

Chapter 2

ABSTRACT PROTOCOL NOTATION

It is useful to specify network protocols using a formal notation. First, by using a formal notation to specify a network protocol, one can formally verify the correctness of this protocol and check that the protocol performs the function that it is intended to perform. Second, formal specification and verification is particularly important for secure network protocols. To verify the security guarantees of a protocol, one cannot depend only on some testing of the protocol because the tester may omit cases where vulnerabilities or weaknesses occur in the protocol. This is why we decided to specify all the secure protocols in this manuscript using a formal notation.

In this chapter, we present a variation of the Abstract Protocol Notation that is introduced in [13]. We use this variation to specify all the protocols presented in this manuscript.

The remainder of this chapter is organized as follows. In Section 2.1, we introduce the concept of processes and channels. In Section 2.2, we introduce the components of a process, namely constants, variables, and actions. In Section 2.3, we introduce the state transition diagram of a protocol, which is our tool to verify the correctness of the protocol. In Section 2.4, we introduce three more features of the AP notation, namely process arrays, parameters, and parameterized actions, that are used in our presentation.

1. PROCESSES AND CHANNELS

A protocol is defined by a collection of processes, and the channels between these processes. Processes in a protocol need to communicate with other processes in the same protocol by sending messages to and receiving

messages from the other processes. A message has a name (or a type) and can have zero or more fields that carry values to be used by the message receiver.

A message is transported from a sending process p to a receiving process q via the channel from p to q . The channel from p to q is the place where a message stays after it is sent by p and before it is received by q or before it is lost. Between each pair of adjacent processes p and q , there are two unidirectional channels: one from p to q , and the other from q to p .

Every channel in a protocol is both unbounded and FIFO (first-in, first-out). The unboundedness property means that an unbounded number of messages can reside simultaneously in a channel. The FIFO property means that messages are received from a channel in the same order in which they were sent into the channel. Messages that reside simultaneously in a channel form a sequence $\langle m.1; m.2; \dots; m.k \rangle$ in accordance with the order in which messages $m.1, m.2, \dots, m.k$ have been sent by the sending process. The head message in the sequence, $m.1$, is the earliest sent, and the tail message in the sequence, $m.k$, is the latest sent. When the receiving process is ready to receive a message, it removes the head message, namely $m.1$, from the sequence. In this case, and the next message, namely $m.2$, becomes the next head message in the sequence, and so on. Therefore messages are to be received in the same order in which they were sent.

2. CONSTANTS, VARIABLES, AND ACTIONS

A process in a protocol is defined by a set of constants, a set of variables, and a set of actions. The protocol performs its designated function by executing the actions in its processes. In the next section, we explain how the actions in a process are executed. In this section, we discuss the constants, variables, and actions of a process.

A *constant* of a process has a name and a value, and can be one of the following four types: boolean, integer, range, and array. The constants of a process can be read but not updated by the actions of this process. Thus, the value of each constant of a process is either fixed or is updated by another process outside the protocol.

A *variable* of a process has a name and a value, and can be one of the following four types: boolean, integer, range, and array. The variables of a process can be read and updated by the actions of this process.

An *action* of a process consists of a guard, an arrow “ \rightarrow ”, and a statement:

$$\langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$$

The <guard> of an action is of one of the following two types: a local guard or a receiving guard.

A *local guard* is a boolean expression that involves the constants and variables of the process in which the local guard occurs.

A *receiving guard* is of the form:

rcv <message> **from** <process name>

A <statement> of an action is of one of the following six types: skip, assignment, send, selection, iteration, and sequence. Next, we describe the six types of statement and how to execute each of them.

A *skip* statement is of the form:

skip

The skip statement is executed by doing nothing.

An *assignment* statement is of the form:

v := E

where v is a variable of the process in which the assignment statement occurs, and E is an expression of the same type as v. The assignment statement is executed by assigning the current value of E to variable v.

A *send* statement is of the form:

send <message> **to** <name of another process>

This statement is executed by sending a message of the specified type to the specified process.

A *selection* statement is of the form:

if <boolean expression> → <statement>
 ...
[] <boolean expression> → <statement>
fi

This statement is executed by first computing the current value of each <boolean expression>, then arbitrarily selecting one <boolean expression> whose value is true and executing its corresponding <statement>.

An *iteration* statement is of the form:

do<boolean expression> → <statement>
od

This statement is executed by repeatedly computing the value of the <boolean expression> and then executing the <statement> when the value of the <boolean expression> is true. Execution of this statement terminates when the <boolean expression> becomes false.

A *sequence* statement is of the form:

<statement>; <statement>

This statement is executed by first executing the first **<statement>** and then executing the second **<statement>**.

Next, we use an example to illustrate the use of constants, variables, and actions. The following protocol consists of two processes *p* and *q*. In this protocol, process *p* can send a request message to process *q*, and then wait for a reply message from *q* before *p* can send the next request message to *q*. Process *p* can be specified as follows.

```
process p
var ready  : boolean  {init. ready=true}
    txt, t   : integer
begin
    ready →
        txt := any;
        send rqst(txt) to q;
        ready := false

    [] rcv rply(t) from q →
        {use text t in received message}
        ready := true
end
```

Process *p* has three variables: variable *ready* is used to remember whether process *p* is waiting for a *rply* message from process *q* or not, variable *txt* is used for keeping the content of the latest *rqst* message process *p* sends to process *q*, and variable *t* is used for keeping the content of the latest *rply* message process *p* receives from process *q*. There are two actions in process *p*. In the first action, if the value of *ready* is true, then *p* chooses a new value for *txt*, sends a *rqst(txt)* message to process *q*, and sets the value of *ready* to false. In the second action, if *p* receives a message *rply(t)* from *q*, then *p* sets the value of *ready* to true.

When process *q* receives a request message from process *p*, *q* returns a reply message to *p*. Process *q* can be specified as follows.

```
process q
var t : integer
begin
    rcv rqst(t) from p →
        t := any;
```

send rply(t) to p
end

Process q has one variable t, which is used for keeping the content of the latest message that process q receives from or sends to process p. There is one action in process q: if q receives a rqst(t) message from p, then q chooses a new value for t, and returns a rply(t) message to p.

3. STATE TRANSITION DIAGRAM

A state of a protocol is defined by one value for each constant and one value for each variable in each process in the protocol and by one sequence of messages for each channel in the protocol.

An action in a process p in a protocol is enabled at a state S of the protocol iff one of the following two conditions holds at S: the guard of the action is a local guard, or the guard is a receiving guard of the form **rcv m from q** and the head message in the channel from process q to process p is m at state S.

If one or more actions in the same process or in different processes in a protocol are enabled at a state S, then exactly one of the enabled actions is executed, yielding a next state S' of the protocol. Likewise, if one or more actions are enabled at state S', then exactly one of the enabled actions is executed, yielding a next state S'', and so on. An execution of a protocol may terminate when the protocol reaches a “deadlock state”, where no action is enabled. If a protocol never reaches a deadlock state, then an execution of this protocol can continue endlessly.

Executing the actions (of different processes) in a protocol proceeds according to the following three rules:

- I. *Atomicity:*
The actions in a protocol are executed one at a time.
- II. *Nondeterminism:*
An action is executed only when its guard is true.
- III. *Fairness:*
An action whose guard is continuously true is eventually executed.

To construct a state transition diagram of a protocol, we have to derive all the possible states that can be reached by the protocol. The derivation of reachable states begins with an initial state in which every constant and every variable is assigned an initial value and every channel in the network

is empty. Then, all the actions that are enabled at this state are identified. Execution of each of these enabled actions at the current state leads the network to a different next state. This procedure is continued at each of the next states until a deadlock state is reached or a previous state is reached.

After we derive all the reachable states of a protocol, we can draw the corresponding state transition diagram. In a state transition diagram, each node represents one network state, and each arrow from a node S to another node S' represents an action execution that leads the network from state S to state S' .

Next, we use the protocol defined in the last section as an example for illustrating the construction of a state transition diagram. Assume that this network of process p and process q starts at a state defined by the following protocol predicate $S.0$.

$$S.0 : \text{ready} \wedge \text{txt} = x \wedge t.p = y \wedge t.q = z \wedge \\ \text{ch.p.q} = \langle \rangle \wedge \text{ch.q.p} = \langle \rangle$$

The first conjunct in $S.0$ asserts that variable ready in process p has the value true. The next three conjuncts assert that txt in process p has the value x , t in process p has the value y , and t in process q has the value z . The last two conjuncts assert that the two channels between processes p and q are empty.

At state $S.0$, exactly one action, namely the first action in process p , is enabled. Executing this action at state $S.0$ leads the network to the following state $S.1$.

$$S.1 : \sim \text{ready} \wedge \text{txt} = x \wedge t.p = y \wedge t.q = z \wedge \\ \text{ch.p.q} = \langle \text{rqst}(\text{txt}) \rangle \wedge \text{ch.q.p} = \langle \rangle$$

At state $S.1$, only the sole action in process q is enabled. Assume that process q chooses a random value z' for its variable t when executing this action at state $S.1$. Thus the network is led to the following state $S.2$.

$$S.2 : \sim \text{ready} \wedge \text{txt} = x \wedge t.p = y \wedge t.q = z' \wedge \\ \text{ch.p.q} = \langle \rangle \wedge \text{ch.q.p} = \langle \text{rply}(t.q) \rangle$$

At state $S.2$, only the second action in process p is enabled. Executing this action at $S.2$ leads the network to the following state $S.3$.

$$S.3 : \text{ready} \wedge \text{txt} = x \wedge t.p = z' \wedge t.q = z' \wedge \\ \text{ch.p.q} = \langle \rangle \wedge \text{ch.q.p} = \langle \rangle$$

At state $S.3$, only the first action in process p is enabled. Assume that process p chooses a random value x' for its variable txt when executing this action at state $S.3$. Thus the network is led to the following state $S.4$.

$$S.4 : \sim \text{ready} \wedge \text{txt} = x' \wedge t.p = z' \wedge t.q = z' \wedge$$

$$\text{ch.p.q} = \langle \text{rqst}(\text{txt}) \rangle \wedge \text{ch.q.p} = \langle \rangle$$

It turns out that this protocol has an infinite number of reachable states, because in each round process p chooses a new value for its variable txt before sending a message $\text{rqst}(\text{txt})$ to process q , and process q chooses a new value for its variable t before sending a message $\text{rply}(t)$ to process p . Therefore, it is impossible to draw the corresponding state transition diagram in full.

To solve this problem, the definition of a state transition diagram for a protocol can be generalized as follows. Instead of each node in the diagram representing only one state of the protocol, some nodes in the diagram can be aggregated under a broader protocol predicate into one node that represents a nonempty subset of the protocol states.

For example, the initial state $S.0$ of this protocol can be found in an aggregated state that is defined by the following protocol predicate $T.0$.

$$T.0 : \text{ready} \wedge \text{ch.p.q} = \langle \rangle \wedge \text{ch.q.p} = \langle \rangle$$

State $S.1$ can be found in an aggregated state that is defined by the following protocol predicate $T.1$.

$$T.1 : \sim \text{ready} \wedge \text{ch.p.q} = \langle \text{rqst}(\text{txt}) \rangle \wedge \text{ch.q.p} = \langle \rangle$$

State $S.2$ can be found in an aggregated state that is defined by the following protocol predicate $T.2$.

$$T.2 : \sim \text{ready} \wedge \text{ch.p.q} = \langle \rangle \wedge \text{ch.q.p} = \langle \text{rply}(t.q) \rangle$$

Similarly, state $S.3$ can be found in the aggregated state defined by $T.0$, state $S.4$ can be found in the aggregated state defined by $T.1$, and so on.

We derive the following three inductions regarding the three aggregated states $T.0$, $T.1$, and $T.2$. First, at a state defined by $T.0$, only the first action in process p is enabled, and executing this action at a state defined by $T.0$ leads the protocol to a state defined by $T.1$. Second, at a state defined by $T.1$, only the sole action in process q is enabled, and executing this action at a state defined by $T.1$ leads the protocol to a state defined by $T.2$. Third, at a state defined by $T.2$, only the second action in process p is enabled, and executing this action at a state defined by $T.2$ leads the protocol to a state defined by $T.0$. Therefore, the sequence of transitions from $T.0$ to $T.1$, from $T.1$ to $T.2$, and from $T.2$ to $T.0$ forms a cycle in which the network performs progress. In this case, a state transition diagram for the protocol can be drawn as shown in Figure 2.1.

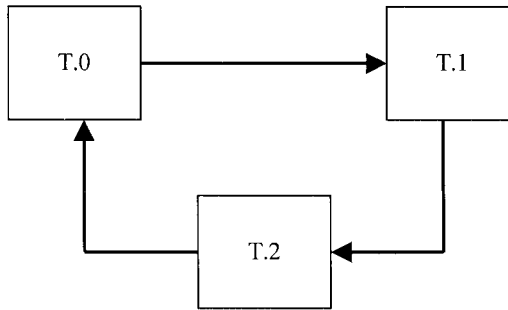


Figure 2-1. A state transition diagram for the example protocol.

4. PROCESS ARRAYS, PARAMETERS, AND PARAMETERIZED ACTIONS

In this section, we introduce two extensions of the AP notation. First, we introduce process arrays which allow one to define a set of identical processes by defining only one representative process. Second, we introduce parameters and parameterized actions which allow one to define a finite set of actions as a single parameterized action in a process.

A process array is a finite set of processes: each of them has the same set of constants, the same set of variables, and the same set of actions. Thus, all the processes in a process array can be specified by specifying only one representative process of the array. For example, let p be an array of n processes named $p[0]$, $p[1]$, ..., $p[n-1]$ respectively. A representative process of this array can be $p[i]$, where i is an index whose value is in the range between 0 and $n-1$.

A parameter has a name and is of type range. This implies that each parameter has a finite number of values.

A parameterized action is an action that refers to one or more parameters. A parameterized action is a shorthand notation for a finite set of actions: each of them can be obtained from the parameterized action by first selecting for each parameter i in the parameterized action a value $v.i$ from the domain of i , and then replacing every occurrence of i in the parameterized action by the selected value $v.i$.

Next, we extend the example shown in Section 2.2 to illustrate the use of process arrays, parameters, and parameterized actions. In the extended example, process p communicates with an array of n processes $q[i: 0 .. n-1]$.

In this protocol, process p can send a request message to any process $q[i]$, and then wait for a reply message from $q[i]$ before p can send the next request message to the same $q[i]$. While process p is waiting for a reply message from $q[i]$, p can send a request message to any other process $q[i']$ that is different from $q[i]$, provided that p is not waiting for a reply message from $q[i']$. Process p in the extended protocol can be specified as follows.

```

process p
const n      : integer      {number of processes in process array q}
var   ready  : array [0 .. n-1] of boolean  {init. ready=true}
        txt, t : integer
par    i      : 0 .. n-1
begin
    ready[i] →
        txt := any;
        send rqst(txt) to q[i];
        ready[i] := false

    [] rcv rply(t) from q[i] →
        {use text t in received message}
        ready[i] := true
end

```

Process p in the extended protocol has one constant n , which specifies the number of processes in process array q , and one parameter i , which stands for the index of process array q . Variable `ready` in process p is changed to an array of n booleans to remember whether process p is waiting for a reply message from each of the n processes in process array q or not. Both actions in process p are parameterized actions; each action is a shorthand notation of n actions as there are n possible values for parameter i . In the first parameterized action, if the value of `ready[i]` is true, then p chooses a new value for `txt`, sends a `rqst(txt)` message to process $q[i]$, and sets the value of `ready[i]` to false. In the second parameterized action, if p receives a message `rply(t)` from $q[i]$, then p sets the value of `ready[i]` to true.

Next, we specify process array q in the extended protocol as follows. Each new process $q[i]$, when receiving a request message from p , will return a reply message to p . As discussed previously in this section, we can specify process array q by specifying one representative process $q[i]$ in the array.

```

process q[i : 0 .. n-1]
const n : integer      {number of processes in process array q}
var    t : integer

```

```
begin  
  rcv rqst(t) from p  $\rightarrow$   
    t := any;  
    send rply(t) to p  
end
```

Each process $q[i]$ has one new constant n , which is the same as the constant n in the new process p . There is one action in process $q[i]$: if $q[i]$ receives a $rqst(t)$ message from p , then $q[i]$ chooses a new value for t , and returns a $rply(t)$ message to p .



<http://www.springer.com/978-0-387-24426-6>

Hop Integrity in the Internet

Huang, C.-T.; Gouda, M.G.

2006, XI, 112 p. 17 illus., Hardcover

ISBN: 978-0-387-24426-6