conditional statements along with their location. For simplicity, the traversal of only the first <if> and the first <elsif> of the XML in Figure 6.1 are shown in two containers, XIf and XElsIf, in the scheme. These statements' locations are retained for generating traceability from Promela to the Ada program (discussed later in Section 6.7).

From the current <if> block, the XML tree search continues to the next <then> block at line-4 of the XML, and again the search for the next state transition continues. When the search reaches the <accept> tag, indicated by the XAccept container in the scheme, the accept state of entry1 is stored in a list.

The search continues to reach the accept state of entry4, and this state name is appended to the list of states. A further search of the XML tree results in reaching the end of the task, and the search stops. The list of states collected is returned to the calling function get_next_states().

The data parsed from the XML like, states, state transitions, entry calls, etc. by the class XmlParser is returned to the main class XmlToPromelaTranslator. This main class now uses its four major functions to translate the parsed XML data to a Promela model.

6.4.2.2 Generation of Promela Model- the Right Side of the Translation Scheme

In the previous section, we have seen how the XML data is parsed. The next step is to understand how this parsed XML data is translated to generate a Promela model. To understand this model generation process, we now move on to the right side of the main class in the scheme in Figure 6.5.

The class PromelaWriter, with the help of the function write(), is responsible for generating the output Promela file. To understand more about this function, we first need to know the Promela's PObject class, its instances, and the role of these instances in generating the Promela model.

Similar to XObject, there is PObject. The PObject class is the parent class for generating the Promela file. This parent class is extended by several class instances. These instances provide the required elements for the Promela model like, sync messages (mtype), channel names, message sending (!), message receiving (?), etc. Similar to XObject's instances, the PObject's instances have names that indicate their role in the generation of Promela. For example, the PObject's instance PMType defines the sync messages, the PObject's instance PChanVar defines the channel names etc.

The third container on the right side from the top represents the Promela file T1_T2.pml, which is nothing but the PModule. Similar to XModule, the PModule represents the Promela code in the form of a statement tree.

In Section 6.3, we have discussed the different elements required (sync messages, channel definitions, sync points, states, and state transitions) to generate a Promela model. Using the scheme, we will describe how these elements are written step-by-step to an output Promela file.

Step-1: Define sync signals (mtype), channel names and processes (proctype)
The two instances, PMType and PChanVar provide the required sync messages and the channel declarations, respectively. This is indicated by the fourth and fifth containers from the top.

The instance PProcess defines the Promela's *proctype* (similar to XTask) T1 and T2, indicated by two containers, T1:PProcess and T2:PProcess respectively, in the scheme.

Step-2: Define states
After defining sync messages, channels, and proctypes, the next step is to define the states.

If the XML structure had just one next state, there would be only one state transition in Promela. However, in our XML example, from the starting pointing of the task, there are four possible transitions as seen in Figure 6.2. Therefore, the `PProcess` instance creates a non-deterministic set of choices from the initial state using the instance `PIf`, as shown by the container `initial:PIf` in the scheme. This operation is performed by the function `_translate_task()` by translating the sequence of states data obtained from XML into a sequence of Promela statements.

Step-3: Define synchronization points
The function `_translate_module()` converts the task data obtained from XML to create synchronization points in Promela. This is indicated by the container having the `PArrowedSequence` in the scheme. The `PArrowedSequence` is nothing but the combination of channel name, sync type, sync message and the arrow separator ("->") in Promela.

The appropriate type of synchronization, i.e., send (!) or receive (?) is generated by calling the instances `PSend` or `PReceive`, respectively. An appropriate instance is called based on whether the `XState` retrieved an `<accept>` or a `<rendezvous>` tag in the XML. Since the XML in Figure 6.1 has `<accept>` tags for task T1, the `PReceive` instance is used as shown in two `PReceive` containers in the scheme.

Step-4: Define next state transitions
After generating the left side of the arrow separator (channel+sync type+sync message), the resulting next state transition, after the sync, is generated using the `PGoto` instance. This is indicated by the last but one container in the scheme. Each `PProcess` instance determines the state transitions in Promela based on the data provided by its XML counterpart, `XTask`.

The resulting state from `PGoto` is another accept state, and therefore the `PArrowedSequence` is used again. The last container in the scheme indicates this. For simplicity, only the Promela proctype T1 generation is shown in the scheme instead of both T1 and T2. Also, only two instances of `PReceive` are shown for simplicity.

Until now, we have seen how the toolset's translator generates the Promela model from the parsed XML data with the help of a scheme. The next section summarizes the overall steps and the outcome of the translation.

6.5 Discussion

This section summarizes the overall steps until the generation of the Promela model, followed by describing the generated model's state space metrics.

We started with the Ada program of the software system under consideration. The Ada program has more than one million lines of code. We used the extraction code to retrieve only the concurrency constructs from the Ada program, along with the program's control flow. This extracted information is output to an XML file.

We then explained how the extracted data in the XML is translated to a Promela model. For this translation, the different elements required like sync messages, channels, sync points, states, and transitions are listed.

Later, we have seen how the translation from XML to Promela is done automatically. For this, the toolset's translator is illustrated. This illustration includes a scheme that shows the step-by-step process of generating the Promela model.

The Ada program is of more than one million lines. The generated Promela model of this Ada program is of 73,232 lines with 63,655 state transitions. From now on, we refer to this generated Promela model as “model.”

Because of the model’s large state space, even SPIN’s supertrace verification (see SPIN’s verifier modes in Section 3.3.3) failed to complete by returning an out-of-memory error. The next challenge is to find technique(s) to reduce the state space of the model without compromising the code to model behavior.

In the next section, we will describe the technique used to reduce the generated model’s state space.

6.6 State-Space Reduction

Program slicing [Wei84] is used to reduce the state space of the generated model. In program slicing, all the program lines that do not affect the property to be verified are removed. This helps in reducing the state space of the model. An example is, given a property that checks the value of a specific variable X, all other variables that do not affect X’s value can be removed.

In our context, the model lines that do not affect the property to be verified (deadlock/livelock) can be removed from the model. It is sufficient to retain only those states and transitions that participate either directly or indirectly in process synchronization, as it is impossible for transitions without synchronization to cause a deadlock/livelock.

The transitions without synchronization in the model results from “unpaired synchronization” in the extracted XML data. Unpaired synchronization refers to those concurrency constructs in the XML that do not have a counterpart to synchronize with. For example, an `<accept>` tag without a `<rendezvous>` counterpart cannot complete a synchronization. Similarly, a `<WaitForEvent>` tag without a `<SetEvent>` counterpart cannot complete a synchronization. In the next section, we will look into the metrics and reasons for unpaired synchronizations in the extracted XML data.

6.6.1 Unpaired Synchronizations

In this section, we will see how the toolset helps in automatically identifying the paired/unpaired synchronizations. Subsequently, we will identify the root cause(s) for unpaired synchronizations by analyzing the Ada code.

The toolset is built-in with an algorithm to automatically analyze the paired and unpaired synchronizations in the XML. The algorithm is based on the task pair Definition 6.1. When the toolset’s python code is executed, along with the Promela model and the traceability matrix, it also generates a CSV file containing paired/unpaired synchronizations.

A small section of the generated CSV file is shown in Table 6.4. The first column contains the name of the sync signal, the second column indicates the task to which the signal belongs, the third column indicates the tag type, and the fourth column indicates the tag location in the XML. The fifth column lists the paired task for the task in the second column, the sixth column lists the paired tag for the tag type in the third column, and the seventh column indicates the paired tag’s location in the XML. Note that columns 5, 6, and 7 provide the paired/unpaired information for the signal in column-1.

Let us see some examples as to how to read the table. Consider the `Wait` signal. From the table, it can be understood that the task `Schedule_Task` waits to accept the `Wait` signal sent by the task `Identify_Task`. In other words, the signal `Wait` corresponds to a paired synchronization.

On the other hand, the signal `Stop` that belongs to `Compute_Task` corresponds to an unpaired synchronization.

Target	Task 1	Task 1 tag	Task 1 tag line number	Task 2	Task 2 tag	Task 2 tag line number
Wait	Schedule_Task	accept	6	Identify_Task	rendezvous	1873
Stop	Compute_Task	accept	11			
Proceed	Inspect_Task	SetEvent	12	Roll_Task	WaitForEvent	1873

Table 6.4: A small section of the synchronization list CSV

A detailed analysis of the number of paired/unpaired synchronizations in the XML is shown in Figure 6.6. Some key observations from the figure are listed below.

1. 25% of the `<accept>` tags do not have a `<rendezvous>` pair,
2. 76% of the `<rendezvous>` tags do not have an `<accept>` pair,
3. 45% of the `<WaitForEvent>` tags do not have a `<SetEvent>` pair, and
4. 9% of the `<SetEvent>` tags do not have a `<WaitForEvent>` pair.

Of all the unpaired syncs, the number of `<rendezvous>` tags without an `<accept>` pair is the highest. A further investigation of this set of unpaired tags showed that there is one particular task with 1271 `<rendezvous>` tags without an `<accept>` pair. That is, this task alone accounts for 73% of the unpaired synchronizations of this set. This particular task requires further analysis (future work).

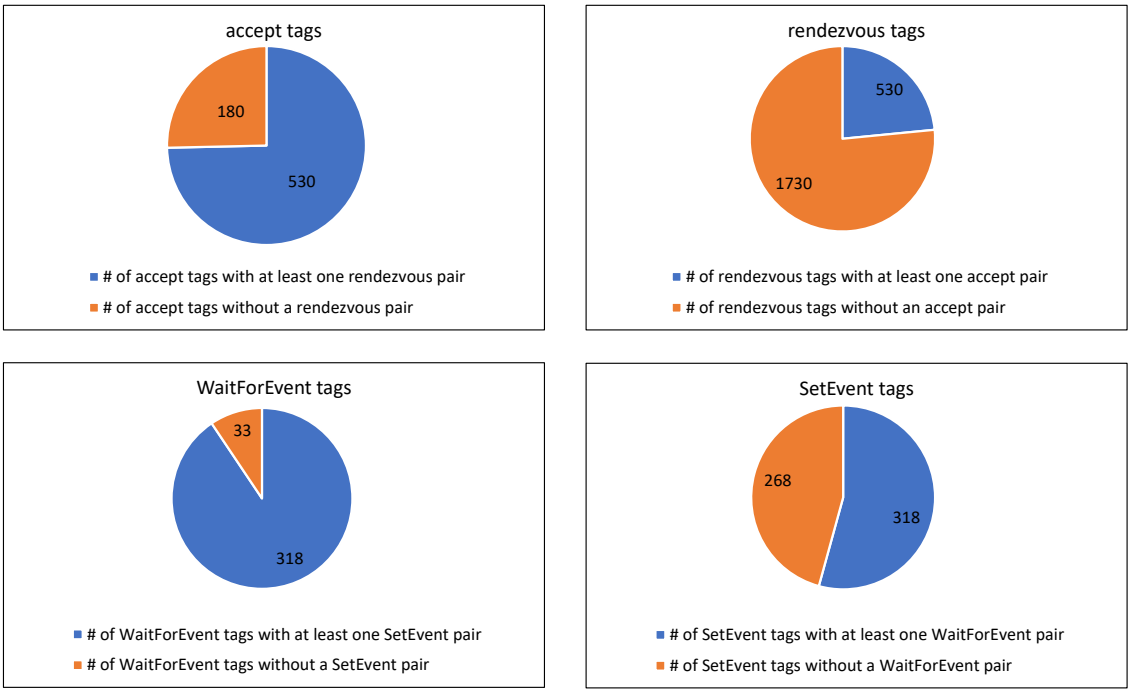


Figure 6.6: Analysis of paired and unpaired synchronizations in the XML

Until now, we have used the XML data to analyze paired/unpaired syncs automatically. The metrics from Figure 6.6 raises an important question:
The Ada code under consideration is a single code base for a family of machines. This means the code should be "somehow" fully connected. In other words, it is unlikely to have unpaired

synchronizations in the code, although this is what is seen from the extracted XML. What is/are the possible reason(s) for unpaired syncs in the XML?

Let us answer this question by identifying the root cause for unpaired syncs, which requires analysis of the Ada source code. We will now see the outcome of the Ada code analysis for unpaired syncs. Note that the toolset's sync list CSV indicates which specific tags do not have a pair by listing the tags' line numbers. This provided a head-start for the code analysis.

The analysis of the Ada code revealed five reasons for unpaired synchronizations in the extracted XML data. They are: task-level analysis instead of thread-level analysis, search depth for concurrency constructs, triggers from the external software interface, triggers from external hardware source, and not considering shared memory for model generation.

We will now look into these reasons one by one.

1. Task-level analysis instead of thread-level analysis:
   An Ada task becomes a thread at run-time. The extraction code analyzes all task bodies to extract the concurrency constructs. However, it is possible to have a thread of the Ada program that is not in a task body. This thread may have concurrency constructs, which are not covered by the extraction code.

   A concurrency construct present at the thread-level may be a synchronization counterpart of the concurrency construct present at the task-level. As the extraction code analysis is based only on task-level, it may lead to having unpaired synchronization with its thread-level counterpart.

   We will now describe, with an example, how paired synchronization can occur between a concurrency construct at the thread level and a concurrency construct at the task level.

   Paired synchronization at the program's thread-level may be represented as shown in Figure 6.7. At the start of the program's execution, the program's primary thread (Ada program primary thread) also begins its execution. The primary thread of the Ada program is not in a task body. This primary thread, through function call(s), may lead to a function, and this function may have the Ada `accept` construct waiting to receive the message `m1`. On the other side, the active Ada task T1 may have the Ada entry call construct sending the message `m1`.

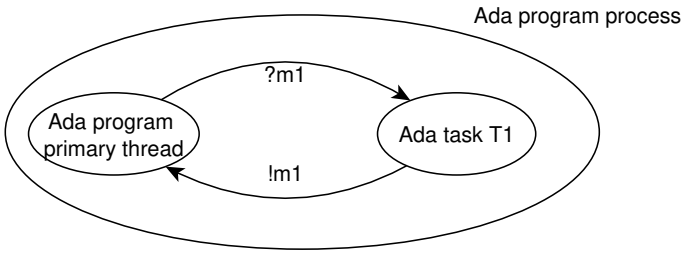


Figure 6.7: Synchronization at program's thread level

   As the extraction code looks only for concurrency constructs in the Ada task body and not at the thread level, the `accept` construct in the primary thread may be missed. In other words, the output XML will have only the entry call found in the body of task T1, without its `accept` counterpart.

   In summary, the Ada code analysis only at the task level and not at the thread level is the first reason for having unpaired syncs in the XML. Furthermore, the analysis showed

that this is the primary reason for the maximum number of unpaired syncs in the XML. The extraction code can be extended to perform a thread-level analysis for concurrency constructs. This is future work.

2. Search depth for concurrency constructs:
   The extraction code has a *call depth* parameter. This parameter limits the depth of search for synchronization primitives in the Ada code. This means it may be possible to have paired synchronizations at a larger search depth, which are not found. Currently, the call depth is set at five. It is also observed that the extraction code took several hours to finish for higher values of call depth (greater than eight). In summary, the extraction code's limitation on call depth is the second reason for unpaired syncs in the XML.

3. Triggers from external software interface:
   The scope of analysis is mainly the Ada software under consideration. However, it is seen that there are synchronizations triggered from outside of this software (an external software interface). Such external software triggers are not considered in our analysis. Including these external software factors lead to a too wide analysis scope. In summary, not considering all the code-dependencies for analysis by the extraction code is the third reason for unpaired syncs in the XML.

4. Triggers from external hardware source:
   Besides external software triggers, it is also observed that there are synchronizations in the code that are triggered from external sources (for example, hardware sources). Such external sources are not considered in our analysis. Including these triggers lead to a too wide an analysis scope. In summary, not considering the triggers from external sources for analysis by the extraction code is the fourth reason for unpaired syncs in the XML.

5. Shared memory:
   The extracted XML only has two types of task communications from the Ada code: rendez-vous-based and event-based. Besides these, tasks in the code can also communicate using shared memory. The extraction code could retrieve the lock and unlock mutexes used for shared data in the Ada code (this is not part of this work). However, the shared data is not considered for our formal model; we restrict the formal model's scope to have only the first two types of task communications. In summary, not considering shared memory from the Ada code is the fifth reason for unpaired syncs in the XML. Extension of the toolset to handle shared memory is future work.

Now that we know the reasons for unpaired syncs in the XML let us look into the XML metrics with paired/unpaired syncs. Table 6.5 shows that out of the 80 tasks in the XML, 52 tasks have at least one paired sync. This is 65% of the total number of tasks. 28 tasks do not have a paired sync, the reasons for which we have described in the preceding paragraph.

# of tasks in the XML	80
# of tasks with at least one paired sync	52
# of tasks without any paired sync	28

Table 6.5: Metrics of tasks in the XML with paired/unpaired synchronizations

In the next section, we will see the effect of considering only the tasks with paired syncs (i.e., applying program slicing) on the model's state space.

6.6.2 Applying Program Slicing on the Model

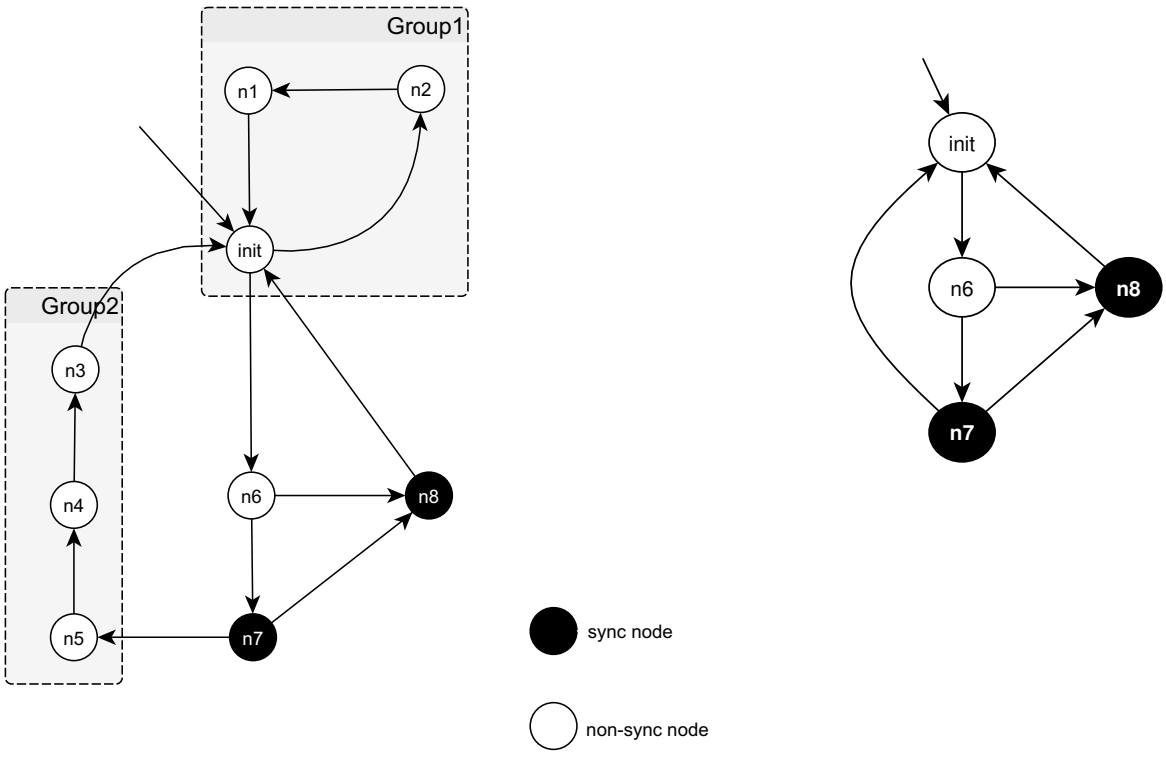
The previous section described the reasons for unpaired syncs in the XML. The unpaired syncs require further extension of the extraction code and the toolset. What is clear in front of our table are the paired syncs. From now on, we focus only on verifying the *paired* synchronizations that may lead to a deadlock/livelock. In other words, we focus on a subset of the synchronizations to verify the existence or absence of concurrency problems.

Regardless of paired or unpaired synchronization, the toolset's translator generates states corresponding to every concurrency construct. For example, whenever the translator encounters an `<accept>` tag in the XML, it defines a state in the Promela model for this tag, regardless of whether its counterpart `<rendezvous>` is found or not. Besides defining states, the translator also defines state transitions respecting the program control flow.

The state transitions defined for unpaired synchronizations are nothing but a set of "goto" transitions without a sending signal (!) or receiving a signal (?). It is not possible for a set of "goto" transitions without synchronization to cause a deadlock/livelock. Therefore, the states and state transitions that do not involve synchronization can be safely removed from the model, thereby reducing the state space. In the next section, we will describe the technique used to remove the states and transitions from the model that do not involve synchronization.

6.6.3 Collapsing and Removing Unpaired Synchronizations

This section explains the technique used to reduce the state space of the generated Promela model. The algorithm that implements this technique is incorporated into the toolset to generate a Promela model verified by SPIN without scalability problems.



(a) FSM before collapsing (b) FSM after collapsing

Figure 6.8: Collapsing and removing unpaired synchronizations

Figure 6.8a is an FSM with a set of nodes, two of which are synchronizing nodes, and the remaining are non-synchronizing nodes. A synchronizing node refers to a node that generates or accepts a synchronization message, as an action, during the transition from-it or to-it, respectively.

To detect deadlock/livelock, it is sufficient to retain only the synchronizing statement pairs (like, *accept-rendezvous* pair or *WaitForEvent-SetEvent* pair) in the Promela model. In other words, a set of transitions without generating/accepting any synchronization signal cannot lead to a deadlock/livelock.

Back to Figure 6.8a, the grouping technique showed in the figure may be visualized as *collapsing* a graph to retain only the synchronizing nodes n7 and n8. As a result of collapsing, the nodes that do not involve synchronization, directly or indirectly, will be grouped and collapsed. In the figure, the sync node n7 has an edge to the non-sync node n5 in the original graph. In this case, the nodes n5, n4, and n3, which does not involve directly or indirectly in synchronization, are merged (collapsed) such that there is a direct edge from n7 back to the init node. In the end, the non-synchronizing nodes (n1, n2) and (n3, n4, n5) are grouped and *collapsed* to the init node.

Listing 6.4: Promela model before collapsing

```
1  mtype = { S1, S2 };
2  chan C1 = [0] of { mtype };
3  chan C2 = [0] of { mtype };
4  proctype T1(){
5  initial:
6  if
7  :: goto n2
8  :: goto n6
9  fi
10 n2: goto n1
11 n1: goto initial
12 n6:
13 if
14 :: goto n7
15 :: goto n8
16 fi
17 n7:
18 if
19 :: C1!S1 -> goto n5
20 :: C1!S1 -> goto n8
21 fi
22 n8: C2!S2 -> goto initial
23 n5: goto n4
24 n4: goto n3
25 n3: goto initial
26 }
```

Listing 6.5: Promela model after collapsing

```
1  mtype = { S1, S2 };
2  chan C1 = [0] of { mtype };
3  chan C2 = [0] of { mtype };
4  proctype T1(){
5  initial: goto n6
6  n6:
7  if
8  :: goto n7
9  :: goto n8
10 fi
11 n7:
12 if
13 :: C1!S1 -> goto initial
14 :: C1!S1 -> goto n8
15 fi
16 n8: C2!S2 -> goto initial
17 }
```

The collapsed FSM is shown in Figure 6.8b. The nodes init, and n6 are also non-sync nodes but have direct edges with the sync nodes. As a result of collapsing, these nodes with direct edges with sync nodes (in other words, these nodes involve *indirectly* in synchronization) are

retained to ensure that the control-flow is kept intact. Listing 6.4 shows the Promela model before collapsing, corresponding to the FSM in Figure 6.8a. Listing 6.5 shows the Promela model after collapsing, corresponding to the FSM in Figure 6.8b.

The following observations can be made from the Promela models before and after collapsing.

1. The number of states in the model before collapsing is **9**, the number of states in the model after collapsing is **4**.
2. The number of transitions in the model before collapsing is **11**, the number of transitions in the model after collapsing is reduced to **6**.

Therefore, the collapsing technique reduces the number of states and transitions in the resulting model. This leads to having a model with reduced state space.

As mentioned in Section 6.5, the Promela model of the Ada code initially had 73,232 lines and 63,655 transitions before applying the collapsing technique. After applying this technique, the Promela model had 17,708 lines and 12,585 transitions — the size of the model is **reduced to 25% of the original model**. The model has 52 processes, which correspond to the 52 Ada tasks in the XML with at least one paired sync as seen from Table 6.5.

In summary, we started this section by listing the different reasons for having unpaired synchronizations in the XML. Next, we have seen how the collapsing of these unpaired synchronizations led to having a Promela model with reduced state space.

The collapsed model is verified in SPIN on a 64-bit Windows 10 PC with 16GB RAM. The model completed verification without out-of-memory/timeout errors using SPIN's supertrace verifier mode. The supertrace verification, however, achieved less than 100% model coverage. Also, the model returned an out-of-memory error when verified using SPIN's (default) exhaustive verification and other state compression techniques (see SPIN's verifier modes in Section 3.3.3). In Chapter 7, we will discuss how we achieved an improved verification performance and improved model coverage.

Now that we have a scalable model, we need traceability from this model to the Ada code. Traceability is essential to trace violations detected by SPIN from the model back to the code. The next section describes the technique used to generate this traceability.

6.7 The Toolset's Traceability Generator

This section will describe the technique used to generate the traceability matrix from the model to code.

In the previous section, we have seen that the collapsed model retains only those states and transitions that involve directly/indirectly in synchronization. We have also seen that the collapsing technique respects the program control-flow. This results in defining a rule for the traceability matrix.

The toolset generates the traceability matrix based on the following rule: *only those transitions that are directly/indirectly involved in synchronization will be retained along with the corresponding source code lines*. As the collapsed model only has transitions that involve directly/indirectly in synchronization, this rule is sufficient to trace any path in the model with a violation back to the code. Below we will look into an example illustrating this rule.

The rest of this section is organized as follows. An example Ada program and its extracted XML format are shown, followed by the Promela model generated from this XML. We will then look

Listing 6.6: T1.adb

```
1  task body T1 is
2  begin
3   if cond1 then
4    SetEvent_1;
5   end if;
6   if cond2 then
7    T2.S1;
8    T2.S2;
9   end if;
10  if cond3 then
11   SetEvent_2;
12  end if;
13 end T1;
```

into the generated traceability matrix with the mapping from the model to the Ada program.

Listing 6.6 shows Ada task T1. The concurrency constructs are in lines-4, 7, 8, and 11. Of these constructs, only the entry calls at line-7 and line-8 participate in synchronization (i.e., sync nodes). The `SetEvent` construct at line-11 does not contribute, directly or indirectly, to synchronization (i.e., this is the non-sync node). Task T1 sends two entry calls, namely, `S1` and `S2`, which are received by another Task T2 (not shown for simplicity).

Listing 6.7: XML with concurrency constructs extracted from `T1.adb`

```
1  <task name="T1" location="TaskBody T1 T1.adb:1">
2     <if location="IfStmt T1.adb:3">
3        <then>
4           <SetEvent target="SetEvent_1" targetlocation="CallExpr T1.adb:4"><
               /SetEvent>
5        </then>
6     </if>
7     <if location="IfStmt T1.adb:6">
8        <then>
9           <rendezvous target="S1" targetlocation="CallExpr T1.adb:7"></
               rendezvous>
10          <rendezvous target="S2" targetlocation="CallExpr T1.adb:8"></
               rendezvous>
11       </then>
12    </if>
13    <if location="IfStmt T1.adb:10">
14       <then>
15          <SetEvent target="SetEvent_2" targetlocation="CallExpr T1.adb:11">
               </SetEvent>
16       </then>
17    </if>
18 </task>
```

Listing 6.7 shows the XML representation of `T1.adb`. Listing 6.8 shows the Promela translation for `T1.adb` before applying the collapsing technique, Listing 6.9 the Promela translation for `T1.adb` after applying the collapsing technique.

Table 6.6 shows the generated traceability matrix from the collapsed Promela model to `T1.adb`. The first column contains the line number from the model, and the second column contains the

model's states/transitions at the line number indicated by the first column. The fourth and sixth columns contain the source code lines corresponding to a specific transition in the model. The third and fifth columns are mainly for debugging suspicious counterexamples from SPIN, if any, by looking into the XML (without necessarily looking into the source code).

The decision on the format of the traceability file is based on the following two rationales:

1. The traceability file must contain all the required information *at one place* for tracing the model back to code. Rather than having only line numbers, it is also essential to include these lines' content from the collapsed model.
2. Violations, if any, detected by SPIN can be in a specific path out of the several possible paths to reach a state. This means it is insufficient to have only a one-to-one line number mapping from the model to the code, but it is necessary to have the different transition paths that lead to a state. This led to the decision to have distinct columns for "current state" and "next state."

Listing 6.8: Promela model for `T1.adb` before collapsing

```
1  mtype = {S1,S2};
2  chan T1_to_T2 = [0] of {mtype};
3  proctype T1(){
4  L1:
5  if
6  :: goto SetEvent_1
7  :: goto S1
8  :: goto SetEvent_2
9  fi
10 SetEvent_1:
11 if
12 :: goto S1
13 :: goto SetEvent_2
14 fi
15 S1: T1_to_T2!S1 -> goto S2
16 S2: T1_to_T2!S2 -> goto
      SetEvent_2
17 SetEvent_2: skip
18 }
```

Listing 6.9: Promela model for `T1.adb` after collapsing

```
1  mtype = {S1,S2};
2  chan T1_to_T2 = [0] of {mtype};
3  proctype T1(){
4  L1:
5  if
6  :: goto SetEvent_1
7  :: T1_to_T2!S1 -> goto S2
8  fi
9  SetEvent_1: T1_to_T2!S1 -> goto
       S2
10 S2: T1_to_T2!S2 -> skip
11 }
```

Promela line number	Promela line content	Current state XML line number	Current state in source code	Next state XML line number	Next state in source code
4	L1:	1	T1.adb:1	-	-
6	:: goto SetEvent_1	1	T1.adb:1	4	T1.adb:4
7	:: T1_to_T2!S1 ->goto S2	1	T1.adb:1	10	T1.adb:8
9	SetEvent_1: T1_to_T2!S1 ->goto S2	4	T1.adb:4	10	T1.adb:8
10	S2: T1_to_T2!S2 ->skip	10	T1.adb:8	18	T1.adb:13

Table 6.6: Traceability table from Promela model to source code

The Table 6.6 shows that the state `S2` in the collapsed model can be reached through two paths: (1) from the initial state `L1` at line-4 and (2) from the state `SetEvent_1` at line-9.

Assume that SPIN reports a violation in one of these paths, in which case it is necessary to trace this specific path back to the source code. The traceability matrix shows that the first path to `S2` at line-7 of the model corresponds to the control flow from line-1 to line-8 of `T1.adb`. Similarly, the second path to `S2` at line-9 of the model corresponds to the control flow from line-4 to line-8 of `T1.adb`. Therefore, by looking at the error trace from SPIN, the Ada program's specific path leading to this error can be identified along with the Ada program's line number.

6.8 Algorithms Used in the Toolset

This section presents the algorithms used in the toolset. Table 6.7 provides the summary of the algorithms along with their purpose.

Algorithm #	Algorithm name	Purpose
1	Translate parsed XML data to Promela	1. Define sync messages in Promela 2. Call the appropriate function to find task pairs in order to define channel names in Promela 3. Call the appropriate function to define proctypes in Promela 4. Call the appropriate function to add sync points in Promela 5. Call the appropriate function to collapse
2	Translate Ada tasks to Promela proctypes	1. Call the appropriate function to list states in Promela corresponding to Ada concurrency constructs 2. Call the appropriate function to list the next state transitions for each Promela proctype
3	Define next state transitions in Promela	Using the states and state transitions listed by Algorithm-2, define: 1. Non-deterministic next state transitions in Promela, if there are more than one next states from the current state 2. A `goto' transition to the next state if there is just one next state from the current state
4	Translate control-flow in XML to next state transitions in Promela	Traverse the XML statement tree recursively to retrieve all the next state transitions
5	Identify task pairs	Identify paired Promela processes corresponding to communicating Ada tasks to define channel names
6	Multi-task synchronization	Add sync points in Promela corresponding to the Ada program's control point at which the Ada task(s) communicate
7	Convolution algorithm to collapse unpaired synchronizations	1. Remove states that do not involve directly/indirectly in sync 2. Remove state transitions that do not involve directly/indirectly in sync
8	Generate traceability from Promela model to source code	Create a mapping from each state and state transition of the Promela model to the corresponding Ada code's concurrency construct and control-flow respectively

Table 6.7: List of algorithms and their purpose

Algorithm 1 Translate parsed XML data to Promela

```
1: function XmlToPromelatoolset._translate_module(xmodule)
2:     pmodule ← create empty object of type PModule
3:     proc_list ← empty list
4:     for Each current_task in xmodule do
5:         proc ← XmlToPromelatoolset._translate_task(current_task)
6:         Add proc into proc_list
7:     end for
8:     paired_processes_data ← XmlToPromelatoolset._get_paired_processes_data(proc_list)
9:     chanvars ← map paired_processes_data to Promela channels list
10:    for Each current_proc in proc_list do
11:        XmlToPromelatoolset._add_synchronization(current_proc)
12:        XmlToPromelatoolset._convolute_process(current_proc)          ▷ Only when
                                                                           collapsing
13:    end for
14:    Add proc_list, chanvars and other declarations to pmodule
15:    return pmodule
16: end function
```

Algorithm 2 Translate Ada tasks to Promela proctypes

```
1: function XmlToPromelatoolset._translate_task(current_task)
2:     next_states ← next states for Xtask object
3:     if length of next_states list = 1 then
4:         first_state ← next_states[0]
5:     else if length of next_states list > 1 then
6:         first_state ← create virtual Xstate
7:         Set first_state as top state of current_task
8:     end if
9:     states ← current_task.get_all_states()
10:    states_data ← empty list
11:    for Each full_name, current_state in states do
12:        next_states ← current_state.get_next_states()
13:        full_name ← defined name format for current_state
14:        Add next_states and full_name to states_data
15:    end for
16:    for Each current_state_data_item in states_data do
17:        for Each current_next_state in current_state_data_item do
18:            if current_next_state is final state then
19:                Replace final state by first_state
20:            end if
21:        end for
22:    end for
23:    pprocess ← create new PProcess object
24:    for Each current_state, states_data in states do
25:        pstatement ← XmlToPromelatoolset._translate_state(current_state,states_data)
26:        Add pstatement to pprocess
27:    end for
28:    return pprocess
29: end function
```

We now describe the algorithms one by one. Algorithm 1 translates the list of XML tasks to a list of Promela proctypes. The XML states data of each task is translated to a sequence of Promela statements using the `_translate_task` function (line-5). Channels are defined based on task pairs (line-8 and line-9). For each Promela proctype, synchronization points are added (line-11) using the channel definition. The function returns a Promela module with a proctype list, channel, and other declarations like mtype (line-14 and line-15).

Algorithm 2 finds the next state transitions for each of the task states in the XML (line-2). A virtual state (line-6) refers to a label to indicate the initial state. While finding the next states, a meaningful naming convention is used for each state (line-13). The sequence of XML states obtained is converted to Promela statements (line-23 to line-26).

Using the XML task states and the list of next state transitions, Algorithm 3 creates Promela transitions. A Promela "if" statement (for non-deterministic choices) is created if the next state list has more than one statement (line-3 to line-11). A Promela "goto" statement is created if the next state list has only one statement (line-13 to line-16).

Algorithm 3 Define next state transitions in Promela

```
1: function XmlToPromelatoolset._translate_state(current_state, states_data)
2:     next_states ← next_states from states_data for current_state
3:     if length of next_states list > 1 then
4:         pif ← new PIf object
5:         pif.name ← full_name from states_data for current_next_state
6:         for Each current_next_state in next_states do
7:             pgoto ← new PGoto object
8:             pgoto.target_name← full_name from states_data for current_next_state
9:             Add pgoto to pif options
10:        end for
11:        return pif
12:    else
13:        pgoto ← PGoto object
14:        pgoto.name ← full_name from states_data for current_state
15:        pgoto.target_name ← full_name from states_data for next_states[0]
16:        return pgoto
17:    end if
18: end function
```

Algorithm 4 describes the general idea of the next state search process. If the <if> tag in XML is an xobject, then the <then> tag is the caller_object of the xobject (line-1).

In line-2, "blocks" refer to the associated statements of a particular conditional construct. For example, consider the Ada snippet in Listing 6.10. The blocks associated with the `if` statement are nothing but the `then`, `elsif` and `else` statements. On a similar note, the block associated with a `case` statement is nothing but the `when` statement, and so on.

Listing 6.10: Associated blocks of `if` block

```
1 if conditionA then
2  -- some code;
3 elsif conditionB then
4  -- some code;
5 else
6  -- some code;
```

The next state search process's logic depends on concrete statements like if, elsif, select, case, etc. (line-5).

The idea behind the next state search is to traverse a portion of the XML statement tree recursively. Analogically, the statement tree is like one big thread split into many small threads, each of which is split into other threads and so on. Some threads reach the *finish line*, meaning, encounter an XML state tag like <accept>, <WaitForEvent> etc . In this case, they flow in the opposite direction, from the child node to the parent node (line-6). Each thread stops when: (1) it reaches a state that is guaranteed to occur after the current state (line-2 to line-9), or (2) it reaches the end of the task (line-10 to line-13). If there are states that occur after the current state, then these states are merged and returned recursively as a list of states to the caller (line-4, line-6).

Algorithm 4 Translate control-flow in XML to next state transitions in Promela

```
1: function XObject.get_next_states(xobject, caller_xobject, next_states)
2:    if caller_object is one of the statements from blocks of xobject then
3:       if caller_object is not last statement in block then
4:          return next_states merged with XObject.get_next_states(
                      <next statement to caller_object in xobject block>, xobject)
5:       else if some conditions are met then        ▷ logic depends on concrete
                                                       xobject class like
                                                       <Xif>, <XCase> etc.
6:          return next_states merged with XObject.get_next_states(
                     <first statement of another block>, xobject)
7:       else
8:          return XObject.get_next_states(<parent of xobject>, xobject, next_states)
9:       end if
10:   else
11:      if some block is not empty then                              ▷ or blocks
12:         return XObject.get_next_states(<first statement of some block>,xobject)
13:      end if
14:   end if
15:   return XObject.get_next_states(<parent of xobject>, xobject)
16: end function
```

Algorithm 5 Identify task pairs

```
1: function XmlToPromelatoolset._get_paired_processes_data(proc_list)
2:    proc_pairs ← combine all processes from proc_list to create process pairs
3:    statement_pairs ← empty list
4:    for Each current_pair in proc_pairs do
5:       statements_pairs_for_proc_pair ← combine all <Xaccept>, <Xrendezvous>,
                                          <XSetEvent> and <XWaitForEvent>
                                          statements of current_pair to find
                                          statements which are paired
6:       Add statements_pairs_for_proc_pair to statement_pairs
7:    end for
8:    return statement_pairs
9: end function
```

Algorithm 5 identifies the Promela processes that are paired (corresponding to Ada task pairs).

Definition 6.1 is used in the algorithm to identify Ada task pairs that correspond to Promela process pairs (line-4 to line-6). The names of the channel are defined based on the paired process data.

We will now discuss the logic behind adding the sync points (channel name + sync type + sync message) in Promela. A task waiting for a signal can be serviced by more than one sending task. Consider the example in Figure 6.9. TaskA is the receiving task, capable of receiving more than one signal (S1, S2). This task can be serviced by TaskB and TaskC by sending signal S1, or can also be serviced by TaskD and TaskE by sending signal S2, depending on the signal that arrives first. These multiple ways of synchronization are modeled as non-deterministic choices in Promela.

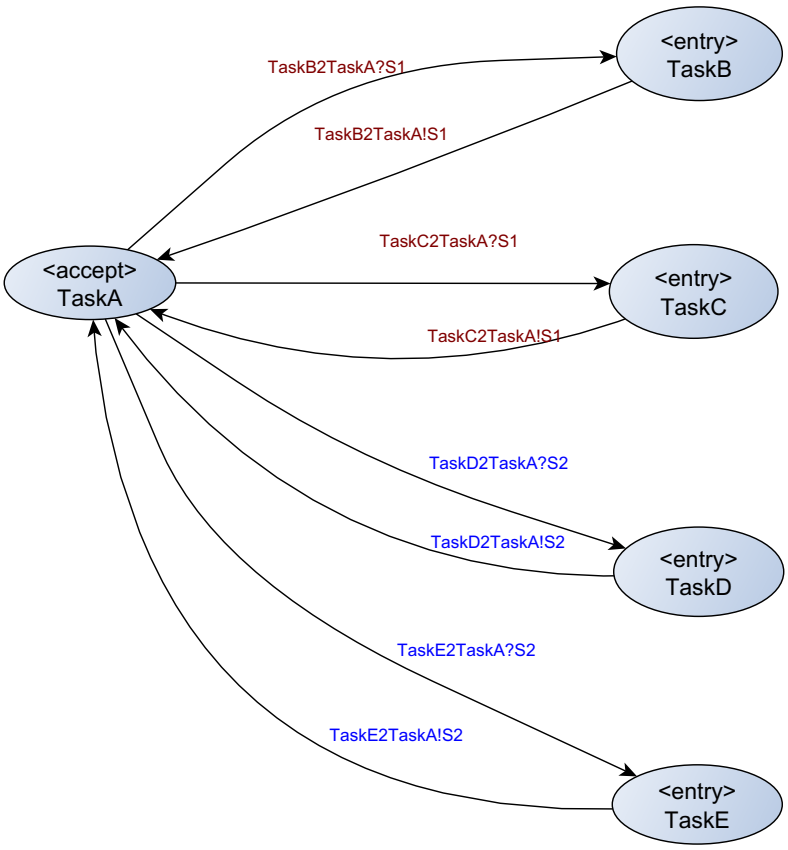


Figure 6.9: Multi-task synchronization

Algorithm 6 describes the logic used to place sync points at the location in the model that corresponds to the control point in the Ada program where a task can synchronize with another.

The paired processes obtained from Algorithm 5 is used to define sync points in the model (algorithm's line-2 and line-3). A Promela "goto" statement is created if the there is just one next state transition after the sync (line-4). The combination of channel name `TaskB2TaskA` + the sync type `?` + the sync signal `S1` + the arrow separator (`->`) forms the *PArrowedSequence*. The term 'wrap' in line-5 refers to combining the *PArrowedSequence* with the 'goto' transition state after the synchronization. For example, see line-2 of Listing 6.11. Adding the transition `goto L1` after the *PArrowedSequence* is the wrapping operation.

Suppose there is just one next state transition after sync, but the sync is multi-task sync. In that case, the multiple synchronizations are defined as non-deterministic choices (as mentioned at the beginning of this algorithm), see algorithm's line-6. An example for this is shown in

Listing 6.11, which is the Promela snippet of TaskA in Figure 6.9. Here L1 is the only next state transition after the sync, but four sender tasks can service TaskA. Therefore the four sync points are defined as non-deterministic choices.

In the previous case, we considered that there is just one next state transition after sync. However, suppose the next state transition after the sync is more than one. In that case, the usual non-deterministic choice of the next state transitions and the non-deterministic choice of multi-task sync are defined (line-7 to line-10).

Algorithm 6 Multi-task synchronization

```
1: function XmlToPromelatoolset._add_synchronization(
                                        current_proc, paired_processes_data)
2:    for Each current_paired_processes_data_item in paired_processes_data do
3:       pstatement ← get statement from current_paired_processes_data_item
4:       if pstatement is PGoto object then
5:          Wrap pstatement to PArrowedSequence object by adding sender/receiver
            guard
6:          For multi-task sync, add non-deterministic choice
             of different synchronizations to PArrowedSequence
7:       else if pstatement is PIf object then
8:          Create PArrowedSequence with sender/receiver guard
9:          For multi-task sync add non-deterministic choice
             of different synchronizations to PArrowedSequence
10:      end if
11:      pifs ← find all PIf objects in current_proc
12:      for Each pifs do
13:         delay ← find XDelay object in current pif
14:         if delay exists then
15:            Wrap corresponding goto with PArrowedSequence and "len" statement
               as guard
16:         end if
17:      end for
18:   end for
19: end function
```

Listing 6.11: Non-deterministic choice of synchronizations in Promela corresponding to multi-task synchronization for TaskA in Figure 6.9

```
1 if
2 :: TaskB2TaskA?S1 -> goto L1
3 :: TaskC2TaskA?S1 -> goto L1
4 :: TaskD2TaskA?S2 -> goto L1
5 :: TaskE2TaskA?S2 -> goto L1
6 fi
7 L1: /* some code */
```

Besides synchronous channels for rendezvous communication, asynchronous channels are required for modeling Ada events (see the behavior mapping in Section 5.1.4) and Ada selective accept with delay (see the behavior mapping in Section 5.1.5). Therefore, the required Promela primitives, “delay” and “len,” are in line-12 to line-17. An example *PArrowedSequence* with “len” as guard is shown in Listing 6.12.

Listing 6.12: *PArrowedSequence* for asynchronous communication

```
1 if
2 :: len(ch) > 0 -> ch?s1 -> goto L1
3 :: ...
4 fi
5 L1: /* some code */
```

Algorithm 7 describes the convolution process incorporated into the toolset to implement the collapsing technique. The name *convolution* is used to indicate the operation performed on two or more state transitions that result in a single state transition.

Algorithm 7 Convolution algorithm to collapse unpaired synchronizations

```
1: function XmlToPromelatoolset._convolute_process(current_proc)
2:    while at least one change is made per cycle do
3:       for Each current_statement in current_proc do
4:          if current_statement is PGoto object then
5:             For all goto with target_name equal to current_statement name
               replace target_name with current_statement target_name
               and remove current_statement
6:          end if
7:          if current_statement is PIf object then
8:             Remove duplicated non-deterministic choice, if any
9:             Replace all PIf options to new PGoto object, if all PIf options
               are same
10:         end if
11:      end for
12:      for Each process type current_proc do
13:         if current_statement is PArrowedSequence object then
14:            gotos ← all gotos with target name equal to current statement
                       name
15:            for Each gotos do
16:               if Parent of current_statement is PIf then
17:                  Move current_statement to PIf
18:               else if Parent of current_statement is PArrowedSequence then
19:                  Merge goto with current_statement
20:               end if
21:            end for
22:         end if
23:      end for
24:   end while
25: end function
```

To describe the algorithm, we will use the Promela model before collapsing in Listing 6.8 and the Promela model after collapsing in Listing 6.9. If the operator is just ”goto,” which does not involve sync, then it is deleted, and all references to it are replaced with references to its target (algorithm’s line-4 and line-5). For example, the goto transitions to `goto SetEvent_2` in Listing 6.8 does not involve any sync. Therefore the state SetEvent_2 and transitions to it are removed in the collapsed model.

Suppose there is only one non-deterministic choice left after collapsing (and this choice may be repeated). In that case, the non-deterministic choice is replaced with the content of goto transition (algorithm's line-7, line-8, and line-9). For example, consider the next state transitions of the state SetEvent_1 at line-10 of Listing 6.8. There are two non-deterministic transitions; one to the state S1 and the other to the state SetEvent_2. After collapsing, the state SetEvent_2 is removed, resulting in just one choice. Therefore, in the collapsed model, the non-deterministic `if` is removed and replaced with the content of goto transition of state S1 for the state SetEvent_1 (see line-9 of the collapsed model).

If the goto transition to a state is within the non-deterministic `if`, and the target state has sync, then the sync is moved to the goto transition (algorithm's line 13 to line 17). For example, consider the goto transition to the state S1 in Listing 6.8. The target state S1 has a sync (i.e., *PArrowedSequence*). Therefore, the goto transition is replaced with the actual sync as in line-7 of the collapsed model.

If a state does not involve sync, but the transition to this state involves sync, then the target state of this state is merged to the target state of the state with the sync (algorithm's line-18 and line-19). Here 'target state' could be either the state name after 'goto' or the 'skip' statement. For example, the state SetEvent_2 at line-17 of Listing 6.8 does not involve sync. However, one of its parent state (i.e., state S2) has sync as in line-16. Therefore, the 'skip' statement at line-17 is merged with the `goto SetEvent_2` at line-16. The effect of this merging is seen in line-10 of the collapsed model.

Algorithm 8 describes the logic used for generating the traceability matrix.

Algorithm 8 Generate traceability from Promela model to source code

```
1: function XmlToPromelatoolset._create_mapping(
                              processes, data_by_pprocess, line_content_by_number, file_path)
2:    Create a CSV file in location file_path in write mode
3:    for Each current_process.pstatements in processes do
4:       for Each data_by_pprocess in current_process.pstatements do
5:          xstate_by_pstatement_name ← data_by_pprocess for current_process
6:          if current_pstatement is PArrowedSequence then
7:             goto ← find goto in current_pstatement
8:             current_xstate ← xstate_by_pstatement_name using current_pstatement name
9:             next_xstate ← xstate_by_pstatement_name using goto target name
10:            Write CSV row using current_xstate data, next_xstate data and
               line_content_by_number
11:         else if current_pstatement is PIf then
12:            current_xstate ← xstate_by_pstatement_name using current_pstatement name
13:            Write CSV row using current_xstate data and line_content_by_number
14:            pif ← current_pstatement
15:            for pif.options do
16:               goto ← find goto in current option
17:               next_xstate ← xstate_by_pstatement_name using goto target name
18:               Write CSV row using current_xstate data, next_xstate data and
                  line_content_by_number
19:            end for
20:         end if
21:      end for
22:   end for
23: end function
```

To describe the algorithm, we will use the XML in Listing 6.7, the XML's collapsed Promela model in Listing 6.9 and the traceability Table 6.6.

Algorithm 8 uses four inputs (algorithm's line-1):

- *processes* refers to Promela proctypes. E.g.: `proctype T1()` at line-3 of the model
- *data_by_pprocess* refers to the state names in Promela corresponding to the XML state. E.g.: `SetEvent_1` at line-6 of the model is the Promela state corresponding to the target value `SetEvent_1` of the XML state `<SetEvent>` at line-4 of the XML
- *line_content_by_number* refers to the XML line numbers
- $file\_path$ refers to the location in which the traceability file (CSV) is created

For each Promela proctype, and for each Promela state of a proctype, the Promela state names corresponding to XML state names are collected and stored in the list *xstate_by_pstatement_name* (algorithm's line-5). In other words, during translation of XML state name to Promela state name, references to the XML state are saved.

Next, there are two possibilities for next state transitions; just one next state or more than one next states (same as discussed in Algorithm 6). Suppose there is just one next state transition (algorithm's line-6). In that case, the 'goto' keyword is searched in the `PArrowedSequence` (line-7) to retrieve the state name immediately after 'goto.' Let us see an example below to understand this.

Consider line-9 of the Promela model: `SetEvent_1: T1_to_T2!S1 -> goto S2`. Here, there is just one next state transition to state S2 after the sync. If SetEvent_1 is the current state, then the state S2, which is immediately after the 'goto' statement, is the current state's next state. The current and next states are stored into *current_xstate* and *next_xstate* respectively (algorithm's line-8 and line-9). In the traceability table (CSV), the current state in Promela, along with its line number, is written into column-2 and column-1, respectively. The reference to the XML state line numbers corresponding to the current and next Promela states are written into column-3 and column-5, respectively (algorithm's line-10). Only the XML line numbers are printed in the CSV instead of XML line content for simplicity. The source code line numbers for each XML line number (retrieved using `targetlocation` XML attribute) is printed in column-4 and column-6 of the CSV (not shown in algorithm).

We have seen the traceability generated for the first possibility of having just one next state for the current state. The second possibility is having more than one next states transitions listed as non-deterministic choices using Promela `if` statement (this is `PIf` in algorithm's line-11). Let us consider line-4 to line-8 of the Promela model to understand the generated traceability for this case.

Line-4 of the model is the initial state labeled L1. This initial state, along with its corresponding XML line number, is written in the traceability table's first row (algorithm's line-12 and line-13). From the initial state, there are two non-deterministic transitions as in line-6 and line-7 of the model. For each of these transitions (options of Pif, algorithm's line-15), the 'goto' statement is searched from the initial state (algorithm's line-16). The state immediately after 'goto' is one of the next state transitions from the initial state (algorithm's line-17).

For each of the non-deterministic choices, the current and next state Promela code, along with current and next state XML numbers, are printed in the CSV (algorithm's line-18). For example, see the second row of the traceability table. The row shows that `goto SetEvent_1` is the Promela state corresponding to the current XML state at line-1 of the XML. The next state for this Promela state is `SetEvent_1` at line-9 of the model. This next state of the model corresponds to the target value `SetEvent_1` of the XML state `<SetEvent>` at line-4 of the XML.

6.9 Validation of the Toolset

The toolset's Python code underwent 19 revisions before its final release. The algorithms were reviewed, validated using several tests, and fixed for bugs. A standard software development life-cycle process was followed during the development of the toolset. The process includes (a) requirement definition, (b) development of algorithms, (c) code development, (d) testing compliance of each code iteration to the algorithms, (e) bug-tracking, and (f) code version control. Necessary artifacts are maintained as proof for the process followed. Furthermore, every Ada concurrency/conditional construct and its Promela counterpart is tested sufficiently to ensure that the generated model imitates the modeled code. In summary, the toolset's code is validated using empirical analysis.

6.10 Chapter Summary

We started this chapter with a brief overview of the extraction code. The extraction code generates an XML file with the Ada program's concurrency constructs, along with the control flow.

We then described the structure of the generated XML. We looked into the different XML tags and defined *task pairs* that involve synchronization.

Next, we listed the required elements to generate a Promela model from the XML data of an Ada program. The elements include sync messages, channel definitions, sync points, state names, and state transitions.

We then saw how the toolset's translator automatically extracts the required elements to generate a Promela model from the XML data. We described the different functions and classes used by the translator, supported by class diagrams. Subsequently, the translator's scheme is presented, showing the two-step translation process from XML to Promela. The first step was about parsing the XML data, and the second step was about generating the Promela model from the parsed XML data.

Next, we described the state space metrics of the generated model by the translator. The generated model could not be verified by SPIN due to state space explosion. We then illustrated the toolset's collapsing technique to reduce the state space of the generated model.

After generating a scalable model, we proceeded to illustrate the toolset's technique to generate the traceability matrix from the model to the Ada program.

Next, we described the validation of the toolset. Finally, we presented the algorithms developed for the toolset.

Overall, this chapter explained the Stage-2 of Figure 1.1 related to the automatic generation of the Promela model and the traceability matrix. The next Chapter 7 presents Stage-3 of Figure 1.1 related to verification of the generated model by SPIN.

7 MODEL VERIFICATION

This chapter describes Stage-3 of Figure 1.1 related to the verification of the generated model.

In Section 6.6.3, we have seen that the toolset generated a collapsed Promela model having 17,708 lines and 12,585 transitions. We also mentioned that the collapsed model failed to complete an exhaustive verification but could complete SPIN's supertrace verification. There is a downside with the supertrace verification results as follows.

Although the accuracy of counterexamples (if any) is guaranteed in supertrace verification [Hol04], the supertrace is a lossy compression technique. In other words, there is a chance that supertrace misses some paths of model exploration compared to the exhaustive verification. This leads to achieving a model coverage that is less than 100%. To overcome this problem, it is essential to stick to exhaustive verification as much as possible. Fortunately, a wide range of techniques is supported by SPIN to increase the model coverage. This leads us to explore the advanced features of SPIN.

The chapter's goals are as follows.

1. Conduct experiments to achieve an exhaustive verification of the model, where ever possible.
2. Conduct experiments to achieve an improved model coverage when lossy compression is used.
3. Verify two properties, (a) absence of deadlock and (b) absence of livelock on the generated model.

The rest of the chapter is organized as follows. Section 3.3.3 provided an introduction to basic features of SPIN's verification. Besides these basic features, this chapter requires some knowledge about the advanced features of SPIN. The advanced features help to overcome the two downsides in the verification results, as mentioned above. SPIN's advanced features are discussed in Section 7.1.

After gaining knowledge about SPIN's advanced features, we need to answer the questions: (a) where to start with the verification?, and (b) how to proceed with the verification? That is, we need a roadmap to guide us with the verification process. The verification roadmap is described in Section 7.2.

After having a roadmap, the next step is to complete the verification journey using the roadmap. The verification is performed using two different environments as follows.

1. A 64-bit Windows 10 PC with 8GB RAM (usable RAM is up to 7GB)
2. A 64-bit Linux machine with up to 2TB RAM

We intend to use the minimum memory possible to complete exhaustive verification. *Divide-and-conquer* technique is used at the beginning of the verification journey to support this intention. The divide-and-conquer technique helps group an independent set of interacting pro-

cesses and create sub-models from the complete model. The sub-models have reduced state space, thereby providing higher chances to perform an exhaustive run even with limited physical memory. Only in cases where we need more memory or improved model coverage will we switch to the Linux machine with more RAM. The verification journey is described in Section 7.3.

7.1 Advanced Features of SPIN

This section describes the advanced features of SPIN that can help for exhaustive verification and to improve the model coverage. The source for this section is mainly derived from [Hol04]. The section is organized as follows. In Section 7.1.1 we define the *search complexity* in SPIN model checking by explaining the factors that can cause unsuccessful/incomplete verification.

We then present the different techniques available in SPIN to overcome an unsuccessful verification. The techniques are categorized as non-algorithmic techniques and algorithmic techniques. Non-algorithmic techniques optimize the modeling approach; this involves changes in an existing model. The non-algorithmic techniques are described in Section 7.1.2. Our toolset already considers some of these non-algorithmic techniques while generating the Promela model.

The algorithmic techniques involve systematic ways of using SPIN's built-in compression algorithms. The compression algorithms are classified as lossless compression and lossy compression. The algorithmic techniques are described in Section 7.1.3. Our work uses both types of compression techniques.

After using non-algorithmic and algorithmic techniques to reduce the search complexity, there are other techniques available to improve the verifier's performance even further. The techniques include selecting the type of property to be verified, carefully selecting the search depth value, the maximum memory to be used, the number of bits to be used per state, choosing a different hash function, etc. These techniques are described in Section 7.1.4. We investigate the effect of these techniques on the collapsed model to obtain the best possible results.

7.1.1 Search Optimization in SPIN

This section lists the reasons for search complexity, leading to an unsuccessful/incomplete verification of a model. First, the search complexity is defined, followed by the techniques supported by SPIN to overcome the search complexity.

The search complexity of a model checker is defined as follows [Hol04].
Search Complexity = $M * B * S$, where

- M: number of reachable states in the global state space
- B: number of states in the property automaton
- S: size of individual states (or the state-vector)

We will now describe how each of the three factors affects the search complexity and how to keep each of them as small as possible.

7.1.2 Non-Algorithmic Techniques to Reduce Search Complexity

Non-algorithmic techniques refer to techniques that require changes in the model. The techniques are illustrated below.

1. To reduce the size of the property automaton (B), use simple properties instead of a combination of multiple properties. There will be an exponential increase in B with the increase in the number of operators in an LTL property. Let us see an example to understand this.

   Consider an LTL property: "sooner or later it is possible to either reach the label L1 in process P3 with process ID as 3 or reach the label L2 in process P2 with process ID as 2."

   The property involve two different processes (i.e., proctype). Two ways of specifying this property are as follows:

   (a) `ltl prop {<> P3[3]@L1 || <> P2[2]@L2}`, or

   (b)
   ```
   ltl prop1 {<> P3[3]@L1};
   ltl prop2 {<> P2[2]@L2};
   ```

   The second way of the specification is preferred over the first; the simpler the property, the lower is the property automaton's size. For more details on LTL specification in SPIN, see the SPIN's manual page [SPI18a].

2. To reduce the size of the global state space (M), the following techniques help.

   (a) Divide and conquer: In simple terms, this technique means *not to connect what is independent; i.e., group the independent set of communicating processes separately*.

   This is one of the key techniques used in our upcoming verification journey to reduce M. Section 7.3 describes this technique in more detail.

   (b) Reduce the number of processes: A smaller number of processes means fewer states to explore. Our translator retains only those processes in the model with at least one paired synchronization in the context of this work. Any further reduction in the number of processes is not possible.

   (c) Use unique channels between the sender and receiver processes in case of multi-task synchronization: The interleaving of messages within a single channel causes significantly higher search complexity. The toolset, as illustrated in Section 6, generates unique channels if a task waiting for a signal can be *serviced* by more than one sending task, thus reducing the search complexity.

   (d) Reduce the number of channels used: Fewer channels considerably reduce the search complexity. For our work, this technique is superseded by the technique to use unique channels in the case of multi-task synchronization (item (c)). Any further reduction in the number of channels could not be seen for the generated model.

   (e) Reduce the capacity of the asynchronous channels (number of slots): By default, our translator sets a channel capacity of two (see Section 6.3). This is further optimized to set channel capacity to one. The state space reduced considerably because of this optimization.

   Regarding the techniques used to reduce the global state space (M), it is important to acknowledge that our toolset already considers some of these techniques. Using unique channels for multi-task synchronization and reducing the channel length of asynchronous channels are some examples. In other words, besides the algorithms that are built-in to reduce the state space, the toolset also has built-in techniques that reduce the complexity of the generated model.

3. To reduce the size of the state vector (S), the following techniques help.

(a) Abstraction: Our toolset considers the abstraction principle that only the concurrency constructs and control flow from the Ada code are considered for modeling.

(b) Program Slicing: We have seen this technique in Section 6.6. Our toolset retains only the paired processes in the model.

7.1.3 Algorithmic Techniques to Reduce Search Complexity

So far, we have seen non-algorithmic techniques to reduce the search complexity. We will now see some of the algorithmic techniques that can be used to reduce the search complexity.

SPIN has several algorithmic techniques to handle large to very large verification problems. The techniques are aimed at (a) reducing the number of states (M) to be searched and (b) reducing the amount of memory required for each state storage (S).

Figure 7.1 shows the overview of SPIN's algorithmic techniques. Section 7.1.3.1 describes SPIN's partial order reduction and statement merging that help in reducing the number of states to be searched. Regarding reducing the memory required, Section 7.1.3.2 describes SPIN's lossless and lossy compression techniques.

The lossless compression techniques can perform an exhaustive search using less memory at the cost of a higher run-time. The lossless techniques are *Collapse Compression* and *Minimized Automaton*.

The lossy compression techniques help reduce memory usage without an increase in the run-time. On the downside, there is a potential loss of model coverage (model coverage means the number of states searched). *Bitstate Hashing* and *Hash-compact* are the two lossy compression techniques supported by SPIN.

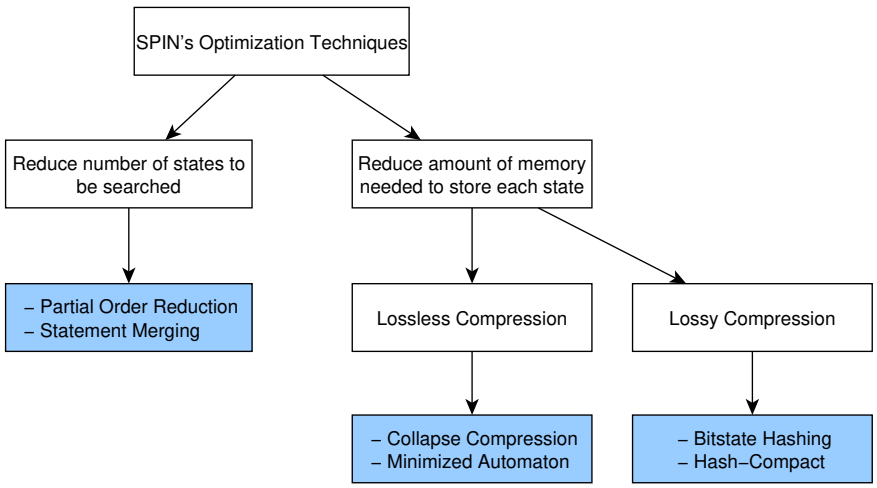


Figure 7.1: SPIN's Optimization Techniques

We will now briefly describe each optimization technique.

7.1.3.1 Techniques to Minimize Number of States (M)

7.1.3.1.1 Partial Order Reduction

Partial order reduction reduces the number of states to be searched for a given property. The reduction algorithm uses a unique search pattern to identify the possible transitions from a state.

Based on the property checked, the search pattern is such that only a *subset* of next state transitions is chosen instead of choosing all possible transitions from the current state.

SPIN's heuristic to find the subset of next state transitions is described in [HP95]. The heuristic first considers only those transitions from processes having a "local" effect. In other words, if in a global state, a process P can execute only local statements, then all other processes may be delayed until later. Here "local statements" refer to Promela statements that can access only local variables, a process receiving a message (?) from a channel from which no other process receives, a process sending a message (!) through a channel to which no other process sends.

The reduction algorithm ends its search as follows. Suppose there is at least one transition from a state, and if this transition leads to a state that was already stored (in a stack), then the algorithm ends. For more details on the algorithm, see [Cla+18a] and [HP95].

The partial order reduction is enabled by default in SPIN. A model with rendezvous operations is incompatible with partial order reduction when weak fairness is enabled. This applies to our model when verifying for the absence of livelock and does not apply when verifying for the absence of deadlock. When partial order reduction is not compatible, then the default option is disabled using the compile-time directive `-DNOREDUCE`.

7.1.3.1.2 Statement Merging

Statement merging is another default technique used by SPIN to reduce the number of states searched. The statement merging technique is a special case of partial order reduction.

The technique tries to find transitions that are part of the same process and combines them into a single step. This 'combining' operation avoids creating intermediate states after each transition, thereby reducing the number of states searched. Chapter 9 of [Hol04] provides more details on statement merging along with an example.

The statement merging is also a default option enabled by SPIN. The only drawback of using statement merging is that it makes it harder to understand the automaton structure used in the verification process due to the 'combining' operation. Therefore when using the automaton structure, say, to debug the model, the user can disable statement merging using the run-time directive `-o3`.

For our model, the enabling of statement merging resulted in reduced state space compared to disabling it. The statement merging option is disabled only when interpreting the automaton structure while debugging the model.

We have seen that the partial order reduction and statement merging reduces the number of system states to be explored. Next, we will look into SPIN's techniques to reduce the memory required to store each state.

7.1.3.2 Techniques to Minimize State Storage (S)

7.1.3.2.1 Collapse Compression

During state space search, an increase in the number of reachable states is mainly due to the combination of local states (like processes and data objects) in several ways. Duplication of all the local state combinations for each global state is something that can be avoided. This is the idea behind collapse compression.

Using the collapse compression technique, the entire state space (i.e., state vector) is separated into two components: (a) global data and channel declarations and (b) processes. The

separated components are stored in a lookup table by assigning a unique index-number to each component.

The modified state vector now contains only the index numbers instead of the complete state data. In other words, the state vector, which initially included all the global and local data, is now a state vector with index numbers for the separated global and local data.

Experiments show that collapse compression does not prove to be an alternative to partial order reduction but provides good memory reduction when combined with it. Chapter 9 of [Hol04] illustrates the experiments.

Collapse compression is enabled by using SPIN's compile-time directive `-DCOLLAPSE`.

7.1.3.2.2 Minimized Automaton

Besides collapse compression, SPIN supports another lossless compression method called the minimized automaton (MA) to optimize the memory used to store each state. MA constructs and maintains a minimal deterministic finite automaton through state matching. The constructed automaton acts as a recognizer for a set of state descriptors. During state space search, the automaton is checked if a certain state is already visited. The automaton is updated whenever a new state is visited.

In summary, in the MA technique, the set of reachable states are represented as sets of state descriptors in a minimal finite-state automaton. The automaton is updated with a new state descriptor for every new state encountered during the search. The main drawback of the MA technique is the significantly high run-time as the automaton is updated for every new state encountered. The MA technique is explained in detail along with an example in [HP99]. For a concise read on MA technique, see Chapter 9 of [Hol04].

For our model, the MA technique helped to reduce memory usage compared to the default exhaustive search. The application of the MA technique for our verification is described in the upcoming sections.

The MA technique is enabled by using SPIN's compile-time directive `-DMA=N`, where N is the maximum length of the expected state vector.

7.1.3.2.3 Combining Collapse Compression and Minimized Automaton

It is possible to combine collapse compression and the minimized automaton techniques. Experiments show that combining these techniques may reduce the high run-time, although this is not always true. It is, however, worth trying. Chapter 9 of [Hol04] illustrates the experiments.

Whether the two techniques are used separately or together, both perform an exhaustive search (i.e., both are lossless techniques). For very large models, however, it is not always possible to achieve exhaustive coverage. The solution for this is to use two of the SPIN's proof approximation techniques that work well with very little memory without providing exhaustive coverage. The two techniques are bitstate hashing and hash-compact.

7.1.3.2.4 Bitstate Hashing

Consider a state space search with R states, with each state requiring S bytes, and let M bytes be the available physical memory (RAM). The bitstate hashing technique is considered when an exhaustive search is not possible, i.e., $R * S >> M$.

Let us first understand a standard hash table lookup of states before going to bitstate hashing. Assume a hash table with h slots as shown in Figure 7.2, where each slot can store a list of states. During the state space search, assume that a new state s is encountered. To determine as to which list the new state s should be stored, a hash-value $h(s)$ which is unique to s and chosen randomly between 0 and $h-1$ is computed.

SPIN's bitstate hashing uses a similar hash table lookup during the search where each state is represented as a sequence of bits in the memory. SPIN's bitstate hashing uses Jenkins' hash [JEN] to compute the hash value.

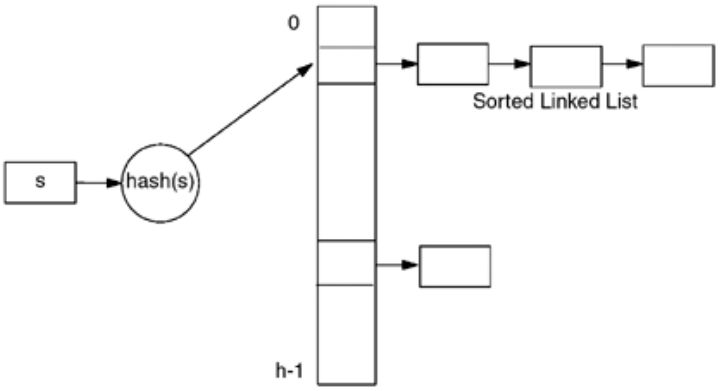


Figure 7.2: Hash Table Lookup [Hol04]

If r is the number of states stored in the hash table and h is the number of slots in the hash-table, then two scenarios arise while using bitstate hashing:

- $h >> r$ indicates that each state can be stored in a different slot
- $h < r$ indicates the possibility of computing the same hash function for two different states. This leads to *hash collisions*. With hash collisions, the model checking algorithm incorrectly concludes that a state was visited before and will skip this state. Furthermore, other states which can only be reached via this skipped state are also missed leading to a loss of coverage. Although it is possible to resolve the hash collisions (using linked lists), it leads to having an overhead of multiple state comparisons.

It can be seen that the scenario where $h >> r$ is desirable over $h < r$, as it leads to having a low collision probability. If $h >> r$, the only information to be stored in the hash table is whether or not the state corresponding to a given hash value has already been visited or not. This is nothing but a *one-bit information*; 0 means the state was not visited before, and 1 means the state was already visited. Therefore, it is sufficient to store only this one bit of information instead of explicitly storing the state (say, with S bits).

The downside of bitstate hashing is the reduced model coverage. The metrics of reduced coverage depend on the number of states missed during the search. With bitstate hashing, it is possible to reach far more states than M/S, but it may not be guaranteed to reach all R states.

When using the lossy bitstate hashing technique, a user needs to know the reliability of verification results obtained; that is, it is important to know the model coverage obtained. The model coverage can be known by looking at the SPIN's verification log for the value of *hash factor*.

The hash factor is the ratio of the maximum number of states expected to be reached to the actual number of states that are actually reached. A large factor, greater than 100, indicates

a model coverage closer to 100% (high reliability). A hash factor closer to 1 indicates model coverage closer to 0% (low reliability).

The bitstate verification is aimed at achieving a hash factor that is larger than 100. Improving the hash factor is nothing but minimizing the collision probability in cases where $h < r$.

A set of options are supported by SPIN to improve the hash factor. The first option is allocating the maximum amount of memory (compile-time `-DMEMLIM`). The second option is by carefully choosing the hash table's size (run-time option `-wN`). For example, assume that the real memory available is 128MB, and a billion states are expected to be reached, with each state using one-bit. The value of N in `-wN` is chosen such that N bytes can handle 1 billion states. When N=27, 2^{27} = 134,217,728 bytes which is 1,073,741,824 bits or 1 billion bits. The real memory is 128MB which is 1,024,000,000 bits or 1 billion bits. Therefore, if the available real memory is 128MB and the expected number of states is 1 billion, then N should be chosen as 27 (i.e., `-w27`).

Other options to improve the hash factor include using multiple independent hash functions (run-time option `-hN`) and setting more than one bit per state (run-time option `-kN`). These options are discussed further in Section 7.1.4.

The bitstate hashing method is enabled by using SPIN's compile-time directive `-DBITSTATE`.

In summary, the bitstate hashing technique helps to perform a clean and complete run on our model even if all other techniques fail due to state explosion. The technique, however, performs an approximate verification that may not provide a complete model coverage. To improve the model coverage (i.e., hash factor), SPIN supports techniques that involve tuning the verification run. The application of bitstate hashing for our verification is described in the upcoming sections.

7.1.3.2.5 Hash-Compact

The hash compaction is a variant of the bit-state hashing. The idea here is to increase the hash table's size to a very large value, like 2^{64} bits. For each state *s*, a single hash value is calculated within the range $0..2^{64}-1$. The calculated hash value is a 64-bit number (or a 64-bit address), and this number is stored in the hash table instead of storing the state *s* explicitly. In other words, the bits' addresses are stored in the hash-table instead of storing the whole state.

The hash-compaction offers a lower probability of hash collisions than bit-state hashing, giving an expected coverage of 100%. However, the memory usage is considerably higher compared to bit-state hashing. The work in [WL93] provides more details on hash-compaction. Chapter 9 of [Hol04] explains the technique concisely.

The hash-compact method is enabled by using SPIN's compile-time directive `-DHC4`.

We will use the hash-compact method in the upcoming sections to perform a clean and complete run of our model if all the lossless compression techniques fail. If hash-compact also fails by returning an out-of-memory error, only then will we use the bitstate hashing.

7.1.3.3 Comparing Exhaustive, Bitstate and Hash-compact Techniques

With exhaustive search as the basis, we will now compare the two lossy compression techniques, bitstate hashing and hash-compaction. Figure 7.3 shows the model coverage metrics plotted against available memory, m, in bits. k is the number of hash functions used. That is, k bits are stored for each state, with each of the k bit-positions computed with an independent hash function.

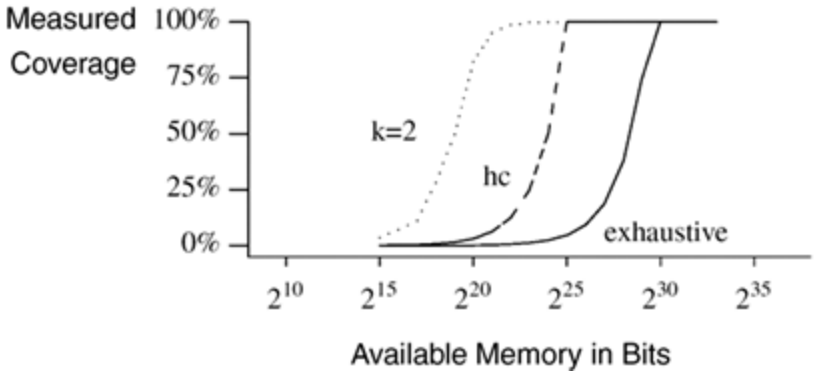


Figure 7.3: Model coverage acheived with different search techniques [Hol04]

When the available RAM is sufficient to perform a clean and complete run, it is always preferred to do the exhaustive search. The area in the figure with $m > 2^{29}$ shows this. When the available RAM is slightly less than the required, then the hash-compact is preferred. The area in the figure where $2^{23} < m < 2^{29}$ shows this.

However, when the available RAM is significantly less than required (possible for very large models), bitstate hashing is preferred over the other two search techniques. The area in the figure with $m < 2^{23}$ shows this. Observe that, with bitstate hashing, it is still possible to achieve around 50% of coverage with just 0.1% of memory required for an exhaustive search.

The plot in Figure 7.3 is based on an experiment in [Hol98] for a problem size of 427,567 reachable states. In the upcoming sections, we will see that the comparison holds for our model as well, with large problem size. We will use the exhaustive search, hash-compact, and bitstate hashing in the said order on our model to observe that even if the first two methods fail, bitstate hashing will likely complete the verification without an out-of-memory error.

7.1.3.4 Summary

This section summarizes the different algorithmic techniques supported by SPIN to reduce the search complexity. The first column of Table 7.1 shows the compiler-directives to be enabled to use the technique listed in the second column. The third column lists the pros and cons of each technique.

Verification option	Meaning	Pros/cons
p.o. reduction	partial order reduction	Default mode; No side effect;
statement merging	statement merging	Default mode; No side effect;
-DCOLLAPSE	collapse compression	Good compression (reduced memory); Small penalty on verification time
-DMA	minimized automaton storage	Very good compression; Large penalty on verification time
-DHC4	hash-compact	Very good compression; Small chance of missing states during search
-DBITSTATE	supertrace verification	Excellent compression; Fast; Relatively higher chance of missing states during search

Table 7.1: Summary of SPIN's algorithmic techniques to reduce the state-vector size

So far, we have been stating the memory and time used by each of the SPIN's algorithmic

techniques with terms like ‘reduced memory,’ ‘small penalty on run-time,’ etc. For better clarity, let us look into some concrete metrics of memory and time consumed.

Table 7.2 shows the verification memory and time metrics for one of the test sets of our model (upcoming Section 7.5 discusses the test sets). The description in the third column of Table 7.1 is based on these metrics.

Optimization technique	Memory used (in Megabyte)	Time used (in second)	Model coverage (in %)
Without partial order reduction (-DNOREDUCE)	175.3	17	100
With partial order reduction	1.5	0.076	100
With collapse compression (-DCOLLAPSE)	24.7	20	100
With minimized automaton (-DMA)	7.3	71	100
With collapse compression and with minimized automaton (-DCOLLAPSE) + (-DMA)	2	44	100
With hash-compact (-DHC4)	12.9	15	100
Bitstate hashing (-DBITSTATE)	3	28	96.7

Table 7.2: Memory and Time Metrics for each algorithmic technique supported by SPIN

7.1.4 Tuning the Verification to Improve Performance

Next to the non-algorithmic and algorithmic techniques, SPIN supports additional techniques to improve the verification performance. Here ‘performance improvement’ can be, for example, increased model coverage, reduced verification time, etc. The subsequent sections describe some of the tuning techniques.

For a complete list of SPIN verification options, see SPIN’s manual page [SPI11].

7.1.4.1 Compiler Optimization (-O2)

The Promela model that is input to SPIN is first converted to an equivalent C code named `pan.c` (‘pan’ refers to ‘process analyzer’). This C code is converted to an executable verifier named `pan.exe`. The verifier is the one which checks the property and returns a counterexample, if any.

The flag `-O2` is a compiler optimization flag. The flag optimizes the C code (`pan.c`) thereby reducing the run-time of the verifier by half. The flag is enabled by default.

7.1.4.2 Setting the Upper Bound for Memory Usage (-DMEMLIM)

The `-DMEMLIM` is a directive used to set the upper bound for the memory allocated for verification. By default, `-DMEMLIM=1024`. This means that the verifier can use a maximum of 1024 Megabytes.

SPIN shows the error message `pan: reached -DMEMLIM bound` to indicate the set memory bound is reached. In the upcoming sections, we will see how to use the `-DMEMLIM` directive for our verification by increasing the memory to fix this error. The maximum value of `-DMEMLIM` cannot exceed the real physical memory.

7.1.4.3 Selecting the Maximum Search Depth (-mN)

SPIN's run-time option `-mN` is used to set the search depth. The default search depth is 10,000 steps. In other words, the search is *bounded* to 10,000 steps. If the default search depth is insufficient, the search truncates at 9,999 steps (the truncation can be seen in the verification log), making the verification less exhaustive. In the case of truncation, the bound should be set to a higher value using the `-m` flag. For example, `$./pan -m20000`.

In some cases, the counterexample trace leading to a property violation has a large number of steps. A large number of steps make it hard to identify the actual cause of the violation. In such cases, the search depth can be set to a smaller value to find a shorter trace leading to the violation. For example, `$./pan -m40`.

Alternatively, SPIN's compile-time directive `-DREACH` can automatically truncate the length of the counterexample trace. The `-DREACH` directive helps if the model completes a *clean and complete* run during exhaustive verification and if a violation is found. The `-DREACH` directive, however, can increase the memory and time requirements.

7.1.4.4 Selecting the Size of the Hash-Table (-wN)

The hash table is the indicator of the physical memory (RAM) used during verification. There is no need to change the hash table's size when the model is compiled without the bitstate option (`-DBITSTATE`). This is because SPIN adjusts the hash table size if the default value is too small. The default value of hash table size during exhaustive verification is `-w24`.

When a bitstate run is used, it is required to specify the hash table size carefully. A too high value greater than the available RAM makes the system page, therefore should be avoided. Therefore, the maximum value set is the real RAM (not virtual) that can be used. A too low value will not utilize the available RAM, giving a lower than possible model coverage. Remember, bitstate hashing is a lossy compression, and its results are reliable if the model coverage (i.e., hash factor) is higher. The default value of hash table size during bitstate verification is `-w20`.

With a bitstate run, N's value should be chosen such that it is at least equal to the nearest power of 2 of the number of reachable system states expected. For example, if 32 million reachable states are expected, then N is set to 22 (`-w22`); 2^{22} bits is 32 million bits (=states), and the exploration of 32 million states require around 4MB of RAM.

7.1.4.5 Choosing a Different Hash Function (-hN)

For a fixed number of reachable states, fixed memory, and a fixed size of the hash table, the bitstate run produces significantly higher coverage than the exhaustive run. This implies the improved accuracy of verification and improved chances of finding property violations when using a bitstate run. On the downside, the bitstate run generally denies the certainty of proof if no violations are found. Fortunately, there is a solution to this uncertainty.

When using a bitstate run, it is possible to improve the chances of finding violations by repeating the run with different hash functions (`-hN`). Bitstate runs without changing the hash function lead to unresolved hash conflicts causing a truncation of the search. However, changing the hash functions over multiple bitstate runs causes the hash conflicts to appear in different places; in other words, this will result in a different sampling of the state space.

SPIN supports 100 different built-in hash functions. The default is `-h1`. Changing the hash function to, say, `-h51` will result in a different sampling of the state space.

In summary, repeating bitstate runs with different hash functions improves the chances of finding property violations. In the upcoming sections, we will see the usage of the `-hN` tuning option on our model when bitstate run is used.

7.1.4.6 Varying the Number of Bits Set per State (-kN)

Besides changing the hash function, another way to influence the state space sampling is by changing the number of bits set per state (`-kN`).

The default is `-k3`, meaning three bit-positions per state by using three independent hash functions. There is no upper limit for N; any value greater than zero is valid.

In the upcoming sections, we will see the impact of varying `-kN` on model coverage for our verification, along with a graph.

7.1.4.7 Choice of Property to Be Verified

It is required to select the specific type of violation to be searched by the SPIN verifier. At the end of the Section 3.3.2, it is mentioned that SPIN supports the search for *safety* properties (e.g., absence of deadlock), and the search for *liveness* properties (e.g., absence of non-progress cycle). The search for safety properties is enabled by default using SPIN's compiler directive `-DSAFETY`. To perform a check for non-progress cycles, the run-time option `-l` is used after compiling the model with the directive `-DNP` (NP is non-progress). While checking for livelock, the run-time option `-f` is used to enable weak-fairness. The search for safety and liveness properties cannot be combined.

A question may arise as to how the choice of the property can affect the performance of the verifier. To understand this, let us see the definition of *relative complexity* which is a variant of the *search complexity* defined at the beginning of the Section 7.1.1.

If M is the number of reachable states, B is the number of states in the property automaton, and S is the size of the state-vector, then problem size P is defined as follows [Hol04]: $\boxed{P = M * B * S}$.

If k is the number of processes in the model, experiments to assess relative complexity show that:

1. A search for safety property require memory and run-time that is same as problem size P.
2. A search for liveness property increases the run-time by a factor of two, i.e., $2P$ without impacting the memory requirements, i.e., P (same as in the first case).
3. A search for liveness property along with (weak) fairness enabled increases the run-time by a factor of $2(k+2)$, i.e., $P * 2 * (k+2)$ without impacting the memory requirements, i.e., P (same as in the first case).

The data from the experiments guides the user to follow the below sequence while checking for properties. We will follow the same sequence of verification in Section 7.3.

- Whenever possible, use safety properties
- Only when needed, use liveness properties, and
- Only when unavoidable, use fairness constraints.

For more details on the experiments and how the complexity is calculated, see Chapter 8 of [Hol04].

In Section 3.3.4.1 we have seen an example of how SPIN detects deadlock. Subsequently, in the same section, we have seen an example of how SPIN detects non-progress cycles and the significance of weak fairness. For our verification journey, it is additionally required to know more about *progress label* while checking for liveness properties. The next section introduces the *progress label*. The upcoming sections use the progress label extensively.

7.1.4.8 Progress Label

A progress label is a label-name prefix in Promela used to specify liveness properties. A progress label can be any label name starting with the eight-character sequence "progress." For example, it is valid to use several progress labels, like, progress1, progress2, etc.; the only condition is the progress label must start with the character sequence "progress."

Listing 7.1: Example usage of progress label

```
1 proctype P1(){
2 L1:
3 do
4 :: Ch1!S1 ->
5    progress: Ch?S2
6 od
7 }
```

We have seen in Section 3.3.1.10 that a label in Promela prefixes a statement, which helps in uniquely identifying a local process state. A state marked with a progress label is the way to instruct the verifier that it is required to traverse this state in any infinite execution. In other words, the state marked with a progress label must be visited *infinitely often* in any infinite system execution. A violation of this will be reported as a *non-progress* cycle by the verifier.

Example usage of progress label is as follows. The progress label in Listing 7.1 is an indication to the SPIN verifier that the statement `Ch?S2` must be visited infinitely often in any infinite system execution.

7.1.4.9 Summary

Table 7.3 summarizes the tuning techniques supported by SPIN to improve the verification performance.

7.1.5 Discussion

We started this section by listing the reasons for the unsuccessful/incomplete during state space exploration; we used the definition of *Search Complexity* for this (Section 7.1.1). We then described the techniques that can be used to reduce the search complexity. The techniques involved two types. The non-algorithmic techniques refer to changes/optimization in the model (Section 7.1.2), and algorithmic techniques refer to the lossless and lossy compression techniques supported by SPIN (Section 7.1.3). We have also looked into the memory and time metrics of one test set to compare the different algorithmic techniques.

In scenarios where it is inevitable, if the lossy compression techniques are used and if the achieved model coverage is less than 100%, then there are techniques to improve the coverage. This is done by tuning the verification. These tuning techniques have been described in Section 7.1.4.

Verification option	Meaning
-O2	Compiler optimization flag
-DMEMLIM	Set upper bound for memory usage
-mN	Set maximum search depth
-wN	Set size of the hash table
-kN	Set number of bits per state
-hN	Choose a different hash function
-DSAFETY	Enable check for deadlock (safety property)
-DNP	Enable check for non-progress cycle (liveness property)
-f	Enable weak-fairness

Table 7.3: Summary of verification tuning options

Now that we know different techniques to reduce search complexity, the next questions are: (a) where to start with the verification?, and (b) how to proceed with the verification?. In other words, we need a roadmap that can guide us with the verification process. The next section describes the roadmap that we will follow to verify our model.

7.2 Verification Roadmap

Figure 7.4 illustrates the verification roadmap that will be followed to verify the generated model. We will now describe the different steps in the roadmap; the step numbers are indicated in the figure. We will also see an overview of the outcome of each step on our collapsed model. The outcome is explained in more detail in the subsequent sections.

Starting from the top of the figure, the first step is to define the model M to verify a property P. In Section 6.6.3 we discussed the collapsing technique to reduce the state space of the generated model. The collapsed model with reduced state space consisted of 17,708 lines and 12,585 transitions. **Throughout this chapter, M is nothing but the collapsed model generated by the toolset, and P is the check for absence of deadlock or check for absence of livelock**.

The second step is to do a sanity check of the model. This includes the check for any syntax errors. If there are no syntax errors, then a random/interactive simulation is run to get a quick glimpse as whether the model's behavior is as expected. If there are no (minor) modeling errors, we proceed to the next step. In our model, the sanity check is a pass.

The third step is especially useful for a large model (as in our case). In this step, SPIN's bitstate verification is used to check for any logical errors in the model, which may require several modifications. An example of a logical error is when there is a mismatch between the behavior of a code construct and the behavior of the Promela construct for this code construct. The bitstate verification also helps to get a quick glimpse of the run-time of the model. This step guides us to carefully choose the next steps to reduce the run-time in case of a long run-time. In our model, no modeling errors were found. The run-time of the collapsed model was 12 minutes and 39 seconds. The initial model coverage was, however, poor (hash factor = 1.63).

The fourth step is to verify the model using the default exhaustive search for a property. If successful, the verification is said to be complete with either of the two outcomes: (a) the said property holds in the model, or (b) the said property does not hold in the model, and a counterexample is generated. However, if the verification is incomplete, we proceed to the next step. In our work, the collapsed model could not complete an exhaustive run.

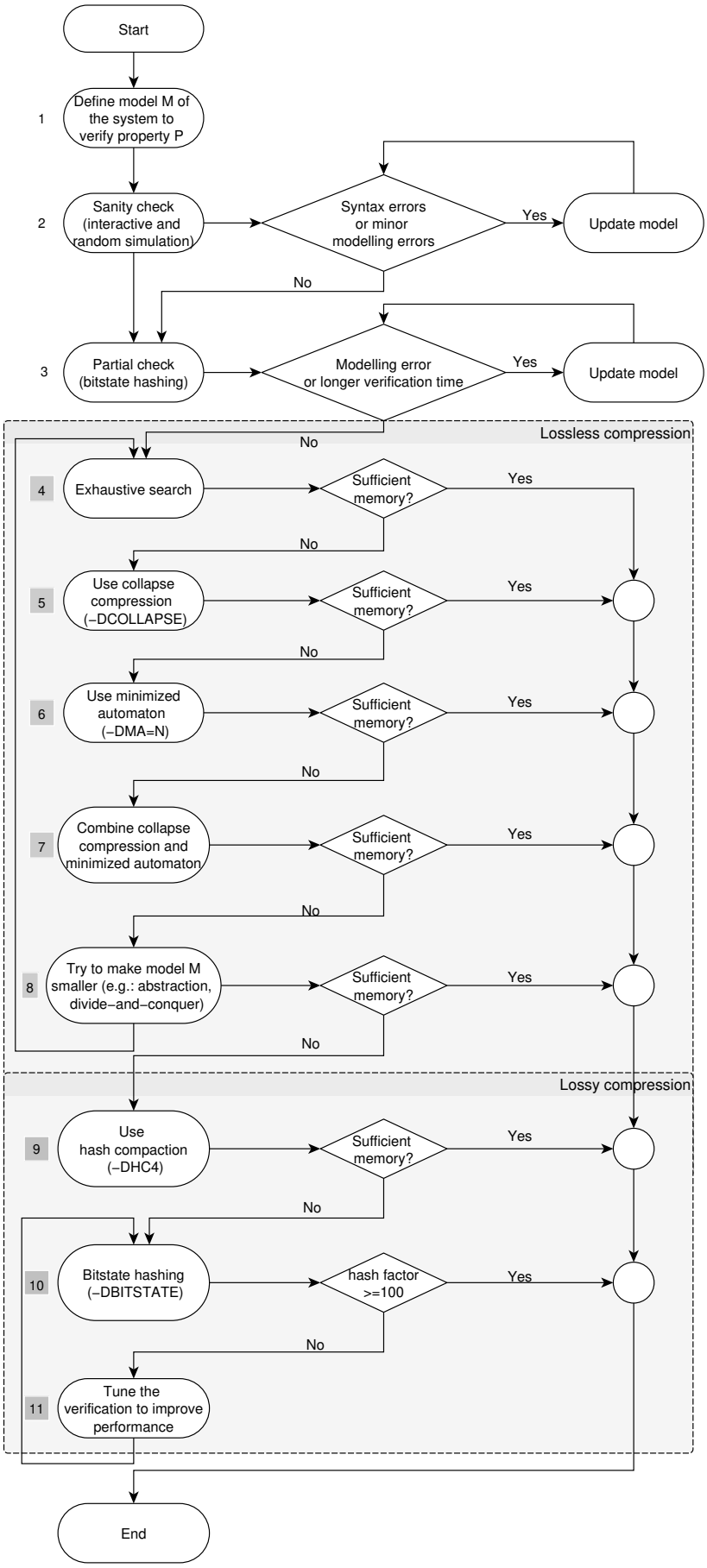


Figure 7.4: Model Verification Roadmap

The fifth step is to use the first lossless compression technique, the collapse compression. If the verification fails, we proceed to the next step. In our work, the collapsed model could not complete the verification with `-DCOLLASPE`.

The sixth step is to use the second lossless compression technique, the minimized automaton. If the verification fails, we proceed to the next step. In our work, the collapsed model could not complete the verification with `-DMA`.

The seventh step is to combine the two lossless compression techniques, collapsed compression plus minimized automaton. If the verification fails, we proceed to the next step. In our work, the collapsed model could not complete the verification with (`-DCOLLAPSE`) + (`-DMA`).

As mentioned at the beginning of this chapter, it is always better to use lossless compression techniques as much as possible, to avoid missing states during the search. The eighth step tells the user to investigate the possibilities to make the model, M, any smaller. A reduced M may help to stick to the lossless techniques. The reduction in M may be achieved by, say, rigorous abstraction or divide-and-conquer (see Section 7.1.2). The investigation in this step is the one that led us to use the compositional verification (described in upcoming Section 7.5) for our verification journey.

If M is the size of the original model, and M' is the size of the model after the eighth step, then there are two possible outcomes:

(a) $M' = M$, indicating that the model could not be made any smaller

(b) $M' < M$, indicating that there exists a technique that could reduce the size of the original model.

In the second outcome,(b), the reduction in model size could:

- be large enough such that a lossless compression technique on M' succeeds, or
- not be large enough, which leads us to inevitably use the lossy compression techniques on M'.

The eighth step helped the verification of our collapsed model by splitting it into (small) sub-models. However, not all sub-models could be verified using the lossless compression techniques. This leads us to the next step.

The ninth step is to use the first lossy compression technique, hash-compact. In our work, the collapsed model before using the divide-and-conquer technique (i.e., before the eighth step) could not succeed the hash-compact run. After using the divide-and-conquer technique (i.e., after the eighth step), one of the sub-models could not succeed the hash-compact run. This leads us to the next step.

The tenth step is to use the second lossy compression technique (the final resort), bitstate hashing. If the attained model coverage is closer to 100, then the verification results are more reliable. As mentioned in the third step, our collapsed model (before using the divide-and-conquer technique) could complete a bitstate verification, but with a poor hash factor of 1.63. After using the divide-and-conquer technique, the sub-model that could not complete a hash-compact run (in the ninth step) can now complete a bitstate run. However, the hash factor is still smaller than 100, even in the latter case. This leads us to the next step to improve the hash factor.

The eleventh step is to use the combination of the techniques illustrated in Section 7.1.4 to tune the verification to improve the hash factor. The question here is, which of these techniques are suitable for our model? There is no single answer to this question, as it requires several trial

runs depending on the model and the verified property. In the subsequent sections, we present the details of the tuning.

In summary, this section described the roadmap for our verification.

7.3 The Verification Journey

In this section, we will complete the verification process by following the roadmap.

The verification is performed using two different environments. The first environment is a 64-bit Windows 10 PC with 8GB RAM (usable RAM is up to 7GB). The second environment is a 64-bit Linux machine with up to 2TB RAM.

To support exhaustive verification with limited physical memory, we will use a state-of-the-art *divide-and-conquer* technique, proving that it is possible to perform an exhaustive search for large models with limited physical memory. Section 7.3.1 describes the divide-and-conquer technique, also called compositional verification. The technique helps to group an independent set of interacting processes and create sub-models from the collapsed model.

Under compositional verification, consider that some sub-models have memory limitation. The limitation may prevent the sub-models from either (a) completing a clean run or (b) achieving good coverage. In such a case, we switch to the second environment, the Linux machine with higher memory, with up to 2TB RAM. In any case, the verification runs that use up to 7GB RAM produce the same verification metrics and verification results from both environments. Verification runs using more than 7GB RAM are explained in detail in the subsequent sections.

Section 7.3.1 is further divided based on the verified property, as follows:

1. Property (deadlock/livelock)
    (a) Verification metrics (memory, time, model coverage etc.)
    (b) Verification result (property holds or property fails)

Detected property violations are traced back to the source code. The path(s) leading to the violation in the source code is/are illustrated with snippets of Ada and Promela, along with a control-flow graph.

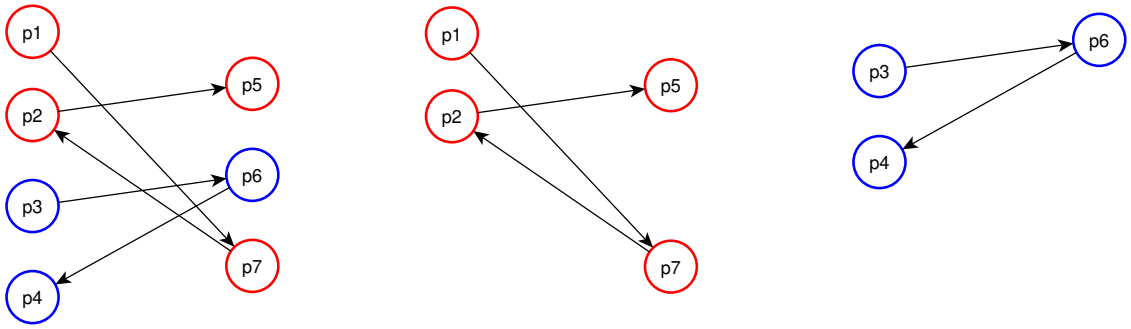
7.3.1 Compositional Verification

With reference to using the exhaustive verification as much as possible (the eighth step of the verification roadmap), further analysis of the collapsed model showed the possibility of applying a compositional verification technique. The compositional verification is a *divide-and-conquer* technique to extract sub-models from the complete model without sacrificing the system behavior. In this technique, a set of sub-models is verified exhaustively to check for concurrency problems. In other words, the same property verified on the full model is now verified on a set of sub-models. Detected violation, if any, is then traced back to the source code for validation.

Some of the earlier works using the divide-and-conquer technique for model checking are as follows. In [Aun+21] the authors show an example of how dividing the reachable state space helps to address the state explosion problem. The authors of [OZ13] present how divide-and-conquer could make model checking feasible for liveness problems under fairness constraints.

We will now describe how this compositional verification technique is applied to our model. Consider a network of interacting processes, representing a set of communicating Ada tasks,

as shown in Figure 7.5a. The interacting processes can be visualized as a graph in which each node represents a process, and each edge represents a shared channel for interaction.



(a) A network of interacting processes

(b) Separating the first set of connected processes

(c) Separating the second set of connected processes

Figure 7.5: An example showing the compositional verification approach

Figure 7.5a shows that processes p1, p2, p5 and p7 together form one set of connected component of the graph. Similarly, the processes p3, p4, and p6 together form another set of connected component. A process in the first set of connected component will not affect the behavior of a process in the second set of connected component, and vice versa. In other words, the two components interact independently. Without loss of generality, a property to be proved on a complete model (with all seven processes) can now be proved on each of these sub-models (one sub-model with four processes and another sub-model with three processes).

The advantages of the compositional verification technique are listed below:

- The sub-models have reduced state space, providing higher chances to use SPIN's exhaustive verification.
- Even if the different sets of connected components are verified within the same model, it is obvious to see that a process in one independent set will not have any interaction with the process of another independent set. Therefore, this approach is safe and does not change the behavior of the modeled system.

Just like in Figure 7.5, the collapsed model has processes that can be grouped into sets of independent sub-models and verified separately. From Section 6.6.3 we know that the collapsed model has 52 processes corresponding to the 52 Ada tasks in the XML with at least one paired synchronization. Different test sets are created by grouping sets of processes (among the total 52 processes) that interact independently. In essence, the sum of processes combined from all test sets gives 52, which is nothing but the total number of processes in the collapsed model. Note that 'test sets' refers to several Promela files (*.pml) of different sizes created from the (complete) collapsed model, each containing a group of independently interacting processes.

The next section illustrates the verification metrics and verification results for each of the test sets. The verified property is the absence of deadlock. If the property is violated (i.e., if a deadlock is present), SPIN generates counterexample(s).

7.3.1.1 Check for Absence of Deadlock

This section will describe the test sets created for the compositional verification to check for the absence of deadlock. First, we will present the verification metrics, followed by the verification

result.

7.3.1.1.1 Verification Metrics

Table 7.4 shows the test sets created under compositional verification to check for the absence of deadlock. The second column of the table shows the number of connected processes in each test set. **Note that the sum of number of processes in the second column gives 52, which is nothing but the total number of processes in the collapsed model**. The third column indicates the type of verification. Column four to column eight provides the verification metrics from SPIN - the fourth column indicates the number of states in the verified sub-model. The fifth column indicates the total number of transitions explored, and this serves as a statistic for the amount of work done by SPIN to complete the verification. The sixth column indicates the total memory used. The seventh column indicates the actual search depth, and the eighth column indicates the time consumed.

Test set #	# of processes	Verification type	# of states	# of transitions	Memory used (Megabytes)	Search depth (steps)	Verification time (seconds)
1	2	Exhaustive verification	16	32	128.73	13	4.84
2	2	Exhaustive verification	10	14	128.73	10	4.35
3	2	Exhaustive verification	43	90	128.73	21	4.28
4	2	Exhaustive verification	54	161	128.73	19	4.81
5	2	Exhaustive verification	219	455	128.73	8	5.41
6	2	Exhaustive verification	4	6	128.73	4	4.77
7	8	Exhaustive verification	2.40E+05	2.70E+06	219.217	229,510	9.45
8	32	Exhaustive verification	3.48E+06	2.04E+08	5,722	5,529,622	1,740

Table 7.4: Verification metrics for the check for deadlock

Note that SPIN's default search depth of 10,000 steps (i.e., `-m10000`) and the default hash table size of 2^{24} (i.e., `-w24`) are used for test set 1 to test set 6. These sets' memory usage can be reduced below the indicated value of 128.73MB by setting smaller than default values for search depth and hash table size.

Test sets 1 to 6 could complete an exhaustive verification, and their verification metrics are presented in Table 7.4. Next, we proceed to test set 7. This set, unlike previous sets, has eight connected processes. The verifier's default search depth for this test set is changed from 10,000 steps to 230,000 steps to support the increased number of states and transitions. Except for the change in search depth, all other settings remain the same as in the previous six test sets. The number of states explored hit a six-digit mark with 239,832 states, the highest among the first seven test sets. The memory usage is 70% higher than the previous six test sets, and the time consumed is 9.45s, the highest compared to the previous six test sets.

We will now look into the metrics of the test set 8. This is the largest test set with 32 connected processes. Note that test set 8 is just like other test sets like set 1, set 2, set 3, etc., with 32 processes interacting as a set. What makes set 8 unique over other sets is that it is the set with the largest number of connected components, making its exhaustive verification challenging. Set 8 has 15,097 lines of Promela code out of the total 17,708 lines of the collapsed model. In

other words, this set constitutes 85% of the collapsed model. Like the previous test sets, we will follow the roadmap to verify test set 8. Table 7.5 presents the verification metrics of set 8.

We will now present our observations on the verification metrics of set 8 as seen in Table 7.5. As a reminder, set 8 is also verified on a 64-bit Windows 10 PC with 8GB RAM.

(i) Exhaustive run: The exhaustive run of all 32 processes in set 8 did not succeed with memory-bound set to 7GB (`-DMEMLIM=7168`) and the search depth set to 20,000,000 steps (`-DMEMLIM=20000000`). To find the **point at which the verification runs out of memory**, the number of processes in the model was increased gradually from zero, with increasing order of process size. That is, we start with a small number of processes of smaller size and gradually increment the number of processes. By doing this, the exhaustive run could succeed for up to 17 processes, with minor corrections like increasing the search depth and memory bound.

Now, the 18^{th} process is added to the set. After exploring 20,057,199 states for 334 seconds, the exhaustive run returned an out-of-memory error. As per the roadmap, the next step is to verify these 18 processes using collapsed compression.

(ii) Collapse compression: The number of processes (=18), the memory bound, and the search depth are retained the same as in exhaustive run. Additionally, the option `-DCOLLAPSE` is enabled. After exploring 89,703,083 states for 5.2e+03 seconds (around 1hour and 45minutes), the verifier returned a out-of-memory error. Observe that the number of states explored by collapse compression is 347% higher than the exhaustive run. The downside, however, is the increased run-time. As per the roadmap, the next step is to verify these 18 processes using minimized automaton.

(iii) Minimized automaton: The number of processes (=18), the memory bound, and the search depth are retained the same as in exhaustive run. On a trial and error basis, the automaton's initial size is set to 450 (`-DMA=450`). The technique could complete an exhaustive run for the 18 processes after exploring 2.95e+08 states. Although the memory-bound was set to 7GB, the minimized automaton run used only 1.1GB. SPIN also suggested that `-DMA=288` is sufficient over `-DMA=450`. The run-time, however, is a major downside at 17,800 seconds (around 5 hours).

(iv) Collapse compression + minimized automaton: Following our verification roadmap, let us see what happened when the collapse compression and minimized automaton are combined. The number of processes (=18), the memory bound, and the search depth are retained the same as in the exhaustive run. Additionally, the options `-DCOLLAPSE` and `-DMA=450` are enabled. The search explored 2.95e+08 states (same as minimized automaton). The memory usage increased from 1.1GB to 1.8GB. The major downside was the run-time which was about 28 hours. In summary, nothing interesting happened by combining collapse compression and minimized automaton, as compared to using only minimized automaton.

So far, we verified set 8 only partially for up to 18 processes. Out of all the exhaustive runs, only the minimized automaton (MA) could succeed, although with high run-time. Regardless of run-time, MA is a lossless compression technique, and we aim to stick to lossless techniques as much as possible. Considering this, the number of processes is now increased from 18 to 32; i.e., complete set 8 is now verified with MA. Unfortunately, this time, the MA run returned an out-of-memory error after 4 hours and 21 minutes. Following the roadmap, we move on to using the first lossy compression technique, the hash-compact.

(v) Hash compact: The complete set 8 with 32 processes is verified using the hash compact (DHC) technique. The memory bound is set at 7GB, and the search depth at 14,000,000

Verification of test set 8 Total number of processes in set 8 = 32 Environment: 64-bit Windows machine with 7GB RAM					
Verification type	**# of processes out of 32**	**# of states**	**Memory used (Megabytes)**	**Verification time (seconds)**	**Verification result**
Exhaustive verification	18	2.01E+07	>7000	3.34E+02	Out-of-memory
Collapse compression	18	8.97E+07	>7000	5.20E+03	Out-of-memory
Minimized automaton	18	2.95E+08	1100	1.78E+04	Success
Collapse compression + minimized automaton	18	2.95E+08	>7000	1.01E+05	Out-of-memory
Minimized automaton	32	6.50E+07	>7000	1.52E+04	Out-of-memory
Hash compact	32	8.90E+07	>7000	6.30E+03	Out-of-memory
Bitstate hashing	32	3.48E+06	5600	1.74E+03	Success

Table 7.5: Verification metrics for test set 8

steps (the value for search depth is set on a trial and error basis). After exploring 8.9e+07 states for about 1 hour and 45 minutes, the DHC technique returned an out-of-memory error. Following the roadmap, we will use the second lossy compression technique, the bitstate hashing.

(vi) Bitstate hashing: The complete set 8 with 32 processes is verified using the bitstate hashing (DBITSTATE) technique. The memory bound is set at 7GB, the search depth at 6,000,000 steps, the number of hash-functions (`-kN`) is set to 90, and the size of the hash-table (`-wN`) is set to 26. Note that the values for search depth, *-kN* and (`-wN`) are chosen based on trial and error. After exploring 3,481,461 states for 29 minutes, the bitstate hashing succeeded in completing the run. The actual memory usage is 5.6GB.

Remember that whenever bitstate hashing is used, we need to check the hash factor's value to know the model coverage. In this run, the hash-factor value is 19.27. In summary, the bitstate technique could complete a clean run at a record time and also used less memory than all the previous techniques. The next step is to try to improve the hash-factor. Following the roadmap, we will now experiment with a range of verification tuning techniques to improve the model coverage.

(vii) Tuning the verification: Trial runs are conducted using the knowledge gained from Section 7.1.4 to tune the verification. In one of the trial runs, the size of the hash-table (`-wN`) is increased from `-w20` to `-w27`. Additionally the number of bits set per state is increased from `-k55` to `-k5000` (again, on a trial and error basis). Furthermore, we have seen that repeating the run with different hash-functions (`-hN`) improves the chances of finding errors. Therefore, few more runs are performed on set 8 with different hash-functions like `-h21`, `-h31`, `-h41` etc. Next, we will present the tuning metrics and the observed coverage for set 8.

Until now, we have used the Windows PC for our verification. We will now switch to the Linux machine and see if we can get an improved hash factor with higher memory. Table 7.6 shows the verification metrics for set 8 in the Linux environment. Two tuning techniques are used: (a) vary the hash-table size, and (b) vary the number of hash functions

Bitstate verification (-DBITSTATE) for test set 8 Environment: 64-bit Linux machine with upto 2TB								
Run #	**Size of the hash-table (-wN)**	**Number of hash functions (-kN)**	**hash factor**	**# of states**	**# of transitions**	**Depth reached**	**Memory used (MB)**	**Verification time (second)**
1	w26	k5000	549.18	1.22E+05	7.69E+06	198,515	953	844
2	w26	k100	19.27	3.48E+06	2.04E+08	5,529,622	5,722	1,740
3	w20	k90	17.75	5.91E+04	3.65E+06	96,848	6,294	25.4
4	w26	k1000	133.52	5.03E+05	3.01E+07	811,509	8,030	820
5	w25	k55	11.97	2.80E+06	1.62E+08	4,420,128	10,719	1,180
6	w26	k150	26.85	2.50E+06	1.49E+08	3,987,386	10,719	1,410
7	w27	k200	34	3.95E+06	2.35E+08	6,302,314	12,064	2,450
8	w27	k100	19.27	6.96E+06	4.10E+08	11,048,365	16,098	1,943

Table 7.6: Bitstate verification of test set 8

used. For each variation of the tuning technique used, the observed hash factor, the number of states reached, the number of transitions taken, the depth reach, the memory used, and the time consumed are presented in the table.

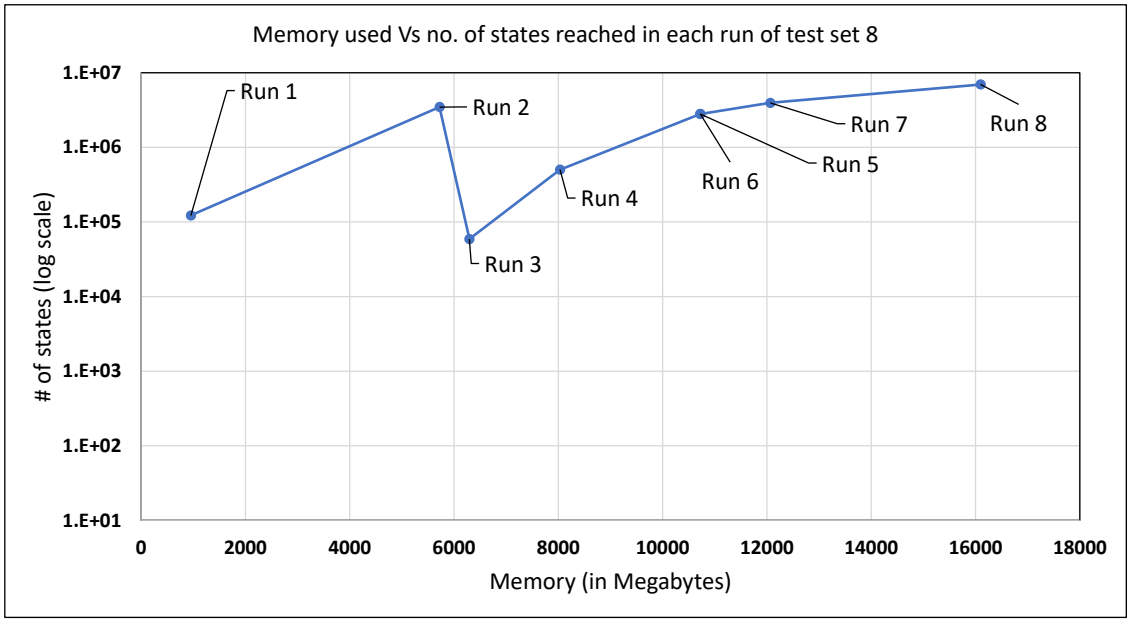


Figure 7.6: Number of states reached for different `-wN` and `-kN` for set 8

Some key observations from the table can be visualized using the graph shown in Figure 7.6. The observations are listed below.

- See test run 2. It shows that with `-w26` and `-k100`, the verifier could reach 3.48E+06 states with just 5.7GB memory.
- See test run 7 and compare it to test run 2. It shows that by increasing hash-table size from `-w26` to `-w27`, and with increased number of hash functions (`-k200`), the verifier could reach slightly more states than in test run 2 (with `-w26`). However, the penalty is the memory usage which increased from 5.7GB to 12GB (nearly doubled).
- See test run 8 and compare it to test run 7. It shows that it is possible to reach more states further with a reduced number of hash functions. However, the memory

penalty is even higher at 16GB.

- See test run 1 and test run 4. They show that the hash factor is greater than 100 (desirable), but the number of states reached is less compared to, say, test run 1 with a lower hash factor.

The important message from the table is that a significantly large number of states could be reached with `-w26` and `-k100` using only 5.7GB memory as in test run 1. In other words, **reaching the maximum number of states within a given memory limitation is all that matters**. If memory is not a limitation, it could be possible to achieve a higher hash factor by increasing the hash-table size (`-wN`).

In summary, with a memory limitation of 7GB, only the bitstate hashing technique could successfully verify set 8. Subsequently, the hash factor is improved by using the tuning techniques. Next, we will describe the results of the verification.

7.3.1.1.2 Verification Results

The check for deadlock aims to find a potential deadlock scenario between the set of connected processes in each test set. More specifically, we are looking for deadlock scenarios similar to those we have seen in Section 2.4.1. SPIN reports a deadlock if none of the processes in a test set can proceed. In other words, when the system as a whole cannot *evolve*, then the system is said to be in a deadlocked state.

A check for deadlock is performed for test set 1 to test set 7 by enabling the `-DSAFETY` compile-time directive. SPIN's exhaustive run did not report a deadlock in set 1 to set 7. That is, in each test set, there is no cyclic dependency between the paired processes that prevented the model from evolving; each test set has at least one next state transition that prevented the model from going into a deadlocked state. Subsequently, some random and thorough interactive simulation runs are performed on each test set to ensure the expected process interactions. Here the focus of interactive simulation is to check if the model takes all possible transitions in each process. The simulation results showed that the connected processes in each test set are synchronizing as expected (i.e., the behavior is similar to the communication between the corresponding Ada tasks).

The bitstate verification of set 8 did not report a deadlock either. However, a number of lines in the model could not be reached due to the high number of non-deterministic transitions. The unreached lines in the model are the states that are not visited by the model checker. In other words, the transitions that would make the model reach these states are never enabled. The unreached lines in the model are similar to dead-code, and they do not have a "deadly embrace" [SPI08]. The unreachable statements are technically redundant and can be removed from the model without any change in functionality.

Under verification metrics, we have seen that set 8 has 15,097 lines. The bitstate verification reported 404 unreached lines (of 15,097 lines) in set 8. For the sake of completeness, we will now look into an example showing non-determinism as a cause for unreachable states.

Consider two processes P1 and P2, as shown in Figure 7.7. State S1 is the initial state of P1, and state S2 is the initial state of P2. The signals used for state transitions are `a1, b0, b1`. From its initial state, Process P1 sends signal `a1` when it goes from state S1 to S2. The process P2 starts executing concurrently from its initial state. Note that P2 has three non-deterministic transitions to go to the states S1, S3, and S4 from its initial state S2. However, out of the three transitions, P2 can only take the transition to S1, as currently, only the signal `a1` is ready to be received. Therefore, P2 goes to state S1 by receiving `a1` from process P1.

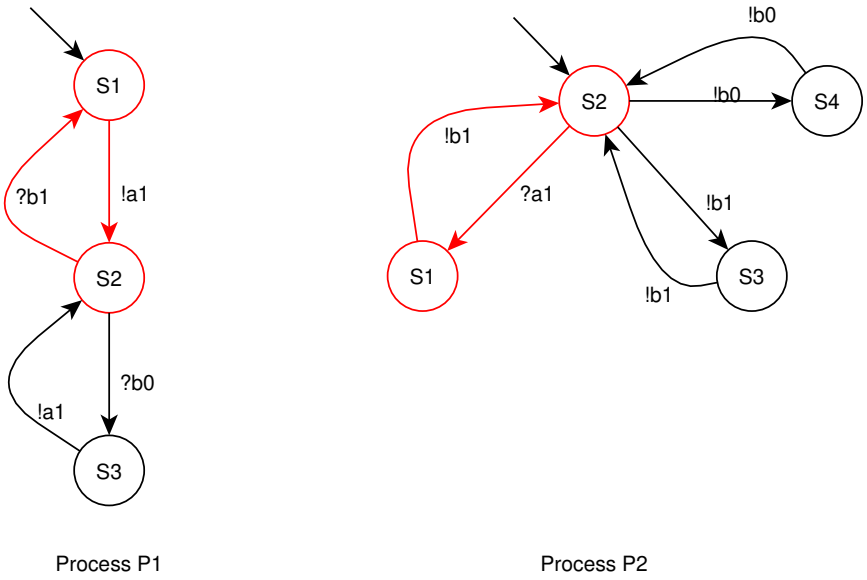


Figure 7.7: An example to illustrate unreached lines in the model

Process P1, which is currently at state S2 has two non-deterministic transitions, one to state S1 and the other to state S3. However, out of the two transitions, P2 can only take the transition to S1, as currently, only the signal `b1` (sent by P2) is ready to be received. Therefore, P1 goes to state S1 by receiving `b1`. This sequence of transitions continues, leading to three unreached states: state S3 in P1 and states S3 and S4 in P2. The verification of a Promela model for this example returns the unreached lines as shown in Listing 7.2.

Some of the reasons that caused unreached lines in set 8 are: (a) multi-task sync, wherein different tasks send the same signal, but only one of them can be accepted by the receiving task, (b) transitions involved in indirect synchronizations ("goto" leading to sync statements) along with transitions that involve sending/receiving a signal, etc.

In summary, our verification proved the absence of a potential deadlock in each test set.

Until now, we have seen the check for the absence of deadlock in each test set under the compositional verification technique. Following the order of property verification as stated in Section 7.1.4.7, in the next section, we will describe the check for livelock.

Listing 7.2: A section of SPIN verification log showing unreached lines in the model

```
1 ...
2 unreached in proctype P1
3         Example.pml:14, state 9, "a2b!a1"
4         (2 of 11 states)
5 unreached in proctype P2
6         Example.pml:29, state 12, "b2a!b1"
7         Example.pml:31, state 14, "b2a!b0"
8         (3 of 16 states)
```

7.3.1.2 Check for Absence of Livelock

This section will describe the test sets created for the compositional verification to check for the absence of livelock. We will use the progress label (see Section 7.1.4.8) to detect path(s)

leading to livelock(s) in each sub-model. More details on the usage of the progress label and the detected livelocks are presented under verification results. First, we will present the verification metrics followed by the verification results.

7.3.1.2.1 Verification Metrics

Table 7.7 show each test set's verification metrics to check for livelock. Like the check for deadlock, the check for livelock will also be verified using a Windows PC until the memory requirement exceeds 7GB. We switch to the Linux machine when the memory requirement exceeds 7GB. Nevertheless, the verification runs that use up to 7GB RAM produce the same verification metrics and results from both environments.

As before, SPIN's default search depth of 10,000 steps (i.e., `-m10000`) and the default hash table size of 2^{24} (i.e., `-w24`) are used for test set 1 to test set 6. The memory usage for these sets can be reduced below the indicated value of 128.73MB by setting smaller than default values for search depth and hash table size.

Test set #	# of processes	Verification type	# of states	# of transitions	Memory used (Megabytes)	Search depth (steps)	Verification time (seconds)
1	2	Exhaustive verification	36	188	128.73	28	5.73
2	2	Exhaustive verification	10	22	128.73	13	6.75
3	2	Exhaustive verification	23	94	128.73	24	5.78
4	2	Exhaustive verification	250	723	128.73	60	6.72
5	2	Exhaustive verification	159	1417	128.73	32	6.88
6	2	Exhaustive verification	17	86	128.73	24	5.94
7	8	Collapse compression (lossless compression)	7.98E+07	7.30E+08	12,064.29	68,692,405	7,880
8	32	Bitstate verification	3.56E+06	2.81E+08	6,294	5,639,742	2,430

Table 7.7: Verification metrics for the check for livelock

Listing 7.3: SPIN's directives to check for livelock (Linux environment)

```
1  Checking syntax and generating the pan.c verifier ...
2  spin -a Set_1.pml
3  No Syntax Error.
4
5  Starting verification...
6  /usr/bin/gcc -DMEMLIM=1024 -DNP -DNOCLAIM -DXUSAFE -DCOLLAPSE -DNOREDUCE -O2 -
       DVECTORSZ=1200 -w -o pan pan.c
7  ./pan -m10000 -l -f -c0 -e
8
9  Verification result:
10 ...
```

We will now look into the SPIN's compile-time and run-time directives used while checking for livelock. Listing 7.3 shows the SPIN's directives used for checking livelock. After a successful syntax check (line-3), the `pan.c` is generated. The generated `pan.c` is compiled to obtain an object file using a 64-bit gcc compiler (line-6). Line-6 indicates that:

Listing 7.4: SPIN's directives to check for livelock for set 7 (Linux environment)

```
1  Checking syntax and generating the pan.c verifier ...
2  spin -a Set_7.pml
3  No Syntax Error.
4
5  Starting verification...
6  /usr/bin/gcc -DMEMLIM=16384 -DNP -DNOCLAIM -DXUSAFE -DCOLLAPSE -DNOREDUCE -O2 -
       DVECTORSZ=100000000 -DNFAIR=9 -w -o pan pan.c
7  ./pan -m100000000 -l -f -c0 -e
8
9  Verification result:
10 ...
```

- the default memory used is 1024MB (`-DMEMLIM`)
- the check for non-progress cycles is enabled (`-DNP`)
- the check for an explicit LTL property is disabled (`-DNOCLAIM`)
- the check for validity of channel assertions is disabled (`-DXUSAFE`)
- partial order reduction is disabled (`-DNOREDUCE`). Reason: partial order reduction is not compatible with weak-fairness (run-time option of weak-fairness is `-f`)
- the compiler optimization flag is enabled (`-O2`), and
- a memory of 1200bytes is allocated for state-vector usage (`-DVECTORZ=1200`). This is an additional compiler directive required in Linux environment.

Next, the run-time options in line-7 indicates that:

- the `pan` executable uses the default search depth of 10,000 steps (`-m10000`)
- the search for non-progress cycles is enabled (`-l`)
- weak-fairness is enabled (`-f`)
- all the counterexamples for all the detected violations will be collected, instead of stopping at the first violation (`-c0 -e`)

Test sets 1 to 6 could complete an exhaustive verification, and their verification metrics are presented in Table 7.7. Next, we proceed to test set 7 with eight connected processes. The (default) exhaustive run for set 7 resulted in an out-memory-error for the default `-DMEMLIM=1024` and `-mN=10000` after about 1 hour and 42 minutes. Therefore, the next step, collapse compression is used with increased memory, `-DMEMLIM=16384`, and an increased search depth `-m100000000`. The change in the SPIN directives used for set 7 is shown in Listing 7.4. The values chosen for memory and search depth are based on a few trial runs until the verification succeeds.

The collapse compression successfully completed the verification for livelock for set 7. The search explored 79,780,420 states by using around 12GB of memory (as opposed to the set 16GB), and with search depth of 68,692,405 (as opposed to the set 100,000,000 steps). The run-time of the search is around 2hours.

Observe that the verification metrics for the livelock check for set 7 are significantly higher than the verification metrics for the deadlock check for set 7 (see Table 7.4). This proves the *relative complexity* that we have seen in Section 7.1.4.7, which guided us to check for safety properties first before checking for liveness properties.

Also, observe that set 7 is verified using the collapse compression (lossless) technique at the cost of exceeding the memory limitation of 7GB. To stay within the memory limit, set 7 is verified with the bitstate hashing. The bitstate hashing completed the verification of set 7 with just 1GB of memory and within 10 seconds. The observed hash factor is 20.

The verification of set 8 was complex for livelock check, similar to the complexity observed while checking for deadlock. set 8 could complete a clean run using the bitstate hashing. The hash table size is set as `-w27`, and the number of hash functions is set as `-k100`. The run-time is about 40 minutes and the memory used is around 6GB. The hash factor is observed to be 19.41.

The next section describes the verification results for the check for livelock.

7.3.1.2.2 Verification Results

The check for livelock aims to find potential non-progress cycle(s) between the set of connected processes in each test set.

Earlier, the check for deadlock did not report any violation. However, the check for livelock returned some execution paths leading to potential non-progress cycles. In this section, we will describe the detected non-progress cycles. The detected non-progress cycles are illustrated using Ada/Promela snippets and control-flow graphs.

Table 7.8 presents the summary of results for the check for livelock in each test set.

Property verified: Absence of livelock; **Verification environment**: 64-bit Linux machine		
Test Set#	**Verification result**	**Number of execution paths leading to livelock (if any)**
1	Livelock absent	Not applicable
2	Livelock absent	Not applicable
3	Livelock absent	Not applicable
4	Livelock present	3
5	Livelock absent	Not applicable
6	Livelock present	3
7	Livelock present	1
8	Livelock present	3

Table 7.8: Verification results for the check for livelock

From the table, the overview of reported livelocks are as follows:

- The check for livelock on test sets 1, 2, 3, and 5 did not report any non-progress cycles.
- The check for livelock on test set 4 reported three execution paths leading to a livelock. Reason: infinite execution of an `else` statement within a selective accept statement in the corresponding Ada code.
- The check for livelock on test set 6 reported three execution paths leading to a livelock. Reason: an `if` condition set to false infinitely, and another `if` condition within a `case` statement infinitely set to false in the corresponding Ada code.
- The check for livelock on test set 7 reported one execution path leading to a livelock. Reason: multiple reasons like, `if` condition being false forever, a specific `case` option is never taken, infinitely executing the `else` part of a selective accept etc.,

- The check for livelock on test set 8 reported three execution paths leading to a livelock. Reason: multiple reasons like in set 7. More details on this will be discussed in the subsequent paragraphs.

We will now illustrate two detected livelock scenarios, one in test set 6 and the other in test set 8. The detected livelocks are illustrated with Ada/Promela snippets and control-flow graphs.

Livelock in test set 6 illustrated:
Let us consider the livelock detected in test set 6. The Ada snippet corresponding to this set is shown in Listing 7.5.

The snippet has two tasks T1 and T2. Task T2 sends the entry call `S1`, which is accepted by T1. The snippet uses Ada's selective accept statement with delay. For better readability, the concurrency statements in lines 6, 20, and 32 are highlighted in red. The listing indicates a linear control-flow of the Ada code corresponding to set 6. However, the actual Ada code corresponding to set 6 is more complex than shown in the listing, with many procedure calls.

Without going into too many details, our focus here is to acknowledge that: (a) from line-17, T2 sends entry `S1` only if `cond1` and `cond2` are true, and (b) from line-26, T2 sends entry `S1` only if `case C1` is `a1`, and the conditions `cond3`, `cond4` and `cond5` are true.

The Promela model (i.e., test set 6) is shown in Listing 7.6. The set has two processes, T1 and T2, corresponding to the two Ada tasks. Progress labels are added in lines 5, 18, and 24 containing synchronization (highlighted in blue). As mentioned in Section 7.1.4.8, the progress labels instruct the verifier to execute the sync statements infinitely often in any infinite system execution. The line-19, marked in red, corresponds to the execution of the `else` statement at line-21 of the Ada code. Similarly, line-25 marked in red indicates that line-32 of the Ada code could not be executed (either because the `case` statement failed, or because anyone of the `if` conditions failed).

The test set is verified with the following options. The check for the non-progress cycle is enabled, and the weak-fairness is enabled. Additionally, the verifier is instructed to report all possible error paths and not stop at the first error (i.e., `./pan -c0 -e`). The verifier reported the presence of a non-progress cycle and provided three counterexamples with different paths leading to the non-progress cycle. Let us use a control-flow graph to show the detected non-progress cycles. The execution paths that can cause a livelock are marked in red in Figure 7.8.

The first execution path leading to a livelock is as follows. Process T2 goes to state `accept_message_1` from its initial state `L1`, and subsequently loops back to its initial state. This execution path is possible when either `cond1` or `cond2` at line-19 of the Ada snippet remain false.

The second execution path leading to a livelock is as follows. Process T2 goes to state `accept_message_2` from its initial state `L1`, and subsequently loops back to its initial state. This execution path is possible when - (a) `case C1` at line-28 of the Ada snippet is never `a1`, or (b) either `cond3` or `cond4` at line-30 of the Ada snippet remain false, or (c) `cond5` at line-31 of the Ada snippet remain false.

Listing 7.5: Ada snippet for test set 6

```
task body T1 is
begin
 loop
  begin
   select
    accept S1;
     -- some code
   end select;
 end loop;
end T1;

task body T2 is
begin
 loop
  begin
   select
    accept message_1
    do
    if cond1 and cond2 then
      T1.S1;
    else
     -- print something
    end if;
   end message_1;
  or
   accept message_2;
   do
   case C1 is
    when a1 =>
     if cond3 and cond4 then
       if cond5 then
        T1.S1;
     end if;
    end if;
   -- other options for case
   end case;
  end message_2;
 or
   delay 1.0;
  end select;
 end loop;
end T2;
```

Listing 7.6: Test set 6

```
mtype = {S1};
chan C1 = [0] of {mtype};

proctype T1(){
initial: C1?S1 -> progress1:
    goto initial
}

proctype T2(){
L1:
do
:: goto accept_message_1
:: goto accept_message_2
:: timeout
od

accept_message_1:
do
:: C1!S1 -> progress2: goto L1
:: goto L1
od

accept_message_2:
do
:: C1!S1 -> progress3: goto L1
:: goto L1
od
}

init
{
  atomic
  {
    run T1();
    run T2();
  }
}
```

The three execution paths reported by the verifier leading to the non-progress cycle are as follows.

1. `goto accept_message_1 -> goto L1 -> goto accept_message_2 -> goto L1 -> goto accept_message_1 -> goto L1 -> goto accept_message_1 -> goto L1 -> goto accept_message_1 -> ....`
2. `goto accept_message_1 -> goto L1 -> goto accept_message_2 -> goto L1 -> goto accept_message_1 -> goto L1 -> goto accept_message_1 -> goto L1 -> goto accept_message_2 -> goto L1 -> ...`
3. `goto L1 -> goto accept_message_1 -> goto L1 -> goto accept_message_1 -> ...`

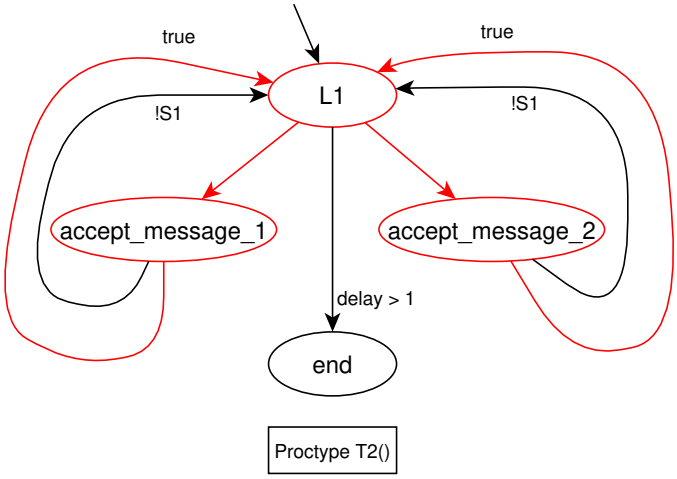


Figure 7.8: Livelock in test set 6

By looking at the verifier's execution sequence leading to the non-progress cycle, the Ada code's corresponding execution sequence can be obtained using the generated traceability matrix (see Section 6.7). Note that the reported execution sequences only show the possibilities of livelock(s). Whether these sequences can actually be taken or not in the code requires further analysis. In other words, our verification provides only a list of code execution sequences leading to livelocks(s), but to conclude whether these sequences are real or spurious requires analysis of the code by the software designer. The analysis is a yet-to-do task, which will be a part of future work.

Remember that the traceability matrix has code control points (with code line numbers) corresponding to each next state transition of the current state of the Promela model. It is, therefore, straightforward to identify the path leading to livelock in the Ada code. In the upcoming section, we will illustrate, with an example, how an Ada developer can use the verifier's counterexample and the toolset's traceability matrix to identify the path leading to a violation in the Ada code.

We have described the detected livelock in set 6. Next, we will see another detected concurrency problem in set 8.

Livelock in test set 8 illustrated:
We will now look into another concurrency problem reported in test set 8. The reported problem is again a livelock, but this time, it is a livelock leading to starvation of a process. Listing 7.7 shows the Ada snippet corresponding to a section of test set 8.

The snippet has three tasks P1, P2, and P3. P1 sends entry calls `S1` and `S2` to P2, and P3 sends entry call `S3` to P2. For easy readability, the snippet is written using sentences instead of using Ada syntax (e.g.:"Send entry S1 to P2" instead of `P2.S1`). As in the earlier example for set 6, lines with send and receive operations are highlighted in red.

A section of the test set 8 is shown in Listing 7.8. The set has three processes corresponding to the three Ada tasks. Progress labels are added in lines 4, 5, 14, 15, 19, and 25 containing synchronization (highlighted in blue). Note that line-26, highlighted in red, corresponds to a failure to reach line-33 of the Ada snippet.

A check for the non-progress cycle (with weak-fairness enabled) returned execution sequences leading to a livelock. The sequences highlighted in red in Figure 7.9 indicates the execution

sequence leading to a livelock. S0 is the initial state of processes P1, P2, and P3. From its initial state, Process P1 sends signal `S1`, goes to state S1, and eventually goes back to state S0. From its initial state, Process P2 accepts the signal `S1` sent by P1, goes to state S1, and eventually goes to state S3. For P2 to proceed to its next state S0, it requires signal `S3` which can only be sent by process P3. However, P3 executes infinitely in its initial state S0 by not making any progress. The non-progress cycle in P3 causes starvation (indefinite wait cycle) in P2; i.e., P2 is blocked in state S3.

Listing 7.7: Ada snippet corresponding to a section of test set 8

```
1  task body P1 is
2  begin
3   loop
4    initial state accept_Start;
5    if cond1 then
6     Send signal S1 to task P2;
7    else
8     Send signal S2 to task P2;
9    end if;
10  end loop;
11 end P1;
12
13 task body P2 is
14 begin
15  loop
16   initial state L1;
17   if signal S1 is ready to be accepted
         && delay < 3 then
18    Receive signal S1 from task P1;
19   elsif signal S2 is ready to be
         accepted && delay < 3 then
20    Receive signal S2 from task P1;
21   else
22    delay 3.0
23   end if;
24   Receive signal S3 from task P3;
25  end loop;
26 end P2;
27
28 task body P3 is
29 begin
30  loop
31   initial state L1;
32   if cond1 then
33    Send signal S3 to task P2;
34   end if;
35  end loop;
36 end P3;
```

Listing 7.8: A section of test set 8

```
1  proctype P1(){
2  accept_Start:
3  do
4  :: Ch1!S1 -> progress1: goto
       accept_Start
5  :: Ch1!S2 -> progress2: goto
       accept_Start
6  od
7  }
8
9  proctype P2(){
10 L1:
11 if
12 :: len(Ch1) > 0 ->
13    do
14    :: Ch1?S1 -> progress3: goto
         State_Change_2
15    :: Ch1?S2 -> progress4: goto
         State_Change_2
16    od
17 :: timeout
18 fi
19 State_Change_2: Ch2?S3 -> progress5:
       goto L1
20 }
21
22 proctype P3(){
23 L1:
24 do
25 :: Ch2!S3 -> progress6: goto L1
26 :: goto L1
27 od
28 }
```

Besides this, the verifier reports another issue. After going back to state S0, the process P1 again sends signal `S1`; i.e., now the length of channel `Ch1` is greater than zero with signal `S1` in the queue. However, process P2, the only task that can accept signal `S1` cannot accept the signal as it is blocked in state S3 because of process P3.

By this time, the corresponding Ada code line(s) causing the non-progress cycle in process P3 should become obvious. During run-time of the Ada code, if `cond1` at line-32 remains false, then this causes a non-progress cycle leading to starvation. The next step is to check the Ada code and see if there is a possibility for `cond1` to remain false. If this is the case, then the counterexample reported by SPIN is real, if not, then the counterexample is spurious.

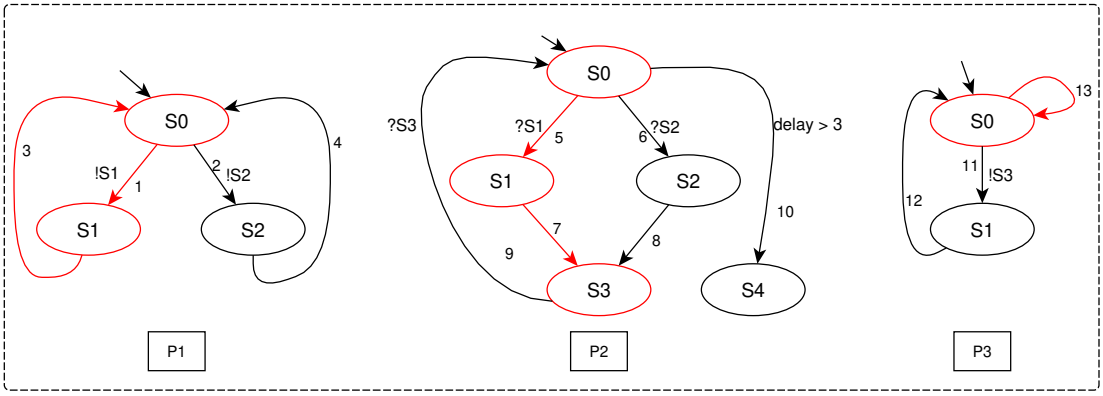


Figure 7.9: Livelock in test set 8

Note that the reported execution sequences only show the possibilities of livelock(s). Whether these sequences can actually be taken or not in the code requires further analysis. In other words, our verification provides only a list of code execution sequences leading to livelocks(s), but to conclude whether these sequences are real or spurious requires analysis of the code by the software designer. The analysis is a yet-to-do task, which will be a part of future work.

7.3.1.2.3 Summary - Check for Livelock

Let us summarize the check for livelock. The eight sub-models (or test sets) of the collapsed model are verified for the absence of non-progress cycles. The verification metrics showed that test set 1 to test set 6 could complete an exhaustive verification. Test set 7 completed the verification using the collapse compression technique (lossless) using more than the limited memory of 8GB. However, using bitstate hashing, the test set 7 could complete a clean run within the limited memory. Moving on to test set 8. The set could complete a clean run only using bitstate hashing. We have also seen the appropriate compile-time and run-time SPIN options to be enabled while checking for livelock.

We then described the verification results for the check for livelock. SPIN's verifier reported execution paths leading to a potential livelock in four out of the eight sub-models. We listed the reason(s) in the corresponding Ada code that can cause these livelock scenarios. To understand the details of the detected livelocks, we illustrated two examples.

1. The first example showed how the failure of `if` conditions or the failure to choose a specific `case` statement's option caused a livelock in test set 6.
2. The second example showed how a non-progress cycle in one process led to the starvation of a dependent process in test set 8.

7.3.1.3 Discussion

We have seen the verification results for the check for deadlock and the check for livelock. In regard to the verification results, the questions that remain unanswered are:

(a) What about the spurious counterexamples?

(b) How are the spurious counterexamples taken care of in our verification process?

Also, a question that requires an illustration is: How can an Ada developer know which specific condition(s) in the Ada code is/are leading to a potential concurrency problem?

The next section answers these questions. Section 7.3.2 answers the questions on spurious counterexamples. The subsequent Section 7.3.3 illustrates finding the specific condition(s) in the Ada code leading to a concurrency problem.

7.3.2 Model Refinement to Minimize the Number of Spurious Traces

While validating the livelock counterexample traces on the Ada code, it is observed that some execution paths in the model are not possible in the Ada code. Such paths leading to spurious traces were (manually) removed from the model. Let us consider an example of how test set 1 is refined to eliminate a spurious trace.

Listing 7.9 shows the Ada snippet corresponding to a section of set 1. Based on the boolean condition `cond1` being false or true respectively, either entry `S1` is sent as in line-6, or the entry `S2` is sent as in line-9.

Listing 7.9: Ada snippet corresponding to a section of test set 1

```
1  task body T1 is
2  boolean cond1;
3  begin
4   loop
5    if not cond1 then
6     T2.S1;
7    end if;
8    if cond1 then
9     T2.S2;
10   end if;
11  end loop;
12 end T1;
```

Listing 7.10: Refinement of test set 1

```
1  proctype T1(){
2  L1:
3  do
4  :: Ch1!S1 -> progress1: goto L1
5  :: Ch1!S2 -> progress2: goto L1
6  /* :: goto L1 */
7  od
8  }
```

Listing 7.10 shows a section of set 1. It is sufficient to focus only on line-6, a loop back to the initial state L1. This loopback in the model corresponds to the scenario where neither line-6 nor line-9 is executed in the Ada snippet. Since the conditions are kept abstract for our model, the toolset considers the transition path when both the `if` conditions in the Ada code are false, thereby generating a loop back to L1 in the model. In reality, this execution path is not possible in the Ada code; since `cond1` is a boolean, it is always the case that either one of the entry calls is sent. Therefore, line-6 of the model is commented to eliminate a spurious trace.

Given the large size of the code, it is not possible to check every code execution path and remove every transition in the model leading to a spurious trace. However, some initial steps towards refinement are taken in our model, such as (a) removing a loop back to the initial state/any previous state in the model corresponding to selective accept statement with delay in the Ada code, (b) removing a loop back to initial state/any previous state in the model corresponding to two `if` conditions using the same boolean variable in the Ada code, etc. Note that these are only small steps towards model refinement. Given the large number of non-deterministic transitions in the model, the corresponding impossible execution paths in the code might be larger than the possible execution paths in the code. Therefore, the model refinement task requires further investigation to minimize the spurious traces.

Some techniques are proposed in Chapter 8 to *automatically* remove transitions in the model that are not possible in reality. Nevertheless, while validating the paths leading to livelock, an Ada developer can simply ignore any spurious traces and consider only the real traces (like in Figure 7.8 and Figure 7.9).

The following section illustrates finding the specific condition(s) in the Ada code leading to a concurrency problem.

7.3.3 Using the Toolset's Traceability to Trace Violation Back to Ada Code- an Illustration

Until now, we have described some of the livelocks detected by SPIN. As a reminder, using the SPIN option `./pan -c0 -e` it is possible to obtain all possible paths leading to a potential livelock. This means, at the end of the execution, we will have a list of paths (error traces) leading to possible livelocks. The next step is to use the error traces and trace them back to the corresponding Ada code. Tracing error traces back to Ada code is a straightforward task (as mentioned in Section 6.7). For the sake of completeness, below, we will show an example (from test set 6) of how the toolset's traceability matrix is used to identify the cause of livelock in the Ada code. More specifically, we will see how to identify which conditional statement(s) in an execution path can cause a potential livelock in the Ada code.

Assume that an Ada developer has the SPIN's error trace and the toolset's traceability matrix. We will now see how the developer can use these data and identify the cause of livelock in the Ada code. Figure 7.10 illustrates the execution sequence of two Ada tasks T3 and T4, leading to a livelock. Column-B shows the starting point (which Ada file and which Ada task to look for). The execution sequence is from left to right. Consider the execution sequence in row-4. The body of task T3 is at line-387 of the Ada file `T3_T4.adb`. From the task body, go to the `select` statement at line-392. Next, go to line-393, in which T3 is waiting to accept the entry `Wait_Detect`.

Next, consider the execution sequence of Task T4 in row-5. The body of task T4 starts at line-461 in the same Ada file. From the task body, go to the `select` statement at line-467, and subsequently go to the `accept` statement at line-493. Next, consider the `if` statement at line-495. Within the `if` statement, if the condition `Events(L1) = False` or the condition `State != Idle`, then the control loops back to the starting point of the task body, leading to a livelock. The developer's next step is to check if the said conditions can take the mentioned values during the code's run-time. If yes, then this is a real counterexample; if not, then it is a spurious counterexample. A similar explanation holds for the other two execution sequences of task T4, as shown in row-6 and row-7. In conclusion, by using the toolset's traceability matrix along with SPIN's counterexamples, an Ada developer can trace the detected violations back to the Ada program.

7.3.4 Summary of Verification Journey

In this section, we summarize our verification journey. Section 7.3.1 started by illustrating the divide-and-conquer technique and creating sub-models from the collapsed model. The section had two sub-divisions.

The first sub-division described the check for a deadlock in each test set. Under check for deadlock, we looked into the verification metrics, followed by the verification results. Our verification showed the absence of a potential deadlock in each test set.

The second sub-division described the check for livelock in each test set. Under check for livelock, we looked into the verification metrics, followed by the verification results. Our verification showed the presence of livelock in four out of the eight sub-models. We then described the detected livelocks with two examples, one from test 6 and the other from test set 8. The reported livelocks require further analysis by the software designer to decide whether they are real or spurious livelocks, which will be future work.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	#	**Ada task name and location**		**Execution sequence**										
4		Task T3 T3_T4.adb:387	SelectStmt T3_T4.adb:392	AcceptStmt T3_T4.adb:393 **accept Wait**										T3 is waiting to **accept Wait**
5	2	Task T4 T3_T4.adb:461	SelectStmt T3_T4.adb:467	AcceptStmtWith Stmts T3_T4.adb:493	IfStmt T3_T4.adb:495	SelectStmt T3_T4.adb:467 because **L1 = False OR S1 != Idle** at IfStmt T3_T4.adb:495								But T4 is executing in an infinite loop that does not send **accept Wait**
6		Task T4 T3_T4.adb:461	SelectStmt T3_T4.adb:467	AcceptStmtWith Stmts T3_T4.adb:506	Call procedure **Trigger** at T3_T4.adb:511	CaseStmt T3_T4.adb:428	CaseStmtAlter native T3_T4.adb:429	IfStmt T3_T4.adb:430 **L1 =False OR (Signal! = Continuous)**	end **Trigger** (T3_T4.adb:459)	loop back to SelectStmt T3_T4.adb:467				
7		Task T4 T3_T4.adb:461	SelectStmt T3_T4.adb:467	AcceptStmtWith Stmts T3_T4.adb:506	Call procedure **Trigger** at T3_T4.adb:511	CaseStmt T3_T4.adb:428	CaseStmtAlter native T3_T4.adb:429	IfStmt T3_T4.adb:430 **L1=True AND (Signal! = Continuous)**	Call procedure **Search** at T3_T4.adb:437	IfStmt T3_T4.adb:378 L2= True OR S1!=Idle	end **Search** T3_T4.adb: 385	end **Trigger** T3_T4.adb: 459	loop back to SelectStmt T3_T4.adb: 467	

Figure 7.10: Using the toolset's traceability matrix to trace path leading to livelock in Ada code

Next, Section 7.3.2 described how our verification process minimizes the generation of spurious counterexamples. We have seen some refinements performed (manually) on the model. Minimizing spurious counterexamples and doing it automatically requires further investigation, which will be future work.

Finally, in Section 7.3.3 we illustrated with an example as to how an Ada developer can use SPIN's counterexample trace along with our toolset's traceability matrix to identify the specific condition(s) in the Ada code leading to concurrency problem(s).

7.4 Conclusions from Model Verification

This section presents the major conclusions from model verification.

1. The property *absence of deadlock* is verified on the eight sub-models. No deadlocks were reported. This does not mean the code is free of deadlock, as we have considered only a subset of syncs or 52 tasks (as seen from Section 6.6.1). We can only rule-out deadlocks in the subset considered. As and when the limitations of the extraction code are overcome, the number of tasks increases and it is essential to re-verify the updated model for deadlocks.
2. The property *absence of livelock* is verified on the eight sub-models. Livelocks were reported in four out of the eight sub-models. By using the SPIN's counterexamples and the toolset's traceability matrix, it is possible to trace detected violations back to the Ada program. The question as to whether these counterexample traces are real/spurious requires further analysis by the Ada software designer. The analysis could be, for example, case studies with the software designer, which will be part of future work.

   Furthermore, the detected livelocks are not the only livelocks in the code, as we have considered only a subset of syncs (52 tasks). We can only say the reported livelocks can be considered for the said subset. As and when the limitations of the extraction code are overcome, the number of tasks increases and it is essential to re-verify the updated model for livelocks.

7.5 Chapter Summary

This chapter described Stage-3 of the work approach shown in Figure 1.1 to verify the generated Promela model.

We started the chapter by describing the advanced features of SPIN in Section 7.1. The knowledge of the advanced features was essential to complete exhaustive verification of the model and improve verification performance. We have seen the non-algorithmic and algorithmic techniques supported by SPIN to reduce the search complexity under advanced features. Subsequently, we have seen various tuning options supported by SPIN to improve the performance of verification.

After gaining knowledge about a variety of features supported by SPIN, the next step is to have a roadmap that can guide us to complete the verification process. Section 7.2 illustrated the verification roadmap as a systematic approach to verify our model to achieve the best possible verification metrics and verification results.

After having the roadmap, the following Section 7.3 described the verification journey. The section used the divide-and-conquer technique (compositional verification) to create a set of sub-models from the collapsed model. The technique showed that it is possible to perform

an exhaustive search even for large models with limited physical memory. We verified two properties on the model under this technique: (a) absence of deadlock and (b) absence of livelock.

For each verified property, we presented the verification metrics and verification results. The check for deadlock did not report a potential deadlock scenario in any of the sub-models. The check for livelock reported execution paths leading to potential livelock in four out of the eight sub-models. The detected livelocks were illustrated with Ada/Promela snippets along with control-flow graphs. The reported livelocks require further analysis by a software designer to decide whether they are real or spurious, and this will be future work.

We have also seen the refinements made in the model to minimize spurious counterexamples. At the end of the section, we illustrated an example of how an Ada developer can find the specific condition(s) in the Ada code leading to a concurrency problem by using SPIN's counterexample and the toolset's traceability matrix.

Overall, this chapter concludes our verification journey. The next chapter provides the conclusions about our work and illustrates the steps required for future work.

8 CONCLUSIONS AND FUTURE WORK

This chapter concludes this thesis. The chapter is organized as follows. First, Section 8.1 summarizes the work. Second, in Section 8.2 we will revisit the goals set in Chapter 1, and evaluate these goals. Finally, in Section 8.3 we will list some possible future work.

8.1 Summary

We started this thesis with the problem statement in Chapter 1, set five goals, and illustrated the 3-stage approach using Figure 1.1.

We then introduced the Ada programming language in Chapter 2 and focused mainly on the language's concurrency features. We also looked into the different concurrency problems in Ada.

After familiarizing with Ada, we introduced model checking and its challenges in Chapter 3. We focused specifically on the SPIN model checker and its Promela modeling language.

Next, we surveyed the related work on formal verification of Ada programs in Chapter 4. The lessons learned and the limitations of each work were illustrated.

In Chapter 5, we used the knowledge gained on Ada and Promela from earlier chapters to define the correct Promela construct for each Ada construct. This is to ensure that the Promela model correctly simulates the behavior of the Ada program.

In Chapter 6, we used the knowledge gained on behavior mapping from Ada to Promela and developed a Python toolset to generate the Promela model of the Ada software automatically.

We then verified the generated model for concurrency problems in Chapter 7. The reported problems were traced back to the Ada code.

The important contributions from this work are:

1. Mapping the Ada constructs to the appropriate Promela constructs
2. Development of a toolset that automatically generates Promela models
3. Verify the generated model and trace detected violations back to the Ada code

In the next sections, we will summarize these contributions.

8.1.1 Behavior Mapping

The behavior equivalence between different Ada constructs and Promela constructs is described in Chapter 5.

The Ada concurrency primitives - *task*, *entry call*, *accept*, *selective accept* and *delay* are modeled in Promela. The mapping is essential to model Ada's rendezvous and event-based synchronizations in Promela.

The Ada conditional primitives - *if/else*, *or else/and then*, *case* and *loop* are modeled in Promela. The mapping is essential to ensure the control-flow from the Ada code is preserved in the Promela model.

8.1.2 Model Generation

The toolset developed for the automatic generation of the Promela model from the Ada code is described in Chapter 6. The three tools of the toolset are listed below.

1. The toolset's translator generates the Promela model of the Ada code. The translator preserves the behavior of different Ada constructs along with the code's control-flow in the generated model.
2. The toolset's state-space reducer uses the convolution algorithm (see Algorithm 7 in Section 6.8) to collapse unpaired synchronizations in the model to reduce the state space.
3. The toolset's traceability generator creates a mapping from the generated Promela model to the Ada code. This is to help to trace detected violations by SPIN back to the Ada code.

8.1.3 Model Verification

The generated Promela model is verified for the absence of deadlock and absence of livelock. The verification reported the presence of livelock in four cases. The detected livelocks were traced back to the Ada code using the traceability matrix. The verification did not report a deadlock. Spurious counterexamples were removed (manually) to some extent. A technique is proposed in Section 8.3 to automatically remove the paths in the model leading to spurious traces.

8.2 Evaluation

In Chapter 1, we listed five goals of this thesis. We will now revisit these goals and evaluate each one of them.

1. *Automatically generate a formal model of a software system implemented in Ada*

   Our toolset's translator, implemented in Python, uses an intermediate XML as input. The XML has the concurrency constructs of the Ada code along with the code's control flow. The translator generates a Promela model of the Ada program from the XML. In general, for a given Ada program, if its concurrency constructs and control flow are extracted into an XML, then the translator can generate a Promela model of the program.

   The toolset's Python code underwent 19 revisions before its final release. The algorithms were reviewed, validated using several tests, and fixed for bugs. A standard software development life-cycle process was followed during the development of the toolset. The process includes (a) requirement definition, (b) development of algorithms, (c) code development, (d) testing compliance of each code iteration to the algorithms, (e) bug-tracking, and (f) code version control. Necessary artifacts are maintained as proof for the process followed.

Every Ada concurrency/conditional construct and its Promela counterpart is tested sufficiently to ensure that the generated model imitates the modeled code. In other words, the toolset's code is validated using empirical analysis.

The limitations to overcome to make the toolset industry-ready are listed below.

(a) In Section 6.6.1 we have seen the limitations of the extraction code that led to having unpaired syncs in the XML. In this sense, the toolset is validated only on a limited number of test sets. Future improvements of the extraction code may lead to having larger test sets that can quickly run into state space problems. A thorough validation followed by update(s) of the toolset is required in such cases.

(b) A more concrete validation of the toolset is required to make it industry-ready. This means, besides empirical analysis, the toolset must be validated through formal reasoning. Examples of formal reasoning proof by induction, proof by contradiction, proof by exhaustion, etc. Furthermore, the toolset's usability by the software developers needs to be assessed.

2. *Automatically apply state space reduction techniques on the generated model*

Our toolset's state space reducer uses a convolution algorithm that applies a program slicing technique to collapse unpaired synchronizations in the model. The collapsing technique reduced the size of the model to 25% of the size of the initially generated model.

The large reduction in state space through program slicing was possible because of the unpaired syncs in the XML. Section 6.6.1 described the limitations of the extraction code that led to the unpaired syncs. In the future, if the said limitations are overcome, the number of paired syncs in the XML will be significantly large in number compared to the number of unpaired syncs. This could mean that the number of interacting processes in each test set will be higher. In such a case, applying the divide-and-conquer technique may still lead to state explosion problems. Therefore, further investigation is required in such tricky cases before using the divide-and-conquer technique.

3. *Automatically apply model checking to the generated model to prove the presence/absence of concurrency problems in the code's control and co-ordination flow*

Software model checking is not only about having a scalable model that faithfully represents the modeled software. It is also about making the best possible use of the different features of the model checking tool, which comes with experience.

Our work followed a systematic verification roadmap to use the SPIN model checker to verify the generated model. The verification proved the presence of livelocks in the Ada program. No deadlocks were reported. The reported livelocks require further analysis by the software designer (or case studies with the software designer) to decide whether they are real or spurious. This is a yet-to-do task, which will be a part of future work.

Currently, the toolset does not support the modeling of shared memory from the Ada code, which is a limitation to verify other concurrency problems like race conditions.

4. *Automatically generate traceability from model to code*

The toolset's traceability generator establishes a mapping between each line of the Promela model and the corresponding Ada code. Furthermore, the traceability matrix has current and next state transitions in the model that can be traced to the corresponding code's

control flow. Therefore, it is possible to trace detected violations back to the Ada code using the traceability matrix.

5. *Evaluate the level of model checking knowledge required for a software developer to verify his/her software using our work*

   A preliminary version of the toolset's user guide is available in Appendix A. The user guide shows that by following eight simple steps, an Ada developer/tester could use our toolset to verify an Ada program for concurrency problems. The steps are defined such that details on model-checking are kept abstract. In other words, the user guide shows that it is not necessary to have model-checking knowledge to use the toolset. The user guide, however, is only a preliminary version that requires further evaluation in the future.

In summary of the evaluation, we can say that our toolset is the first prototype that attempts to verify concurrent industrial software fully automatically using model checking. The toolset performs satisfactorily when it comes to generating a formal model and traceability matrix. On the other hand, the state space reduction technique that worked well for our approach may not be easy to adopt when the limitations of the extraction code are overcome. In such a case, the toolset requires re-evaluation and further extension. Furthermore, the toolset's algorithms should be validated using formal reasoning, and the toolset's user-guide requires further evaluation for the toolset to be industry-ready.

In the next section, we will point out some more areas of improvement for the toolset and the verification approach.

8.3 Future Work

This section provides directions for future work. We will first discuss the future work on the extraction code. Next, we will discuss the possible improvements in our modeling approach and the verification process.

8.3.1 Extraction Code

Section 6.6.1 on unpaired synchronizations introduced the *call-depth* parameter used in the extraction code. For our work, the search depth level is set to five. There may be paired synchronizations at a larger search depth. However, an increased depth is observed to consume more time to traverse the Ada code's AST and extract the concurrency constructs. Future work in this direction is to improve the extraction code's performance such that larger depth-search can be done without consuming more time. Subsequently, the translator shall be used to generate the Promela model from an updated XML as a result of the improved extraction code.

Section 6.6.1 on unpaired synchronizations also mentioned that the extraction code looks only for concurrency constructs in the Ada task body and not at the thread-level. There may be paired synchronizations by performing a thread-level analysis of the Ada code. Future work in this direction is to extend the extraction code to perform a thread-level analysis of the Ada code for concurrency constructs, besides the task-level analysis.

A remark about the toolset's code is worth mentioning at this stage. The future extension of the extraction code may increase the XML size by (a) retrieving more concurrency constructs and (b) retrieving longer control-flow leading to the concurrency constructs. This means the toolset's Python translator may need to be extended to support additional concurrency constructs (if any) and support additional control-flow constructs. The good news here is that the translator is developed using a modular object-oriented approach. That is, each feature of the translator is

implemented using (independent) separate class modules (as seen in Section 6.4). This means the translator can be extended with minimal changes to the existing Python code.

8.3.2 Modeling and Verification

In this section, we will describe the possible improvements in our modeling approach and the verification process.

8.3.2.1 Pruning the State Transitions

Section 6.6.3 described the collapsing technique to reduce the model's state space. A closer investigation of the collapsed model revealed a technique that can further reduce the model's state space. The technique involves *pruning the state transitions from a different set of source states to the same set of destination states*. Below we will briefly introduce the technique and mention its effect on the model's current state space. A similar technique has been used in [DL11], in which the authors call it as *symmetry reduction*.

A pizza delivery analogy to understand the pruning technique: Consider a pizza delivery shop S with two employees E1 and E2. The employees are responsible for delivering the pizza orders at the delivery locations. Imagine that the shop gets three orders, O1, O2, and O3, from three different delivery locations, L1, L2, and L3. The goal here is to deliver the correct orders at the correct locations. So, even though both E1 and E2 can complete the delivery, only one of them will be assigned the delivery task.

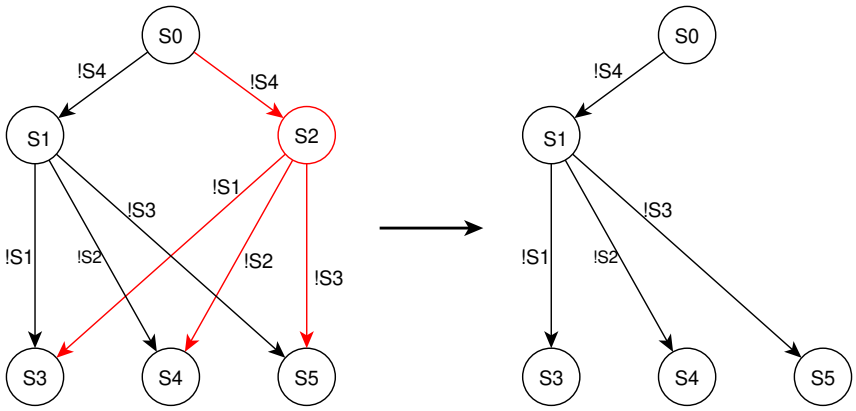


Figure 8.1: Pruning the state transitions

Now let us apply this analogy to the NDFA on the left-side of Figure 8.1. State S0 is the pizza delivery shop. States S1 and S2 are the two employees. The signals !S1, !S2, and !S3 correspond to the three orders O1, O2, and O3. States S3, S4, and S5 are the delivery locations. To achieve the goal, it is sufficient to retain either state S1 or state S2. The NDFA on the right side of the figure shows that the state transitions are pruned to retain only state S1 and remove state S2. Two rules for the pruning are (a) the set of destination states should be the same for a given set of source states and (b) the source states should send/receive the same signals during their next-state transition to their common destination states.

The pruning technique has been applied on test set 8 (see set 8 metrics in Table 7.4). It is observed that the number of state transitions reduced to 1.90E+08 compared to the 2.04E+08 transitions before pruning; this is a 6% reduction in the number of transitions.

Future work in this direction is to extend the translator to apply the pruning technique automatically on the model. Consequently, the algorithm of the traceability generator shall be updated.

8.3.2.2 Property- Absence of Race Condition

Our verification journey verified the two properties (a) absence of deadlock and (b) absence of livelock. Besides these, there are other properties to be verified, particularly the race condition problem discussed in Section 2.4.3. The extraction code already retrieves the lock and unlock mutexes used for shared data in the Ada code (not part of this work). The next step is to extend the translator to have the mutex behavior from the Ada code in the Promela model and verify the absence of race conditions.

The verification of race conditions may require user-defined properties (like LTL specifications, assertions, etc.). Therefore, the property definitions and defining them automatically requires further investigation. The work in [DPC98] provides some directions in automatically determining properties for the model.

8.3.2.3 Automatically Minimize Spurious Traces

In Section 7.3.2 we described the steps taken to minimize the spurious traces by manually removing the paths in the model that cannot be taken during the code's run-time.

A technique to automatically remove such paths in the model is by enabling the translator to *interpret the actual conditions from the code*, besides the code control flow. For example, consider the Listing 7.9. While generating a model for this code, the translator shall interpret that `cond1` is a boolean, and hence either one of the lines, line-5 or line-8, must be executed. This way, it is possible to avoid the loop-back transition at line-6 of the model in Listing 7.10, which led to a spurious counterexample showing a livelock in the model. For cases like these, the extraction code already extracts the Ada code conditions (not part of this work). For cases like these, the next step is to extend the translator to interpret the code's actual conditions to avoid having transitions in the model that are not possible during the code's run-time.

Note that a condition in one task could change several times due to the behavior of other task(s). Therefore, it may be a complex task to automatically interpret all the code conditions and eliminate all transitions in the model that cannot happen in the code-control flow. This task requires further investigation and extension of the toolset's algorithms.

8.3.2.4 Optimize Step-4 of the 8-Step Guideline to the Ada Developer

In Appendix A, Section A.2 describes eight steps to be followed by an Ada developer to use our work to verify his/her code. Among the steps listed, step-4 mentions retaining only the independent set of interacting processes and commenting out the rest while verifying each sub-model. It is possible to optimize the step further.

The toolset already generates a synchronization list (sync list) with an independent set of interacting processes. The next step is to extend the toolset such that it can use the sync list to automatically disable the processes in the model that are not part of a set. This way, the Ada developer can directly perform the sub-model verification by avoiding the commenting process.

8.3.2.5 Swarm Verification

In Section 7.5 we have seen that even if all the SPIN's compression algorithms fail, the bitstate verification is very likely to complete an approximate verification of the model. Further to bitstate hashing, SPIN supports an option to perform a series of bitstate searches in parallel. The option is called *swarm verification*.

Swarm verification [HJG10] performs several small runs in parallel to improve the model coverage for very large verification problems. The swarm technique effectively leverages large computing resources (like the 2TB Linux machine in our work) to verify very large models.

A first-level investigation of the effect of swarm verification on our model showed a slight improvement in the hash-factor. Future work in this direction is to investigate the SPIN's swarm technique further.

BIBLIOGRAPHY

[AD03] Peter Amey and Brian Dobbing. “High integrity ravenscar”. In: *International Conference on Reliable Software Technologies*. Springer. 2003, pp. 68–79.

[Ada20a] Ada Resource Association. *Ada Comparison Chart*. `https://www.adaic.org/advantages/ada-comp-chart/`. last updated 2020.

[Ada20b] AdaCore. `https://learn.adacore.com/courses/intro-to-ada/index.html`. last updated 2020.

[Ada20c] AdaCore. *libadalang*. `https://github.com/AdaCore/libadalang`. last updated 2020.

[Arc97] Arcadis. *IRIS-Ada*. `https://www.ics.uci.edu/~arcadia/arcadia_software.html`. last updated 1997.

[Aun+21] Moe Nandi Aung et al. “A Divide and Conquer Approach to Eventual Model Checking”. In: *Mathematics* 9.4 (2021), p. 368.

[Bar03] John Barnes. *High integrity software: the spark approach to safety and security: sample chapters*. Pearson Education, 2003.

[Bar08] John Barnes. *Safe and secure software: An invitation to Ada 2005*. AdaCore, 2008.

[BDV04] Alan Burns, Brian Dobbing, and Tullio Vardanega. “Guide for the use of the Ada Ravenscar Profile in high integrity systems”. In: *ACM SIGAda Ada Letters* 24.2 (2004), pp. 1–74.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[Bošnacki. “Enhancing state space reduction techniques for model checking”. PhD thesis. PhD thesis, Eindhoven University of Technology, 2001.

[Bri+10] Maria AS Brito et al. “Concurrent software testing: a systematic review.” In: (2010).

[CAU88] Jingde Cheng, K Araki, and Kazuo Ushijima. “Tasking communication deadlocks in concurrent Ada programs”. In: *ACM SIGAda Ada Letters* 8.5 (1988), pp. 61–70.

[CGL94] Edmund M Clarke, Orna Grumberg, and David E Long. “Model checking and abstraction”. In: *ACM transactions on Programming Languages and Systems (TOPLAS)* 16.5 (1994), pp. 1512–1542.

[Che90] Jingde Cheng. “A classification of tasking deadlocks”. In: *ACM SIGAda Ada Letters* 10.5 (1990), pp. 110–127.

[Cla+18a] Edmund M Clarke Jr et al. *Model checking*. MIT press, 2018.

[Cla+18b] Edmund M Clarke et al. *Handbook of model checking*. Vol. 10. Springer, 2018.

[Cor93] James C Corbett. *An SEDL translator*. Tech. rep. Citeseer, 1993.

[DL11] Alexandre David and Kim G Larsen. “More features in UPPAAL”. In: *Deliverable No.: D5. 12 Title of Deliverable: Industrial Handbook* (2011), pp. 49–76.

[DPC98] Matthew B Dwyer, Corina S Pasareanu, and James C Corbett. *Translating Ada programs for model checking: A tutorial*. Tech. rep. Citeseer, 1998.

[Eva+03a] Sami Evangelista et al. “Quasar: a new tool for concurrent Ada programs analysis”. In: *International Conference on Reliable Software Technologies*. Springer. 2003, pp. 168–181.

[Eva+03b] Sami Evangelista et al. “Verifying linear time temporal logic properties of concurrent Ada programs with Quasar”. In: *Proceedings of the 2003 annual ACM SIGAda international conference on Ada: the engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies*. 2003, pp. 17–24.

[Eva05] Sami Evangelista. “High level petri nets analysis with Helena”. In: *International Conference on Application and Theory of Petri Nets*. Springer. 2005, pp. 455–464.

[FMP12] José Miguel Faria, Joao Martins, and Jorge Sousa Pinto. “An approach to model checking Ada programs”. In: *International Conference on Reliable Software Technologies*. Springer. 2012, pp. 105–118.

[HJG10] Gerard J Holzmann, Rajeev Joshi, and Alex Groce. “Swarm verification techniques”. In: *IEEE Transactions on Software Engineering* 37.6 (2010), pp. 845–857.

[HM03] Douglas J Howe and Stephen Michell. “An approach to formal verication of real time concurrent Ada programs”. In: *Proceedings of the 12th international workshop on Real-time Ada*. 2003, pp. 87–92.

[HMU01] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. “Introduction to automata theory, languages, and computation”. In: *Acm Sigact News* 32.1 (2001), pp. 60–65.

[Hol04] Gerard J Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004. ISBN: 978-0-321-22862-8.

[Hol80] Gerard J Holzmann. *Basic spin manual*. 1980.

[Hol88] Gerard J Holzmann. “An improved protocol reachability analysis technique”. In: *Software: Practice and Experience* 18.2 (1988), pp. 137–161.

[Hol97] Gerard J Holzmann. “State compression in SPIN: Recursive indexing and compression training runs”. In: *Proceedings of third international Spin workshop*. 1997.

[Hol98] Gerard J Holzmann. “An analysis of bitstate hashing”. In: *Formal methods in system design* 13.3 (1998), pp. 289–307.

[HP95] Gerard J Holzmann and Doron Peled. “An improvement in formal verification”. In: *Formal Description Techniques VII*. Springer, 1995, pp. 197–211.

[HP99] Gerard J Holzmann and Anuj Puri. “A minimized automaton representation of reachable states”. In: *International Journal on Software Tools for Technology Transfer* 2.3 (1999), pp. 270–278.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[HSH05] Alex Ho, Steven Smith, and Steven Hand. *On deadlock, livelock, and forward progress*. Tech. rep. University of Cambridge, Computer Laboratory, 2005.

[JEN] JENKINS. `http://burtleburtle.net/bob/hash/doobs.html`.

[Jen13] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*. Vol. 1. Springer Science & Business Media, 2013.

[LA99] Kristina Lundqvist and Lars Asplund. “A formal model of the Ada ravenscar tasking profile; delay until”. In: *ACM SIGAda Ada Letters* 19.3 (1999), pp. 15–21.

[Lev89] Gertrude Levine. “Controlling deadlock in Ada”. In: *ACM SIGAda Ada Letters* 9.4 (1989), pp. 87–91.

[Lib18] Libadalang User Manual. `https://docs.adacore.com/live/wave/libadalang/html/libadalang_ug/introduction.html#parsing-a-file`. last updated 2018.

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. “UPPAAL in a nutshell”. In: *International journal on software tools for technology transfer* 1.1-2 (1997), pp. 134–152.

[Mey92] Bertrand Meyer. “Applying ‘design by contract’”. In: *Computer* 25.10 (1992), pp. 40–51.

[OZ13] Kazuhiro Ogata and Min Zhang. “A Divide and Conquer Approach to Model Checking of Liveness Properties”. In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. IEEE. 2013, pp. 648–657.

[Sie02] Stephen F Siegel. *The INCA query language*. Tech. rep. Technical Report UM-CS-2002-18, Department of Computer Science, University of Massachusetts, 2002.

[SPA08] SPARK Team. “SPARK Examiner: The SPARK Ravenscar Profile”. In: (2008).

[SPI08] SPIN. `http://spinroot.com/spin/Man/progress.html`. last updated 2008.

[SPI11] SPIN. `http://spinroot.com/spin/Man/Pan.html`. last updated 2011.

[SPI16] SPINRCP. `http://lms.uni-mb.si/spinrcp/`. last updated 2016.

[SPI18a] SPIN. `http://spinroot.com/spin/Man/ltl.html`. last updated 2018.

[SPI18b] SPIN. `http://www.spinroot.com/spin/Man/README.html`. last updated 2018.

[Tai94] Kuo Chung Tai Kuo Chung Tai. “Definitions and detection of deadlock, livelock, and starvation in concurrent programs”. In: *1994 Internatonal Conference on Parallel Processing Vol. 2*. Vol. 2. IEEE. 1994, pp. 69–72.

[VDW12] Willem Visser, Matthew B Dwyer, and Michael Whalen. “The hidden models of model checking”. In: *Software & Systems Modeling* 11.4 (2012), pp. 541–555.

[Wei84] Mark Weiser. “Program slicing”. In: *IEEE Transactions on software engineering* 4 (1984), pp. 352–357.

[WL93] Pierre Wolper and Denis Leroy. “Reliable hashing without collision detection”. In: *International Conference on Computer Aided Verification*. Springer. 1993, pp. 59–70.

APPENDICES

A An Use Case of the Toolset

This section presents a use case of our toolset. For an Ada developer with little or no knowledge of model checking, the use case provides the steps to be followed to use our toolset to find concurrency problems in an Ada program.

Section A.1 lists the pre-requisite tools to be installed before using the toolset. The subsequent Section A.2 lists the steps to be followed by the developer to use the toolset. Note that the latter section's user guide is only a preliminary version and requires further evaluation to make it a baseline version.

A.1 Pre-Requisites

The Ada developer shall complete the below steps before using the toolset.

- Download and install the latest Python version, depending on the platform used (Windows/Linux/Mac). Add Python to the system environment variables (Windows PC) to execute Python files using a command line.
- Download and install the SPIN model checker from [SPI18b] depending on the platform used (Windows/Linux/Mac). Add SPIN to the system environment variables (if Windows PC is used) to execute Promela files using a command line. Alternatively, an interactive SPIN IDE (Eclipse-based) called SpinRCP is available for download at [SPI16].

A.2 The Toolset's User Guide

This section serves as a user manual for an Ada developer to use the toolset. The section is divided into two parts. The first part concisely presents the steps to use the toolset. The second part supplements the first part by helping the user to (a) set the correct SPIN verification parameters and (b) identify different errors from SPIN's verifier, the cause for each of them, and the fix for each of the errors.

A.2.1 Part-1 of the User Guide – Steps to Use the Toolset

This section lists eight simple steps to be followed by an Ada developer to use the toolset. Some of the steps require additional information, supplemented by part-2 of the user guide in the subsequent section.

1. Step-1: Set the path of the Ada source code in the extraction code's main file, and run the extraction code. Using the Ada source code, the extraction code generates an XML containing the source code's concurrency constructs along with the control-flow. Let the name of the generated XML be, say, `inputXML.xml`.

2. Step-2: Create a new folder, and place the generated XML in this folder. Next, place the toolset's Python translator files in the same folder. There are two translator files, namely, `xml_to_promela_translator_collapsed_deadlock.py` and `xml_to_promela_translator_collapsed_livelock.py`

3. Step-3: From the command prompt, execute:

   (a) `python xml_to_promela_translator_collapsed_deadlock.py -m inputXML.xml outputPML.pml` to generate a Promela model to check for the absence of deadlock, OR

   (b) `python xml_to_promela_translator_collapsed_livelock.py -p -m inputXML .xml outputPML.pml` to generate a Promela model to check for the absence of livelock.

   Here, `*.py` is the toolset's translator's Python code, the `-m` is the option to generate the traceability matrix, the `-p` is the option to automatically add progress labels in the model to check for livelock, and the `OutputPML.pml` is the generated Promela file.

   The translator generates three output files:
   (a) a Promela file (`OutputPML.pml`) that can be used either to check for deadlock or livelock
   (b) a traceability matrix from Promela model to the Ada source code (`Mapping.xls`), and
   (c) a synchronization list (`SynchronizationList.csv`). The synchronization list shows the independent set of interacting processes.

   Tip: A batch file can be created that combines steps 1, 2, and 3.

4. Step-4: Open the generated Promela file, and scroll down to the bottom of the file containing the list of different processes (search for `run` command in the file). The list of processes corresponds to the Ada tasks from the source code.

   We know the independent set of interacting processes from the synchronization list (step-3). Use this data and run only those processes that are communicating as a set. For example, assume that the generated Promela model has ten processes P1, P2, ... P10. Out of the ten, say, the set of processes, P1, P3, P6, and P8 interact with each other. Except for these four processes, comment all the other processes (use `/* */` for commenting a block of lines). Follow a similar procedure for other independent sets of processes.

   Note: Our intention is to use one model (single `*.pml` file) and one traceability matrix. This is why we comment the interacting processes in the same file to avoid creating multiple `*.pml` files that require multiple traceability matrices.

5. Step-5: Open the command prompt and compile the edited Promela model from step-4. Use the compilation command provided in Section A.2.2 to compile the model. The Promela model is first converted to a C-file (`pan.c`) and will be compiled to produce an executable (`pan.exe`).

   Alternatively, the SpinRCP IDE can be used to compile the Promela model.

6. Step-6: In this step, the user shall decide on two aspects: (a) the type of concurrency problem (deadlock or livelock) to be verified, and (b) the available memory and time constraint.

   Depending on the deadlock check or livelock check, set the appropriate verification options (see Section A.2.2). Depending on memory and time restrictions, choose the appropriate verification type (see Section A.2.2).

After deciding on the problem to be verified and the memory and time constraints, use the run command provided in Section A.2.2 to run the produced executable (`pan.exe`).

Alternatively, the SpinRCP IDE can be used to execute the Promela model.

If there is a violation in the model, then at the end of step-6, counterexample files (`*.pml.trail`) are created.

7. Step-7: If a counterexample file is generated, run the generated file using the simulation command provided in Section A.2.2. The simulation shows the model execution path leading to the violation.

   Alternatively, the SpinRCP IDE can be used to run the simulation.

8. Step-8: Open the generated traceability matrix from step-3. On one side, there is the execution path leading to the violation (from step-7), and on the other side is the traceability matrix. The traceability matrix has the mapping from the model's execution path to the corresponding Ada code's execution path. Check if it is possible to take this execution path during the Ada code's run-time. If yes, then the code has violation(s) in this path. Take necessary steps (e.g., update the code) to address this problematic path in the code. If no, then ignore the counterexample trace and check the next counterexample (if any).

   Repeat step-4 to step-8 for the next set of connected processes.

A.2.2 Part-2 of the User Guide – Supplement to Part-1

This section helps the user to (a) use the SPIN compilation commands, (b) set the correct SPIN verification parameters, (c) choose the appropriate verification type, (d) use the SPIN simulation commands, and (e) understand and fix errors that occur during verification. In other words, this section supplements steps 5, 6, and 7 of part-1 of the user guide. Although the commands described are specific to Windows PC, they also work in Unix/Linux machines. The user shall ensure that correct paths to the SPIN executable and C-compiler are used before running the commands.

1. SPIN's compilation command: `C:\..\bin\spin.exe -a outputPML.pml`, where

   - `bin` is the folder containing the spin executable (see Section A.1)
   - `-a` is the command to generate a verifier in `pan.c` and
   - `outputPML.pml` is the Promela model to be compiled.

   As outcome of the compilation, SPIN returns the message `No Syntax Error` and generates a C-code of the model, `pan.c`.

2. (a) To check for the absence of deadlock in the model, execute the command `C:\..\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=1024 -DSAFETY -DNOCLAIM -DXUSAFE -O2 -w -o pan pan.c` to generate an executable of the C-code, i.e., to generate `pan.exe`. Here,

   - `x86_64-w64-mingw32-gcc.exe` is a 64-bit gcc compiler. Note that the compilation may fail if a 32-bit compiler is used
   - `-DMEMLIM` is the memory bound set for compilation, in this case it is 1024MB
   - `-DSAFETY` is the option to check for safety properties, which checks for deadlock by default

Note that other options like `-DNOCLAIM`, `-DXUSAFE` etc. are not explained to avoid too many details, and the details are not necessary to use the toolset. Nevertheless, the user shall refer [SPI08] for details on other options.

(b) To check for the absence of livelock in the model, execute the command `C:\cygwin64\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=1024 -DNP -DNOCLAIM -DXUSAFE -DNOREDUCE -O2 -w -o pan pan.c`. Here,

- `-DNP` is the option to check for livelock

Note that other options like `-DNOREDUCE` are not explained to avoid too many details, and the details are not necessary to use the toolset. Nevertheless, the user shall refer [SPI08] for details on other options.

3. (a) Command to run `pan.exe` if the model is verified for deadlock: `.\pan -m10000 -c0 -e`. Here,

- `-m10000` is the bound to the search depth, which is 10,000 steps in this case
- `-c0 -e` instructs the verifier to collect all the violations in the model, instead of stopping at the first violation

(b) Command to run `pan.exe` if the model is verified for livelock: `.\pan -m10000 -l -f -c0 -e`. Here,

- `-l` is the option to search for non-progress cycles (infinite execution sequences) in the model
- `-f` enables weak-fairness such that all processes in the model are visited at least once

4. This step lists the common error messages and ways to fix them. At the end of step-3, the two verification errors that can occur are:

- The error `pan: reached -DMEMLIM bound` indicates that the verifier has reached the set memory bound. To fix this error, increase the value gradually (say from 1024) up to the value, which is the maximum physical memory available in the machine until the error is no longer seen. If the error is seen even after setting the maximum value, proceed to step-5 to use a different verification type.
- The error `error: max search depth too small` indicates that the verifier has reached the set search depth bound. To fix this error, increase the value gradually (say from 10,000 steps) until the error is no longer seen.

5. This step helps to set the appropriate verification type. Despite setting the maximum available physical memory to `-DMEMLIM`, if the memory error is still seen, then a different verification type shall be used using the commands listed below. While using each command, the user shall increase memory bound and search depth bound until the corresponding errors are no longer seen. The user shall use the subsequent commands only if a command still shows a memory error despite setting the maximum value to `-DMEMLIM`.

(a) To verify the model for deadlock, using the following commands one-by-one in the order specified

- `C:\..\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=7168 -DSAFETY -DNOCLAIM -DXUSAFE -DCOLLAPSE -O2 -w -o pan pan.c`
- `C:\..\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=7168 -DSAFETY -DNOCLAIM -DXUSAFE -DMA=450 -O2 -w -o pan pan.c`

- `C:\..\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=7168 -DSAFETY -DNOCLAIM -DXUSAFE -DMA=450 -DCOLLAPSE -O2 -w -o pan pan.c`
- `C:\..\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=7168 -DSAFETY -DNOCLAIM -DXUSAFE -DHC4 -O2 -w -o pan pan.c`
- `C:\..\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=7168 -DSAFETY -DNOCLAIM -DXUSAFE -DBITSTATE -O2 -w -o pan pan.c`

(b) To verify the model for livelock, using the following commands one-by-one in the order specified.

- `C:\..\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=7168 -DNP -DNOCLAIM -DXUSAFE -DCOLLAPSE -DNOREDUCE -O2 -DNFAIR=9 -w -o pan pan.c`
- `C:\..\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=7168 -DNP -DNOCLAIM -DXUSAFE -DMA=450 -DNOREDUCE -O2 -DNFAIR=9 -w -o pan pan.c`
- `C:\..\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=7168 -DNP -DNOCLAIM -DXUSAFE -DHC4 -DNOREDUCE -O2 -DNFAIR=9 -w -o pan pan.c`
- `C:\..\bin\x86_64-w64-mingw32-gcc.exe -DMEMLIM=7168 -DNP -DNOCLAIM -DXUSAFE -DBITSTATE -DNOREDUCE -O2 -DNFAIR=9 -w -o pan pan.c`

(c) Only when using the `-DBITSTATE` option, observe the value of *hash factor* shown by the verifer. If the value of hash factor is less than 100, this means the model coverage is incomplete and therefore needs improvement. Use the command `./pan -m10000 -k35 -w20 -c1 -w33` to set different values for `-kN` and `-wN` and see for which values the hash factor improves.

(d) Only when verifying for livelock, the user may see another error `pan: error: too many processes -- current max is N procs (-DNFAIR=2) recompile with -DNFAIR=M`. To fix this error, increase the value of `-DNFAIR` until the error is no longer seen.

6. SPIN's simulation command: `C:\cygwin64\bin\spin.exe -X -p -s -r -v -g -l -k OutputPML.pml.trail -u10000 OutputPML.pml`. Here,
   - `OutputPML.pml.trail` is the name of the counterexample file containing the path to the detected violation in the model. This trail file is automatically generated by SPIN, if the model has violation(s).
   - `-u10000` indicates the number of simulation steps showing the path to the violation. The user shall increase or decrease the number of steps depending on the length and complexity of the counterexample trace.

   The user shall refer [SPI08] for details on other simulation options.

As a reminder, if the user prefers IDE over commands, then SpinRCP IDE can be used to verify the model using a few mouse clicks.