

Model Checking: Theory and Practice

Matthew B. Dwyer John Hatcliff Robby

August 4, 2004

Contents

1	Foundations	5
1.1	BIR	6
1.1.1	A simple BIR system	6
1.1.2	States and transitions	7
1.2	Interleavings, Schedules, and Computation Trees	12
1.3	Executing (Simulating) BIR Systems with Bogor	16
1.4	Verifying BIR Systems with Bogor	19

Chapter 1

Foundations

This chapter presents basic concepts associated with explicit-state model-checking – focusing on the primary data structures required for implementing a simple depth-first state-space exploration algorithm.

Our reasons for taking this approach are as follows.

- The depth-first search algorithm that will be presented forms the foundation of the more sophisticated strategies used in many popular model-checkers, especially software-oriented model-checkers. Thus, material in subsequent chapters will explain these more sophisticated strategies by making modifications and extensions of the basic algorithm presented here.
- Many readers that are new to model-checking are unable to predict even for simple systems the state-space size and thus the overall costs of model-checking. By understanding the basic algorithms and data structures, readers will be able to understand how concurrent computation gives rise to interleavings, and the relative number of associated interleavings.
- As stated earlier, model-checking is often effective when adapted to particular domains. Thus, researchers interested in developing and applying model-checking to their particular domain of interest need to understand enough of the internal details to be able to adapt confidently an existing model-checker.

Our linguistic vehicle for introducing model-checking concepts is a simple notation for describing concurrent systems called BIR. In its full form, BIR is a sophisticated language called the Bandera Intermediate Representation (BIR) that is used in the Bandera Tool Set [?] and the Bogor model-checker to represent models of concurrent object-oriented systems. These models may be coded directly in BIR or constructed automatically by translations from Java source code, Java byte code, or higher-level design notations. Because we will not need the full expressive power of BIR, we will restrict our initial system descriptions to a very simple subset of BIR.

The simple model of concurrent systems represented by BIR is rich enough to illustrate wide body of concepts associated with model-checking including state-space exploration, assertion and deadlock-checking, LTL checking, various forms of search optimization including partial-order reductions, etc. The simple structure of the BIR constructs we will use allows us to provide a clean presentation of basic issues, without being distracted by the more complicated features of full BIR. In later chapters, we will add more complicated linguistic features to BIR including dynamically created objects, references, functions/methods, etc. which are all useful for modeling modern software systems.

We begin by introducing the syntax and semantics of BIR (formal definitions can be found in appendices). We then present the notion of *computation trees* as a means of discussing and reasoning about the exploration of a concurrent system's various schedules and execution traces. This will lead us into a discussion of Bogor, and how it can *simulate* a concurrent system by exploring a path found in a system's computation tree, and how it can *verify* properties of a concurrent system by exploring all the paths in a system's computation tree. We will describe in detail simple representations for the main data structures needed to implement this exploration, and outline the depth-first exploration algorithm. We then present the first extensions to the algorithm which will implement dead-lock checking and a simple depth-bounded search. Finally, we conclude this chapter by applying Bogor to a number of simple concurrent systems.

1.1 BIR

1.1.1 A simple BIR system

Figure 1.1 presents the BIR code for a version of the classical dining philosophers system which we used in Chapter ?? to motivate issues in concurrent systems. We use this code to introduce the features of BIR.

The core of BIR is a *guarded assignment language*. Guarded assignments are a traditional way of expressing concurrent systems [?]. At its highest level of structure, a BIR system has two parts: (1) a passive part that declares the data layout of the system, and (2) an active part that declares the threads of control and transformations of the system.

Typically, the data declaration section will describe a bounded data space by bounding basic data types (e.g., integer values are bounded by subranges). BIR's *primitive types* include boolean, integer subranges, and enumerated types. In the example of Figure 1.1, the declarations of the two boolean variables `fork1` and `fork2` for the data declaration section.

Thread declarations are used to define independently executing transition systems that correspond to processes or threads of control. In Figure 1.1, the two thread declarations represent the independently executing actions associated with two different philosophers. Placing the qualifier `active` on a thread declaration indicates that the thread is to begin its execution immediately. If

```

system TwoDiningPhilosophers {
  boolean fork1;
  boolean fork2;

  active thread Philosopher1() {
    loc loc0: // take first fork
    when !fork1 do { fork1 := true; }
    goto loc1;

    loc loc1: // take second fork
    when !fork2 do { fork2 := true; }
    goto loc2;

    loc loc2: // put second fork
    do { fork2 := false; }
    goto loc3;

    loc loc3: // put first fork
    do { fork1 := false; }
    goto loc0;
  }

  active thread Philosopher2() {
    loc loc0: // take second fork
    when !fork2 do { fork2 := true; }
    goto loc1;

    loc loc1: // take first fork
    when !fork1 do { fork1 := true; }
    goto loc2;

    loc loc2: // put first fork
    do { fork1 := false; }
    goto loc3;

    loc loc3: // put second fork
    do { fork2 := false; }
    goto loc0;
  }
}

```

Figure 1.1: A BIR system modeling 2-dining-philosophers

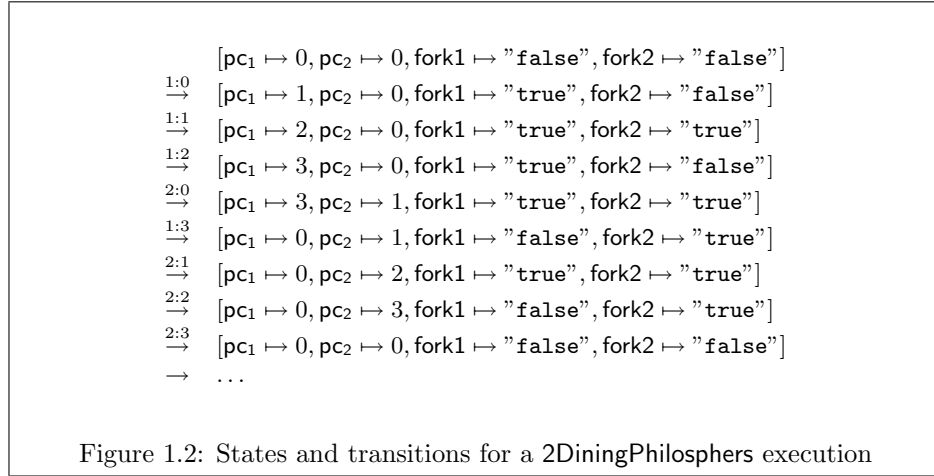
the active qualifier is omitted from a thread declaration, the thread must be started explicitly by another thread using the BIR construct `start`.

A thread body consists of a sequence of *locations*. When system execution begins, control in each thread begins at the *start location* – the first location in the thread’s location sequence. At any given time, a thread is at one of its locations, called the *current location* of the thread. The location of a thread is an abstraction of the *program counter* that would mark the next machine instruction to be executed in a lower-level representation of the thread, and we often refer to a thread’s “current location” as its “program counter”.

Each thread location is the source of one or more *guarded transformations*. Each transformation consists of a boolean *guard expression* followed by a sequence of *actions* ending in the target location indicating the source of the next transformation in the thread to be executed. The intuition is that each time a thread is scheduled to make a step in the system’s computation, the underlying scheduler selects for execution a transformation associated with the thread’s that has a guard expression that evaluates to *true*. The chosen transformation’s actions are executed in sequence, and the system’s data variables are updated (atomically) to produce the next execution state.

1.1.2 States and transitions

Let us now take this intuition about thread actions, and develop a more formal description of a system’s execution. An *execution* of a BIR system can be viewed as a sequence of atomic steps or *transitions* between system *states*. Intuitively, a BIR state holds all information that is necessary to carry on computation starting from a particular point in the system’s execution – that state holds



current location for each thread, along with the values of all the variables from the system's data declaration section.

Figure 1.2 presents the states and transitions of the beginning of one possible execution of the BIR system of Figure 1.1. The initial state of the system

$$[\text{pc}_1 \mapsto 0, \text{pc}_2 \mapsto 0, \text{fork1} \mapsto \text{"false"}, \text{fork2} \mapsto \text{"false"}]$$

shows the initial values of the program counters for both threads and the initial values of the system's variables. The notation $\text{pc}_1 \mapsto 0$ represents the fact that the program counter for the `Philosopher1` thread is set to the initial location `loc0`. The notation $\xrightarrow{t:l}$ represents the fact that thread t has executed a transition out of location l . Thus, the fourth transition in the trace of Figure 1.2 corresponds to `Philosopher2` executing its transition leading out of location `loc0`.

Given this intuition for states and transitions, we now use the notion of *state transition system* [?] to define formally the behavior of a BIR system. A state transition system Σ is a quadruple (S, T, S_0, L) with a set of states S , a set of transitions T such that for each $\alpha \in T, \alpha \subseteq S \times S$, a set of initial states S_0 , and a labeling function L that maps a state s to a set of primitive propositions that are true at s (we will not use the labeling function until Chapter ?? where property languages are addressed).

In the state transition system $\Sigma_{2\text{DP}}$ for Figure 1.1, there are 64 states in S – one for each possible combination of thread locations (4 different locations for each thread, yielding 16 different pairs of locations) and variable values (2 different values for each boolean variable, yielding 4 different pairs). The initial state of the system S_0 , has all threads in their initial locations and all variables have the default value of their type (the default value of boolean variables is "false"). Note that not all of the states of S are *reachable* in $\Sigma_{2\text{DP}}$. For example, it is impossible to start from the initial state and reach a state such as

$$[\text{pc}_1 \mapsto 2, \text{pc}_2 \mapsto 0, \text{fork1} \mapsto \text{"false"}, \text{fork2} \mapsto \text{"false"}]$$

where thread `Philosopher1` is at location `loc2` and where neither fork is held.

To save space, as we continue our discussion of states in the dining philosopher example, we will abbreviate the state notation by dropping the names of state variables. Thus, the state immediately above will be denoted as follows:

$$[2, 0, \text{"false"}, \text{"false"}]$$

For an arbitrary BIR system, in the corresponding state transition system there will be a transition α in T for each guarded transformation in the BIR code. For example, in Σ_{2DP} , there are exactly eight transitions in T – one for each of the four transformations in each of the two threads.

A transition can be thought of as a *state transformer* that, given a state as input, produces one or more states as output. A transition is *deterministic* if for every state s there is at most one state s' such that $\alpha(s, s')$. When α is deterministic (i.e., when α can be viewed as a function from states to states instead of the more general relation between states as introduced in the definition of transition systems above), we write $s' = \alpha(s)$ instead of $\alpha(s, s')$. In our technical discussions, we hardly ever consider non-deterministic transitions, and special notice will be given when we do so. Note that all the transitions of the dining philosopher system of Figure 1.1 are deterministic.

For some more intuition on transitions, let $\alpha_{1:1}$ represent the transition associated with location `loc1` in `Philosopher1`, and consider the functional behavior of $\alpha_{1:1}$ on some example states.

$$\begin{aligned} \alpha_{1:1}([1, 0, \text{"true"}, \text{"false"}]) &= [2, 0, \text{"true"}, \text{"true"}] \\ \alpha_{1:1}([1, 2, \text{"false"}, \text{"false"}]) &= [2, 2, \text{"false"}, \text{"true"}] \end{aligned}$$

Note that the state supplied as the argument in the second application above is actually unreachable, but that does not affect the definition of $\alpha_{1:1}$. A more interesting situation is to consider is when $\alpha_{1:1}$ is *undefined* – that is, when it does not produce an output for the supplied argument. Consider the following states ($\alpha_{1:1}$ is undefined on all of them).

$$\begin{aligned} &[1, 1, \text{"true"}, \text{"true"}] \\ &[0, 0, \text{"false"}, \text{"false"}] \\ &[1, 2, \text{"false"}, \text{"true"}] \end{aligned}$$

$\alpha_{1:1}$ is undefined on the first state because the guard of $\alpha_{1:1}$ requires that `fork2` be not held (`"false"`) before the transition can “fire”. $\alpha_{1:1}$ is undefined on the second state because there is an implicit guard associated with the transition that requires `Philosopher1`’s program counter to be at location `loc1` before the transition can fire. Even though the third state above is unreachable in the system, $\alpha_{1:1}$ is undefined on it for reasons similar to the first state.

In general, for a transition $\alpha \in T$, we say that α is *enabled* in a state s if it is defined on s , i.e., there is a state s' such that $\alpha(s, s')$ holds. Otherwise, α is disabled in s . The set of transitions enabled in s is $enabled(s)$, and the set of transitions enabled in s belonging to thread t is $enabled(s, t)$.

Here are some examples.

$$\begin{aligned}
 \text{enabled}([0, 0, \text{"false"}, \text{"false"}]) &= \{\alpha_{1:0}, \alpha_{2:0}\} \\
 \text{enabled}([1, 0, \text{"true"}, \text{"false"}]) &= \{\alpha_{1:1}, \alpha_{2:0}\} \\
 \text{enabled}([2, 0, \text{"true"}, \text{"true"}]) &= \{\alpha_{1:2}\} \\
 \text{enabled}([1, 1, \text{"true"}, \text{"true"}]) &= \emptyset \\
 \text{enabled}([1, 0, \text{"true"}, \text{"false"}], 1) &= \{\alpha_{1:1}\} \\
 \text{enabled}([2, 0, \text{"true"}, \text{"true"}], 2) &= \emptyset
 \end{aligned}$$

We denote the program counter of a thread t in a state s by $pc(s, t)$. We write $\text{current}(s, t)$ for the set of transitions associated the current control point $pc(s, t)$ of thread t (this set will include $\text{enabled}(s, t)$ as well as any transitions of t at $pc(s, t)$ that may be disabled). Also, $\text{current}(s)$ represents the union of current transitions at s for all active threads.

Here are some examples.

$$\begin{aligned}
 pc([0, 0, \text{"false"}, \text{"false"}], 1) &= 0 \\
 pc([0, 2, \text{"true"}, \text{"true"}], 2) &= 2 \\
 \text{current}([2, 0, \text{"true"}, \text{"true"}]) &= \{\alpha_{1:2}, \alpha_{2:0}\}
 \end{aligned}$$

Note that even though we will usually not consider non-deterministic transitions, this does *not* eliminate non-determinism in a thread – non-determinism is simply represented by multiple enabled transitions at a single control location in a thread. For example, the following BIR fragment shows a location with more than one transition leading out.

```

loc loc0:
  when A do { x := x + 1; } goto loc1;
  when B do { y := y + 1; } goto loc2;

```

In a state where control is at `loc0` and both `A` and `B` are true, then both transitions are enabled, and is selected for execution non-deterministically.

For each transition α , we assume that we can determine, among other things, a unique identifier for a thread t that executes α , and the set of variables that are read or written by α .

A *path* π from a state s is a finite or infinite sequence such that $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots$ such that $s = s_0$ and for every i , $\alpha_i(s_i) = s_{i+1}$.

More details of the relationship between BIR systems and state transition systems can be found in Appendix ??.

Exercises

The goal of these exercises is to help the reader become acquainted with the basic aspects of BIR syntax and semantics.

1. Write a BIR program to compute the sum of the integers between 1 and 25 (inclusive).
2. (GUIDED) For the bounded buffer example of Figure 1.3, give three examples of a state s that is valid in the sense that the values for program

```

system BoundedBuffer {
  boolean full := false;
  boolean locked := false;

  active thread Producer() {
    loc loc0:
      when !locked do {
        locked := true;
      } goto loc1;
    loc loc1:
      when full do {
        locked := false;
      } goto loc0;
      when !full do {
        goto loc2;
      }
    loc loc2:
      do {
        full := true;
      } goto loc3;
    loc loc3:
      do {
        locked := false;
      } goto loc0;
  }

  active thread Consumer() {
    loc loc0:
      when !locked do {
        locked := true;
      } goto loc1;
    loc loc1:
      when !full do {
        locked := false;
      } goto loc0;
      when full do {
        goto loc2;
      }
    loc loc2:
      do {
        full := false;
      } goto loc3;
    loc loc3:
      do {
        locked := false;
      } goto loc0;
  }
}

```

Figure 1.3: A BIR system modeling a simple bounded buffer

```

system ReadersWriters {
  const Param {
    NUM_READERS = 2;
    NUM_WRITERS = 1;
  }

  // Invariant:
  // (nr = 0 ∨ nw = 0) ∧ nw ≤ 1

  int nr := 0;
  int nw := 0;

  active [Param.NUM_READERS]
  thread Reader(int index) {
    loc loc0:
      when nw == 0 do {
        nr := nr + 1;
      } goto loc1;
    loc loc1:
      do {
      } goto loc2;
    loc loc2:
      do {
        nr := nr - 1;
      } goto loc0;
  }

  active [Param.NUM_WRITERS]
  thread Writer(int index) {
    loc loc0:
      when nr == 0 && nw < 1 do {
        nw := nw + 1;
      } goto loc1;
    loc loc1:
      do {
      } goto loc2;
    loc loc2:
      do {
        nw := nw - 1;
      } goto loc0;
  }
}

```

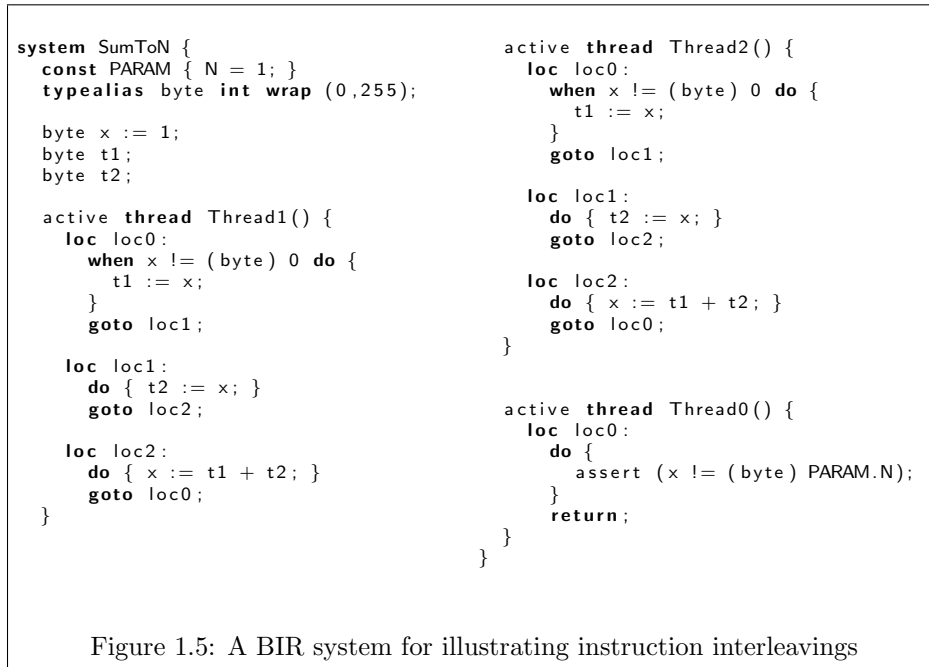
Figure 1.4: A BIR system modeling a simple systems of readers and writers

counters and variables of s match the domains for the bounded buffer program but yet s is actually unreachable from the initial state of the bounded buffer program. Give a short explanation to justify your answer.

3. (GUIDED) In this section, given a transition α and a state s , we discussed that α could be either enabled or disabled in s .
 - (a) For the bounded buffer example of Figure 1.3, give examples of three transitions and associated states such that the transitions are enabled in the associated states. Give a short explanation to justify your answer.
 - (b) For the bounded buffer example of Figure 1.3, give examples of three transitions and associated states such that the transitions are disabled in the associated states. Give a short explanation to justify your answer.
4. (GUIDED) For the bounded buffer example of Figure 1.3, give an example of a state s where the set of current transitions of s is not the same as the set of enabled transitions of s . Give a short explanation to justify your answer.
5. (GUIDED) In this section, we discussed the notion of an execution path as a sequence of states where each state s_{i+1} was generated from its predecessor s_i by applying an transition α that is enabled in s_i . For the bounded buffer example of Figure 1.3,
 - (a) give a sequence of six states that forms an execution for the system, and
 - (b) give a sequence of six states that *does not* form an execution path for the system. Give a short explanation to justify your answer.
6. Carry out the instructions in Exercise 2 above for the readers/writers example of Figure 1.4.
7. Carry out the instructions in Exercise 3 above for the readers/writers example of Figure 1.4.
8. Carry out the instructions in Exercise 5 above for the readers/writers example of Figure 1.4.

1.2 Interleavings, Schedules, and Computation Trees

Figure 1.5 presents the BIR code for a system that we will use to illustrate the concept of instruction interleaving associated with concurrent execution, and the power of state-space exploration techniques such as model-checking for automatically reasoning about the effects of interleavings.



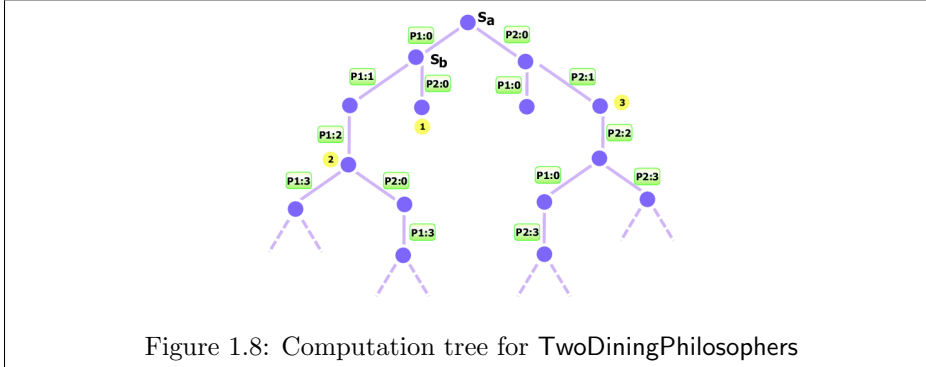
Threads `Thread1` and `Thread2` simply copy the value of variable `x` to temporary variables `t1` and `t2`, and then assign the sum of the temporaries back to `x`. The role of `Thread0`, on the other hand, is different. Notice that its sole instruction is an assertion that `x` does not have the value `N`. Because this assertion is located in its own (independently executing) thread, it can fire at any moment in the program’s execution. `Thread0`, then, acts as a sort of “monitor” to check whether a bad value of `x` ever occurs.

In a multi-threaded system running on a single processor, a *scheduler* that is part of the run-time system picks the thread that should take the next execution step. Thus, the scheduling strategy determines a particular *schedule* – an ordering of instruction executions.

Can the assertion in `SumToN` be violated? Answering this question requires one to reason about possible schedules. Figure 1.6 displays two schedules that cause the assertion to be violated when `N` is set to 2.

In this figure, we have represented a schedule as sequence of states with each annotated arrow $\xrightarrow{t:l}$ representing a transition between two states caused by thread t executing the instruction at location l of the thread. For example, $\xrightarrow{2:1}$ represents the execution of the instruction at `loc1` of `Thread2`.

By this point, the reader has probably surmised that there are a number of different schedules that can cause the assertion of `SumToN` to be violated. It is helpful to visualize the possible schedules of a particular system in the form of a *computation tree* consisting of system states as tree nodes and transitions



between states as tree arcs.

Figure 1.7 displays the initial portion of the computation tree for the `SumToN` system. The orange dashed line represents the first several steps of the schedule on the left side of Figure 1.6, and the green dotted line similarly represents a prefix of the schedule on right side of Figure 1.6. The initial state of `SumToN` is the root of the tree; we will henceforth refer to it as s_0 .

Notice that many different tree arcs emanate from s_0 . This means that multiple transitions are enabled. A moment's reflection on the BIR code for `SumToN` reminds us that all three threads are marked as initially runnable (`active`) – this gives rise to the fact any of the three could be the first to execute. We call such a point where the schedule must choose between several enabled transitions from different threads a *choice point*. In this particular example, each of the threads `Thread0`, `Thread1`, and `Thread2` has an enabled transition at s_0 , represented by the three arcs leading out of s_0 . Schedule (a) of Figure 1.6 represents a case where transition $\xrightarrow{1:0}$ is taken, whereas Schedule (b) of Figure 1.6 represents a case where transition $\xrightarrow{2:0}$ is taken.

The computation tree of Figure 1.7 clearly illustrates that there may be many schedules for the `SumToN` system. Of course, in any real implementation of such a system, the actual scheduling strategy would be determined by the implementing language's underlying run-time system or the underlying operating system. Thus, the particular scheduling strategy would likely vary across different implementations and platforms. Moreover, in distributed systems (which we will see later in subsequent chapters), there is even less control of the ordering of instruction execution across processes.

In formal reasoning about concurrent programs, one often abstracts away from implementation or platform details including such issues as the particular scheduling policy used. This allows one to make conclusions about the correctness of a modeled system when it is deployed in any context. In the following sections, we will see the mechanisms that a tool like Bogor provides for reasoning about some or all of a system's possible schedules.

Exercises

The goal of these exercises is to help the reader understand the basic concept of *schedule* and *computation tree*.

1. (GUIDED) Give another schedule for the SumToN system that causes the assertion to be violated when N has the value of 2.
2. (GUIDED) Describe a schedule for the SumToN system where the assertion *is not* violated when N has the value of 2
3. (GUIDED) Consider a modification to the SumToN system where N is set to 3. Is there a schedule that causes the assertion to be violated in this modified system? If so, give the schedule.
4. For the SumToN system, does there exist a value of N from 0 and 255 such that assertion cannot be violated?
5. (GUIDED) Draw a computation tree five levels deep for the bounded buffer example of Figure 1.3 (i.e., the computation tree should represent the first five instructions of every possible schedule).
6. Draw a computation tree five levels deep for the readers/writers example of Figure 1.4 (i.e., the computation tree should represent the first five instructions of every possible schedule).
7. (GUIDED) Give a schedule in the computation tree for the bounded buffer program of Figure 1.3 such that there is at least one “insertion” of an item into the buffer by the producer, and at least one consumption of an item by the consumer.
8. Give a schedule in the computation tree for the readers/writers program of Figure 1.4 such that there is one “write” of an item into the shared resource by the producer and and one read of the shared resource.

1.3 Executing (Simulating) BIR Systems with Bogor

The examples and exercises of the previous section clearly illustrate that, even for quite small concurrent systems, it is very difficult to reason about possible instruction interleavings. As a first step towards an exhaustive search of a system’s schedules, many model-checkers provide simulation facilities for analyzing the system being developed. These simulation facilities can be used in two modes – *random simulation* or *guided simulation* – that differ in the manner which an instruction is chosen for execution at a choice point. In a random simulation, when the simulator encounters a choice point, it randomly selects the next instruction to execute from the set of enabled transitions. In a guided

simulation, when the simulator encounters a choice point, the simulator typically displays a list of all the enabled transitions at that point, and the user determines the next instruction to execute by picking from the list.

Now we turn to Bogor's guided simulation mode. Figure 1.9 displays the result of invoking Bogor's guided simulation mode on the `TwoDiningPhilosophers` example of Figure 1.1. Referring to the computation tree in Figure 1.8, Figure 1.9 (a) illustrates that the simulator has executed all the transitions (in our example, zero) up to the first choice point s_a , and has paused and is waiting for the user to select from the two enabled transitions at that point. Figure 1.9 (b) shows the Bogor display that results if transition from `Philosopher1` is chosen (i.e., $\xrightarrow{P1:0}$ is taken). The simulation continues along the schedule determined by the user in this way until a state is reached where there are no more enabled transitions or until an assertion is violated.

The discussion above and the exercises below reveal both the benefits and the frustrations that can result from using a model-checker's simulation capabilities. In random simulations, the model-checker may sometimes find an error but often it will fail to pick a schedule that leads to a defect in the model. Guided simulation is useful because it automates a process that developers often carry out in their minds or on paper – namely, stepping through a program in a particularly tricky region in an effort to gain more insight into system's execution (e.g., possible interleavings and values of variables). Using guided simulation, the user may guide the model-checker to a property violation, but generally this requires that the user already have a good idea about how the property violation can occur. Guided simulation is tedious but can be effective for short traces. It is less effective or even infeasible for longer traces. It certainly cannot be used in practice to obtain an exhaustive search of all possible schedules.

Exercises

1. (GUIDED) Set the parameter `N` of the `SumToN` system to various values and analyze the system using Bogor's random simulation mode. Is Bogor able to find any violating traces in random simulation mode?
2. (GUIDED) Set the parameter `N` of the `SumToN` system to 5, and construct a violating trace using Bogor's guided simulation mode. Record the schedule path using the transition notation introduced earlier in the text. Is this the shortest trace that leads to a violation of the assertion? How can you be sure?
3. Set the parameter `N` of the `SumToN` system to 7, and construct a violating trace using Bogor's guided simulation mode. Record the schedule path using the transition notation introduced earlier in the text. Is this the shortest trace that leads to a violation of the assertion? How can you be sure?

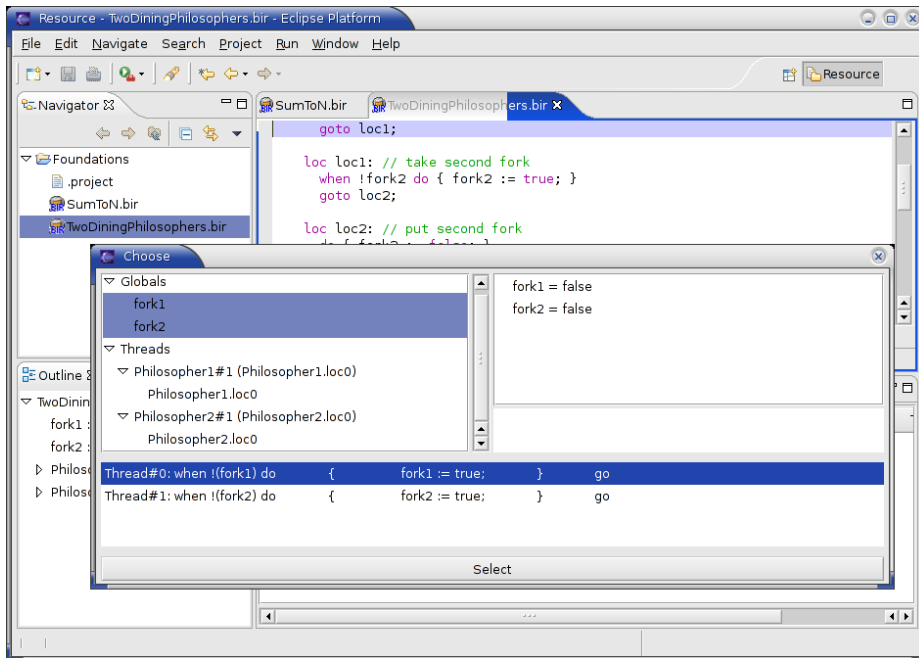
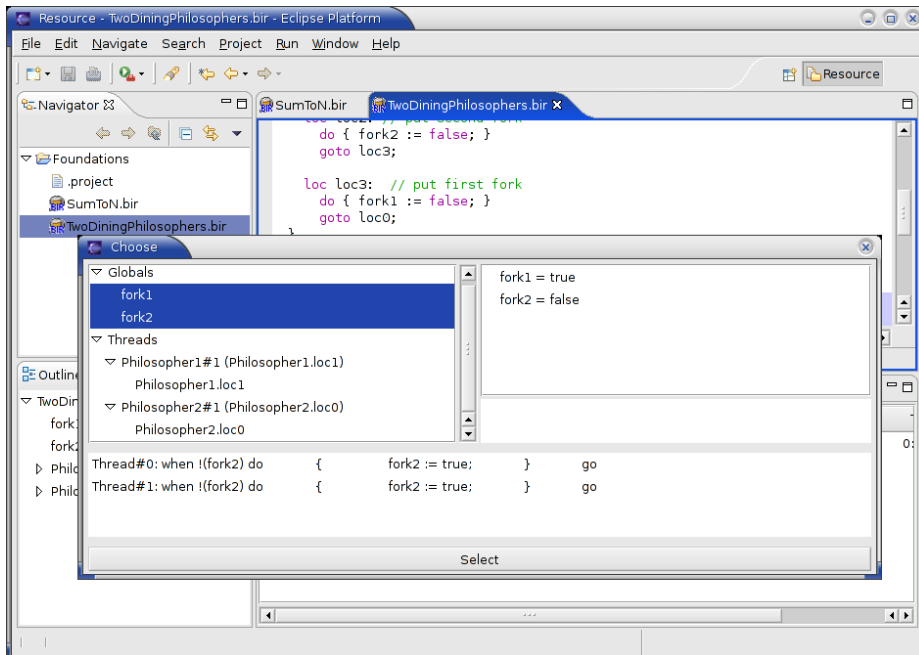
(a) Choice point at s_a (b) Second choice point at s_b (immediately precedes a deadlock)

Figure 1.9: Bogor Guided Simulation: TwoDiningPhilosophers

1.4 Verifying BIR Systems with Bogor

In the previous section, we saw that random simulation may be useful for “tire kicking” – a quick and superficial check to assess the quality of a model, but since it only explores one execution trace it’s likely not be that useful for finding bugs. We also saw that guided simulation is useful when trying to determine the possible execution traces for a particularly tricky section of the system, but since much manual effort is required, it’s not very effective for exploring a lot of traces or traces that are quite long.

The main strength of state exploration tools like Bogor is not their simulation capabilities, rather it is their ability to perform exhaustive searches of a system’s statespace. In this section, we introduce the core algorithm used in explicit state model checkers. We will orient the presentation around the primary data structures used by the algorithm. In fact, there are a variety of search strategies that have been implemented in explicit-state model-checkers. By far, the most common is depth-first search (DFS). We will focus on the basic depth-first search algorithm in this section; other types of searches will be discussed in later chapters.

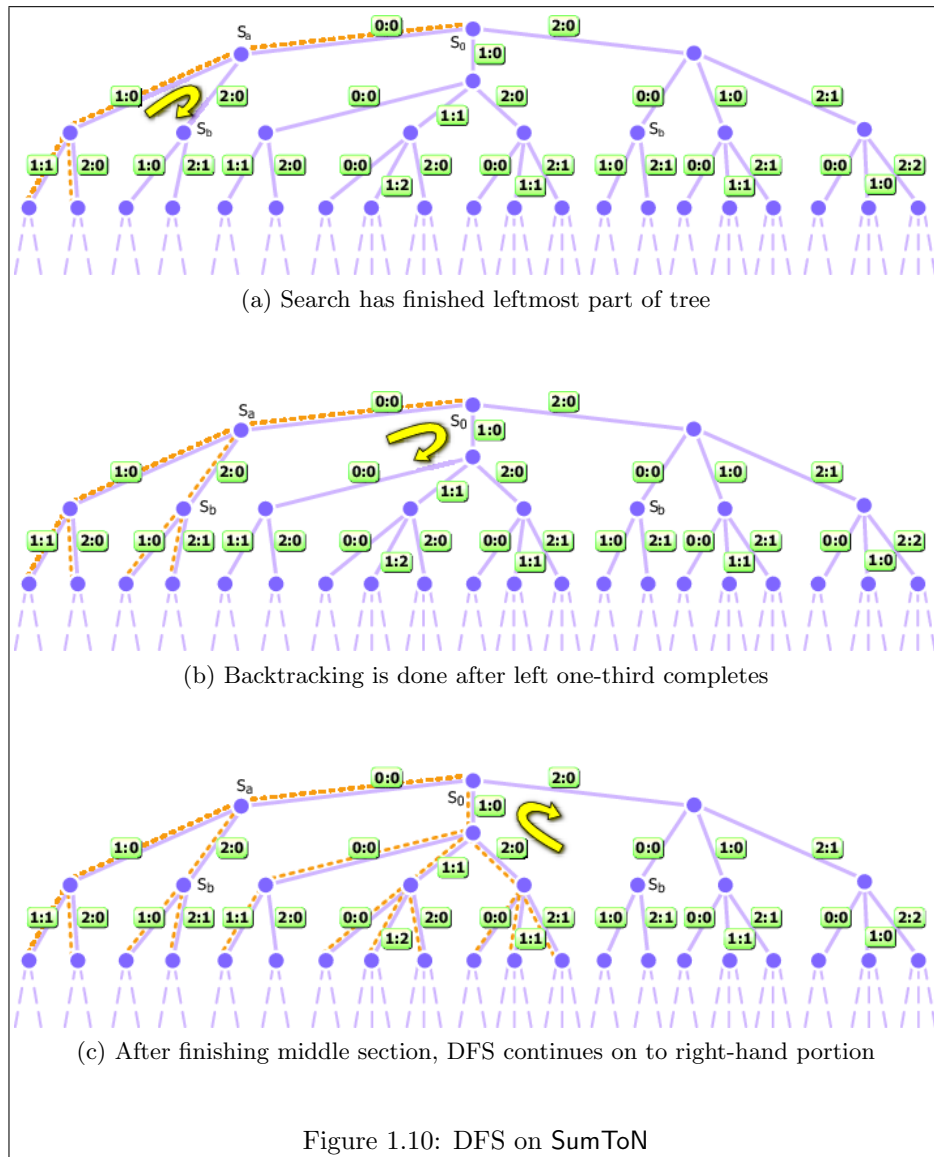
Recall that the “interesting aspects” of the simulation algorithms of the preceding section related to how the algorithms proceeded when they arrived at a choice point where more than one transition was enabled: random simulation picked one of the enabled transitions at random, whereas user-guided simulation asked the user to select a transition. In an exhaustive depth-first search, the system’s entire computation tree is explored. When a choice point is encountered, the model-checker selects an unexplored transition, but it also keeps track of the fact that it needs to return to that point and explore the remaining transitions leading out of that point.

The basic depth-first search algorithm

Figure 1.10 gives an overview of a depth-first search of the SumToN system. Figure 1.10 (a) illustrates that, as the algorithm completes the exploration of the entire sub-tree leading out of the left-most transition of s_a , it back-tracks until it finds a state where transitions remained to be explored (s_a in this case), and then it continues with next unexplored transition leading out of s_a . Once all paths out of s_a have been explored, the algorithm back-tracks to s_0 and continues exploring the unexplored paths from that point Figure 1.10 (b). The process continues as illustrated in Figure 1.10 (c) until the entire tree is covered (not pictured).

Figure 1.11 presents the outline of the basic DFS algorithm. This algorithm uses three different data structures:

- a *state vector* s holds the value of all variables as well as program counters for each thread,
- a *stack* of state vectors holds the states encountered down the current path in the computation tree, and



```

1  seen := {}
2  stack := cons(s0, nil)
3
4  while stack ≠ nil do {
5    s := head(stack)
6    stack := tail(stack)
7    if s ∉ seen {
8      checkState(s)
9      seen := seen ∪ {s}
10     workSet := enabled(s)
11     for all α in workSet do {
12       s' := α(s)
13       stack := cons(s', stack)
14     }
15   }
16 }

```

Figure 1.11: Depth-first search state-space exploration

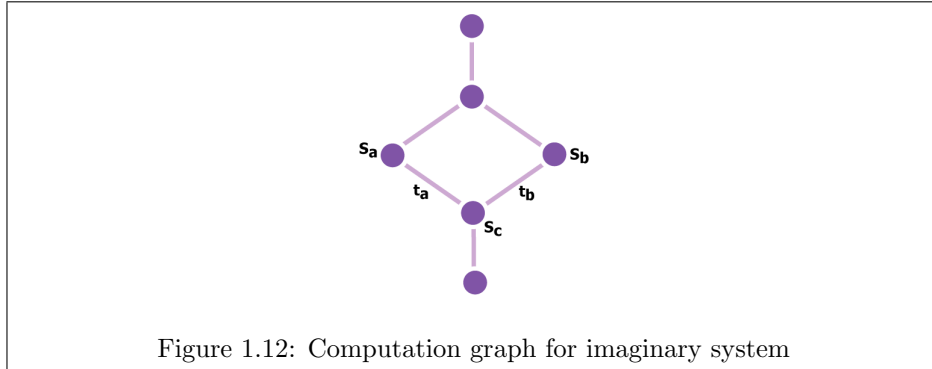
- a *seen state set* holds the state vectors for all the states that have been checked already (i.e., seen) in the depth first search.

The algorithm of Figure 1.11 proceeds as follows. First, *seen* is initialized as the empty set, indicating that no states have yet been visited. Next, the initial state is placed on the stack of states which must still be visited. The stack will serve an important purpose: when the end of a path in the computation tree is reached, the topmost stored state on the stack indicates the state to which the exploration should backtrack and restart.

Each iteration of the **while** loop at line 4 of Figure 1.11 starts by fetching the next state *s* to be analyzed. If the subtree starting at *s* has already been seen, then the iteration aborts immediately. Next, the state is checked to see if all invariants and assertions are satisfied by invoking *checkState(s)*. By inserting the current state into the *seen* set at line 9, we ensure that the algorithm will never explore a given state more than once. Next, all the outgoing transitions from *s* that are enabled are collected in *workSet*. Then, for each transition *α* in the *workSet*, *α* is applied to the current state *s* to get a next state *s'* along the current path. The new state *s'* is then pushed onto the top of the pending-work stack; a future iteration of line 4's **while** loop will explore the entire subtree rooted at *s'*. As a consequence, *all* paths leading out of *s* are explored.

Model-checkers usually employ multiple clever representations of the state vector, stack and seen set structures to obtain a highly optimized algorithm with respect to space and speed. We will discuss optimizations in later chapters, but for now we represent the values of the data structures in a simple abstract manner that captures the essences of the issues we wish to present.

Figure 1.2 of the previous section illustrates the sequence of state vectors for



a particular trace of the example in Figure 1.5.

We now provide some additional intuition about the role of the seen set. Often it is the case that the DFS algorithm will proceed down a path in a system's computation tree and it will encounter a state s that has already been encountered before on a different path through the computation tree. In such a case, there is no need to check s again (or any of s 's descendant states in the computation tree) since these states have already been checked previously. In the DFS algorithm, the seen set holds the states that have been seen before, and the algorithm consults this set to avoid exploring/checking a part of the computation tree that is identical to a part that has already been explored before.

For example, consider the fragment of the computation tree presented in Figure 1.10. Assume that the DFS algorithm has already explored the left path of the computation tree and is now proceeding down the right path. When it encounters the state s_b , it finds this state in the seen set. This causes the algorithm to backtrack to begin exploring other unexplored paths.

In cases where a transition along one path leads to a state s that has been encountered on another path, it is sometimes convenient from a conceptual standpoint to view s as being *shared* between the two paths as illustrated in Figure 1.12. In this view, the computation *tree* is actually represented as a *computation graph*, where a node that has more than one incoming arc represents a state that is encountered multiple times during the DFS search.

Due to the use of the seen set, checking a system with non-terminating executions (i.e., the system's computation tree contains infinite paths) may actually terminate if the system has a finite number of states. Our `SumToN` example contains a finite number of states since it has (1) only statically allocated threads, each of which has a fixed number of possible control points and (2) global variables each fixed at 8 bits long (the byte type). In this case, there will eventually be a back edge in the computation graph when the value of x overflows. This assures us that the DFS on `SumToN` will terminate when a seen set is used.

Exercises

The goal of these exercises is to help the reader understand the depth-first search state-space exploration algorithm, and in particular, the data structures associated with the algorithm.

1. (GUIDED) Given the computation tree in Figure 1.8 for the dining philosopher system of Figure 1.1, show the values of the three main data structures of the search algorithm of Figure 1.11 (i.e., the state vector, the stack, and the seen set) for each of the yellow-encircled labels in the computation tree.
2. (GUIDED) For the bounded buffer example of Figure 1.3, show the values of the three main data structures of the search algorithm of Figure 1.11 (i.e., the state vector, the stack, and the seen set) for each of the states in the computation tree up to depth two (consider the initial state to be at depth zero).
3. For the bounded buffer example of Figure 1.3, show the values of the three main data structures of the search algorithm of Figure 1.11 (i.e., the state vector, the stack, and the seen set) for each of the states in the computation tree up to depth two (consider the initial state to be at depth zero).
4. For the readers/writers example of Figure 1.4, show the values of the three main data structures of the search algorithm of Figure 1.11 (i.e., the state vector, the stack, and the seen set) for each of the states in the computation tree up to depth two (consider the initial state to be at depth zero).