Acknowledgments

These notes are based on an Alloy example in the following book.

The Problem

- We will model in Alloy the disposable card key system used in most hotels for locking and unlocking guest rooms.
- The system uses recordable locks, which prevent previous guests from entering a room once it is assigned.
- We will model both static and dynamic aspects of the system.
Problem Description [Jack06]

“[...] the hotel issues a new key to the next occupant, which recodes the lock, so that previous keys will no longer work. The lock is a simple, stand-alone unit [...] with a memory holding the current key combination. A hardware device [...] generates a sequence of pseudorandom numbers. The lock is opened either by the current key combination, or by its successor; if a key with the successor is inserted, the successor is made to be the current combination, so that the old combination will no longer be accepted.”
Problem Description [Jack06]

“This scheme requires no communication between the front desk and the door lock. By synchronizing the front desk and the door locks initially, and by using the same pseudorandom generator, the front desk can keep its records of the current combinations in step with the doors themselves.”
Signatures and Fields

- Signatures: Time, Key, Room, Guest, FrontDesk
- Key refers to the key combination stored in the magnetic strip of the card
- FrontDesk stores at any time a mapping between each room and its most recent key combination (if any), and between each room and its current guest
- Each room (lock) has an associated set of keys, and exactly one current key at a time
- Each key belongs to at most one room
module hotel
open util/ordering [Time] as TO
open util/ordering [Key] as KO

sig Key {}
sig Time {}

sig Room {
    keys: set Key,
    currentKey: keys one -> Time
}

sig Guest {
    keys: Key -> Time
}

one sig FrontDesk {
    lastKey: (Room -> lone Key) -> Time,
    occupant: (Room -> Guest) -> Time
}
Room Constraint

- Each key belongs to at most one room

```plaintext
fact {
    all k: Key | lone keys.k
}
```
New Key Generation

Given a key \( k \) and a set of keys \( ks \), the function \texttt{nextKey} returns the smallest key (in the key ordering) in \( ks \) that follows \( k \).

\begin{verbatim}
fun nextKey [k: Key, ks: set Key]: set Key
{
    KO/min [KO/nexts[k] & ks]
}
\end{verbatim}
Initial State

module examples/hotel
open util/ordering [Time] as TO
open util/ordering [Key] as KO

sig Key {}
sig Time {}

sig Room {
  keys: set Key,
  currentKey: keys one -> Time
}

sig Guest {
  keys: Key -> Time
}

one sig FrontDesk {
  lastKey: (Room -> lone Key) -> Time,
  occupant: (Room -> Guest) -> Time
}

No constraints

the record of each room’s key at the front desk is synchronized with the current combination of the lock itself

No guests have keys

No rooms are occupied
Hotel Operations: Initial State

pred init [t: Time] {
  -- no guests have keys
  no Guest.keys.t

  -- the roster at the front desk shows
  -- no room as occupied
  no FrontDesk.occupant.t

  -- the record of each room’s key at the
  -- front desk is synchronized with the
  -- current combination of the lock itself
  all r: Room |
    r.(FrontDesk.lastKey.t) = r.currentKey.t
}

pred entry [ g: Guest, r: Room, k: Key, t, t': Time ]

- Preconditions:
  - The key used to open the lock is one of the keys the guest is holding

- Pre and Post Conditions:
  - The key on the card
    - either matches the lock’s current key, and the lock remains unchanged (not a new guest), or
    - matches its successor, and the lock is advanced (new guest)

- Frame conditions:
  - no changes to the state of other rooms, or to the set of keys held by guests, or to the records at the front desk
pred entry [ g: Guest, r: Room, k: Key, t, t': Time ] {
  -- the key used to open the lock is one of
  -- the key the guest holding
  k in g.keys.t
  -- pre and post conditions
  let ck = r.currentKey |
    -- not a new guest
    (k = ck.t and ck.t' = ck.t) or
    -- new guest
    (k = nextKey [ck.t, r.keys] and ck.t' = k)
  -- frame conditions
  noRoomChangeExcept [r, t, t']
  noGuestChangeExcept [none, t, t']
  noFrontDeskChange [t, t']
}
Frame conditions

pred noFrontDeskChange [t,t': Time]
{FrontDesk.lastKey.t = FrontDesk.lastKey.t'
 FrontDesk.occupant.t = FrontDesk.occupant.t'}
pred noRoomChangeExcept [rs: set Room, t,t': Time]
{all r: Room - rs | r.currentKey.t = r.currentKey.t'}
pred noGuestChangeExcept [gs: set Guest, t,t': Time]
{all g: Guest - gs | g.keys.t = g.keys.t'}
Hotel Operations: Check-out

\[ \text{pred checkout} [ \ g: \text{Guest}, \ t,t': \text{Time} ] \]

- **Preconditions:**
  - the guest occupies one or more rooms

- **Postconditions:**
  - the guest’s rooms become available

- **Frame conditions:**
  - Nothing changes but the occupant relation
Hotel Operations: Check-out

```plaintext
one sig FrontDesk {
    lastKey: (Room -> lone Key) -> Time,
    occupant: (Room -> Guest) -> Time
}

pred checkout [ g: Guest, t,t': Time ] {
    let occ = FrontDesk.occupant | {
        -- the guest occupies one or more rooms
        some occ.t.g
        -- the guest's room become available
        occ.t' = occ.t - (Room -> g)
    }
    -- frame condition
    FrontDesk.lastKey.t = FrontDesk.lastKey.t'
    noRoomChangeExcept [none,t, t']
    noGuestChangeExcept [none,t, t']
}
```
Hotel Operations: Check-in

pred checkin [ g: Guest, r: Room, k: Key t, t': Time ]

- Preconditions:
  - the room is available
  - the input key is the successor of the last key in the sequence associated to the room

- Postconditions:
  - the guest holds the input key and becomes the new occupant of the room
  - the input key becomes the room’s current key

- Frame conditions:
  - Nothing changes but the occupant relation and the guest’s relations
Hotel Operations: Check-in

pred checkin [ g: Guest, r: Room, k: Key, t,t': Time ] {  
  -- the guest holds the input key  
g.keys.t' = g.keys.t + k

let occ = FrontDesk.occupant | {  
    -- the room has no current occupant  
    no r.occ.t  
    -- the guest becomes the new occupant of the room  
    occ.t' = occ.t + r->g  
}

let lk = FrontDesk.lastKey | {  
    -- the input key becomes the room's current key  
lk.t' = lk.t ++ r->k  
    -- the input key is the successor of the last key in  
    -- the sequence associated to the room  
k = nextKey [r.lk.t, r.keys]  
}

noRoomChangeExcept [none, t, t']
noGuestChangeExcept [g, t, t']
}
Trace generation

- The first time step satisfies the initialization conditions
- Any pair of consecutive time steps are related by
  - an entry operation, or
  - a check-in operation, or
  - a check-out operation
**Trace generation**

```plaintext
fact Traces {
    init [TO/first]
    all t: Time - TO/last |
    let t' = TO/next [t] |
    some g: Guest, r: Room, k: Key |
    entry [g, r, k, t, t'] or
    checkin [g, r, k, t, t'] or
    checkout [g, t, t']
}
```
Analysis

- No unauthorized entries are possible:
  - If a guest $g$ enters room $r$ at time $t$, and the front desk records show $r$ as occupied at that time, then $g$ must be a recorded occupant of $r$.

```plaintext
assert noBadEntry {
  all t: Time, r: Room, g: Guest, k: Key |
  let t' = TO/next [t],
      o = r.FrontDesk.occupant.t |
      (entry [g, r, k, t, t'] and some o)
  implies g in o
}
```
Analysis

check noBadEntry for 3
but 2 Room, 2 Guest, 5 Time

- It is enough to check for problem already with just 2 guests and 2 rooms

- Time’s scope must be at least 5 because at least 4 time steps are needed to execute each operation once.

- There is a counter-example
Initially, the current key of Room is Key0, which is also reflected in the front desk’s record.
Guest1 checks in to Room and receives key Key1; the occupancy roster at the front desk is updated accordingly; Key1 is recorded as the last key assigned to Room.
Guest1 checks out, and the occupancy roster is cleared
Guest0 checks in to Room and receives key Key2; the occupancy roster at the front desk is updated accordingly; Key2 is recorded as the last key assigned to Room
Guest1 presents Key1 to the lock of Room, and is admitted.
Necessary Restriction

- There is no intervening operations between a guest’s check-in and room entry.

```plaintext
fact noIntervening {
  all t: Time - TO/last | 
  let t' = TO/next [t], t'' = TO/next [t'] | 
  all g: Guest, r: Room, k: Key | 
  checkin [g, r, k, t, t'] implies 
  ( 
    entry [g, r, k, t', t''] or 
    no t"
  )
}
```
Analysis

- We check once again:
  ```
  check noBadEntry for 3
  but 2 Room, 2 Guest, 5 Time
  ```
  - No counter-example

- For greater confidence, we increase the scope:
  ```
  check noBadEntry for 5
  but 3 Room, 3 Guest, 9 Time
  ```
  - No counter-examples