acuitas

reliability through insight

SOFTWARE ROOT CAUSE ANALYSIS

(WITH FAULT TREES)

Chris Jackson

PhD MSc MMS BE MIEAust CPEng CRE

Dr Chris Jackson holds a Ph.D. and MSc in Reliability Engineering from the University of Maryland's Center of Risk and Reliability. He also holds a BE in Mechanical Engineering, a Masters of Military Science from the Australian National University (ANU) and has substantial experience in the field of leadership, management, risk, reliability and maintainability. He is a Certified Reliability Engineer (CRE) through the American Society of Quality (ASQ) and a Chartered Professional Engineer (CPEng) through Engineers Australia. Dr Jackson was the inaugural director of the University of California, Los Angeles (UCLA's) Center of Reliability and Resilience Engineering and the founder of its Center for the Safety and Reliability of Autonomous Systems (SARAS). He has had an extensive career as a Senior Reliability Engineer in the Australian Defence Force, and is the Director / CEO of Acuitas Reliability. He has supported many complex and material systems develop their reliability performance and assisted in providing reliability management frameworks that support business outcomes (that is, make more money). He has been a reliability management consultant to many companies across industries ranging from medical devices to small satellites. He has also led initiatives in complementary fields such as systems engineering and performance-based management frameworks (PBMFs). He is the author of two reliability engineering books, co-author of another, and has authored several journal articles and conference papers.

www.acuitas.com

Software Root Cause Analysis (With Fault Trees)

"Root Cause Analysis (RCA)" usually focuses on hardware issues. Most systems (including smart locks) are equally reliant on software.

> **"Software"** is a set of instructions called **"code"** that tells a computer what to do.

Software and hardware failures are different. Software always follows the instructions we provide it in the "code." All software failures start with "coding errors," "bugs," or *"software defects."*

> A "coding error," "bug," or *"software defect"* is a mistake in the code that causes software to behave unexpectedly or crash.

A *"coding error"* is a human mistake analogous to *"hardware faults"* like impurities or cracks. Software doesn't 'wear out' or 'get old.' Software can appear to 'wear-in' as "coding errors" are pre-existing issues that will become uncovered as customers use the software, which **"software patches"** can repair.

> A **"software patch"** is an update to "software" or "code" designed to fix specific "coding errors" or improve the "software's" functionality.

The reliability of software subject to monitoring and subsequent "patches" will improve over time.

Software Root Cause Analysis (RCA)

Both hardware and software RCA have several similarities. They both start with an *"undesirable event,"* which becomes the *"top event"* of a Fault Tree. The *"basic events"* are still the *"root causes"* of *"failure"* for both hardware and software. However, software *"failure modes"* are found differently, emphasizing how software interacts with users.

Suppose a smart lock fails to lock at our customer's site during its warranty period. We can't easily inspect it to see if solder joints are fractured. We won't know if hardware or software caused the failure, so both must be considered.

Software RCA starts by understanding how the smart lock should work when software does its job. There are six main "use cases" for smart lock software as part of the typical door-locking process. The user can command the smart lock to lock by pressing a button in an app, issuing a command through a computer, detecting a user's smartphone leaving the doorway, entering a pin on the keypad, issuing a voice command through a smart device, and using a physical key.

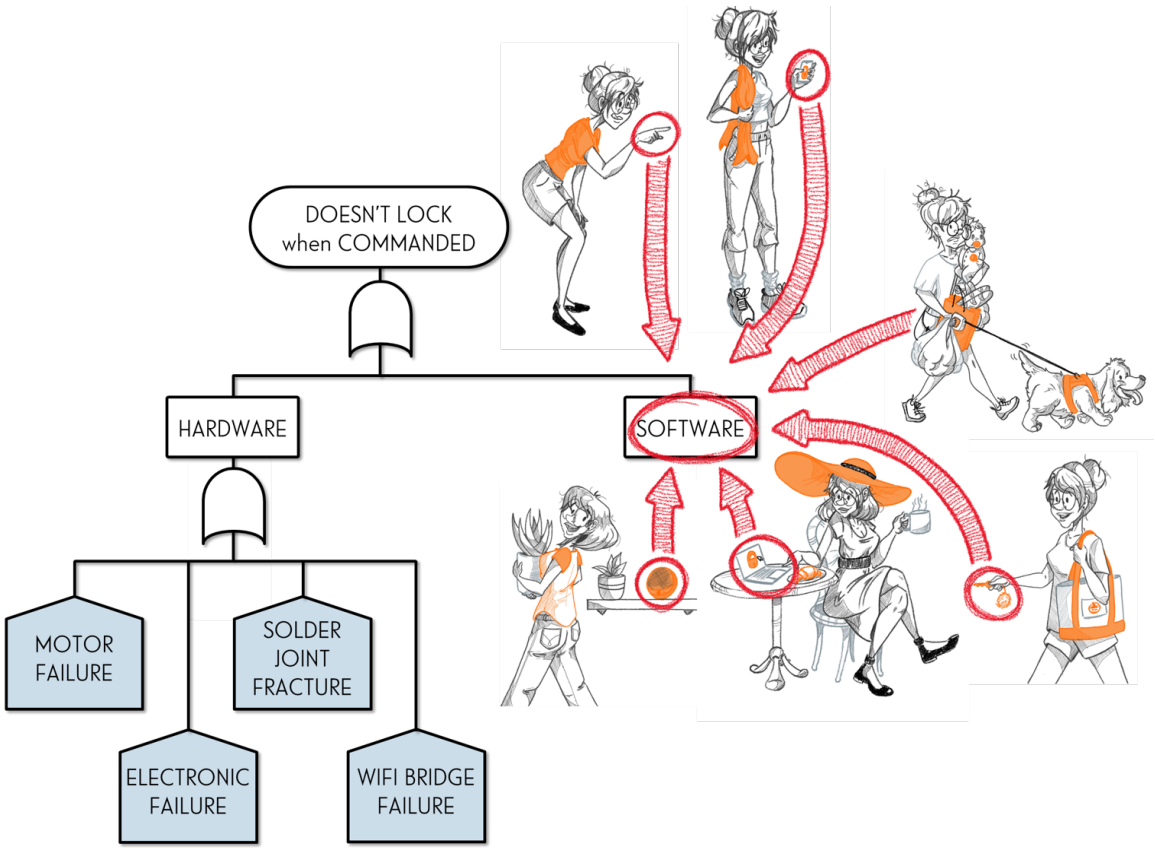

Figure 22.1: Start of smart lock hardware and software RCA Fault Tree (highlighting the importance of user interactions for software)

Software Failure Modes

Software RCA is like any other RCA in that brainstorming occurs one layer at a time. We start with "software failure modes," which are like hardware "functional failure modes."

"Software failure modes" are the "software" functional consequences from "coding errors," "bugs" or *"software defects."*

They represent actions the software should perform but isn't, or actions the software is performing when it shouldn't. One common group of *"software failure modes"* is **"faulty state management,"** where the software believes a device is in a state it is not. This translates to the software believing the lock is locked when it is unlocked with the bolt retracted instead.

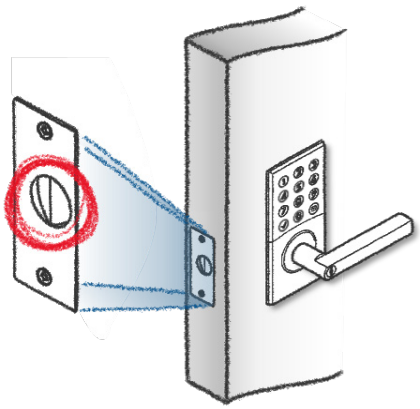

Figure 22.2: Smart lock in 'unlocked state' with retracted bolt

If the software incorrectly believes the smart lock is in a 'locked state,' it won't try to lock it by extending the bolt. There is extensive existing knowledge on "faulty state management" failure modes, including the following eight generic examples:

- "states being stuck due to missing code to start a transition when conditions for that transition are met"
- "software missing a state or a state transition," meaning the code that makes the smart lock lock is absent
- "software unstable from an unexpected loss of power or an unexpected user abort" while in a particular state (which could mean the software assumes the smart lock is locked after replacing flat batteries, even when it is not)
- "software missing a fault or safe state" where the code contains no instructions to deal with problems of any kind
- "software missing a transition to a fault or safe state"
- "fault or safe state behavior is inappropriate for its mission," where 'mission' in this context means the smart lock use case
- "software executes a prohibited transition to another program or accepts a prohibited transition from another program," like when the software incorrectly believes it needs to unlock the lock every time it tries to lock.
- "software allowing the system to be in more than one state at a time," where part of the code acts as if the smart lock is unlocked while another part acts as if it's locked.

These and other well-known software failure modes are part of a compiled list called "Common Defect Enumeration (CDE)" that can be found on the US Defense Acquisition University website:

(https://www.dau.edu/sites/default/files/Migrated/CopDocuments/Reliable%20Software%20SOW%20Appendix%20B%20-%20CDE.xlsx)

> The CDE lists common *"software failure modes"* and describes many common "coding errors" types and other faults that might lead to them.

The CDE describes several categories of *software failure modes* (like "error handling," "timing," "functionality," "data definition," "sequencing," "processing," "faulty usability," and "machine learning"). Not all will apply to a particular product. For example, the smart lock doesn't involve "machine learning," so the "machine learning software failure modes" will not apply. The language used to describe *"software failure modes"* in the CDE is very generic and should be tailored where possible. The red-shaded input (intermediate) events in the Fault Tree illustrated in Figure 22.3 are examples of *"software failure modes"* identified in the CDE but added to the Fault Tree with tailored language.

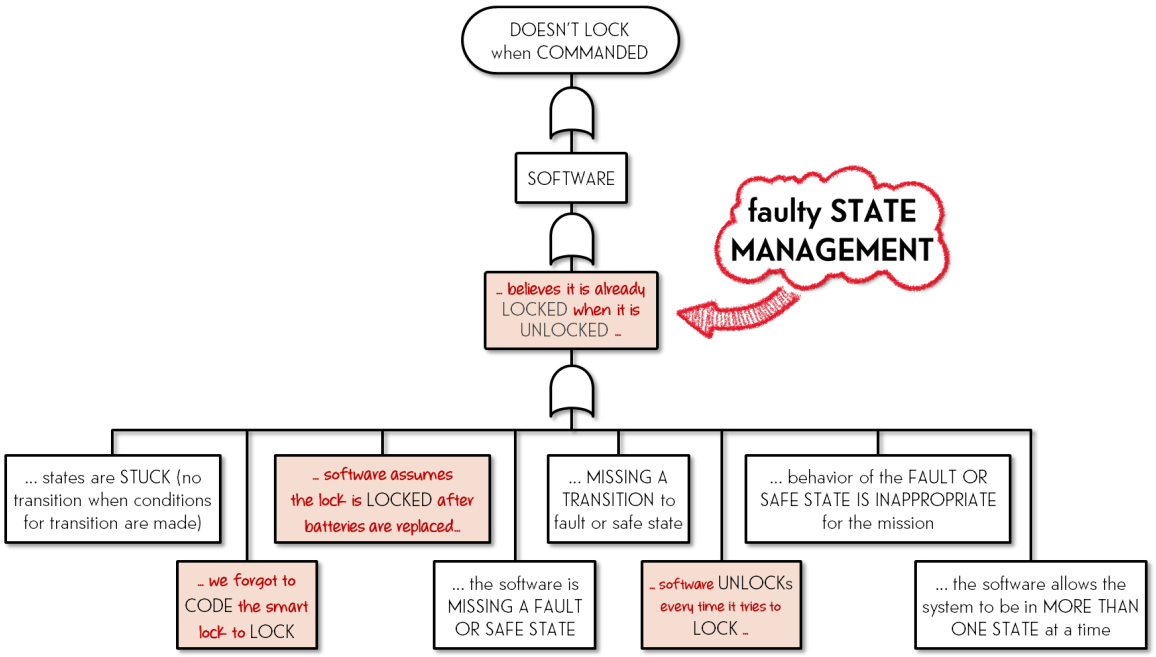

Figure 22.3: Software-specific element of smart lock RCA Fault Tree for event smart lock 'doesn't lock when commanded'

All RCA involves iteratively focusing on a single event and brainstorming one layer at a time until *"potential root causes"* are identified.

Suppose we first focus on 'software assuming the lock is locked after batteries are replaced.' Before we use the CDE to assist RCA, we need to always think about our product or system specifically. Most electronic products have "volatile memory" and "non-volatile memory." *"Volatile memory"* stores data but needs continuous power. *"Non-volatile memory"* stores data with larger, slower integrated circuits but doesn't require continuous power. If the "volatile memory" stores the smart lock state, the software 'forgets' it whenever batteries run flat.

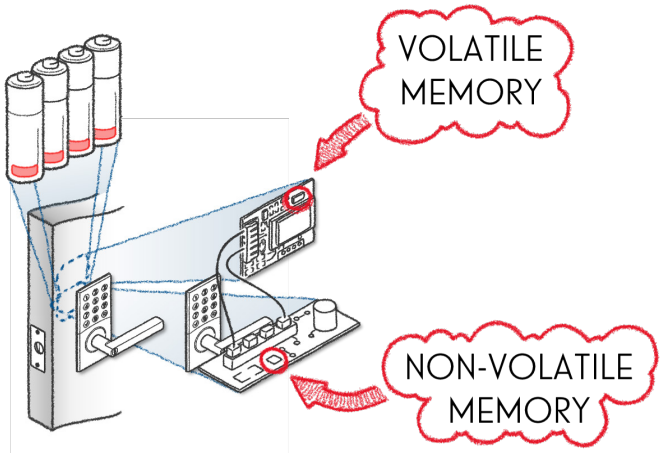

Figure 22.4: Smart lock volatile and non-volatile memory integrated circuits

One *"potential root cause"* of the "software assuming the smart lock is locked after batteries are replaced" is the "lock state is not stored in the non-volatile memory." Another is that there's "no sensor" physically checking the smart lock state without software needing to remember it. A lock state sensor could mean changing the design to include a physical switch or an electrical contact to identify when the bolt is fully extended (or not).

RCA should be conducted on each *"software failure mode"* before the CDE is used to identify system-specific issues like this.

Once the brainstorming is exhausted, CDE can help us find three additional *"potential root causes"* for this *"failure mode,"* two of which involve issues with "software specifications." When we tailor these potential root causes to our smart lock, we get "specifications fail to identify what the software needs to do after the batteries run flat," "specifications aren't tailored to different states where the lock might be locked or unlocked when the batteries run flat," and finally, the "code doesn't implement the requirements in specifications to deal with batteries running flat," meaning there's a "coding error."

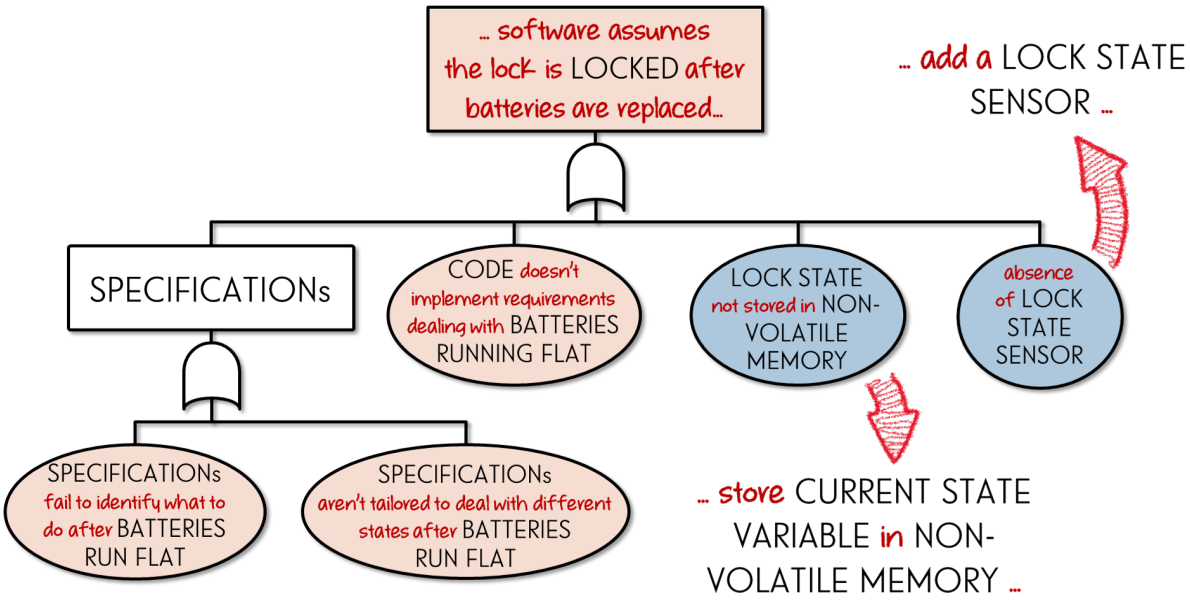

Figure 22.5: Smart lock RCA Fault Tree focusing on "failure mode" 'software assumes the lock is locked after batteries are replaced'

The two *"potential root causes"* on the right of Figure 22.5 have obvious *"corrective actions (CAs),"* which we can prioritize based on cost and efficiency. Adding a lock state sensor is not an option for fielded products. It could be a useful design feature if identified early, especially if "non-volatile memory" is already allocated to other variables. However, if there is still unallocated "non-volatile memory," issuing a "software patch" where the lock state is instead stored in "non-volatile memory" could be an option for fielded systems.

The three *"potential root causes"* the CDE helped us find must be investigated further to confirm their relevance. Confirming that requirements and specifications are fine, which means the issue lies with the code or system, takes time and kills the momentum of the brainstorming session. However, including them creates a more complex Fault Tree that will then help generate *"corrective actions (CAs.)"* The RCA facilitator must decide whether it's worth immediately reviewing specifications during brainstorming or moving on while including these *"potential root causes"* in the Fault Tree to maintain momentum.

Suppose the facilitator chooses to analyze the next *"software failure mode"* when the software 'allows the system to be in more than one state at a time.' Brainstorming with assistance from the CDE yields the following Fault Tree element.

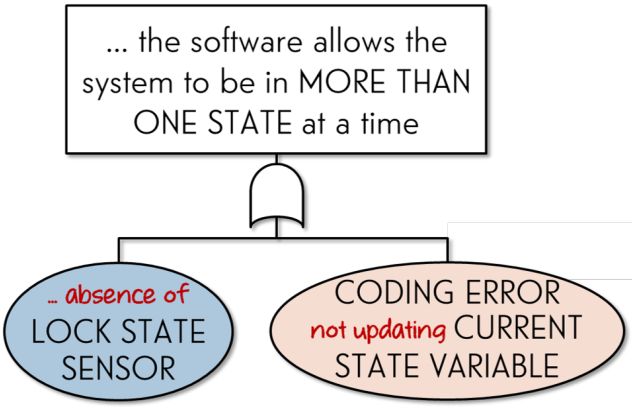

Figure 22.6: Smart lock RCA Fault Tree focusing on "failure mode" 'software allows the system to be in more than one state at a time'

Software tracks product states using a "current state variable." Allowed states are constants to which the "current state variable" can be set. Smart lock allowed states are illustrated in Figure 22.7.

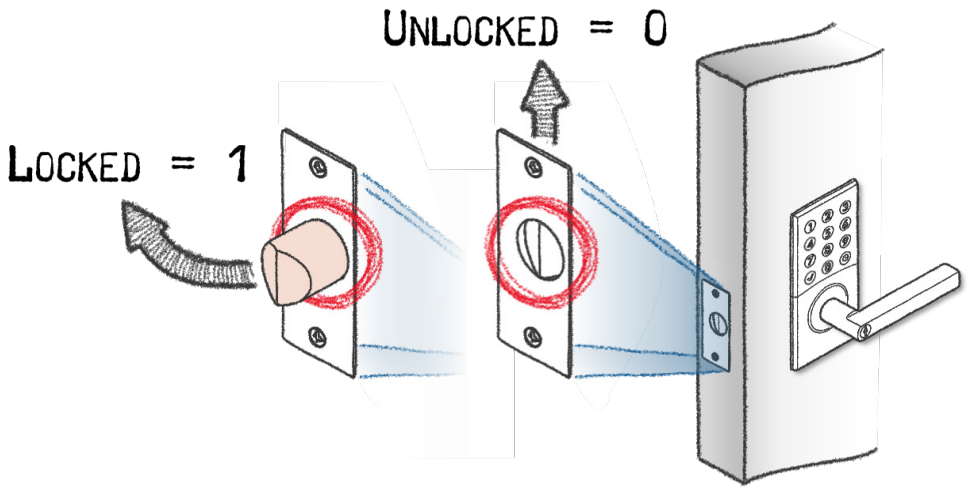

Figure 22.7: Smart lock defined states (values that the "current state variable" can be set to)

A *"coding error"* could set the smart lock's *"current state variable"* to the wrong value, resulting in unintended actions like unlocking a lock when it should be locked. And again, the absence of a lock state sensor could contribute to this *"software failure mode."* While including a lock state sensor in the preliminary design might be costly, it could address multiple causes of future software failure.

Each *"faulty state management failure mode"* can be analyzed using a software-centric perspective before moving on to others. However, the links between software and hardware will typically remain.

For example, another *"software failure modes"* category is *"faulty error handling."*

> A **"software error"** is a deviation from the software's intended behavior or functionality.

"Software errors" are not the same as *"coding errors."* A *"software error"* might be the *"current state variable"* being set to the wrong value. A *"coding error"* or "bug" is the actual mistake in the instructions or "code" generated by whoever created the software that leads to the *"current state variable"* being set to the wrong value.

Software must be able to deal with *"software errors"* to be reliable. *"Software error handling"* can be extended to scenarios where "hardware faults" and "unexpected user actions" may also contribute to failures. Completing a hardware *"failure modes and effects analysis (FMEA)"* should identify potential *"hardware faults"* that software may need to handle.

For instance, electric motors degrade over time. The insulation around the wire coils that generate magnetic fields will deteriorate, causing the motor to require more electrical power and perform less efficiently. This might lead to the "software" assuming sufficient electricity is provided to the motor to fully lock the smart lock when it is not.

Writing additional code to increase the voltage compensating for insulation degradation and extending the motor's effective life compared to competitors (noting this would deplete batteries more quickly).

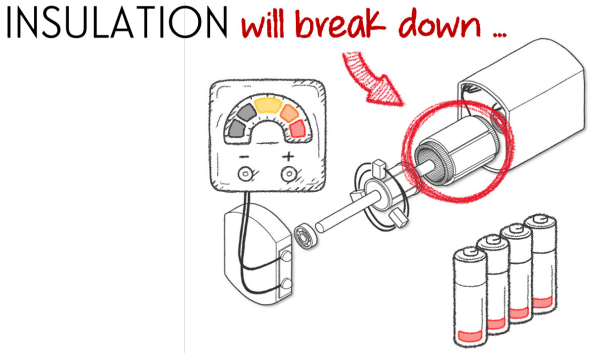

Figure 22.8: Software detecting a "hardware fault" (electric motor insulation degradation that reduces efficiency) and increasing the voltage to compensate (at the expense of battery life)

Fault Trees that model software reliability and assist software RCA often rely on **'PAND' gates** regarding *"hardware faults."* This is because software-based components that must deal with *"hardware faults"* only need to 'not fail' before *"hardware faults"* occur. This is a characteristic of previously modeled switching system reliability Fault Trees that used **'PAND' gates.**

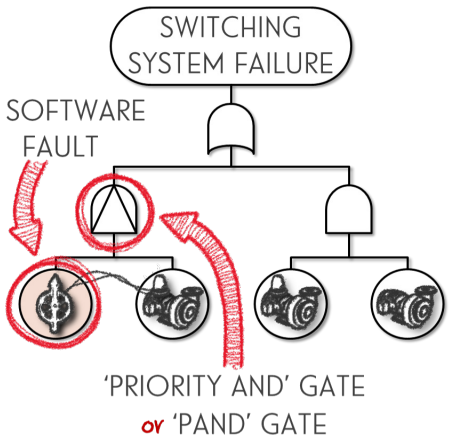

Figure 22.9: Switching system reliability model Fault Tree

Understanding Users

While CDE can help RCA, it should never replace critical thinking in identifying potential root causes of failure. This includes product-specific use cases and designers and engineers working very hard to empathize with customers or users. A smart device might mishear a voice command. A thorough RCA will identify *"corrective actions (CAs),"* such as having the smart device prompt the user for confirmation.

Figure 22.10: Use case where the user commands the smart lock to lock (or unlock) where there is a chance a smart device mishears the command

A **"race condition"** is when multiple *"threads"* are running concurrently and accessing shared data, and the outcome depends on the order in which these *"threads"* finish.

A **"thread"** is a process carried out because of "software" instructions (such as locking a smart lock).

The smart lock can receive two lock commands almost simultaneously. Suppose the first command is processed, and the smart lock is locked before the second command arrives. However, the second command arrives before the *"current state variable"* is updated to reflect the "locked state." This **"race condition"** between competing threads might mean the software inadvertently unlocks the smart lock by sending electricity to the motor without realizing it is now locked.

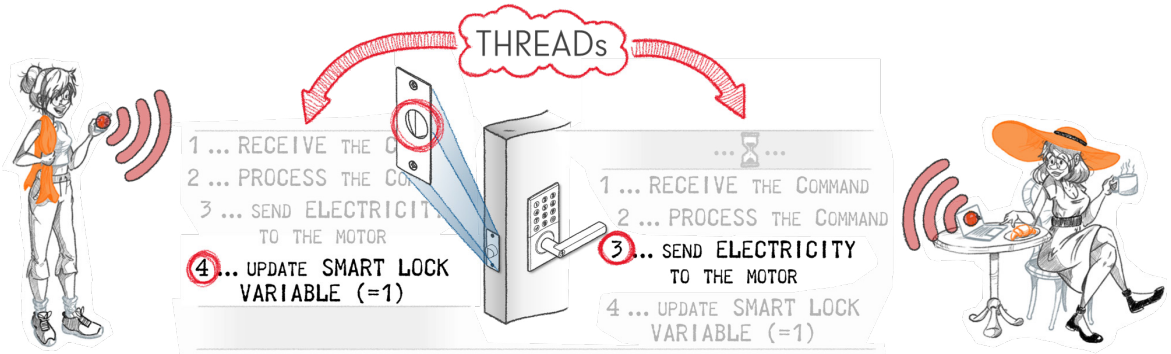

Figure 22.11: Smart lock "race condition" involving locking the smart lock

"Race conditions" are fixable *"software failure modes"* and are easily preventable when processes are analyzed adequately during the initial coding effort.

Corrective Actions (CAs) and Mitigations

A **"corrective action"** or "fix" is an action that involves a change to a system design, operation, maintenance procedures, or the manufacturing process of the item for the purpose of improving its reliability.

In software, **"mitigation"** refers to the actions taken to reduce or eliminate the impact of a potential problem, risk, or vulnerability.

"Corrective Actions (CAs)" deal with actions that involve changing the "code" to eliminate failures. *"Corrective Actions (CAs)"* tend to be robust solutions to software problems and might include refining software specifications, ensuring the "current state variable" is stored in "non-volatile memory," or adding a smart lock state sensor.

If *"Corrective Actions (CAs)"* are impractical as the system is fielded or uses third-party code, *"mitigations"* such as regular resets or adjusting customer operating procedures may be used. Regardless of classification, all *"Corrective Actions (CAs)"* and *"mitigations"* should be prioritized and ranked to determine the most impactful solutions.

Software Failure and Prevention

The CDE not only assists RCA but also *"software failure modes and effects analysis (FMEA),"* which prevents problems during development. The optimal and most cost-effective time to integrate a state sensor into a smart lock is during the initial design. Software FMEAs can uncover additional issues, such as the lack of a **"state diagram"** that can help prevent *"faulty state management."*

A **"software state diagram"** visually represents a software component or system's different states and how it transitions between them.

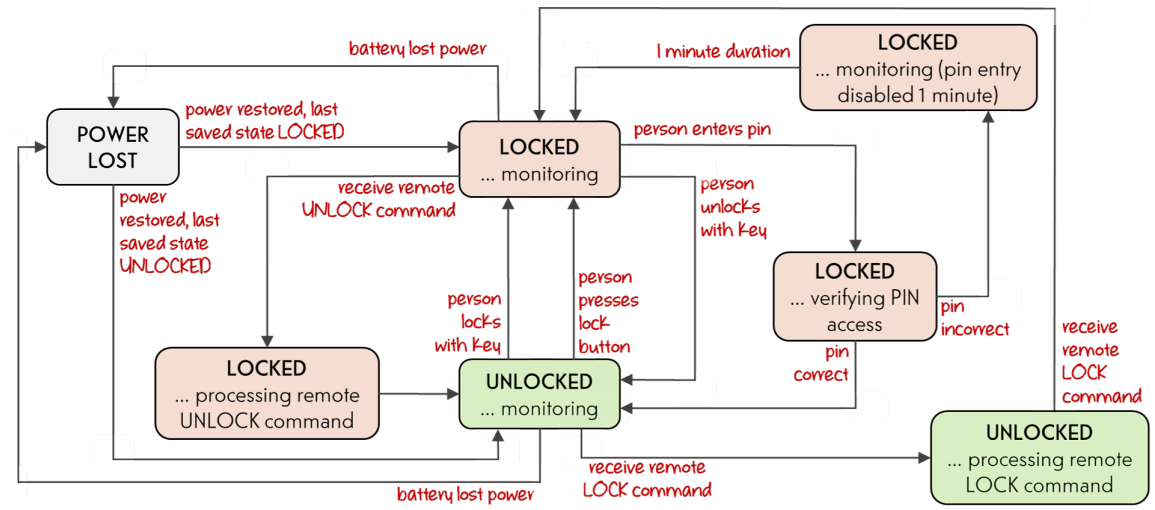

Figure 22.12- Smart lock "state diagram"

"Software FMEAs" can also help identify missing crucial use cases or systemic problems like inadequate communication between designers and software engineers. By addressing these concerns early, *"software FMEAs"* facilitate the development of robust, customer-centric products.


acuitas
reliability through insight