Chapter 7
Concurrent Systems

Finally, we are going to employ the concepts introduced in the previous chapter for their core purpose: the modeling and analysis of systems with concurrently executing components. Examples of such systems are programs with multiple processes or threads that cooperate via shared variables, as well as multiple independent programs that interact by exchanging messages over a network.

The core idea is to replace the somewhat intricate notion of "concurrency" by the more tractable notion of "nondeterminism". While in a concurrent system multiple actions may be executed simultaneously, the nondeterministic model of such a system considers the execution of individual actions only, but of *any* possible action; thus the analysis of the model investigates all possible "interleavings" of actions. Although this does not consider the truly simultaneous execution of two actions a_1 and a_2, one may argue that it suffices to consider just two possibilities, that a_1 is executed before a_2, and that a_2 is executed before a_1: either both actions affect a common physical component, then this component has to arrange them in one of these orders to achieve a well-defined effect; or they only affect components at different locations, then the notion of "simultaneity" is meaningless (due to the principle of relativity).

When reasoning about such models, we will focus on their *safety properties*, i.e., properties whose violation can be observed at a particular point of the system execution; such properties ensure that "nothing bad can ever happen". The verification of a safety property requires the formulation of a *system invariant* that constrains the set of reachable system states as closely as possible and that implies the safety property. The correctness of the invariant is shown by an *induction proof*: all initial system states must satisfy the invariant (the induction base), and, if the invariant holds in a state (the induction hypothesis), it must hold again after every possible action of the system (the induction step); typically the safety property of interest has to be be sufficiently strengthened to yield valid induction steps. Indeed the formulation of *inductive invariants* is the core problem of concurrent system verification; only via such invariants a system is truly understood.

```
 87 shared system AlternatingBitNetwork
 88 {
 89   // the messages sent and received (local to each process)
 90   var sent:Msg = Default;
 91   var rcvd:Msg = Default;
 92
 93   // the protocol bits
 94   var sbit:Bit = ⊥;
 95   var sack:Bit = ⊥;
 96   var rbit:Bit = ⊥;
 97
 98   // the communication channels
 99   var msgq:Queue = ⟨len:0,pack:Array[M,Package](⟨msg:Default,bit:⊥⟩)⟩;
100   var ackq:Queue = ⟨len:0,pack:Array[M,Package](⟨msg:Default,bit:⊥⟩)⟩;
101
102   // the history of the messages sent and received
103   var smsgs:Seq = Array[N,Msg](Default);
104   var snum:Index = 0;
105   var rmsgs:Seq = Array[N,Msg](Default);
106   var rnum:Index = 0;
107
108   // safety property: the messages received are the messages sent
109   invariant rnum ≤ snum ∧ ∀i:Index with i < rnum. rmsgs[i] = smsgs[i];
110
111   // invariants from the shared memory version
112   invariant snum-1 ≤ rnum;
113   invariant sack = sbit ⇒ rnum = snum;
114   invariant rbit ≠ sbit ∧ rnum < N ⇒ rnum < snum ∧ sent = smsgs[snum-1];
115
116   // additional invariants
117   invariant rnum < snum ⇒ rbit ≠ sbit;
118   invariant ∀i:ℕ[M] with i < msgq.len.
119     (msgq.pack[i].bit ≠ rbit ⇒ msgq.pack[i].msg = sent ∧ msgq.pack[i].bit = sbit);
120   invariant ∀i:ℕ[M] with i+1 < msgq.len.
121     (msgq.pack[i].bit ≠ rbit ⇒ msgq.pack[i+1].bit ≠ rbit);
122   invariant ∀i:ℕ[M] with i+1 < ackq.len.
123     (ackq.pack[i].bit ≠ sack ⇒ ackq.pack[i+1].bit ≠ sack);
124   invariant sack = sbit ⇒ (∀i:ℕ[M] with i < msgq.len. (msgq.pack[i].bit = rbit));
125   invariant sack = rbit ⇒ (∀i:ℕ[M] with i < ackq.len. (ackq.pack[i].bit = rbit));
```

- system AlternatingBitNetwork
  - Execute operation
  - Verify specification preconditions
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
    - Is index value legal?
  - Verify specification
    - Does system invariant initially hold?
    - Does system invariant initially hold?
    - Does system invariant initially hold?
    - Does system invariant initially hold?
    - Does system invariant initially hold?
    - Does system invariant initially hold?
    - Does system invariant initially hold?
    - Does system invariant initially hold?
    - Does system invariant initially hold?
    - Does system invariant initially hold?
    - action sender
    - action resend
    - action receiveAck
    - action receiver
    - action sendAck
    - action loseMsg
    - action loseAck
  - Verify initialization precondtions
  - Verify action preconditions

Fig. 7.7 Verification of the Alternating Bit Protocol (Distributed Version)

7.3 A Resource Allocator

We consider a system described in [44] which consists of a server and a set of clients (see Figure 7.8). The server manages a pool of resources which clients request from the server by sending corresponding messages; the server grants these requests by sending corresponding replies. The central safety property of the system is that the server grants every resource to at most one client at a time, i.e., no two clients may simultaneously use the same resource.

The problem becomes complex, because both requests and grants do not only refer to single resources; in particular, every client may request any set resources. However, the server may respond by a message that contains only some of the requested resources; if it does not immediately grant all resources, the server will send later further messages that grant more of them, until the complete request is satisfied. Furthermore, as soon as a client holds some of the requested resources, it may (even if its request has not yet been satisfied completely) return some of them to the server. However, a client may not send another request for new resources before it has received and returned all the resources from its previous request.

To fairly satisfy requests from the various clients, the server keeps track of the sequence of still pending requests in the order in which they were received. The server grants a resource to a client if there is no earlier request from another client for the same resource.

© 2021 Wolfgang Schreiner.

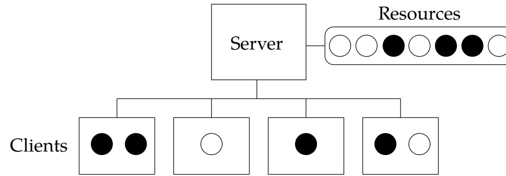

Fig. 7.8 A Resource Allocator

We will model this system in RISCAL based on the following declarations:

```
val C:ℕ; val R:ℕ;
axiom notNull ⇔ C > 0 ∧ R > 0;
type Client = ℕ[C-1];
type Resource = ℕ[R-1];
type Position = ℕ[C];
```

Here the model parameter $C > 0$ denotes the number of clients while $R > 0$ denotes the number of resources. Clients are denoted by values of type *Client* while resources are denoted by values of type *Resource*. Values of type *Position* represent the positions of requests queued in the server; smaller values denote requests that have been received earlier and that therefore have higher priority (value C indicates "no position").

The model makes use of the following auxiliary operations:

```
fun position(c:Client,pos:Map[Client,Position]):Position =
  let p = pos[c] in
  if p < C then
    p
  else if ∀c0:Client. pos[c0] = C then
    0
  else
    1 + max c0:Client with pos[c0] < C. pos[c0];
```

The value of $position(c, pos)$ denotes the position of client c in the "queue" of requests, according to the mapping *pos* of clients to positions (this queue is purely "virtual", i.e., it is only represented by *pos*). If this mapping already contains a position for c, we use this position; otherwise, if the mapping is empty, the position is 0 (the client is at the "front" of the queue); if the mapping is not empty, the position is one plus the maximum position in the mapping (the client is at the "back" of the queue).

© 2021 Wolfgang Schreiner.

```
proc remove(pos:Map[Client,Position], c:Client):Map[Client,Position]
  ensures ∀c0:Client. result[c0] = if c0 = c then C else
    if pos[c] < pos[c0] ∧ pos[c0] < C then pos[c0]-1 else pos[c0];
{
  var p:Map[Client,Position] = pos;
  for c0:Client with p[c] < p[c0] ∧ p[c0] < C do
    p[c0] := p[c0]-1;
  p[c] := C;
  return p;
}
```

The value of $remove(pos, c)$ denotes the mapping of clients to positions that arises from *pos* if we remove client c from the queue of requests: the position of all clients before c remain unchanged while the positions of all clients after c are decreased by one; the position of c itself is set to C (“no position”).

```
pred grant(c:Client,S:Set[Resource],unsat:Map[Client,Set[Resource]],
    alloc:Map[Client,Set[Resource]],pos:Map[Client,Position])
⇔ pos[c] < C ∧ S ≠ ∅[Resource] ∧
  (∀r∈S. r∈unsat[c] ∧ ¬∃c0:Client. r∈alloc[c0]) ∧
  (∀c0:Client with pos[c0]<pos[c]. ∀r∈S. r∉unsat[c0]);
```

The truth value of $grant(c, S, unsat, alloc, pos)$ states whether the server may allocate to client c the resource set S, depending on its state described by the variables *unsat*, *alloc*, and *pos* which will be further explained below.

A Shared Model

Our first model is that of a system with nondeterministic transitions operating on a set of shared variables; in particular, the state of the network isbdefined by the set of messages that are currently in transit. These messages are formed according to the following declarations:

```
type Tag = ℕ[2];
val request = 0; val allocate = 1; val giveback = 2;
type Message = Record[tag:Tag,client:Client,resources:Set[Resource]];
```

Thus a message is a value m of type *Message* where the record field $m.tag \in \{request, allocate, giveback\}$ denotes the purpose of the message, $m.client$ denotes the client sending respectively receiving the message, and

© 2021 Wolfgang Schreiner.

m.resources denotes the set of resources carried by the message. On the basis of these declarations, the model can now be formalized as follows:

```
shared system SharedAllocator
{
  var unsat:Map[Client,Set[Resource]] =
    Map[Client,Set[Resource]](∅[Resource]);
  var alloc:Map[Client,Set[Resource]] =
     Map[Client,Set[Resource]](∅[Resource]);
  var pos:Map[Client,Position] = Map[Client,Position](C);
  var requests: Map[Client,Set[Resource]] =
    Map[Client,Set[Resource]](∅[Resource]);
  var holding: Map[Client,Set[Resource]] =
    Map[Client,Set[Resource]](∅[Resource]);
  var network: Set[Message] = ∅[Message];
  action serverRequest(m:Message) with
    m ∈ network ∧ m.tag = request ∧ alloc[m.client] = ∅[Resource];
  {
    network := network\{m}; val c = m.client; val S = m.resources;
    unsat[c] := unsat[c]∪S; pos[c] := position(c,pos);
  }
  action serverAllocate(c:Client,S:Set[Resource]) with
    grant(c,S,unsat,alloc,pos);
  {
    alloc[c] := alloc[c]∪S; unsat[c] := unsat[c]\S;
    if unsat[c] = ∅[Resource] then pos := remove(pos,c);
    network := network∪{⟨tag:allocate,client:c,resources:S⟩};
  }
  action serverGiveBack(m:Message) with m ∈ network ∧ m.tag = giveback;
  {
    network := network\{m}; val c = m.client; val S = m.resources;
    alloc[c] := alloc[c]\S;
  }
  action clientRequest(c:Client,S:Set[Resource]) with
    requests[c]=∅[Resource] ∧ holding[c]=∅[Resource] ∧ S≠∅[Resource];
  {
    requests[c] := S;
    network := network∪{⟨tag:request,client:c,resources:S⟩};
  }
  action clientAllocate(m:Message) with m ∈ network ∧ m.tag = allocate;
  {
    network := network\{m}; val c = m.client; val S = m.resources;
    requests[c] := requests[c]\S; holding[c] := holding[c]∪S;
  }
  action clientGiveBack(c:Client,S:Set[Resource]) with
    S ≠ ∅[Resource] ∧ S ⊆ holding[c];
  {
    holding[c] := holding[c]\S;
    network := network∪{⟨tag:giveback,client:c,resources:S⟩};
  }
}
```

© 2021 Wolfgang Schreiner.