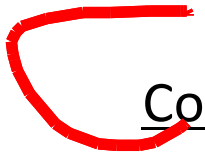
Rigorous Hierarchical Requirements Analysis for Critical System Design



Colin Snook

Asieh Salehi Fathabadi, Dana Dghaym

Thai Son Hoang, Michael Butler

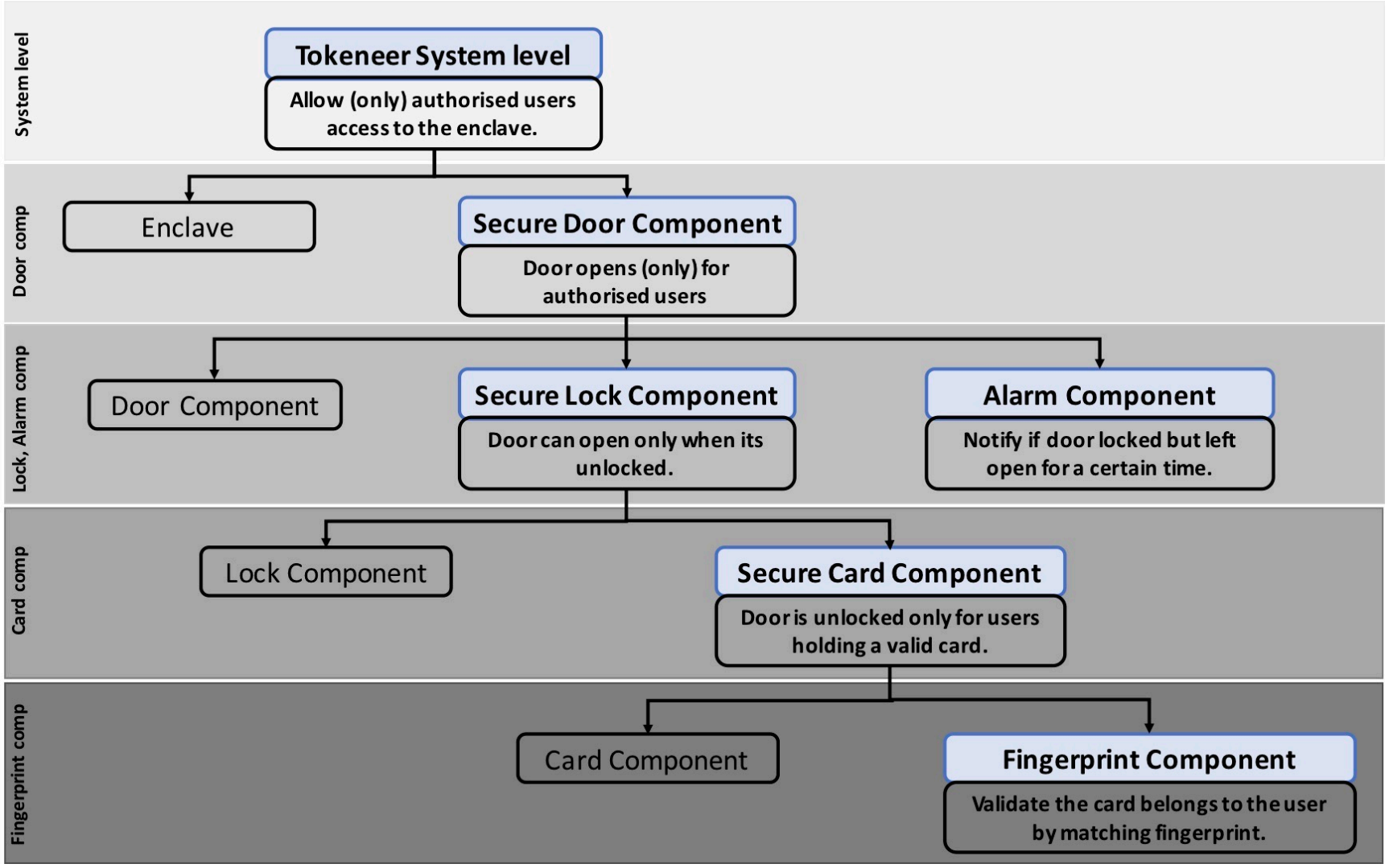
University of Southampton

Introduction: STPA and Event-B

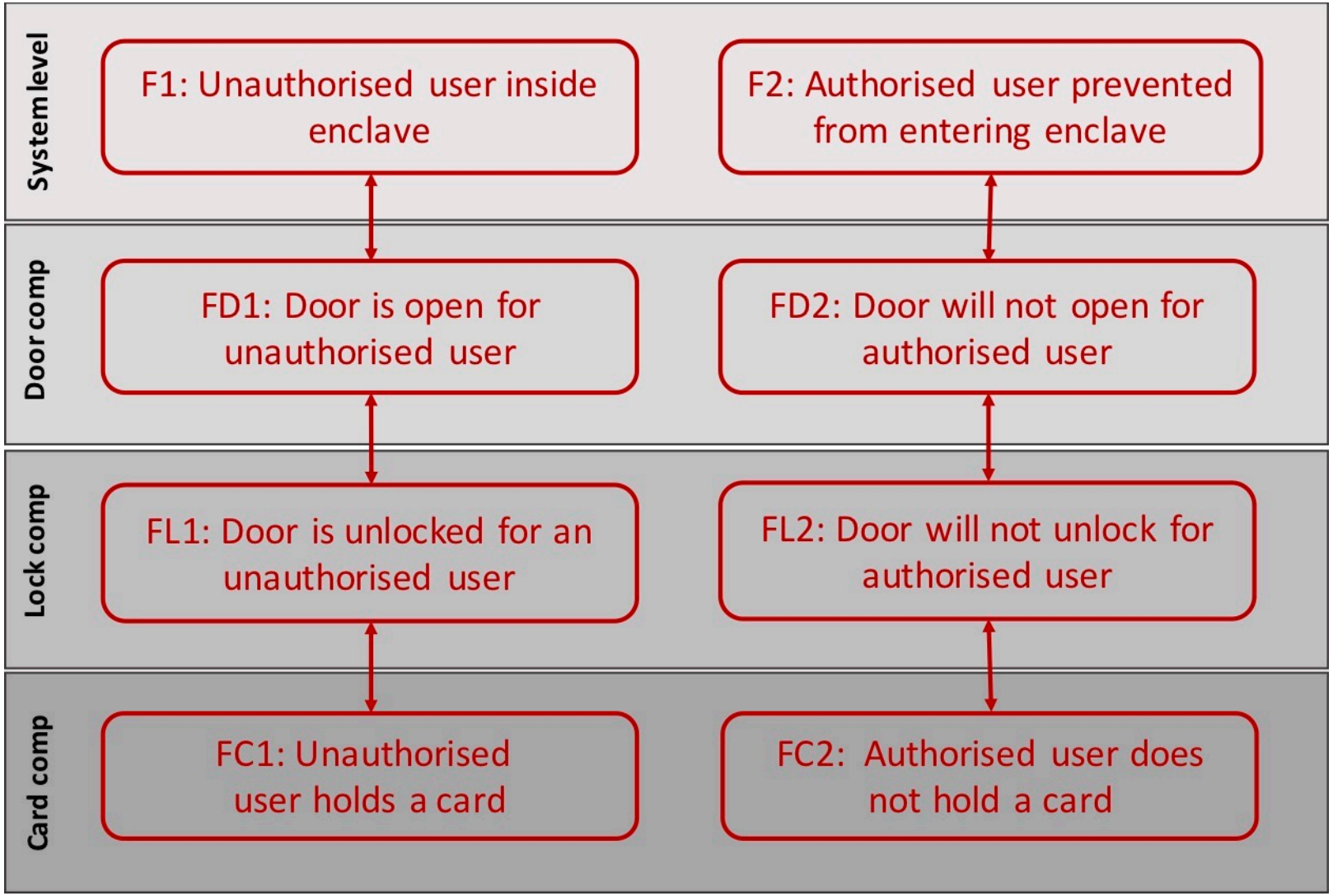
- **STPA – Systems Theoretic Process Analysis**
  - Analyses system behaviour to identify potential ***safety hazards***
    - **STPA-sec -** adapts STPA to identify potential ***security threats***
  - *Methodical* but lacks rigor – relies on human judgement
  - no abstraction - Only deals with one (concrete) level
- **Event-B** formal modelling
  - Validation by animation (scenario checker)
  - Verification by proof (invariant safety and security properties)
  - *Rigorous* but not methodical – relies on human expertise about modelling choices
  - Abstraction - Refinement - can be used to deal with complexity
- Combine **STPA and Event-B**
  - Synergy - *methodical* analysis with *rigorous* verification
  - Hierarchical – use refinement to analyse a hierarchy of sub-components

Hierarchical flow down of requirements to components

Case study: Tokeneer - secure enclave system



Hierarchical component failures



Steps of the analysis process

E.g. Tokeneer: system level analysis

- **Step 1:** State the system purpose. Identify system level failures.

System level

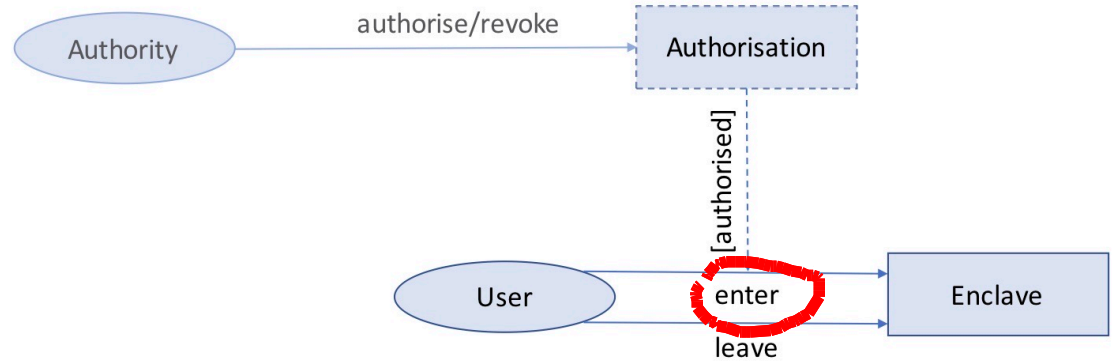
Purpose: Allow (only) authorised users access to the enclave.

Actions: Users can enter and leave enclave.

Failures:

- **F1**: Unauthorised user inside enclave
- **F2**: Authorised user prevented from entering enclave

- **Step 2:**
  Identify the control actions
- *Control action structure diagram*
  - *(equivalent to control action diagrams normally used in STPA but more abstract)*
- *First step towards an Event-B model*



- **Step 3:**
  Perform control action analysis to identify conditions under which failures may occur.

- F2: Authorised user prevented from entering enclave

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
User Enter Enclave	**A11:** Authorised user prevented from entering enclave (*F2*)	**A12:** Unauthorised user enters enclave (*F1*)	N/A
User Leave Enclave	No failure	No failure	N/A

Mitigations:

- **Step 4:**
  Construct formal model:

  - system properties as invariants.

    $$@inv1 : inEnclave \subseteq authorisedUser$$

  - events represent system actions
    - (in this case user actions)

```
event userEnterEnclave
any user
where
  @grd1 : user ∉ inEnclave
  @grd2 : user ∈ authorisedUser
then
  @act1 : inEnclave := inEnclave ∪ {user}
end

event userLeaveEnclave
any user
where
  @grd1 : user ∈ inEnclave
then
  @act1 : inEnclave := inEnclave \ {user}
end
```

- **Step 5:**
  Validate the model by animation using scenarios.
  - Useful for checking model behaves as expected/desired,
  - checking liveness properties, (e.g. authorised users *can* enter enclave)
  - Improve our understanding of the system

- **Step 6:**
  Use automated theorem proving and model checking tools to verify invariant and refinement properties.
  - Useful for debugging the model,
  - checking static properties, (e.g. *no* unauthorised users are in the enclave)
  - Improve our understanding of the system

- **Step 7:** Adjust STPA analysis and models for improved understanding.
  - iteratively

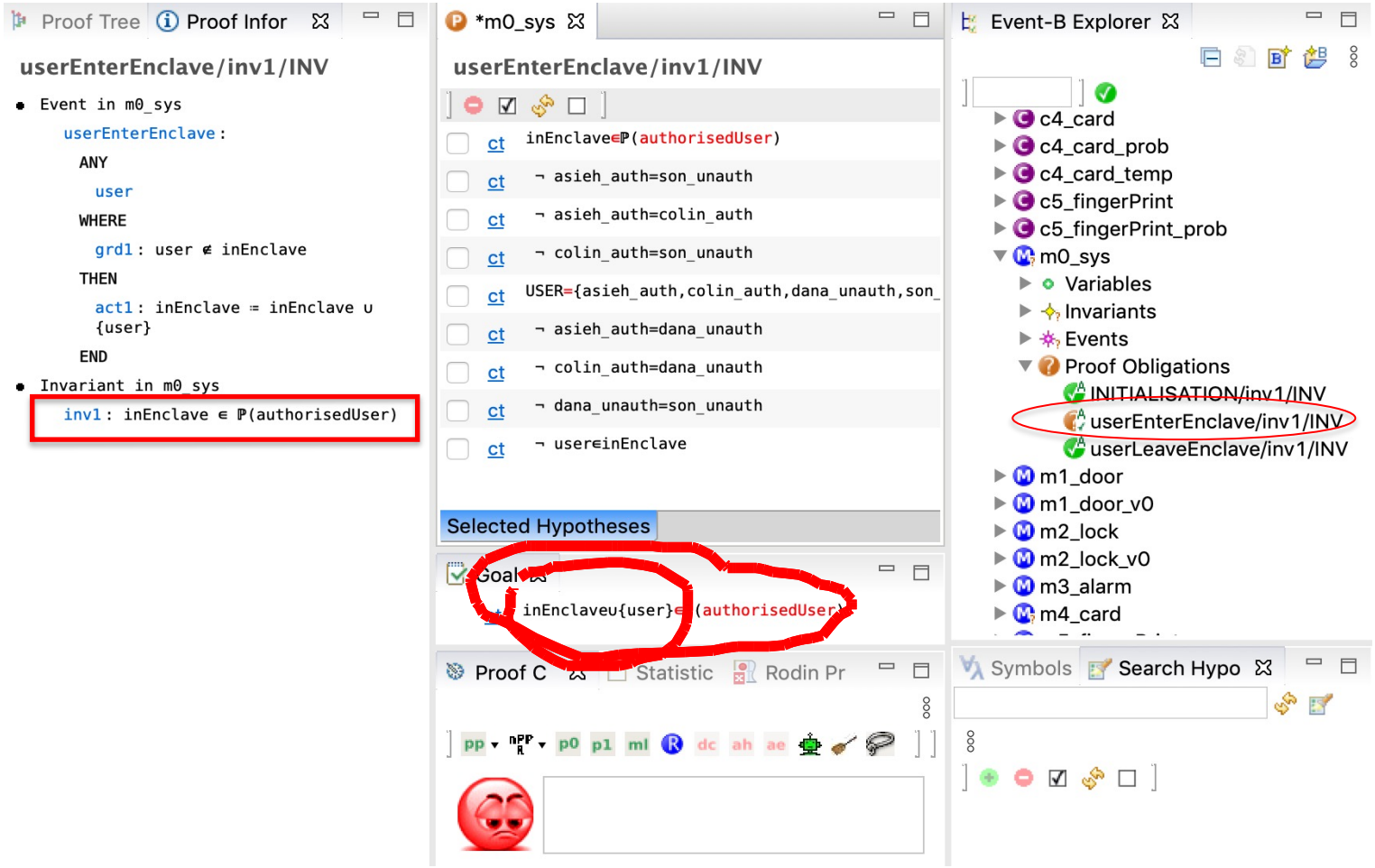
Verification by automated theorem proving (step 6)

- Unproven POs help us to find errors in the modelling:
- e.g.

  INV: Invariant preservation PO

  missing event guards:

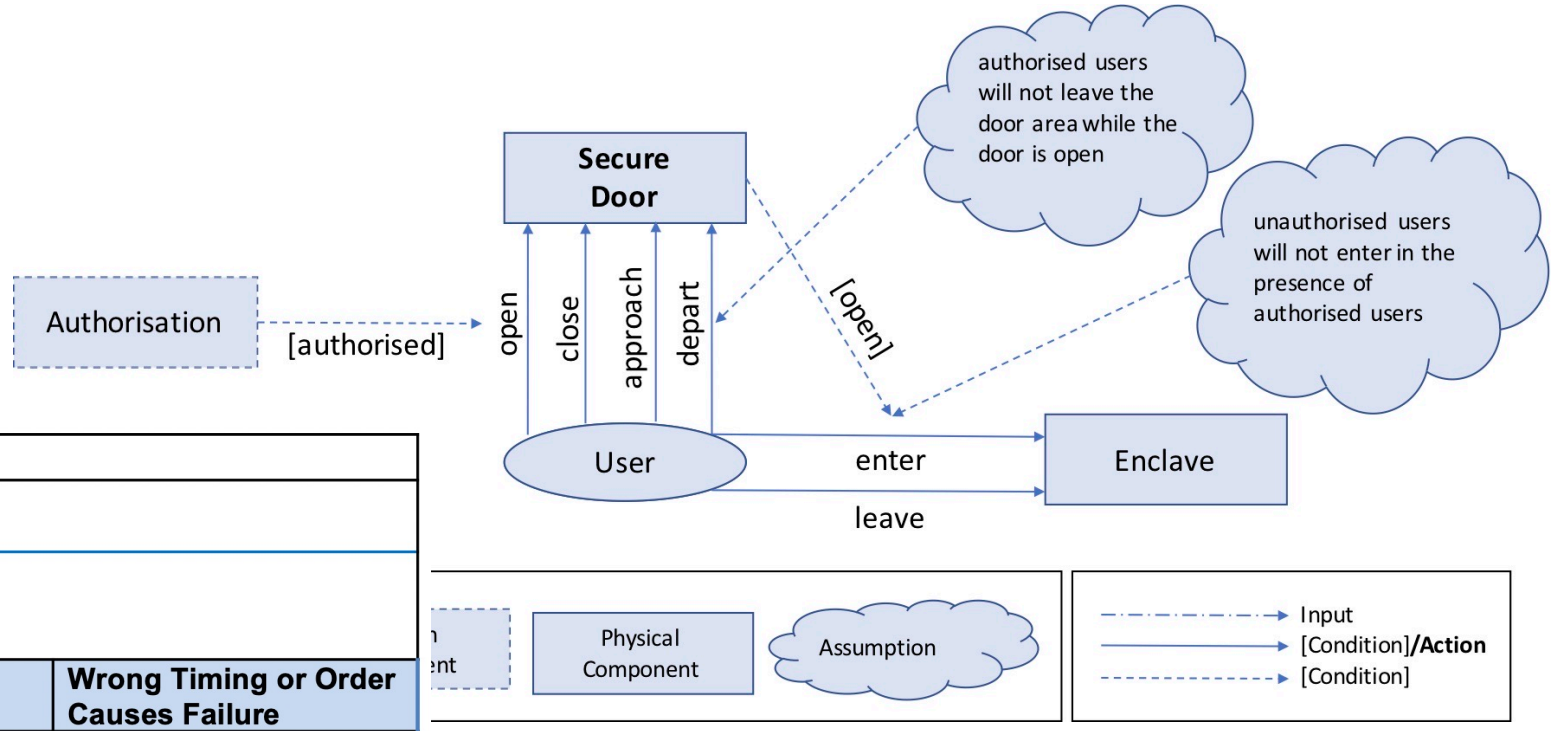
  @grd2: user ∈ authorisedUser
- Sometimes unproven POs help us discover things we missed…
  .. understand the system



- **Step 8:**
  Consider how to mitigate the potential problems with control actions that have been identified in step 3.
  - Mitigations include
    - dismissive arguments,
    - further verification or
    - design of the next level components (derived requirements).
      - E.g. Identify sub-components to be analysed in the next level

- REPEAT steps 1-8 on any sub-components
  - E.g. door

Door component analysis (steps 1-3 & 8)



Door Component

Purpose: Door opens (only) for authorised users.
Actions: Users can open and close doors.

Failures:

- **FD1**: Door is open for unauthorised user (causes *F1*)
- **FD2**: Door will not open for authorised user (causes *F2*)

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
User Open Door	**AD11:** Authorised user is unable to open the door (*FD2*).	**AD12:** Unauthorised user opens the door (*FD1*)	N/A
User Close Door	**AD21:** User does not close the door (*FD1*)	No failure	**AD23:** Authorised user closes door before entering (*FD2*)
User Approach Door	No failure	No failure	No failure
User Leave Door	No failure	No failure	No failure

Mitigations:

- ***Lock*** component controls when the door can be opened (addressing *AD11, AD12*)
- ***Alarm*** component warns when door is left open for a certain time (addressing *AD21*)
- If a user closes the door before entering, they can open it again (addressing *AD23*)

Refine the system model into a door model (step 4)

System level purpose: *Allow (only) authorised users access to the enclave.*
Door component purpose: *Door opens (only) for authorised users.*

- The system & door purposes are modelled as invariants and event guards
- Invariants and guards are refined to specify the derived requirements
- New events are added to open/close door (not shown)

```
machine m0_sys
sees c0_prob
variables
    inEnclave
invariants
    @inv1: inEnclave ∈ ℙ(authorisedUser)
events
    event INITIALISATION
    begin
        @act1: inEnclave ≔ ∅
    end

    event userEnterEnclave
    any
        user
    where
        @grd1: user ∉ inEnclave
        @grd2: user ∈ authorisedUser
    begin
        @act1: inEnclave ≔ inEnclave ∪ {user}
    end

    event userLeaveEnclave
    any
        user
    where
        @grd1: user ∈ inEnclave
    begin
        @act1: inEnclave ≔ inEnclave ∖ {user}
    end
end
```

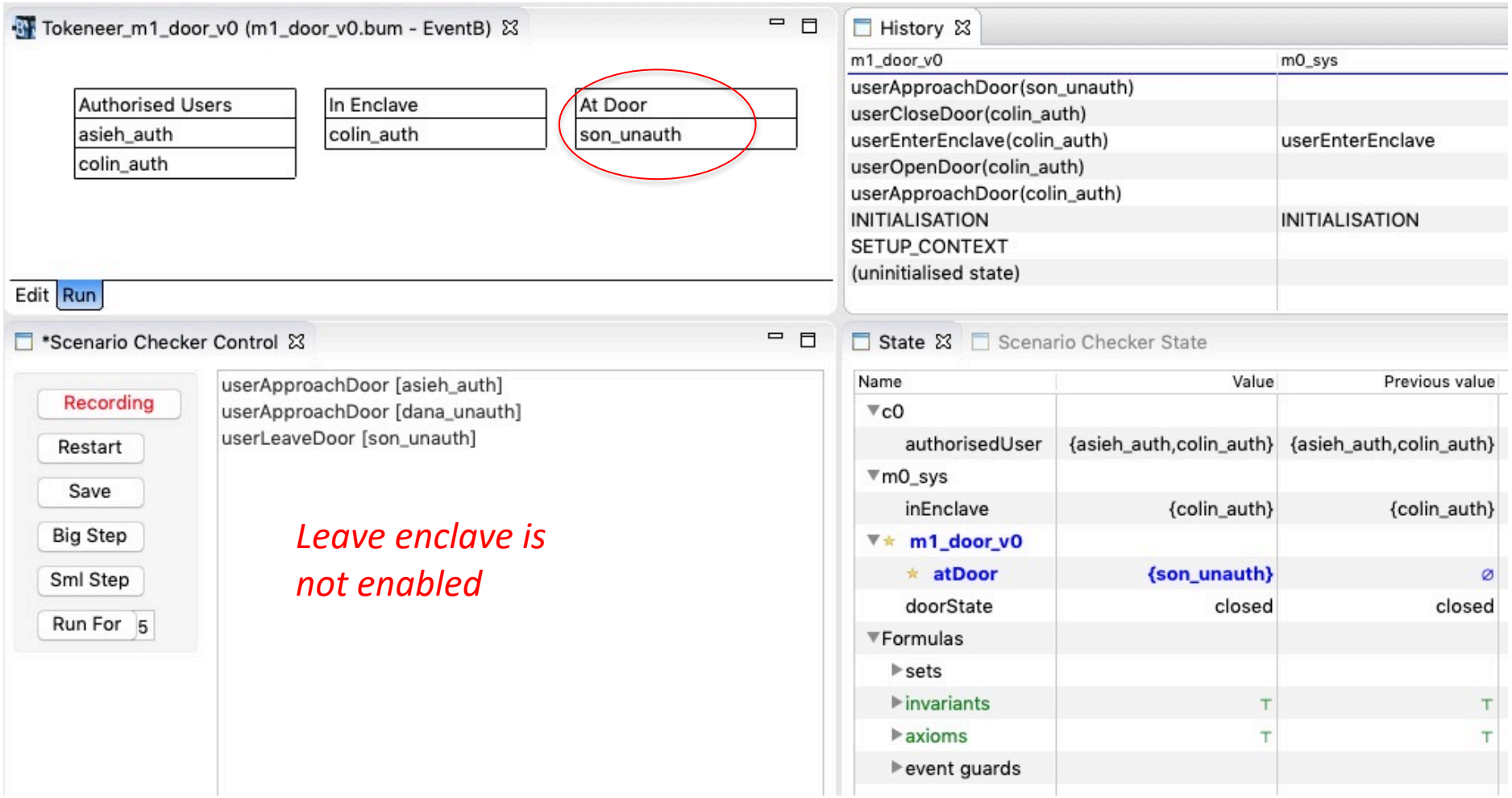
```
machine m1_door_v0
refines m0_sys
sees c1_door_prob
variables
    inEnclave
    atDoor
    doorState
invariants
    @inv1: atDoor ∈ ℙ(USER)
    @inv2: atDoor ∩ inEnclave = ∅
    @inv3: doorState ∈ DOOR_STATES
    @inv4: doorState = open ⇒ atDoor ⊆ authorisedUser
events
    event INITIALISATION extends INITIALISATION
    begin
        @act3: atDoor ≔ ∅
        @act4: doorState ≔ closed
    end

    event userEnterEnclave refines userEnterEnclave
    any
        user
    where
        @grd1: user ∈ atDoor
        @grd2: doorState = open
    begin
        @act1: inEnclave ≔ inEnclave ∪ {user}
        @act2: atDoor ≔ atDoor ∖ {user}
    end

    event userLeaveEnclave refines userLeaveEnclave
    any
        user
    where
        @grd1: user ∈ inEnclave
        @grd2: doorState = open
    begin
        @act1: inEnclave ≔ inEnclave ∖ {user}
```

Validating door component using scenario checker (step 5)

The presence of an unauthorised user by the door prevents the authorised user from opening the door to leave the enclave



Revised door model (Step 7)

- After scenario checking we realised that an unauthorised user can prevent users leaving the enclave.
- Relax the security constraint... the door can be open as long as an authorised user is present.
- Assumption: the presence of authorised users will deter unauthorised ones from entering the enclave

Original model

```
machine m1_door_v0
refines m0_sys
sees c1_door_prob
variables
    inEnclave
    atDoor
    doorState
invariants
    @inv1: atDoor ∈ ℙ(USER)
    @inv2: atDoor ∩ inEnclave = ∅
    @inv3: doorState ∈ DOOR_STATES
    @inv4: doorState = open ⇒ atDoor ⊆ authorisedUser
events
    event INITIALISATION extends INITIALISATION
    begin
        @act3: atDoor ≔ ∅
        @act4: doorState ≔ closed
    end

    event userEnterEnclave refines userEnterEnclave
    any
        user
    where
        @grd1: user ∈ atDoor
        @grd2: doorState = open
    begin
        @act1: inEnclave ≔ inEnclave ∪ {user}
        @act2: atDoor ≔ atDoor \ {user}
    end

    event userLeaveEnclave refines userLeaveEnclave
    any
        user
```

Revised model

```
machine m1_door
refines m0_sys
sees c1_door_prob
variables
    inEnclave
    atDoor
    doorState
invariants
    @inv1: atDoor ∈ ℙ(USER)
    @inv2: atDoor ∩ inEnclave = ∅
    @inv3: doorState ∈ DOOR_STATES
    @inv4: doorState = open ⇒ inEnclave ≠ ∅ ∨ (atDoor ∩ authorisedUser) ≠ ∅
events
    event INITIALISATION extends INITIALISATION
    begin
        @act3: atDoor ≔ ∅
        @act4: doorState ≔ closed
    end

    event authUserEnterEnclave refines userEnterEnclave
    any
        user
    where
        @grd1: user ∈ atDoor
        @grd2: doorState = open
        @grd3: user ∈ authorisedUser
    begin
        @act1: inEnclave ≔ inEnclave ∪ {user}
        @act2: atDoor ≔ atDoor \ {user}
    end

    event unauthUserEnterEnclave refines userEnterEnclave
    any
        user
    where
        @grd1: user ∈ atDoor
        @grd2: doorState = open
        @grd3: user ∉ authorisedUser
        @grd4: atDoor ∩ authorisedUser = ∅
        @grd5: inEnclave = ∅
    begin
        @act1: inEnclave ≔ inEnclave ∪ {user}
        @act2: atDoor ≔ atDoor \ {user}
    end
```

Relaxed constraint

Assumption

Thank you

Any questions?

Tokeneer: Lock, Alarm, Card and Fingerprint component analysis

Lock Component

Purpose: Door can open only when its unlocked.
Actions: Door can lock and unlock for users.

Failures:

- **FL1**: Door is unlocked for an unauthorised user (causes *FD1* and so *F1*)
- **FL2**: Door remains locked for an authorised user (causes *FD2* and so *F2*)

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
Unlock Door	**AL11:** Door remains locked for an authorised user (*FL2*)	**AL12:** Door unlocks for an unauthorised user (*FL1*)	N/A
Lock Door	**AL21:** Door remains unlocked for an unauthorised user (*FL1*)	N/A	**AL23:** Door locks before user opens door (*FL2*)

Mitigations:

- ***Card*** component door is unlocked only for users holding a valid card (addressing *AL11, AL12*)
- Lock component is verified to automatically re-lock when the door closes (addressing *AL21*)
- Lock component is validated to give sufficient time before automatically re-locking (addressing *AL23*)

Card Component

Purpose: Door is unlocked only for users holding a valid card.
Actions: Card can be issued for a user.

Failures:

- **FC1**: Unauthorised user holds a card (causes *FL1* and so *FD1* and *F1*)
- **FC2**: Authorised user does not hold a card (causes *FL2* and so *FD3* and *F2*)

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
Issue Card	**AC11:** Authorised user not issued a card (*FC2*)	**AC12:** Unauthorised user is issued a card (*FC1*)	N/A
Lose Card	No failure	**AC22:** Authorised user loses card (*FC2*)	N/A
Find Card	No failure	**AC32:** Unauthorised user finds card (*FC1*)	N/A

Mitigations:

- Out of scope – an authorisation authority will deal with users without cards (addressing *AC11, AC22)*
- ***Fingerprint*** component ensures door is unlocked only for users with a fingerprint that matches the card that they hold (addressing *AC12, AC32*)

Alarm Component

Purpose: Notify if door locked but left open for a certain time.
Actions: Alarm can start or clear.

Failures:

- **FA1:** Alarm off when door is left open for a certain time (leading to *FD2* and so *F1*)
- **FA2:** Alarm on when door is closed or soon after door opened (this may lead to alarm notifications being ignored, hence leading to *FD2* and so *F1*)

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
Alarm Start	**AA11:** Alarm does not start when door is left open (*FA1*).	**AA12:** Alarm starts when door is closed (*FA2*)	**AA13a:** Alarm started too late means that door is left open without notification for too long (*FA1*). **AA13b:** Alarm started too quickly after door opened (*FA2*)
Alarm Clear	**AA21:** Alarm does not stop after door is closed (*FA2*)	N/A	**AA23a:** Alarm cleared too quickly means that door is left open without notification (*FA1*). **AA23b:** Alarm cleared too late may (*FA2*)

Mitigations:

- Alarm component is verified to ensure that it starts when the door is left open for a certain time and stops as soon as the door is closed is always given correctly (addressing *AA11, AA12, AA21, AA23a, AA23b)*
- The time delay between opening the door and starting the alarm must be chosen by validation and experimentation involving domain experts (addressing *AA23a, AA23b*)

Fingerprint Component

Purpose: Validate the card belongs to the user by matching fingerprint.
Actions: The fingerprint on the card is compared with the user's fingerprint and if a match is found, the card is valid.

Failures:

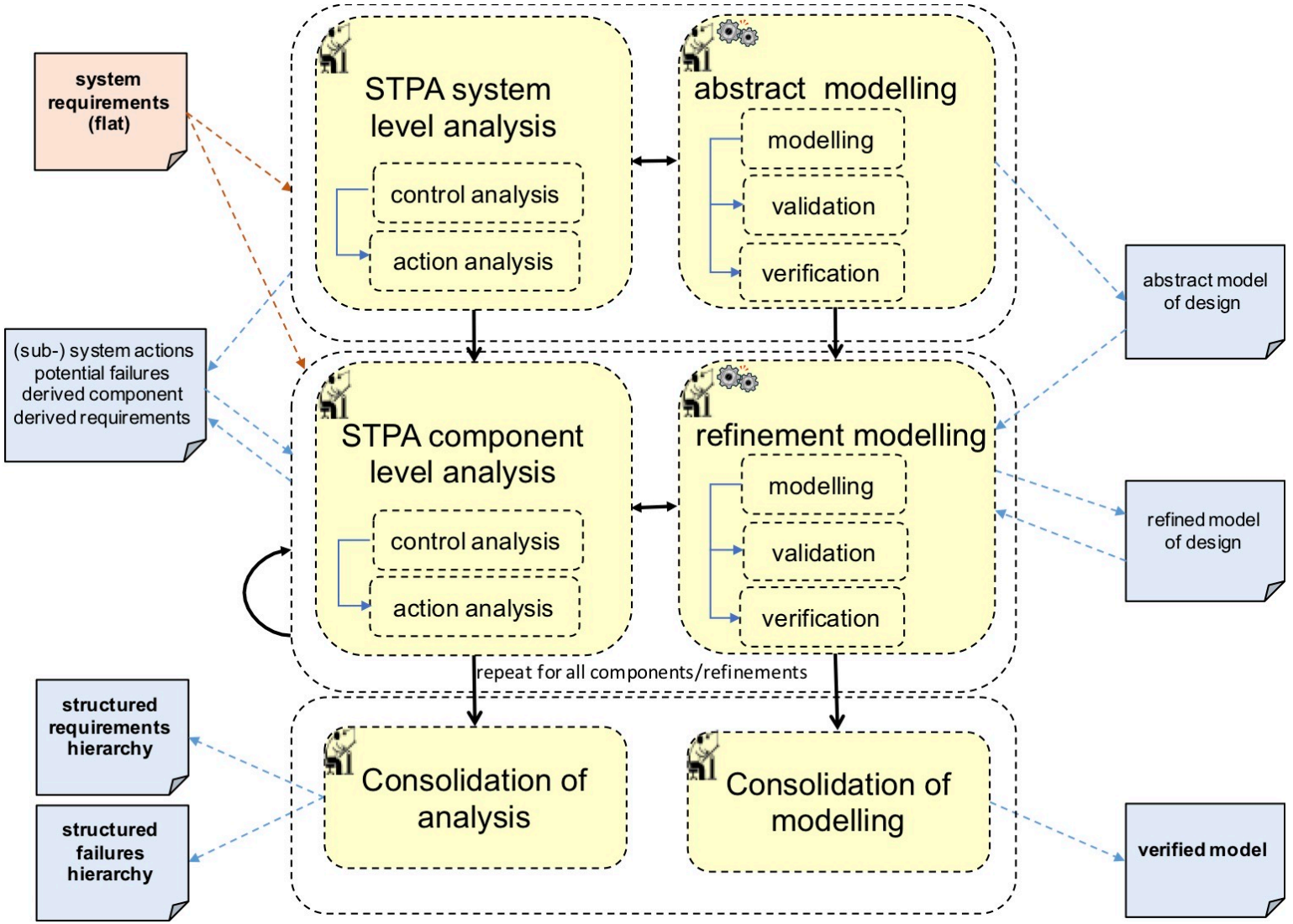
- **FF1:** Authorised user does not hold validated card (new failure leading to *F1*)
- **FF2:** Unauthorised user has validated card (causes *FC1* and so *FL1, FD1, F1*)

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
Match Fingerprint	**AF11:** Authorised users card is not validated (*FF1*)	**AF12:** Card is incorrectly validated for an unauthorised user (*FF2*)	**AF13:** Card is validated after the lock is unlocked

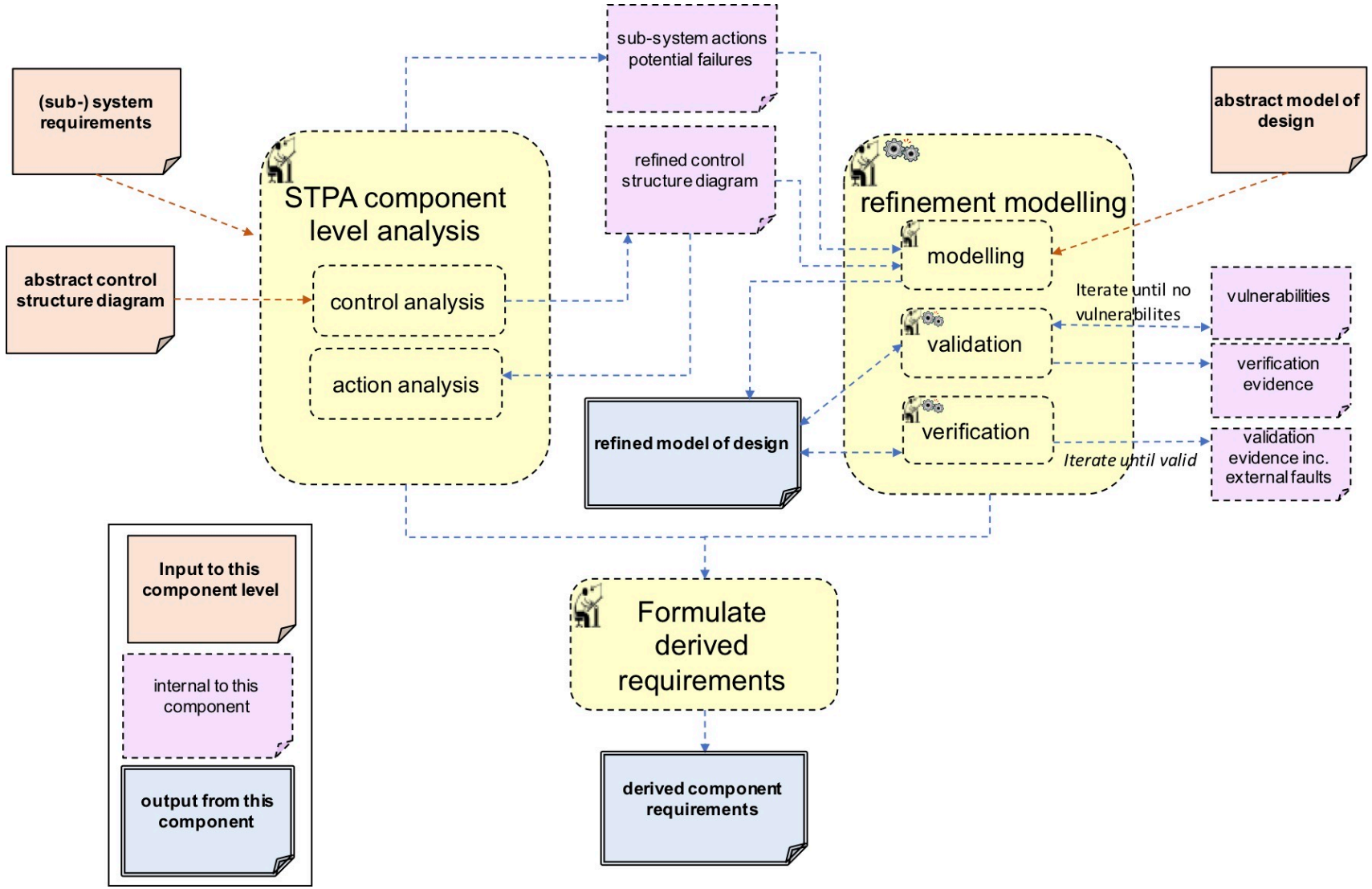
Mitigations:

- Fingerprint component is verified to ensure that validation is always given correctly (addressing *AF11, AF12)*
- Lock component is verified to ensure that it cannot unlock without the card being validated (addressing *AF13)*

Hierarchical STPA-Event-B overall process



STPA-Event-B phase process for one component



Outline of talk

- Event-B and STPA
  - Hierarchical process
- Tokeneer system:
  - Flow down requirements
  - Hierarchical failures
  - System level analysis
  - Component level analysis