

# CS:5810 Formal Methods in Software Engineering

## Case Study: Hotel Lock System

*Copyright 2001-22, Matt Dwyer, John Hatcliff, Rod Howell, Laurence Pilard, and Cesare Tinelli.*

*Created by Cesare Tinelli and Laurence Pilard at the University of Iowa from notes originally developed by Matt Dwyer, John Hatcliff, Rod Howell at Kansas State University. These notes are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.*

# Acknowledgments

These notes are based on an Alloy example in the book:

**[Jack06]** Daniel Jackson. *Software abstractions – Logic, Language, and Analysis*. The MIT press, 2006

# The Task

- 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 its has been re-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 [...] [within the lock] generates a sequence of pseudorandom numbers.”

# Problem Description [Jack06]

"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.

This scheme requires no communication between the front desk and the door lock."

# Problem Description [Jack06]

“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: **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

# Signatures and Fields

Room refers to the room lock

Each room (lock) has

- an associated set of possible keys, and
- exactly one current key at a time

Each key belongs to at most one room

Each guest has zero or more keys at any time



# Signatures and Fields

```
module hotel  
open util/ordering [Key] as K0
```

# Signatures and Fields

```
module hotel
open util/ordering [Key] as KO
sig Key {}

sig Room {
  keys: set Key,
  var currentKey: Key
}

sig Guest {
  var keys: set Key
}

one sig FrontDesk {
  var lastKey: Room -> lone Key,
  var occupant: Room -> Guest
}
```

```
fun sig FDlastKey : Room -> Key {
  FrontDesk.lastKey
}

fun sig FDoccupant: Room -> Guest {
  FrontDesk.occupant
}
```

# Room Constraint

Each key belongs to at most one room

```
fact {  
  all k: Key | lone (Room <: keys).k  
}
```

```
module hotel  
open util/ordering [Key] as K0  
sig Key {}  
  
sig Room {  
  keys: set Key,  
  var currentKey: Key  
}  
  
sig Guest { var keys: set Key }  
  
one sig FrontDesk {  
  var lastKey: Room -> lone Key,  
  var occupant: Room -> Guest  
}  
  
fun sig FDlastKey : Room -> Key {  
  FrontDesk.lastKey  
}  
  
fun sig FDoccupant: Room -> Guest  
  {  
    FrontDesk.occupant  
  }
```

# New Key Generation

Given a key `k` and a set `ks` of keys,

`nextKey` returns the smallest key (in the key ordering) in `ks` that follows `k`

```
fun nextKey [k: Key, ks: set Key]: set Key
{
    KO/min [KO/nexsts[k] & ks]
}
```

# Initial State

```
module examples/hotel
open util/ordering [Key] as K0

sig Key {}
```

```
sig Room {
  keys: set Key,
  var currentKey: Key
}
```

```
sig Guest {
  var keys: set Key } No guests have keys
}
```

```
one sig FrontDesk {
  var lastKey: Room -> lone Key,
  var occupant: Room -> Guest
}
```

) *No constraints*

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

) *No rooms are occupied*

# Hotel Operations: Initial State

```
pred init {  
  -- no guests have keys  
  no Guest.keys  
  
  -- the roster at the front desk shows  
  -- no room as occupied  
  no FDoccupant  
  
  -- 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.FDlastKey = r.currentKey  
}
```

# Hotel Operations: Guest Entry

pred entry [ g: Guest, r: Room, k: Key ]

- 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, to the set of keys held by guests, or to the records at the front desk

# Hotel Operations: Guest Entry

```
pred entry [ g: Guest, r: Room, k: Key ] {  
  -- preconditions  
  -- the key used to open the lock is one of the keys  
  -- held by the guest  
  k in g.keys  
  
  -- pre and post conditions  
  ( -- not a new guest  
    k = r.currKey and r.currKey' = r.currKey  
    or  
    -- new guest  
    k = nextKey[r.currKey, r.keys] and r.currKey' = k  
  )  
  
  -- frame conditions  
  noFrontDeskChange  
  noRoomChange[Room - r]  
  noGuestChange[Guest]  
}
```



# Hotel Operations: Check-out

pred checkOut [ g: Guest ]

- 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

```
pred checkOut [g: Guest] {  
  -- preconditions  
  -- the guest occupies one or more rooms  
  some FDoccupant.g  
  
  -- postconditions  
  -- the guest's rooms become available  
  FDoccupant' = FDoccupant - (Room -> g)  
  
  -- frame conditions  
  FDtoLastKey' = FDtoLastKey  
  noRoomChange[Room]  
  noGuestChange[Guest]  
}
```

# Hotel Operations: Check-in

pred checkIn [ g: Guest, r: Room, k: Key ]

- 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 ] {
  -- preconditions
  -- the room has no current occupant
  no r.FDoccupant
  -- the input key is the successor of the last key in
  -- the sequence associated to the room
  k = nextKey[r.FDlastKey, r.keys]

  -- postconditions
  -- the guest becomes the new occupant of the room
  FDoccupant' = FDoccupant + (r -> g)
  -- the guest holds the input key
  g.keys' = g.keys + k
  -- the input key becomes the room's current key
  FDlastKey' = FDlastKey ++ (r -> k)

  -- frame conditions
  noRoomChange[Room]
  noGuestChange[Guest - g] }
```

# 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

```
fact Traces {  
  init  
  
  always  
    some g: Guest, r: Room, k: Key |  
      entry[g, r, k] or  
      checkin[g, r, k] or  
      checkout[g]  
}
```

# Analysis

Let's check if 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  $r$ .

```
assert noBadEntry {  
  always all r: Room, g: Guest, k: Key |  
    let o = r.FDoccupant |  
      (entry[g, r, k] and some o) implies g in o  
}
```

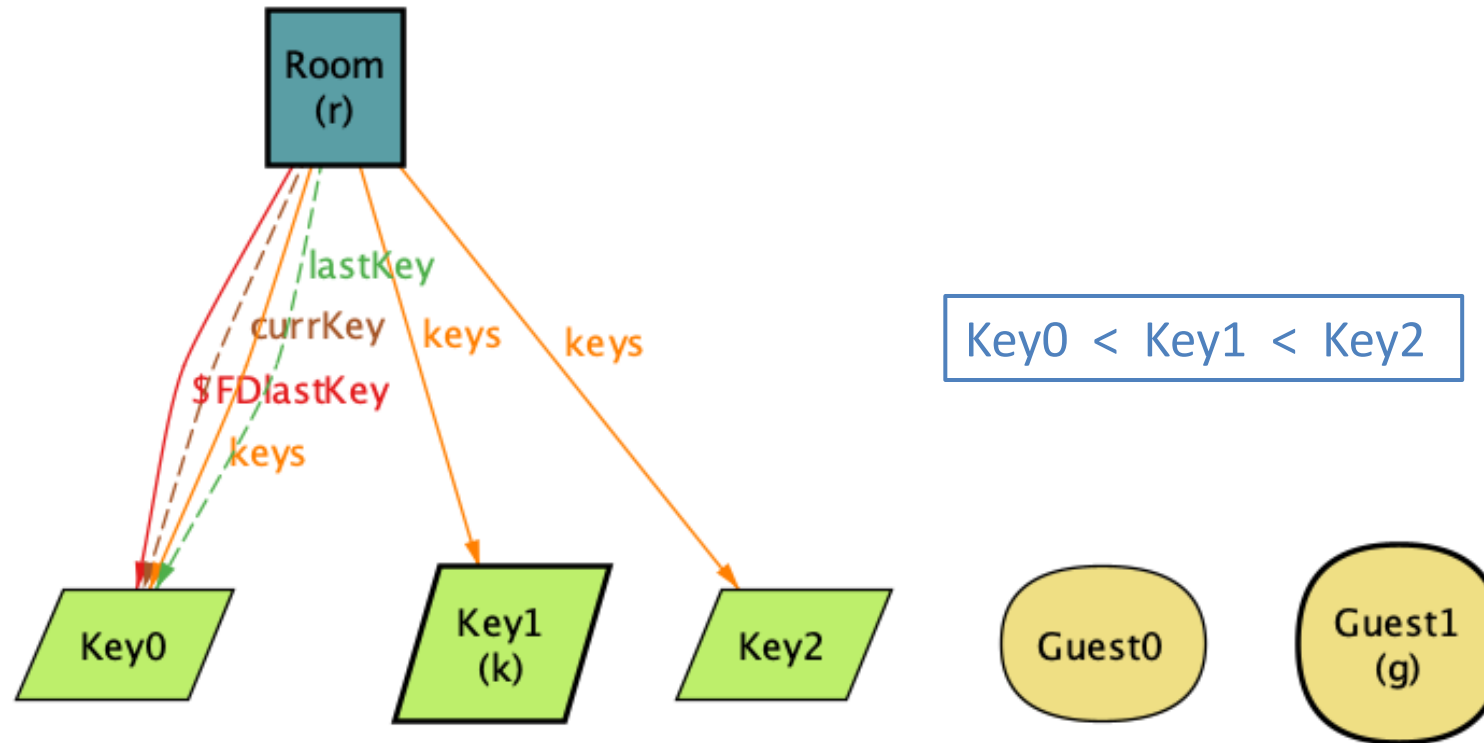
# Analysis

check noBadEntry for 3  
but 1 Room, 2 Guest, 5 steps

- It is enough to check for problem already with just 2 guests and 2 rooms
- `steps`'s scope must be at least 5 because at least 4 steps are needed to execute each operation once
- There is a **counterexample**  
(see file `dynamic/hotel1-elec.a1s`)

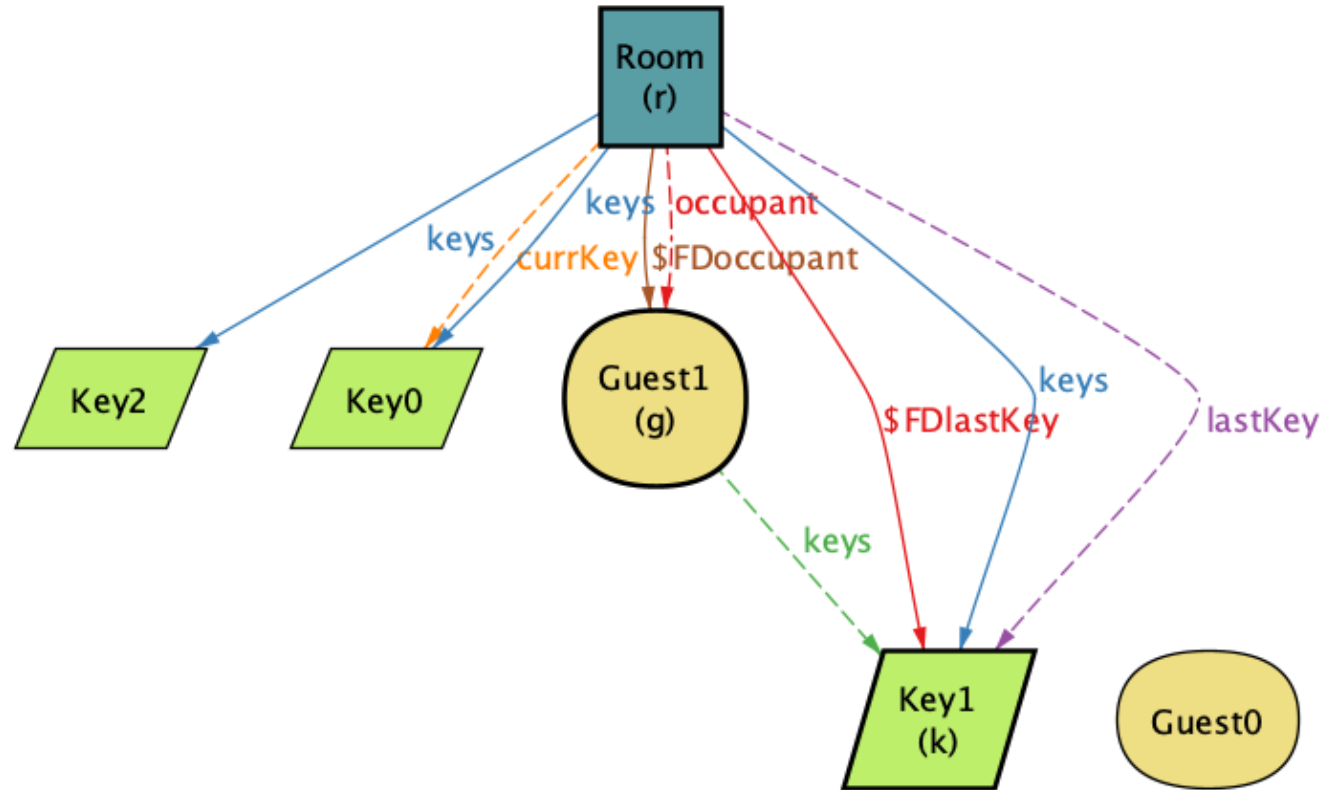


# 0: Initial State



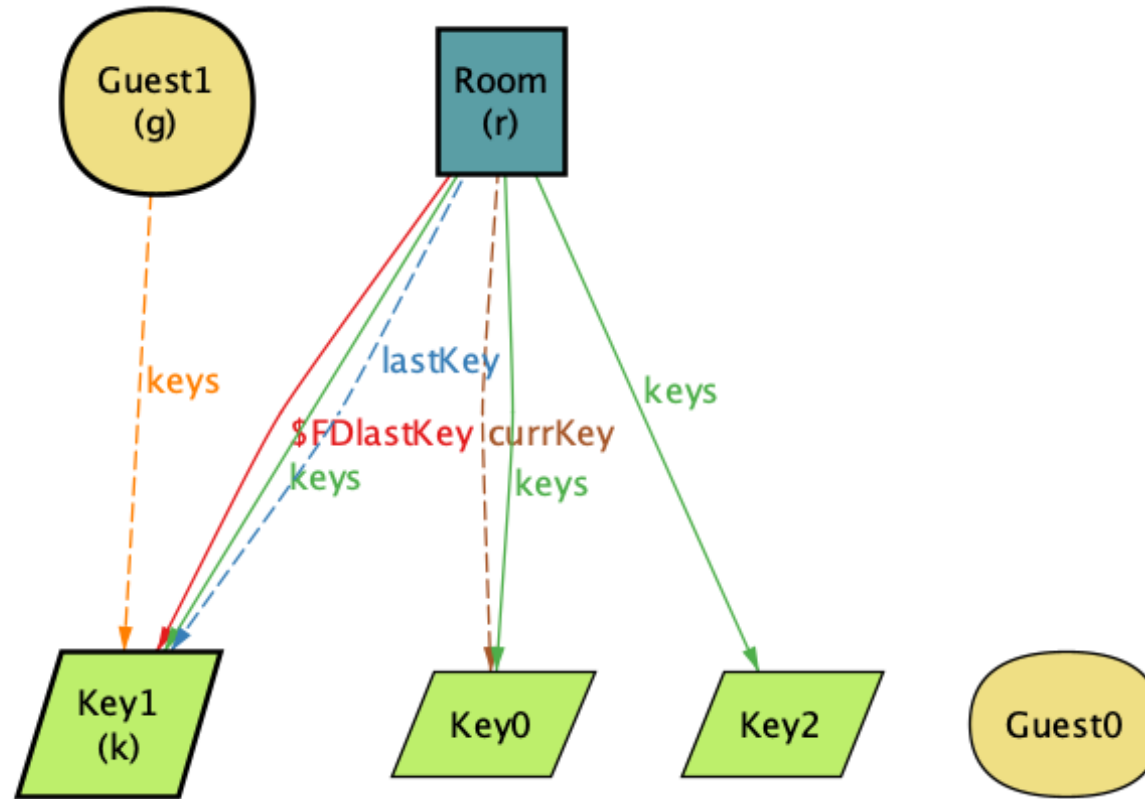
- initially, the current key of Room is Key0
- this is also reflected in the front desk's record

# 1: CheckIn Operation



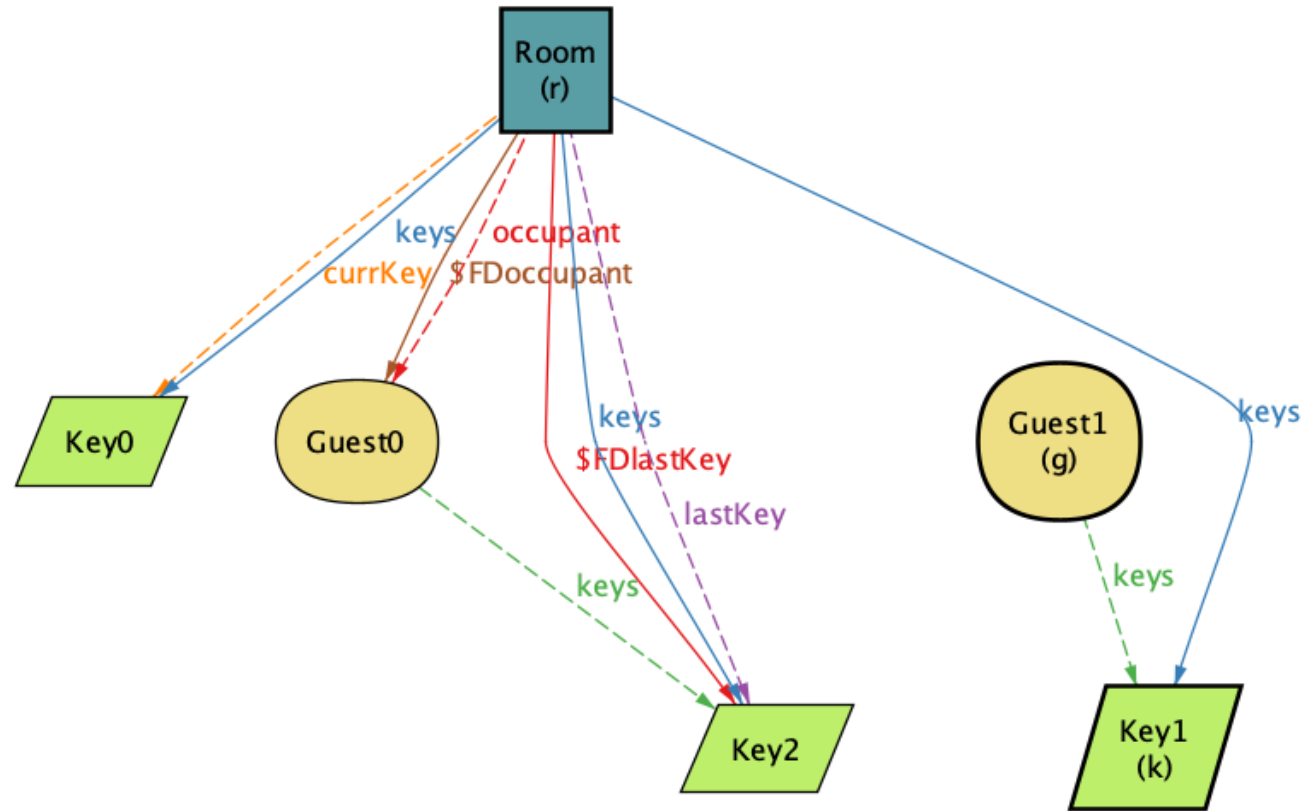
- Guest1 checks in to Room and receives key Key1
- the occupancy roster at the front desk is updated accordingly
- Key1 is recorded as the last key assigned to Room

## 2: CheckOut Operation



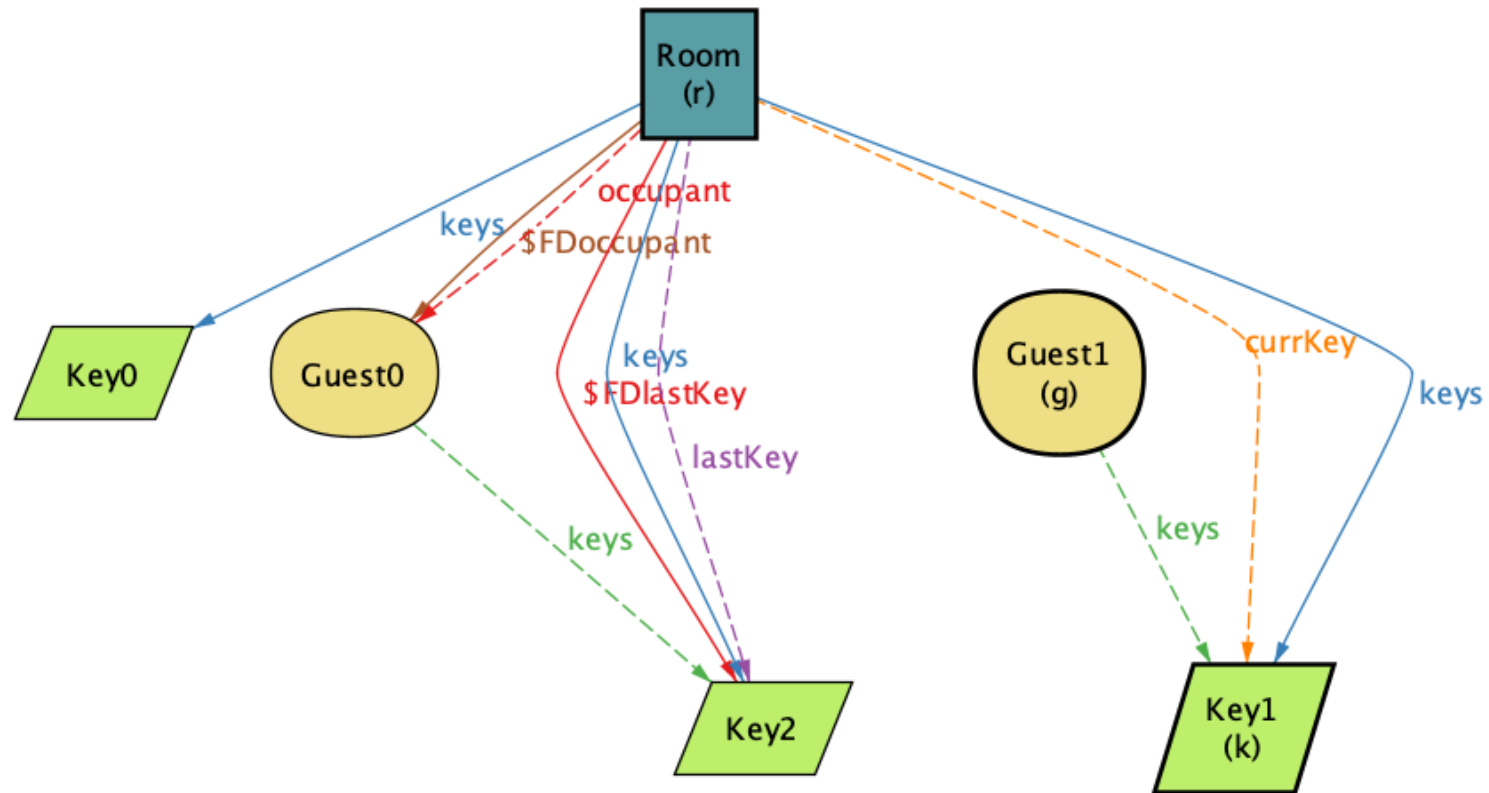
- Guest1 checks out, and the occupancy roster is cleared
- Since Guest1 never entered, Room's current key is still Key0
- Guest1 still holds Key1

# 3: CheckIn Operation



- Guest0 checks into 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

# 4: Entry Operation



- Guest1 presents Key1 to the lock of Room
- Since Room's current key is still Key0, Guest1 is admitted

# Necessary Restriction

There must be **no intervening operation** between a guest's check-in and room entry

```
pred noInterveningOps {  
  always  
    all g: Guest, r: Room, k: Key |  
      checkIn[g, r, k] implies after entry[g, r, k]  
}
```

# Conditional Assertion

Make assertion under `noInterveningOps` assumption

```
assert noBadEntry {
```

```
  noInterveningOps implies
```

```
    always all r: Room, g: Guest, k: Key |
```

```
      let o = r.FrontDesk.occupant |
```

```
        (entry[g, r, k] and some o) implies g in o
```

```
}
```

# Analysis

- We check once again:
  - check noBadEntry for 3  
but 2 Room, 2 Guest, 5 steps
  - No counter-example (see file `dynamic/hotel2-elec.a1s`)
- For greater confidence, we increase the scope:
  - check noBadEntry for 5  
but 3 Room, 3 Guest, 20 steps
  - No counterexamples