Software Quality, Verification, and Validation

Oscar Palm

Feb 2023

Contents

Chapter no. 1

Software Quality, Verification, and Validation

1.1 Introduction

1.1.1 Problems in our society

Our society depends on software, cars, water, energy, computers, everything is controlled by software.
Flawed software will hurt profits, fixing a bug after release and delivery is much more expensive than fixing it before its release. Even if bugs aren't noticeable by the end users or the company hosting it, bugs can lead to security exploits, and later breaches.
Flawed software can even hurt users directly, e.g. pacemaker crashing or av making a wrong turn.

1.1.2 Why this course

- What is "good" software?

  we determine this through quality dependencies

- What is the key to good software?s

  Verification and Validation.

- Exploration of testing and analysis activities of the V and V process

1.2 When is software ready for release?

1.2.1 The short answers

- We can't find any bugs
- When we have finished testing
- When quality is high

1.2.2 The long answer

We all want high-quality software, but be don't all agree what that means.

- We can't find any bugs

  but we can't find all bugs.

- When we have finished testing

  but we can't test everything.

- When quality is high

  but we don't know what that means.

We need to define what we mean by quality, and how we can measure it.

Quality attributes

- Performance

  Ability to meet timing requirements
- Security

  Ability to protect information
- Scalability

  Ability to grow the system to process more concurrent requests
- Availability

  Ability to carry out the task whenever needed
- Modifiability

  Ability to enhance the software to meet new requirements or fix bugs
- Testability

  Ability to easily find faults in the software
- Interoperability

  Exchange information with other systems
- Usablility

  Ability to be used by the intended audience and perform required tasks

  How easy it is to learn to use the software

These can easily conflict with each other, its important to decide what to prioritize and set a threshold of what's good enough.

1.2.3 When is software ready for release?

It's ready for release when it's **dependable**. This means it needs to be correct, reliable.

1.3 Verification and Validation

1.3.1 Verification

The process of checking that a system meets its requirements.
Are we building the product right?
Verification is an experiment. We perform trials, evaluate the results, and gather information about what and why it happened.

Testing

An investigation into system quality, it's based on sequences of stimulations and observations. The software version of a lab rat we later dissect to analyze what failed.

1.3.2 Validation

The process of proving that it meets the specifications set by the customer.
Are we building the right product?
Does the product work in the real world? Even if the software does exactly what we set out to do, it might not be what the customer wants.

1.3.3 Conclusion

Verification checks if the software works as intended.
Validation checks that the software is useful. (This is much harder)

Both are important and complete each other. This class however, focuses largely on verification.

- Testing is the primary activity of verification.

1.4 Required level of V and V

Depends on:

- Software Purpose

  The more critical, the more important that it works

- User Expectations

  Some users are more forgiving than others

- Marketing environment

  With competing products in a market, it might be more important to release a product quickly than to make it perfect.

1.4.1 Basic questions

1. When do verification start and end?
   - It should start as soon as the project starts
     we need to know what we're building and how design/technical choices affect our product's quality
     A great starting point is static verification
   - It ends when the product is released
     A great way of verifying during the development process is through dynamic verification
2. How do we obtain an acceptable level of quality at an acceptable cost?
3. How do we decide when it's ready to release?
4. How can we control quality during the development process?

1.5 Trade offs

There's always a trade-off when designing software, "Better, faster, or cheaper - pick any two".

1.5.1 Verification Trade-offs

We are interested in proving that a program demonstrates property X

- Pessimism inaccuracy

  Not guaranteed to program even if X is true

- Optimism inaccuracy

  May be true, even if X is false

- Property Complexity

  if X is too difficult to check, substitute with simpler property Y

Finding all faults is nearly impossible, instead we need to decide ourselves when we are ready for release, how good is good enough?

We need to establish criteria for what is good enough, and what is not. One way of doing this is through **Alpha/Beta testing** where a small group of users gets the chance to use the product in a somewhat controlled environment and reports feedback and failures.

Chapter no. 2

Quality Attributes and Measurement

2.1 Quality Attributes

Developers prioritize attributes and design systems that meet their needs.

2.1.1 Availability

Ability to avoid completely or recover quickly from failures. Redundancy.

2.1.2 Performance

Ability to meet timing or throughput requirements. The system needs to be able to respond quickly to events.

2.1.3 Scalability

Ability to scale the system performance to meet increased load. The system needs to be able to handle more concurrent requests.

2.1.4 Security

Ability to protect information. The system needs to be able to protect information from unauthorized access while still allowing authorized access.

2.2 Scalability

2.2.1 When is software ready for release?

It's ready for release when it's **dependable**. This means it needs to be correct, reliable, safe, and robust.

Correctness

A program is correct if it is always consistent with its specification. This depends on quality and detail of requirements, it's really easy to build a correct program with a weak specification, but if it's too detailed it becomes impossible to prove your program is completely correct.

More often than not, correctness is something we aim for, not prove.

Reliability

A statistical approximation of correctness, the probability that the program will perform correctly during a given period of time under a given set of conditions. We test reliability by running the program as different types of user profiles, and mainly focuses on reliability for our target audience.

Dependence on specifications

Correctness and reliability are dependent on the quality and strength of the specification. The more detailed the specification, the more likely the program is to be correct and reliable in the real world.

Correctness and reliability doesn't consider the severity of different crashes and bugs, a program that crashes once a year is more reliable than one that crashes once a day while in the real world it might be better with a daily crash than leaking all personal data once a year.

Safety

Safety is the ability to avoid hazards. We specify a set of undesirable situations, hazards, and prove that our program avoids them.

Robustness

Software that is correct may fail when our design assumptions are violated; *how* it fails matters. Software that gracefully fails is robust, e.g. if we tries to save a program to a read-only disk, the program should tell us what wet wrong gracefully instead of crashing.

Robustness cannot be proven, but is rather a goal to aspire to.

2.2.2 Dependability

We could have software that is reliable, but not correct, or correct but not safe, or robust but not safe. We need to consider all of these attributes when we talk about dependability.

Measuring Dependability

We need to establish criteria for when our system is dependable enough for release.
Correctness is too hard to prove conclusively for most programs.
Robustness and safety is important, but doesn't prove that our program functions correctly.
Reliability is the basis for arguing dependability, we can measure it, and we can demonstrate it through testing.

2.3 Reliability

Reliability is the probability of failure-fre operation for a specified time in a specified environment for a given purpose. This depends heavily on the system and user type.

2.3.1 Improving reliability

Reliability is improved when faults in our most frequently used parts are removed, this means that a program is more or less reliable for different users.

2.3.2 Reliability is measurable

Reliability can be defined and measured. We can specify requirements (both functional and non-functional) and measure how well our program meets them.

2.3.3 How to measure reliability

Hardware metrics often aren't suitable for software, since in hardware it can only hard crash and we can assume that the design of the hardware is correct.

With software most of the failures are design failures, and when a system has failed the system is often still available.

Availability

Can the software carry out its given task when needed. Can the system avoid failures, and recover quickly from failures. Can the system beep working for other users when it has crashed for one?

Availability is only a measurement of whether the system is available, not whether it's correct or reliable meaning incorrect computations or security isn't considered.

Availability is also a standalone quality attribute. We can through design prevent, tolerate, remove, or forecast failures. We can keep our system partially available more easily than hardware.

Probability of Failure on Demand (POFOD)

Likelihood that a single request will result in failure. A POFOD of 0.001 means that there is a 0.1% chance that a single request will fail. This is used in situations where failure is unacceptable, e.g. a medical system.

Rate of Occurrence of Fault (ROCOF)

Frequency of occurence of unexpected behaviour. A ROCOF of 0.02 means that we have 2 failure per 100 time units. Is appropriate when requests are made regularly, like a web server.

Mean Time Between Failures (MTBF)

Average time between failures. If we have a system that is used for long sessions, especially where users might only save their data once every few hours or so, it might be important to prevent crashes happening too often.

2.3.4 Reliability Metrics

- Availability: $\frac{\text{Uptime}}{\text{Total time observed}}$
- POFOD: $\frac{\text{Failures}}{\text{Requests over period}}$
- ROCOF: $\frac{\text{Failures}}{\text{Time elapsed in target unit}}$
- MTBF: Average time between observed failures.
- MTTR: Average time to recover from failures.

It may be cheaper to accept some unreliability and pay for failure cost instead of trying to make the system more reliable. This depends on social/political factors.

2.4 Performance

The ability to meet timing or throughput requirements. This is the driving factor in software design, even though it is often at expense of other quality attributes. All systems have performance requirements, but they are often not specified. Bad performance leads to a decrease in users.

2.4.1 Performance Measurements

- Latency

  The time between arrival of stimulus and the systems response to it.

- Response Jitter

  The allowable variation in latency.

- Throughput

  Usually number of transactions per unit time.

- Deadlines in processing
- Number of events not processed

Latency

Time it takes to complete an interaction, how quickly the system responds to a user. We measure latency probabilistically, e.g. 95% of requests are completed within 100ms.

Another type of latency is the turnaround time, the time it takes to complete a larger task, e.g. With daily throughput of 850,000 requests, process should take ¡ 4 hours, including writing to a database.

Response Jitter

Response time is non-deterministic, if we have an average latency of 5 seconds, we might have a latency of 2 minutes at a specific time, response jitter defines how large the maximum latency can be.

Throughput

The workload a given system can handle, usually measured in transactions per unit time. Since a larger throughput can lead to a longer service time, this can conflict with latency requirements.

Deadlines

Some special tasks must take place at a specific time.

Missed events

If our system is busy, we might need to ignore some events. This is usually a problem for real-time systems.

2.5 Scalability

The ability to process an increasing number of requests. There are two types of scalability:

- Horizontal scalability

  Adding more resources to the system, e.g. adding more servers to a web server farm.

- Vertical scalability

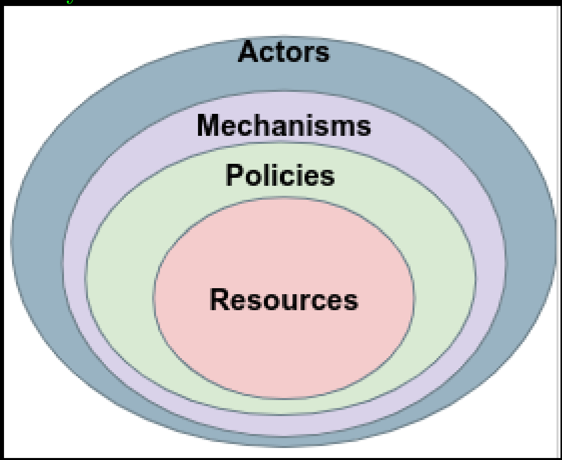
  Adding more power to a single resource, e.g. adding more RAM to a single server.

How can we effectively utilize additional resources? This requires that additional resources:

- Result in performance improvement
- Didn't require undue effort to add
- Did not disrupt operations
- **Must be designed to scale**

2.6 Security

Ability to protect data and information from unauthorized access while still providing access to authorized people and systems.



Processes allow owners of resources to control access. Actors are systems or users. Resources are sensitive elements, operations, and data of the system. Policies define legitimate access to resources.

2.6.1 Security Characterization (CIA)

Confidentiality, Integrity, and Availability.
Authentication, Nonrepudiation and Authorization are also important.

2.6.2 Security Approaches

We achieve security by detecting, resisting, reacting, and recovering from attacks. Data needs to be protected, both when in transit or when at rest.

Security is a process, not a product. Nothing is completely secure or unsecure, and all systems will be compromised. We need to be able to detect and recover from attacks as well as figure out who attacked.

2.6.3 Assessing Security

Measure of system's ability to protect data from unauthorized access while still providing service to authorized users. we need to assess how well system responds to attack. We can respond by e.g. logging, blocking, or shutting down the system.

There isn't one single metric for measuring security, in some cases what data is more important, in some cases how much data was lost.

2.7 Key points

Dependability is one of the most improtant software characteristicts. Reliability depends on the pattern of usage of the software, different users will interact differently.
Reliability measured using ROCOF, POFOD, AVAILABILITY, MTBF, MTTR.
AVAIlability is the ability of the system to be available for use. Performance is about management of resources in the face of demand to achieve acceptable timing Scalability is the ability to "grow" the system to process an increasing number of requests Security is the ability to protect data and information from unauthorized access Security is not "measured", but requires defining attacks and actions to prevent or reduce impact of risk, then assessing those actions

Chapter no. 3

Quality Scenarios

3.1 Scenarios

A scenario is a description of an interaction between external stuff and the system. It defines:

- An event that triggers the scenario.
- An interaction initiated by the external stuff.
- What response is required.

This is similar to use cases or user stories, but examines **both** quality and functionality.

We use scenarios to capture a wide range of requirements:

- A set of interactions with users to which a system must respond
- Processing in response to timed events
- Peak load situations
- Regulator demands
- Failure response
- Maintainer changes stuff
- Whatever the design is required to handle

3.1.1 Scenario usages

Scenarios are used to; provide input to architecture definitions, by helping fleshing out your requirements and finding new ones; Evaluate system architecture, by finding missing or incompatible interfaces; communicating with stakeholders, gives easy to understand examples of what the system can do; driving the testing process by helping prioritizing your testing.

3.1.2 Scenario format

Overview

A brief description of what the scenario is.

External stimulus

What initiated the scenario? A user request, a timer, a failure, a maintainer, etc.

System state

Aspects of the system's internal state that might affect quality, such as does data exist already? is the system under high or low load, and so on.

System environment

How does the environment around the system look? How does our infrastructure look? Does the system have gigabit ethernet? is the system air gapped?

System response

How does the system respond to our stimuli? How should the system respond to meet our quality requirements?

Response measure

How does we judge the quality of the response?

3.1.3 Response measure

Most quality measurements are non-deterministic. All time-based measures should be probabilistic (95% of the time the response should be N, 99% of the time it's M).

For real-time systems all time measurements should give a worst-case response time. Otherwise you can give an absolute threshold.

3.1.4 What do we do with scenarios?

We use them primarily to test and improve our design. It's also a great way of showing to stakeholders what the system can do and how it can be used.

We also use scenarios for exploratory testing, where humans are used to test the system. We also use them for formal test cases.

3.1.5 What is a good scenario?

credible, valuable, specific, precise, comprehensive

- credible

  it should be a realistic scenario

- valuable

- precise

- specific

- comprehensive

3.1.6 Effective scenario use

Identify a focused set of scenarios that are representative of the system's behavior. Too many scenarios can be a hindrance, no more than 20 scenarios.

Use distinct scenarios, it's much better to have wildly different scenarios than to have a few scenarios that are very similar.

Use scenarios early, they are most important in the early phase, later on it can be hard to change the system behaviour after a new set of scenarios.

3.1.7 Reliability scenarios

The ability to minimize the number of observed failures. These resolve around one function accessed through an interface.

Reliability scenario format

- Overview

  Highlight the function(s) being used and which context they are used in

- System state

  Data stored or past events may impact our reliability

- Environment state

  Our available resources can change how reliable our system is, an underpowered processor may cause a system to fail.

- External stimulus

  A user or external system performs one or more input actions

  State specific interactions performed

  If relevant, the type of user and how they perceive reliability

- Required response

  The functional response of the system

- Reliability measure

  how should we measure the reliability of the system and how reliable should it be?

3.1.8 Availability scenarios

Ability of the system to mask or repair failures such that the outage period does not exceed a required value over a time period. In other words, we look at how the system responds to a failure, how long it takes, and what it does to return to normal operation.

We need to distinguish between failures and the software's perception of failure, e.g. when the network goes down, the system doesn't count that as a failure until it needs to access the network, meaning we have time to recover.

Scenarios tends to deal with; failure of internal components or external systems, reconfigure of the physical system, or maintenance/reconfigure of the software.

Availability scenario format

- Overview

  Highlight the function(s) being used and which context they are used in.

- System/environment state
- External stimuli

  stimuli is omissions, crashes, wrongful timings, incorrect responses and so on.

- Required response

  Failure needs to be detected and isolated before we can recover.

- Response measure

  Can specify an availability percentage

  Can specify a time to detect/repair or when it needs to be available and for how long

3.1.9 Availability vs Reliability

Reliability is about how it operates normally and how often it fails. Availability is about how it operates when it fails and how we avoid hard crashes and downtime.

Chapter no. 4

System testing and test case design

4.1 Testing stages

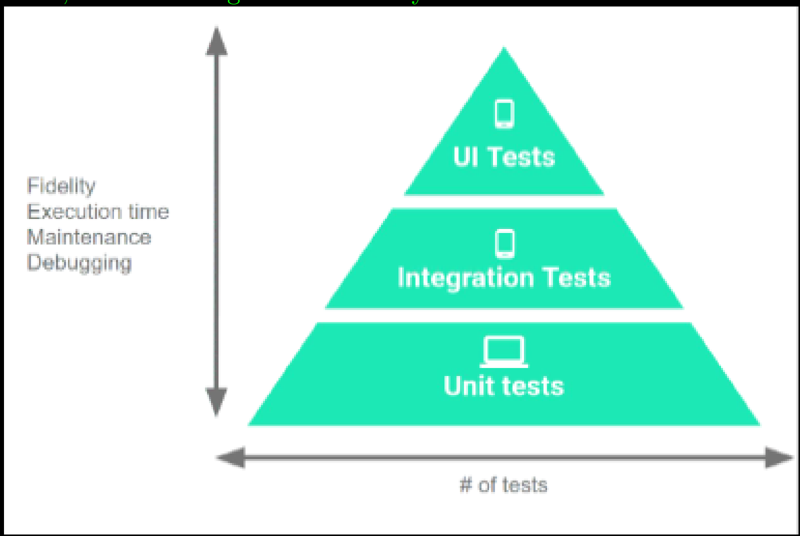
We interact with our systems through interfaces, such as CLIs and GUIs. These systems are in turn composed of subsystems which also has their own interfaces.

System-level testing is about testing the whole system, or parts of it, through an interface. It integrates lower-level components. Many errors first occur when we try to connect several subsystems together, these are the main errors we are looking for in system-level testing.

4.1.1 Unit vs System testing

Unit is about testing a single class, implementation testing, while system testing is about testing the whole system, integration testing.

Often we use the 70/20/10 model, 70% unit testing, 20% integration testing and 10% UI testing. This is because without functioning units, of course the integration testing will fail, and without functioning integration between units, the UI testing will most likely fail.



4.2 Interface types

4.2.1 Parameter Interface

Data passed through method parameters, the functions/methods you can call in a class.

4.2.2 Procedural Interface

An interface which surfaces a set of lower level functions to be able to be called by other components, an example of this is an API, or a GUI.

4.2.3 Shared memory interface

A block of memory shared between systems, a shared database, or data bus.

4.2.4 Message-passing interface

Instead of having two systems connected directly, an intermediary handles the sharing of messages between them.

4.3 Interface errors

4.3.1 Interface misuse

Malformed data, or a wrong number of parameters

4.3.2 Interface misunderstanding

Incorrect assumptions made about called component, e.g. binary search in unordered array.

4.3.3 Timing errors

The producer or consumer reads/writes in the wrong order.

4.4 Creating System-level tests

1. Identify a function that can be tested in relative isolation.
2. Identify controllable aspects of the input and environment that determine the outcome of the function.
3. Identify types of values for each choice that lead to different function outcomes.
4. Combine values to form "recipes" for test cases.
5. Replace representative values with concrete values.

Independently testable functionality are well defined functions, most often based on the "verbs". For UI-tests, look for functions visible to the user.

4.5 Units and "Functionality"

Many tests are written in terms of "units" of code. An independently testable function is a *capability* of a unit.

When testing functions we don't need to test every possible input, most inputs gets similar response, e.g. when testing a number input, maybe test negative, normal, zero, strings , and a large integer, not every single possible input.

To figure out what to test, we usually divide all input into different partitions, and test from all partitions. Look att output events, ranges of values, membership groups, timing, and other aspects of the function.

Chapter no. 5

Quality Scenario Exercise

5.1 Exercise description

You have been asked to develop a new automated parking system at the GOT airport.

In this new system, a user can simply insert their credit or debit card into the card reader at the ramp entrance. This will record the time they entered airport parking. They then can use the same credit or debit card to pay at an exit lane. The system should be fully automated; there is no waiting in line for a cashier. The system should also support ticketed parking: where the user receives a ticket and pays either by credit card or cash on exiting.

The system needs to interact with a number of entities and systems, including:

- Customers parking in the ramp
- Airport police and emergency responders
- Ramp managers
- External systems for validating credit card details and submitting payments
- The airport's accounting system
- External physical gate systems with basic controllers (raise / lower)
- External physical systems for signage
- An existing personnel system for staffing exit kiosks

The system will be deployed within the physical architecture of the airport parking garage, incorporating:

- Entrance Kiosks
  - Card dispensers
  - Credit card reader
  - Card reader for contract parking
  - Signage: OPEN / CLOSED
  - Entry gate
- Parking Entrance
  - Signage FULL / NOT FULL

* Exit Kiosks

    Signage: OPEN / CLOSED

    Staffed kiosks

    Automated kiosks

    Exit gate

* Security cameras

* Hardware for Parking System

    Dual server w/failover (can switch in event of failure)

    Clustered DB

    Storage area network

You will describe scenarios using the following template:

- Overview

    Brief description of what the scenario illustrates, provides important context.

- System State

    Aspects of the system state that affect quality (i.e., information stored in the system, number of concurrent logged-in users, previous failures that may influence execution)

- Environment State

    Significant observations about the environment that the system is running in that could influence the quality of the system

- External Stimulus

    Input events or environmental factors that initiate the scenario. (i.e., infrastructure changes or failures, security attacks, etc.)

    Name both the actor who initiated the scenario and the action performed.

- Required System Response

    How should the system respond (from a functional point of view)? (i.e., how should it handle a defined increase in requests?)

    Focus on actions that relate to the quality attribute of interest.

- Response Measure

    How success is defined and measured, along with a threshold that must be defined for success.

    Based on the quality attribute of interest.

Based on the above template, create the following:

1. Create a reliability scenario (using any of the response measures EXCEPT for availability).
2. Create an availability scenario.
3. Create a performance scenario.
4. Create a scalability scenario.
5. Create a security scenario

5.2 Scenario 1 (Avaliability? ish)

- Overview

  The card reader loses its connection and the system is unable to charge the user. The user therefore has to try and pay multiple times.

- System State

  System is operating normally. influence execution)

- Environment State

  The system is under a normal load of about 150 passthroughs per hour.

- External Stimulus

  The system fails to connect to its external payment system.

- Required System Response

  If the system fails, prompt the user to try, and try again.

- Response Measure

  The system is allowed to fail no more than 1 time per user, and a maximum of 5 times per hour.

5.3 Scenario 2 (Availability)

- Overview

  Parts of the clustered database fails, thanks to malformed data.

  Rats have chewed through the cables, causing the system to fail periodically.

- System State

  The system is operating normally.

- Environment State

  The system is under a normal load of about 150 passthroughs per hour.

  Garage is not empty or full. All physical equipment is working.

- External Stimulus

  the broken cable leads to a deformation of system data.

- Required System Response

  When the system fails, a backup system should be able to take over and continue to operate without the user noticing.

- Response Measure

  The system should have an availability of 98%.

  Each system failure needs to be at least 30 minutes apart as so to give enough time for our technicians to recover the system.

5.4 Scenario 3 (Performance)

- Overview

  The system is unable to handle the load of the users.

- System State

  The system is operating normally.

- Environment State

  The system is under a normal load of about 150 passthroughs per hour.

  Garage is not empty or full. All physical equipment is working.

- External Stimulus

  The system is unable to handle the load of the users.

- Required System Response

  The system should be able to handle 200 passthroughs per hour.

- Response Measure

  Each passthrough should take no more than 30 seconds.

  The system should be able to handle 200 passthroughs per hour.

5.5 Scenario 4 (Scalability)

- Overview

  Thanks to an increase in load after the holidays, management has decided to open another set of kiosks.

  Previously, the system consisted of 3 entrance, and 3 exit-kiosks, with another entrance and exit kiosk being added.

- Environment State

  Each of the open kiosks is able to handle 50 passthroughs per hour but.

- External Stimulus

  The system is under a constant high load of about 200.

- Required System Response

  The system should be able to handle 200 passthroughs per hour meaning each added kiosk should handle 50 passthroughs per hour.

- Response Measure

  Each passthrough should still take 30 seconds.

  The system should be able to handle the adding of more kiosks without having to take it down.

  The system should now be able to handle 50 extra passthroughs per hour.

5.6 Scenario 5 (Security)

- Overview

  A hacker has gained access to the system and is able to increase each cars parking time.

- System state

  The system is operating normally.

- External stimulus

  A malicious USB stick has been attached to our server.

- Required system response

  Even if the system is compromised, the system should still be able to handle the load of the users, and no personal information should be leaked.

  Our logs should clearly state which data has been accessed/modified and where it might have been sent to.

- Response measure

  We should contact the users affected, and compensate them.

  We should use the logs to figure out where it went wrong and fix the hole.

5.7 Scenario 6 (Reliability)

- Overview

  A car tries to enter

- System state

  The system is operating normally.

- Environment state

  Everything is normal

- External stimulus

  The car wants to enter but cannot

- Required system response

  The car should pass through the entrance kiosk

- Response measure

  The ROCOF of the system should be less than 0.005