On States for concurrency

Uri Abraham *
Ben Gurion University of the Neguev.

March 9, 2014

Abstract

The notions of state and history can be used not only to analyze and reason about a sequential program but also about concurrent programs that involve two or more processes. As an example to accompany this introduction we take the back-and-forth algorithm which is a very simple mutual exclusion algorithm for two processes. It is a simplified version of the Peterson–Fischer algorithm which will be dealt with in a later lecture.

1 States and histories

The notion of *state* is one of the most important in this course. What is a state? A state is a description of a system as if frozen in time. It is a snapshot: a picture taken at the highest possible shutter speed. Is reality a sequence of states (like a movie is a sequence of frames)? or perhaps reality is the primitive notion and a state is an idealized derivative? The mathematical definition of a state is first given in its simplest (a function defined on a set of variables) and is then developed into a structure with its corresponding satisfaction relation definition.

Suppose a set V (members of V are called state variables) and a map *type* which assigns to every variable $v \in V$ a set of possible values: $type(v)$. A *state* (over V and *type*) is a map S defined on V such that

$$S(v) \in type(v) \text{ for every } v \in V. \tag{1}$$

*Mathematical models for concurrency 2014

A protocol is a program designed to be executed by two or more processors concurrently. We shall not give a formal definition of this notion here, but the example of the Back-And-forth Protocol should clarify it, and we shall use this example in order to exemplify states.

Any protocol is associated with a set of variables. These are the local variables of the involved processes, as well as the communication means (registers, queues etc.) which are shared variables. In addition, we need special variables, "program counters", which point to the enabled instructions of each process. Let V be the set of variables thus obtained, with $type(v)$ defined for each $v \in V$. A state is a function over V that respects the type of each variable.

A pair of states (over V) is called a "step" when it describes an atomic change of the system. The possibility that the two states in a step are equal is not excluded. Any instruction of the protocol corresponds to a set of steps: those steps that represent an execution of the instruction.

Special states are defined to be "initial states" of the protocol.

A *history* (for a given protocol with variable set V) is a sequence of states over V,

$$H = \langle S_i \mid i \in \omega \rangle,$$

where ω is the set of natural numbers (we assume for simplicity that H is infinite) such that S_0 is an initial state and for every $i \in \omega$ there is an instruction R in the protocol such that step $\langle S_i, S_{i+1} \rangle$ is an execution of R. A history contains steps by different processors, and thus their steps are "interleaved" in a history.

The collection of all resulting histories describes the possible executions of the given protocol, and in order to prove that a protocol satisfies a certain property φ (such as mutual-exclusion) we have to prove that each of the resulting histories satisfies this property. The problem with this approach is that the relation "H satisfies φ" remains to be defined. Yet, at this stage of our lectures we leave this issue at the informal level and rely on our intuitions. We will return to this important question later on.

It turns out to be convenient to allow two consecutive states in a history that are equal: $S_{i+1} = S_i$. Step $\langle S_i, S_{i+1} \rangle$ is called in this case (by Lamport) a "stuttering step". Stuttering steps are natural in the following situation. Suppose that V_1 and V_2 are sets of variables and X_1, X_2 are collections of states over V_1 and V_2 (respectively). Suppose that $\pi : X_2 \to X_1$ is a map from the state space X_2 to X_1. A simple example of such a map is when

P_{cpy}	P_{chg}
RepeatForever	RepeatForever
1_0. repeat $t_{cpy} := \mathsf{R}_{chg}$ until $t_{cpy} \neq \mathsf{R}_{cpy}$	1_1. repeat $t_{chg} := \mathsf{R}_{cpy}$ until $t_{chg} = \mathsf{R}_{chg}$
2_0. CS	2_1. CS
3_0. $\mathsf{R}_{cpy} := t_{cpy}$.	3_1. $\mathsf{R}_{chg} := 1 - t_{chg}$.

Figure 1: The back-and-forth Protocol. Variables t_{chg} and t_{cpy} are local to the processes. they hold binary values $\{0, 1\}$. Registers R_{cpy} and R_{chg} are serial.

$V_1 \subseteq V_2$ and for every state $S \in X_2$, $\pi(S) = S \restriction V_1$ (namely restriction of S to the variables in V_1). It is possible that $S \neq S'$ and yet $\pi(S) = \pi(S')$. If $H = \langle S_i \mid i \in \omega \rangle$ is a history over X_2 (namely $S_i \in X_2$ for every $i \in \omega$) then we may define $\pi(H) = H'$ by the equation $H' = \langle \pi(S_i) \mid i \in \omega \rangle$. There are many situations in which it is natural and useful to view H' as a history sequence. But since it is possible that $S'_i = S'_{i+1}$ we must accept the possibility of stuttering steps in histories.

2 The back-and-forth Protocol

This section introduces the back-and-forth Protocol and uses it to illustrate the notions defined above: state, steps, and histories. There are two (serial) processes: P_{cpy} (also known as the *Copier*) and P_{chg} (aka the *Changer*). They communicate with two single-writer, single-reader serial registers, R_{cpy} owned by P_{cpy}, and R_{chg} owned by P_{chg}. The owner of the register is its sole writer, and the other process can read it. The registers are boolean, and their initial values are arbitrary. The boolean values 0 and 1 will be called "colors" here.

The algorithms (or protocol) are presented in Figure 1. In order to analyze this algorithm with finer granularity we rewrite it in Figure 2 and replace the repeat–until instruction with an explicit checking and go-to instruction.

P_{cpy}	P_{chg}
RepeatForever	RepeatForever
0_0. $t_{cpy} := \mathsf{R}_{chg}$;	0_1. $t_{chg} := \mathsf{R}_{cpy}$
1_0. if $t_{cpy} = \mathsf{R}_{cpy}$ goto 0_0;	1_1. if $t_{chg} \neq \mathsf{R}_{chg}$ goto 0_1;
2_0. CS	2_1. CS
3_0. $\mathsf{R}_{cpy} := t_{cpy}$.	3_1. $\mathsf{R}_{chg} := 1 - t_{chg}$.

Figure 2: The explicit back-and-forth Protocol.

2.0.1 States and histories of the back-and-forth Protocol

Our aim now is to define histories of the back-and-forth Protocol, in the fashion outlined in the previous section. For this, we first define the set

$$Var = \{\mathsf{R}_{cpy}, \mathsf{R}_{chg}, t_{cpy}, t_{chg}, PC_{cpy}, PC_{chg}\}$$

of variables associated with the protocol, and for each variable $x \in V$ we also define $type(x)$, the set of possible values of x.

1. R_{cpy} and $\mathsf{R}_{chg}$are the registers owned by P_{cpy} and P_{chg}. The possible values of these registers are 0 and 1. So $type(\mathsf{R}_{cpy}) = type(\mathsf{R}_{chg}) = \{0, 1\}$.

2. t_{cpy} and t_{chg} are the local boolean variables of P_{cpy} and P_{chg} respectively. $type(t_0) = type(t_1) = \{0, 1\}$.

3. PC_{cpy} and PC_{chg} are the program counters of P_{cpy} and P_{chg} respectively. $type(PC_{chg}) = \{0_1, 1_1, 2_1, 3_1\}$ refers to these lines in the PC_{chg} protocol. Similarly $type(PC_{cpy}) = \{0_0, 1_0, 2_0, 3_0\}$ refers to lines in the P_{cpy} operation. For example, $PC_{chg} = 3_1$ indicate that the write on R_{chg} is enabled and *Changer* is ready to execute "$\mathsf{R}_{chg} := 1 - t_{chg}$".

A state of the back-and-forth protocol is a function defined on Var which gives to every variable a value in its type. Note that any such function is a state. We do not require that a state actually appears in one of the possible executions of the algorithm. The number of states is $2^4 \times 4^2$.

An initial state is a state S such that $S(PC_{cpy}) = 0_0$ and $S(PC_{chg}) = 0_1$ (so there are no restriction on the values of the other variables). We define next the set of program steps.

A *program step* is a pair of states $\langle S_1, S_2\rangle$ over V that represents an execution of an instruction (a line) of the protocol. In details we have the following.

Steps by P_{cpy}:

1. A pair of states $s = \langle S_1, S_2\rangle$ constitutes a reading step of register R_{chg} by P_{cpy} corresponding to line 0 of its code when the following conditions hold:

   (a) $S_1(PC_{cpy}) = 0_0$,

   (b) $S_2(PC_{cpy}) = 1_0$, $S_2(t_{cpy}) = S_1(\mathsf{R}_{chg})$, and for any state variable X other than PC_{cpy} and t_{cpy}, $S_2(X) = S_1(X)$ (we will say "and no other variable changes").

   We say that these are $(0_0, 1_0)$ steps.

2. A pair of states $s = \langle S_1, S_2\rangle$ forms a "CS entry" step when

   $$S_1(PC_{cpy}) = 1_0,\ S_1(t_{cpy}) \neq S_1(\mathsf{R}_{cpy}),\ S_2(PC_{cpy}) = 2_0,$$

   and no variable other than PC_{cpy} changes.

   These steps are denoted $(1_0, 2_0)$ steps.

3. A pair of states $\langle S_1, S_2\rangle$ is a "failed check" if $S_1(PC_{cpy}) = 1_0$, $S_1(t_{cpy}) = S_1(\mathsf{R}_{cpy})$, $S_2(PC_{cpy}) = 0_0$, and no other variable changes.

   These are called $(1_0, 0_0)$ steps.

4. A pair of states $\langle S_1, S_2\rangle$ is a "CS stay" step when $S_2 = S_1$. It is a "CS exit" step when $S_1(PC_{cpy}) = 2_0$, $S_2(PC_{cpy}) = 3_0$, and no other variable changes.

   These are $(2_0, 2_0)$ and (respectively) $(2_0, 3_0)$ steps.

5. We say that $\langle S_1, S_2\rangle$ is a write step (on register R_{cpy} by P_{cpy}) corresponding to line 3_0 of the P_{cpy} operation if $S_1(PC_{cpy}) = 3_0$, $S_2(PC_{cpy}) = 0_0$, $S_2(\mathsf{R}_{cpy}) = S_1(t_{cpy})$, and no other variable changes.

   These are said to be $(3_0, 0_0)$ steps.

Exercise 2.1 *Write in details the steps of* P_{chg}.

A *history* is defined as a sequence $R = \langle S_i \mid i \in I \rangle$ where the index set I is either ω (the set of natural numbers) or else a finite initial segment of ω such that S_0 is an initial state and for every i the pair (S_i, S_{i+1}) is a program step by one of the processes. For simplicity of expression we will assume henceforth that $I = \omega$.

2.0.2 The assertional argument

We shall argue that in any history $R = \langle S_i \mid i \in \omega \rangle$, every state S_i satisfies the mutual exclusion property: it is not the case that $S_i(PC_{cpy}) = 2_0$ and $S_i(PC_{chg}) = 2_1$. We begin with an intuitively appealing argument, of the type that is sometimes said to be "behavioral". This intuitive argument refers to the algorithm of Figure 1.

Consider the steps by P_{cpy}. An execution of the protocol consists of several reads of R_{chg} until a successful read is obtained (a read for which $t_{cpy} \neq \mathsf{R}_{cpy}$ holds). Then, if indeed a successful read is obtained, it is followed by a CS event and then a write on R_{cpy}. Similarly, a successful read for P_{chg} is one in which condition $t_{chg} = \mathsf{R}_{cpy}$ holds, which indicates that the two registers had different values (when R_{chg} was read). There are two types of initial states, one in which $\mathsf{R}_{chg} = \mathsf{R}_{cpy}$ and another for which $\mathsf{R}_{chg} \neq \mathsf{R}_{cpy}$. The argument is very much the same for both types. Assume for example that initially $\mathsf{R}_{chg} = \mathsf{R}_{cpy}$ holds. As long as this condition remains, P_{cpy} will not be able to end the repeat loop of line 1 and to satisfy the until $t_{cpy} \neq \mathsf{R}_{cpy}$ condition. Process P_{chg}, however, will read register R_{cpy} and immediately satisfy its "until $t_{chg} = \mathsf{R}_{chg}$" condition and enter its CS. Then, assuming that at some stage P_{chg} exits its critical section, it executes the write $\mathsf{R}_{chg} := 1 - t_{chg}$ for line 3, and reverses the situation. Now it is P_{chg} that remains with $PC_{chg} = 1_1$ and cannot satisfy its until condition. Only when P_{cpy} has a successful read, it can enter its critical section, and then when it exits its critical section it brings back the condition $\mathsf{R}_{chg} = \mathsf{R}_{cpy}$. Then and only then the "door" for P_{chg} is opened again and it can enter its critical section. Continuing this way, we see that the critical sections of the two processes alternate with each process enabling the entry of the other process to its critical section.

This argument, especially if accompanied with a picture is surely very convincing, and investigating concurrent algorithms in terms of such "behavioral reasoning" is very common. The reader may not immediately see

why these behavioral arguments are not considered "formal". Indeed, the back-and-forth algorithm is so simple (not to say trivial) that it is not obvious to see what a formal proof could add. Experience of many researchers however has shown that this type of behavioral argument, in more complex situations, has lead to grave errors.

One feature of formal arguments which should alert the reader to possible pitfalls when it is not present in the proof is the following. A mathematical proof is proving some property of some objects. Now you should always ask yourselves the following questions when reading an informal argument: are the objects that appear in the argument well defined? Can you locate in the proof the places where the exact definitions of these objects appear? If the answer to one of these question is negative, then there is a danger that the proof is too informal. In our case, the basic objects are the states and the steps defined above. So the answer to the first question is "yes". We do have precise definition of the objects we deal with. But for the second question we have to admit that the answer is negative. Nowhere in this behavioral argument did we refer to the exact definition of the steps. So the behavioral argument fails to pass our formality test.

Yet behavioral arguments *are* attractive and natural and we do not want to prohibit their usage. In this course we want to study the tension between the formal and informal and between the behavioral and state-based correctness proofs. A main aim of this course is to learn how to transform behavioral correctness arguments into more formal correctness proofs.

We describe in our course two approaches to concurrency. The invariant proof approach is based on the state and history notions, and the Tarskian models approach is based on relational structures. We begin with the invariant method which is almost universally accepted as the "right" approach to concurrency, and only after learning some basic logic and investigating the notion of time will we be able to define Tarskian system executions with which behavioral arguments can be formalized.

An *invariant* is a statement φ about states such that

1. φ holds in every initial state.

2. If (S, T) is any step and φ holds in S then it holds in T as well.

We shall prove that the conjunction of the following four statements A, B, C, D is an invariant of the back-and-forth algorithm.

A: $\mathsf{R}_{chg} = \mathsf{R}_{cpy} \rightarrow PC_{cpy} = 0_0, 1_0$.

B: $\mathsf{R}_{chg} \neq \mathsf{R}_{cpy} \rightarrow PC_{chg} = 0_1, 1_1$.

C: If $PC_{chg} = 1_1$ and $(t_{chg} = \mathsf{R}_{chg})$, then $\mathsf{R}_{chg} = \mathsf{R}_{cpy}$.

D: If $PC_{cpy} = 1_0$ and $(t_{cpy} \neq \mathsf{R}_{cpy})$, then $\mathsf{R}_{cpy} \neq \mathsf{R}_{chg}$.

Here, condition $PC_{chg} = 1_1, 2_1, 3_1$, for example, is a shorthand for the disjunction $PC_{chg} = 1_1 \vee PC_{chg} = 2_1 \vee \cdots$.

We also note that the constants $0_0, \ldots, 3_0, 0_1, \ldots, 3_1$ are all different. Formally, this statement is also an invariant. The proof that indeed this is an invariant does not depend on the program that we analyze, but on the fact that no instruction can change the value of a program constant. So, although we may use this fact in our proof, we do not need to justify it.

Language: In any implication $\alpha = (X \textit{ implies } Y)$ we say that X is the premiss ($X = prem(\alpha)$) and Y the conclusion ($Y = conc(\alpha)$). Recall that an implication holds (is true) in a state if the premiss is false or the conclusion is true.

Exercise 2.2 *Prove that the conjunction (which we call φ) $A \wedge B \wedge C \wedge D$ is an invariant.*

Directions. First prove that φ holds in any initial state.

Then consider in turn each of the following twelves kinds of steps. For each kind and for every step (S, T) in that kind, assume that $S \models A \wedge B \wedge C \wedge D$, and then prove that $T \models X$ for $X = A, B, C, D$.

1. s is some $(0_0, 1_0)$ step.
2. s is some $(1_0, 0_0)$ step.
3. s is some $(1_0, 2_0)$ step.
4. s is some $(2_0, 2_0)$ step.
5. s is some $(2_0, 3_0)$ step.
6. s is some $(3_0, 0_0)$ step.
7. s is some $(0_1, 1_1)$ step.
8. s is some $(1_1, 0_1)$ step.

9. s is some $(1_1, 2_1)$ step.

10. s is some $(2_1, 2_1)$ step.

11. s is some $(2_1, 3_1)$ step.

12. s is some $(3_1, 0_1)$ step.

Let ME be the mutual-exclusion statement: $PC_{cpy} = 2_0 \rightarrow PC_{chg} \neq 2_1$.

Exercise 2.3 *Prove that the invariant implies ME.*

Exercise 2.4 *Prove that the following statement E is an invariant.*

$$E: \ PC_{cpy} = 2_0, 3_0 \rightarrow t_{cpy} \neq \mathsf{R}_{cpy}.$$

Then show that this statement is not a consequence of the invariant $A \wedge B \wedge C \wedge D$. For this you have to find a state S that satisfies $A \wedge B \wedge C \wedge D$ but does not satisfy E. This example shows that the invariant $A \wedge B \wedge C \wedge D$ is true in some states that never appear in any history execution of the algorithm. That is, our invariant does not characterize those states that appear in histories.