

Verifying a Hotel Key Card System*

Tobias Nipkow
Institut für Informatik, TU München

December 12, 2009

Abstract

Two models of an electronic hotel key card system are contrasted: a state based and a trace based one. Both are defined, verified, and proved equivalent in the theorem prover Isabelle/HOL. It is shown that if a guest follows a certain safety policy regarding her key cards, she can be sure that nobody but her can enter her room.

1 Introduction

This paper presents two models for a hotel key card system and the verification of their safety (in Isabelle/HOL [6]). The models are based on Section 6.2, *Hotel Room Locking*, and Appendix E in the book by Daniel Jackson [2]. Jackson employs his Alloy system to check that there are no small counterexamples to safety. We confirm his conjecture of safety by a formal proof.

Most hotels operate a digital key card system. Upon check-in, you receive a card with your own key on it (typically a pseudorandom number). The lock for each room reads your card and opens the door if the key is correct. The system is decentralized, i.e. each lock is a stand-alone, battery-powered device without connection to the computer at reception or to any other device. So how does the lock know that your key is correct? There are a number of similar systems and we discuss the one described in Appendix E of [2]. Here each card carries two keys: the old key of the previous occupant of the room (key_1), and your own new key (key_2). The lock always holds one key, its “current” key. When you enter your room for the first time, the lock notices that its current key is key_1 on your card and recodes itself, i.e. it replaces its own current key with key_2 on your card. When you enter the next time, the lock finds that its current key is equal to your key_2 and opens the door without recoding itself. Your card is never modified by the lock. Eventually, a new guest with a new key enters the room, recodes the lock, and you cannot enter anymore.

After a short introduction of the notation we discuss two very different specifications, a state based and a trace based one, and prove their safety and their equivalence. The complete formalization is available online in the *Archive of Formal Proofs* at afp.sf.net.

*Appeared in proceedings of ICTAC 2006 [5]

1.1 Notation

HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types with their primitive operations.

Types The type of truth values is called *bool*. The space of total functions is denoted by \Rightarrow . Type variables start with a quote, as in *'a*, *'b* etc. The notation $t::\tau$ means that term t has type τ .

Functions can be updated at x with new value y , written $f(x := y)$. The range of a function is *range f*, *inj f* means f is injective.

Pairs come with the two projection functions $fst :: 'a \times 'b \Rightarrow 'a$ and $snd :: 'a \times 'b \Rightarrow 'b$.

Sets have type *'a set*.

Lists (type *'a list*) come with the empty list $[]$, the infix constructor \cdot , the infix $@$ that appends two lists, and the conversion function *set* from lists to sets. Variable names ending in “s” usually stand for lists.

Records are constructed like this $(\{f_1 = v_1, \dots\})$ and updated like this $r(\{f_i := v_i, \dots\})$, where the f_i are the field names, the v_i the values and r is a record.

Datatype *option* is defined like this

$$\mathbf{datatype} \ 'a \ option = None \mid Some \ 'a$$

and adjoins a new element *None* to a type *'a*. For succinctness we write $[a]$ instead of *Some a*.

Note that $\llbracket A_1; \dots; A_n \rrbracket \Longrightarrow A$ abbreviates $A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow A$, which is the same as “If A_1 and \dots and A_n then A ”.

2 A state based model

The model is based on three opaque types *guest*, *key* and *room*. Type *card* is just an abbreviation for $key \times key$.

The state of the system is modelled as a record which combines the information about the front desk, the rooms and the guests.

```
record state =  
  owns :: room  $\Rightarrow$  guest option  
  currk :: room  $\Rightarrow$  key  
  issued :: key set  
  cards :: guest  $\Rightarrow$  card set  
  roomk :: room  $\Rightarrow$  key  
  isin :: room  $\Rightarrow$  guest set
```

$safe :: room \Rightarrow bool$

Reception records who *owns* a room (if anybody, hence *guest option*), the current key *currk* that has been issued for a room, and which keys have been *issued* so far. Each guest has a set of *cards*. Each room has a key *roomk* recorded in the lock and a set *isin* of occupants. The auxiliary variable *safe* is explained further below; we ignore it for now.

In specification languages like Z, VDM and B we would now define a number of operations on this state space. Since they are the only permissible operations on the state, this defines a set of *reachable* states. In a purely logical environment like Isabelle/HOL this set can be defined directly by an inductive definition. Each clause of the definition corresponds to a transition/operation/event. This is the standard approach to modelling state machines in theorem provers.

The set of reachable states of the system (called *reach*) is defined by four transitions: initialization, checking in, entering a room, and leaving a room:

init:

$inj\ initk \Longrightarrow$
 $(\ \mathit{owns} = (\lambda r. None),\ \mathit{currk} = \mathit{initk},\ \mathit{issued} = range\ \mathit{initk},$
 $\ \mathit{cards} = (\lambda g. \{\}),\ \mathit{roomk} = \mathit{initk},\ \mathit{isin} = (\lambda r. \{\}),$
 $\ \mathit{safe} = (\lambda r. True)\) \in reach$

| *check-in:*

$\llbracket s \in reach; k \notin issued\ s \rrbracket \Longrightarrow$
 $s(\ \mathit{currk} := (\mathit{currk}\ s)(r := k),\ \mathit{issued} := \mathit{issued}\ s \cup \{k\},$
 $\ \mathit{cards} := (\mathit{cards}\ s)(g := \mathit{cards}\ s\ g \cup \{(\mathit{currk}\ s\ r, k)\}),$
 $\ \mathit{owns} := (\mathit{owns}\ s)(r := Some\ g),$
 $\ \mathit{safe} := (\mathit{safe}\ s)(r := False)\) \in reach$

| *enter-room:*

$\llbracket s \in reach; (k, k') \in \mathit{cards}\ s\ g; \mathit{roomk}\ s\ r \in \{k, k'\} \rrbracket \Longrightarrow$
 $s(\ \mathit{isin} := (\mathit{isin}\ s)(r := \mathit{isin}\ s\ r \cup \{g\}),$
 $\ \mathit{roomk} := (\mathit{roomk}\ s)(r := k'),$
 $\ \mathit{safe} := (\mathit{safe}\ s)(r := \mathit{owns}\ s\ r = \lfloor g \rfloor \wedge \mathit{isin}\ s\ r = \{\} \wedge k' = \mathit{currk}\ s\ r$
 $\ \vee \mathit{safe}\ s\ r)$
 $\) \in reach$

| *exit-room:*

$\llbracket s \in reach; g \in \mathit{isin}\ s\ r \rrbracket \Longrightarrow$
 $s(\ \mathit{isin} := (\mathit{isin}\ s)(r := \mathit{isin}\ s\ r - \{g\})\) \in reach$

There is no check-out event because it is implicit in the next check-in for that room: this covers the cases where a guest leaves without checking out (in which case the room should not be blocked forever) or where the hotel decides to rent out a room prematurely, probably by accident. Neither do guests have to return their cards at any point because they may lose cards or may pretend to have lost them. We will now explain the events.

init Initialization requires that every room has a different key, i.e. that *currk* is injective. Nobody owns a room, the keys of all rooms are recorded as issued, nobody has a card, and all rooms are empty.

enter-room A guest may enter if either of the two keys on his card equal the room key. Then *g* is added to the occupants of *r* and the room key is set

to the second key on the card. Normally this has no effect because the second key is already the room key. But when entering for the first time, the first key on the card equals the room key and then the lock is actually recoded.

exit-room removes an occupant from the occupants of a room.

check-in for room r and guest g issues the card ($currk\ s\ r, k$) to g , where k is new, makes g the owner of the room, and sets *currk s r* to the new key k .

The reader can easily check that our specification allows the intended distributed implementation: entering only reads and writes the key in that lock, and check-in only reads and writes the information at reception.

In contrast to Jackson we require that initially distinct rooms have distinct keys. This protects the hotel from its guests: otherwise a guest may be able to enter rooms he does not own, potentially stealing objects from those rooms. Of course he can also steal objects from his own room, but in that case it is easier to hold him responsible. In general, the hotel may just want to minimize the opportunity for theft.

The main difference to Jackson’s model is that his can talk about transitions between states rather than merely about reachable states. This means that he can specify that unauthorized entry into a room should not occur. Because our specification does not formalize the transition relation itself, we need to include the *isin* component in order to express the same requirement. In the end, we would like to establish that the system is *safe*: only the owner of a room can be in a room:

$$\llbracket s \in reach; g \in isin\ s\ r \rrbracket \implies owns\ s\ r = \lfloor g \rfloor$$

Unfortunately, this is just not true. It does not take a PhD in computer science to come up with the following scenario: because guests can retain their cards, there is nothing to stop a guest from reentering his old room after he has checked out (in our model: after the next guest has checked in), but before the next guest has entered his room. Hence the best we can do is to prove a conditional safety property: under certain conditions, the above safety property holds. The question is: which conditions? It is clear that the room must be empty when its owner enters it, or all bets are off. But is that sufficient? Unfortunately not. Jackson’s Alloy tool took 2 seconds [2, p. 303] to find the following “guest-in-the-middle” attack:

1. Guest 1 checks in and obtains a card (k_1, k_2) for room 1 (whose key in the lock is k_1). Guest 1 does not enter room 1.
2. Guest 2 checks in, obtains a card (k_2, k_3) for room 1, but does not enter room 1 either.
3. Guest 1 checks in again, obtains a card (k_3, k_4), goes to room 1, opens it with his old card (k_1, k_2), finds the room empty, and feels safe . . .

After Guest 1 has left his room, Guest 2 enters and makes off with the luggage.

Jackson now assumes that guests return their cards upon check-out, which can be modelled as follows: upon check-in, the new card is not added to the guest’s set of cards but it replaces his previous set of cards, i.e. guests return

old cards the next time they check in. Under this assumption, Alloy finds no more counterexamples to safety — at least not up to 6 cards and keys and 3 guests and rooms. This is not a proof but a strong indication that the given assumptions suffice for safety. We prove that this is indeed the case.

It should be noted that the system also suffers from a liveness problem: if a guest never enters the room he checked in to, that room is forever blocked. In practice this is dealt with by a master key. We ignore liveness.

2.1 Formalizing safety

It should be clear that one cannot force guests to always return their cards (or, equivalently, never to use an old card). We can only prove that if they do, their room is safe. However, we do not follow Jackson’s approach of globally assuming everybody returns their old cards upon check-in. Instead we would like to take a local approach where it is up to each guest whether he follows this safety policy. We allow guests to keep their cards but make safety dependent on how they use them. This generality requires a finer grained model: we need to record if a guest has entered his room in a safe manner, i.e. if it was empty and if he used the latest key for the room, the one stored at reception. The auxiliary variable *safe* records for each room if this was the case at some point between his last check-in and now. The main theorem will be that if a room is safe in this manner, then only the owner can be in the room. Now we explain how *safe* is modified with each event:

init sets *safe* to *True* for every room.

check-in for room *r* resets *safe s r* because it is not safe for the new owner yet.

enter-room for room *r* sets *safe s r* if the owner entered an empty room using the latest card issued for that room by reception, or if the room was already safe.

exit-room does not modify *safe*.

The reader should convince his or herself that *safe* corresponds to the informal safety policy set out above. Note that a guest may find his room non-empty the first time he enters, and *safe* will not be set, but he may come back later, find the room empty, and then *safe* will be set. Furthermore, it is important that *enter-room* cannot reset *safe* due to the disjunct $\vee \text{safe } s r$. Hence *check-in* is the only event that can reset *safe*. That is, a room stays safe until the next *check-in*. Additionally *safe* is initially *True*, which is fine because initial injectivity of *initk* prohibits illegal entries by non-owners.

Note that because none of the other state components depend on *safe*, it is truly auxiliary: it can be deleted from the system and the same set of reachable states is obtained, modulo the absence of *safe*.

We have formalized a very general safety policy of always using the latest card. A special case of this policy is the one called *NoIntervening* by Jackson [2, p. 200]: every *check-in* must immediately be followed by the corresponding *enter-room*.

<proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof>

2.2 Verifying safety

All of our lemmas are invariants of *reach*. The complete list, culminating in the main theorem, is this:

- Lemma 1**
1. $s \in reach \implies currk\ s\ r \in issued\ s$
 2. $\llbracket s \in reach; (k, k') \in cards\ s\ g \rrbracket \implies k \in issued\ s$
 3. $\llbracket s \in reach; (k, k') \in cards\ s\ g \rrbracket \implies k' \in issued\ s$
 4. $s \in reach \implies roomk\ s\ k \in issued\ s$
 5. $s \in reach \implies \forall r\ r'. (currk\ s\ r = currk\ s\ r') = (r = r')$
 6. $s \in reach \implies (currk\ s\ r, k') \notin cards\ s\ g$
 7. $\llbracket s \in reach; (k_1, k) \in cards\ s\ g_1; (k_2, k) \in cards\ s\ g_2 \rrbracket \implies g_1 = g_2$
 8. $\llbracket s \in reach; safe\ s\ r \rrbracket \implies roomk\ s\ r = currk\ s\ r$
 9. $\llbracket s \in reach; safe\ s\ r; (k', roomk\ s\ r) \in cards\ s\ g \rrbracket \implies owns\ s\ r = \lfloor g \rfloor$

Theorem 1 *If $s \in reach$ and $safe\ s\ r$ and $g \in isin\ s\ r$ then $owns\ s\ r = \lfloor g \rfloor$.*

The lemmas and the theorem are proved in this order, each one is marked as a simplification rule, and each proof is a one-liner: induction on $s \in reach$ followed by *auto*.

Although, or maybe even because these proofs work so smoothly one may like to understand why. Hence we examine the proof of Theorem 1 in more detail. The only interesting case is *enter-room*. We assume that guest g_1 enters room r_1 with card (k_1, k_2) and call the new state t . We assume $safe\ t\ r$ and $g \in isin\ t\ r$ and prove $owns\ t\ r = \lfloor g \rfloor$ by case distinction. If $r_1 \neq r$, the claim follows directly from the induction hypothesis using $safe\ s\ r$ and $g \in isin\ t\ r$ because $owns\ t\ r = owns\ s\ r$ and $safe\ t\ r = safe\ s\ r$. If $r_1 = r$ then $g \in isin\ t\ r$ is equivalent with $g \in isin\ s\ r \vee g = g_1$. If $g \in isin\ s\ r$ then $safe\ s\ r$ follows from $safe\ t\ r$ by definition of *enter-room* because $g \in isin\ s\ r$ implies $isin\ s\ r \neq \emptyset$. Hence the induction hypothesis implies the claim. If $g = g_1$ we make another case distinction. If $k_2 = roomk\ s\ r$, the claim follows immediately from Lemma 1.9 above: only the owner of a room can possess a card where the second key is the room key. If $k_1 = roomk\ s\ r$ then, by definition of *enter-room*, $safe\ t\ r$ implies $owns\ s\ r = \lfloor g \rfloor \vee safe\ s\ r$. In the first case the claim is immediate. If $safe\ s\ r$ then $roomk\ s\ r = currk\ s\ r$ (by Lemma 1.8) and thus $(currk\ s\ r, k_2) \in cards\ s\ g$ by assumption $(k_1, k_2) \in cards\ s\ g_1$, thus contradicting Lemma 1.6.

This detailed proof shows that a number of case distinctions are required. Luckily, they all suggest themselves to Isabelle via the definition of function update ($:=$) or via disjunctions that arise automatically.

<proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof>

2.3 An extension

To test the flexibility of our model we extended it with the possibility for obtaining a new card, e.g. when one has lost one's card. Now reception needs to remember not just the current but also the previous key for each room, i.e. a new field $prevk :: room \Rightarrow key$ is added to $state$. It is initialized with the same value as $currk$: though strictly speaking it could be arbitrary, this permits the convenient invariant $prevk\ s\ r \in issued\ s$. Upon check-in we set $prevk$ to $(prevk\ s)(r := currk\ s\ r)$. Event $new-card$ is simple enough:

$$\begin{aligned} & \llbracket s \in reach; owns\ s\ r = \lfloor g \rfloor \rrbracket \\ \implies & s(\{cards := (cards\ s)(g := cards\ s\ g \cup \{(prevk\ s\ r, currk\ s\ r)\})\}) \in reach \end{aligned}$$

The verification is not seriously affected. Some additional invariants are required

$$\begin{aligned} & s \in reach \implies prevk\ s\ r \in issued\ s \\ \llbracket s \in reach; owns\ s\ r' = \lfloor g \rfloor \rrbracket & \implies currk\ s\ r \neq prevk\ s\ r' \\ \llbracket s \in reach; owns\ s\ r = \lfloor g \rfloor; g \neq g' \rrbracket & \implies (k, currk\ s\ r) \notin cards\ s\ g' \end{aligned}$$

but the proofs are still of the same trivial induct-auto format.

Adding a further event for losing a card has no impact at all on the proofs.

3 A trace based model

The only clumsy aspect of the state based model is *safe*: we use a state component to record if the sequence of events that lead to a state satisfies some property. That is, we simulate a condition on traces via the state. Unsurprisingly, it is not trivial to convince oneself that *safe* really has the informal meaning set out at the beginning of subsection 2.1. Hence we now describe an alternative, purely trace based model, similar to Paulson's inductive protocol model [7]. The events are:

datatype *event* =
Check-in guest room card | *Enter guest room card* | *Exit guest room*

Instead of a state, we have a trace, i.e. list of events, and extract the state from the trace:

initk :: *room* \Rightarrow *key*
owns :: *event list* \Rightarrow *room* \Rightarrow *guest option*
currk :: *event list* \Rightarrow *room* \Rightarrow *key*
issued :: *event list* \Rightarrow *key set*
cards :: *event list* \Rightarrow *guest* \Rightarrow *card set*
roomk :: *event list* \Rightarrow *room* \Rightarrow *key*
isin :: *event list* \Rightarrow *room* \Rightarrow *guest set*
hotel :: *event list* \Rightarrow *bool*

Except for *initk*, which is completely unspecified, all these functions are defined by primitive recursion over traces:

owns [] *r* = *None*
owns (*e* · *s*) *r* =
 (case *e* of *Check-in g r' c* \Rightarrow if *r'* = *r* then $\lfloor g \rfloor$ else *owns s r*
 | - \Rightarrow *owns s r*)

$$\begin{aligned}
& \text{currk } [] \ r = \text{initk } r \\
& \text{currk } (e \cdot s) \ r = \\
& (\text{let } k = \text{currk } s \ r \\
& \text{in case } e \text{ of } \text{Check-in } g \ r' \ c \Rightarrow \text{if } r' = r \text{ then } \text{snd } c \text{ else } k \mid - \Rightarrow k) \\
\\
& \text{issued } [] = \text{range } \text{initk} \\
& \text{issued } (e \cdot s) = \text{issued } s \cup (\text{case } e \text{ of } \text{Check-in } g \ r \ c \Rightarrow \{\text{snd } c\} \mid - \Rightarrow \emptyset) \\
\\
& \text{cards } [] \ g = \emptyset \\
& \text{cards } (e \cdot s) \ g = \\
& (\text{let } C = \text{cards } s \ g \\
& \text{in case } e \text{ of } \text{Check-in } g' \ r \ c \Rightarrow \text{if } g' = g \text{ then } \{c\} \cup C \text{ else } C \mid - \Rightarrow C) \\
\\
& \text{roomk } [] \ r = \text{initk } r \\
& \text{roomk } (e \cdot s) \ r = \\
& (\text{let } k = \text{roomk } s \ r \\
& \text{in case } e \text{ of } \text{Enter } g \ r' \ (x, y) \Rightarrow \text{if } r' = r \text{ then } y \text{ else } k \mid - \Rightarrow k) \\
\\
& \text{isin } [] \ r = \emptyset \\
& \text{isin } (e \cdot s) \ r = \\
& (\text{let } G = \text{isin } s \ r \\
& \text{in case } e \text{ of } \text{Check-in } g \ r \ c \Rightarrow G \\
& \quad \mid \text{Enter } g \ r' \ c \Rightarrow \text{if } r' = r \text{ then } \{g\} \cup G \text{ else } G \\
& \quad \mid \text{Exit } g \ r' \Rightarrow \text{if } r' = r \text{ then } G - \{g\} \text{ else } G)
\end{aligned}$$

However, not every trace is possible. Function *hotel* tells us which traces correspond to real hotels:

$$\begin{aligned}
& \text{hotel } [] = \text{True} \\
& \text{hotel } (e \cdot s) = \\
& (\text{hotel } s \wedge \\
& (\text{case } e \text{ of } \text{Check-in } g \ r \ (k, k') \Rightarrow k = \text{currk } s \ r \wedge k' \notin \text{issued } s \\
& \quad \mid \text{Enter } g \ r \ (k, k') \Rightarrow (k, k') \in \text{cards } s \ g \wedge \text{roomk } s \ r \in \{k, k'\} \\
& \quad \mid \text{Exit } g \ r \Rightarrow g \in \text{isin } s \ r))
\end{aligned}$$

Alternatively we could have followed Paulson [7] in defining *hotel* as an inductive set of traces. The difference is only slight.

3.1 Formalizing safety

The principal advantage of the trace model is the intuitive specification of safety. Using the auxiliary predicate *no-Check-in*

$$\text{no-Check-in } s \ r \equiv \neg(\exists g \ c. \text{Check-in } g \ r \ c \in \text{set } s)$$

we define a trace to be *safe₀* for a room if the card obtained at the last *Check-in* was later actually used to *Enter* the room:

$$\begin{aligned}
& \text{safe}_0 \ s \ r = (\exists s_1 \ s_2 \ s_3 \ g \ c. \\
& \quad s = s_3 \ @ \ [\text{Enter } g \ r \ c] \ @ \ s_2 \ @ \ [\text{Check-in } g \ r \ c] \ @ \ s_1 \wedge \text{no-Check-in } (s_3 \ @ \ s_2) \ r)
\end{aligned}$$

A trace is *safe* if additionally the room was empty when it was entered:

$$\begin{aligned}
& \text{safe } s \ r = (\exists s_1 \ s_2 \ s_3 \ g \ c. \\
& \quad s = s_3 \ @ \ [\text{Enter } g \ r \ c] \ @ \ s_2 \ @ \ [\text{Check-in } g \ r \ c] \ @ \ s_1 \wedge \\
& \quad \text{no-Check-in } (s_3 \ @ \ s_2) \ r \wedge \text{isin } (s_2 \ @ \ [\text{Check-in } g \ r \ c] \ @ \ s_1) \ r = \{\})
\end{aligned}$$

The actual theorem follows by definition of *safe*. The base case of the induction follows from (0). For the induction step let $t = (e \cdot s_3) @ [Enter\ g' \ r \ (k, \ k')] @ s_2 @ [Check-in\ g' \ r \ (k, \ k')] @ s_1$. We assume *hotel t, no-Check-in ((e · s₃) @ s₂) r*, and $g \in isin\ s\ r$, and show $g' = g$. The proof is by case distinction on the event e . The cases *Check-in* and *Exit* follow directly from the induction hypothesis because the set of occupants of r can only decrease. Now we focus on the case $e = Enter\ g'' \ r' \ c$. If $r' \neq r$ the set of occupants of r remains unchanged and the claim follows directly from the induction hypothesis. If $g'' \neq g$ then g must already have been in r before the *Enter* event and the claim again follows directly from the induction hypothesis. Now assume $r' = r$ and $g'' = g$. From *hotel t* we obtain *hotel s* (1) and $c \in cards\ s\ g$ (2), and from *no-Check-in (s₃ @ s₂) r* and (0) we obtain *safe s r* (3). Let $c = (k_1, k_2)$. From Lemma 1.8 and Lemma 3.3 we obtain $roomk\ s\ r = currk\ s\ r = k'$. Hence $k_1 \neq roomk\ s\ r$ by Lemma 2.2 using (1), (2) and *no-Check-in (s₃ @ s₂) r*. Hence $k_2 = roomk\ s\ r$ by *hotel t*. With Lemma 1.9 and (1-3) we obtain $owns\ t\ r = \lfloor g \rfloor$. At the same time we have $owns\ t\ r = \lfloor g' \rfloor$ because *hotel t* and *no-Check-in ((e · s₃) @ s₂) r*: nobody has checked in to room r after g' . Thus the claim $g' = g$ follows.

The details of this proof differ from those of Theorem 1 but the structure is very similar.

3.3 Eliminating *isin*

In the state based approach we needed *isin* to express our safety guarantees. In the presence of traces, we can do away with it and talk about *Enter* events instead. We show that if somebody enters a safe room, he is the owner:

Theorem 3 *If hotel (Enter g r c · s) and safe₀ s r then owns s r = ⌊g⌋.*

From *safe₀ s r* it follows that s must be of the form $s_2 @ [Check-in\ g_0 \ r \ c'] @ s_1$ such that *no-Check-in s₂ r*. Let $c = (x, y)$ and $c' = (k, k')$. By Lemma 1.8 we have $roomk\ s\ r = currk\ s\ r = k'$. From *hotel (Enter g r c · s)* it follows that $(x, y) \in cards\ s\ g$ and $k' \in \{x, y\}$. By Lemma 2.2 $x = k'$ would contradict $(x, y) \in cards\ s\ g$. Hence $y = k'$. With Lemma 1.9 we obtain $owns\ s\ r = \lfloor g \rfloor$.

Having dispensed with *isin* we could also eliminate *Exit* to arrive at a model closer to the ones in [2].

Finally one may quibble that all the safety theorems proved so far assume safety of the room at that point in time when somebody enters it. That is, the owner of the room must be sure that once a room is safe, it stays safe, in order to profit from those safety theorems. Of course, this is the case as long as nobody else checks in to that room:

Lemma 4 *If safe₀ s r and no-Check-in s' r then safe₀ (s' @ s) r.*

It follows easily that Theorem 3 also extends until check-in:

Corollary 1 *If hotel (Enter g r c · s' @ s) and safe₀ s r and no-Check-in s' r then owns s r = ⌊g⌋.*

3.4 Completeness of *safe*

Having proved correctness of *safe*, i.e. that safe behaviour protects against intruders, one may wonder if *safe* is complete, i.e. if it covers all safe behaviour,

or if it is too restrictive. It turns out that *safe* is incomplete for two different reasons. The trivial one is that in case *initk* is injective, every room is protected against intruders right from the start. That is, $[Check\text{-}in\ g\ r\ c]$ will only allow *g* to enter *r* until somebody else checks in to *r*. The second, more subtle incompleteness is that even if there are previous owners of a room, it may be safe to enter a room with an old card *c*: one merely needs to make sure that no other guest checked in after the check-in where one obtained *c*. However, formalizing this is not only messy, it is also somewhat pointless: this liberalization is not something a guest can take advantage of because there is no (direct) way he can find out which of his cards meets this criterion. But without this knowledge, the only safe thing to do is to make sure he has used his latest card. This incompleteness applies to the state based model as well.

<proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof>

4 Equivalence

Although the state based and the trace based model look similar enough, the nagging feeling remains that they could be subtly different. Hence I wanted to show the equivalence formally. This was very fortunate, because it revealed some unintended discrepancies (no longer present). Although I had proved both systems safe, it turned out that the state based version of safety was more restrictive than the trace based one. In the state based version of *safe* the room had to be empty the first time the owner enters with the latest card, whereas in the trace based version any time the owner enters with the latest card can make a room safe. Such errors in an automaton checking a trace property are very common and show the superiority of the trace based formalism.

When comparing the two models we have to take two slight differences into account:

- The initial setting of the room keys *initk* in the trace based model is an arbitrary but fixed value. In the state based model any injective initial value is fine.
- As a consequence (see the end of Section 3.1) *state.safe* is initially true whereas *Trace.safe* is initially false.

Since many names occur in both models they are disambiguated by the prefixes *state* and *Trace*.

In the one direction I have shown that any hotel trace starting with an injective *initk* gives rise to a reachable state when the components of that state are computed by the trace functions:

$$\begin{aligned}
& \llbracket inj\ initk; hotel\ t \rrbracket \\
& \implies (state.owns = Trace.owns\ t, currk = Trace.currk\ t, \\
& \quad issued = Trace.issued\ t, cards = Trace.cards\ t, roomk = Trace.roomk\ t, \\
& \quad isin = Trace.isin\ t, \\
& \quad safe = \lambda r. Trace.safe\ t\ r \vee Trace.owns\ t\ r = None) \\
& \in reach
\end{aligned}$$

Conversely, for any reachable state there is a hotel trace leading to it:

$$s \in reach \implies$$

$$\begin{aligned}
& \exists t \text{ ik.} \\
& \text{initk} = \text{ik} \longrightarrow \\
& \text{hotel } t \wedge \\
& \text{state.cards } s = \text{Trace.cards } t \wedge \\
& \text{state.isin } s = \text{Trace.isin } t \wedge \\
& \text{state.roomk } s = \text{Trace.roomk } t \wedge \\
& \text{state.owns } s = \text{Trace.owns } t \wedge \\
& \text{state.currk } s = \text{Trace.currk } t \wedge \\
& \text{state.issued } s = \text{Trace.issued } t \wedge \\
& \text{state.safe } s = (\lambda r. \text{Trace.safe } t r \vee \text{Trace.owns } t r = \text{None})
\end{aligned}$$

The precondition $\text{initk} = \text{ik}$ just says that we can find some interpretation for initk that works, namely the one that was chosen as the initial setting for the keys in s .

The proofs are almost automatic, except for the *safe* component. In essence, we have to show that the procedural state.safe implements the declarative Trace.safe . The proof was complicated by the fact that initially it was not true and I had to debug Trace.safe by proof. Unfortunately Isabelle's current counterexample finders [1, 8] did not seem to work here due to search space reasons. Once the bugs were ironed out, the following key lemma, together with some smaller lemmas, automated the correspondence proof for *safe*:

$$\begin{aligned}
& \text{hotel } (\text{Enter } g r (k, k') \cdot t) \Longrightarrow \\
& \text{Trace.safe } (\text{Enter } g r (k, k') \cdot t) r = \\
& (\text{Trace.owns } t r = [g] \wedge \text{Trace.isin } t r = \emptyset \wedge k' = \text{Trace.currk } t r \vee \\
& \text{Trace.safe } t r)
\end{aligned}$$

In addition we used many lemmas from the trace model, including Theorem 2.

5 Conclusion

We have seen two different specification styles in this case study. The state based one is conceptually simpler, but may require auxiliary state components which express properties of the trace that lead to that state. And it may not be obvious if the definition of the state component correctly captures the desired property of the trace. A trace based specification expresses those properties directly. The proofs in the state based version are all automatic whereas in the trace based setting 4 proofs (out of 15) require special care, thus more than doubling the overall proof size. It would be interesting to test Isabelle's emerging link with automatic first-order provers [3] on the trace based proofs.

There are two different proof styles in Isabelle: unstructured apply-scripts [6] and structured Isar proofs [9, 4]. Figure 1 shows an example of the latter. Even if the reader is unfamiliar with Isar, it is easy to see that this proof is very close to the version given in the text. Although apply-scripts are notoriously obscure, and even the author may not have an intuitive grasp of the structure of the proof, in our kind of application they also have advantages. In the apply-style, Isabelle's proof methods prove as much as possible automatically and leave the remaining cases to the user. This leads to much shorter (but more brittle) proofs: The (admittedly detailed) proof in Figure 1 was obtained from an apply-script of less than half the size.

The models given in this paper are very natural but by no means the only possible ones. Jackson himself uses an alternative trace based one which replaces the list data structure by an explicit notion of time. It would be interesting to see further treatments of this problem in other formalisms, for example temporal logics.

Acknowledgments Daniel Jackson got me started on this case study, Stefano Berardi streamlined my proofs, and Larry Paulson commented on the paper at short notice.

References

- [1] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.
- [2] Daniel Jackson. *Software Abstractions. Logic, Language, and Analysis*. MIT Press, 2006.
- [3] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*. In press.
- [4] Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646, pages 259–278, 2003.
- [5] Tobias Nipkow. Verifying a hotel key card system. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *Theoretical Aspects of Computing (ICTAC 2006)*, 2006. Invited paper.
- [6] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. <http://www.in.tum.de/~nipkow/LNCS2283/>.
- [7] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.
- [8] Tjark Weber. Bounded model generation for Isabelle/HOL. In W. Ahrendt, P. Baumgartner, H. de Nivelle, S. Ranise, and C. Tinelli, editors, *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR 2004)*, volume 125(3) of *Electronic Notes in Theoretical Computer Science*, pages 103–116, 2005.
- [9] Markus Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002. <http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html>.