GA03 Deliverable

M.U.P

massively underdeveloped project

Documentation

	Måndag	Tisdag	Onsdag	Torsdag	Fredag	
Calle			1			1
Kim			1			1
Nils			1			1
Rasmus						0
						3

Updating GA02

	Måndag	Tisdag	Onsdag	Torsdag	Fredag	
Calle	4					4
Kim	3	1,5				4,5
Nils	4					4
Rasmus	1	1,5				2,5
						15

Updating / GA02

	Måndag	Tisdag	Onsdag	Torsdag	Fredag	
Calle				2,5		2,5
Kim				2,5		2,5
Nils						0
Rasmus				2,5		2,5
						7,5

Anylizing Changes

	Måndag	Tisdag	Onsdag	Torsdag	Fredag	
Calle						0
Kim	1					1
Nils						0
Rasmus	1					1
						2

Assignment planning

	Måndag	Tisdag	Onsdag	Torsdag	Fredag	
Calle		1				1
Kim		1				1
Nils						0
Rasmus		1				1
						3

Self evaluation

	Måndag	Tisdag	Onsdag	Torsdag	Fredag	
Calle		3				3
Kim		2				2
Nils		2				2
Rasmus		2				2
						9

Anylizing Changes

	Måndag	Tisdag	Onsdag	Torsdag	Fredag	
Calle		2	1			3
Kim		2	1			3
Nils		1	1			2
Rasmus		1				1
						9

	Total hour	Percentage
Calle	14,5	29,90
Kim	15	30,93
Nils	9	18,56
Rasmus	10	20,62
	48,5	100

Rasmus Tilljander - rati10@student.bth.se
Nils Forsman - nifo08@student.bth.se
Calle Ketola - cake10@student.bth.se
Kim Hansson - kiha10@student.bth.se

2012/10/12

Updates to GA02

Motivations:
* Some motivations given, but no alternatives to design decisions are discussed
Alternatives are now discussed.

Revisit assignment 1:
* 3 issues have solutions, but no concrete strategies to implement them
Added strategies to most issues that lacked such.

Strategy mapping to module view:
* You motivate the decisions well, but you should specify also the concrete strategy, not only the issue, which leads to the decision.
The corresponding strategies are now mentioned as well.

Module view:
* No traceability between conceptual and module view, i.e. not clear which and where conceptual components are reflected in the module view.
Added a short mention to each layer description which component that it reflects.

* Why are there 4 "Data transfer" modules? If they all have the same functionality, they shouldn't be replicated four times. If they are different, e.g. clients in the broker pattern, they should have different names
Each of the modules now has an individual name to show that they are different modules.

Strategy mapping to execution view:
* The design decisions are not connected to any strategy and/or issue (except broker strategy).
New issue-cards and strategies will be added so that we can map them to the execution view.

Execution view:
* Diagrams are not clear since notation is not followed. We need to see at least one diagram showing the execution architecture view and one with communication paths.
Communication paths have been added to the entities now and will be described in the text.

* There is no visible mapping between module view and execution view.
The mapping between the modules in the module view and the modules in the execution view and now mentioned.

Self-evaluation of architecture:
* Shallow description of alternative evaluation methods, choice weakly motivated.
Updated description of alternatives.

* Chosen method (BTH evaluation method) not described and process not explained.
Made a longer description of the chosen method.

* BTH method not followed (i.e. most relevant quality attributes identified, but not motivated why they are the most important ones; no normal/worst case scenarios defined)
Motivation has been added.

* Scenarios are shallowly analyzed
Added some more meat to the analysis.
* If you look at the description and advantages/disadvantages of the broker pattern (e.g. here: http://www.openloop.com/softwareEngineering/patterns/architecturePattern/arch_Broker.htm), do you still consider it as a good solution?
Yes. After reading the text we find that the pros of reusability and extensibility far outweigh the cons of lost fault tolerance and efficiency.

Architecture transformation:
* Even the weak scenarios identified one issue, leading to create a new issue-card ("Multiple crash issue"). How did this affect the architecture and which changes did you implement?
We had already taken this issue into consideration, it was just never documented properly. This will be mentioned in the evaluation now.

Introduction

The assignment consists of creating and documenting an architecture for a predefined system and in doing so learn how to transform quality goals into a practical solution.

The system is an automated test bench for use on various kinds of software. It will feed predefined input into the tested system and verifies the output, reporting any deviations. The system need to be easy to maintain and adding new features such as input types, emulation mechanics and testing techniques with minimal effort and cost.

It also needs to log all the testing data and generate this as a report containing test statistics, how to recreate the errors and the type of errors encountered.

The idea is that the system will be used in maintenance departments of software organizations with a demand for advanced and automated testing.

Assumptions

We have assumed that:

- All standard computers have an operative system that is either Windows, Linux or MacOS.
- A standard computer has at least a Giga bit network card, DDR2 memory, 2 Ghz single core.
- We do not emulate any software within the system, instead we channel the information through the system between the tested system and the exterior software.
- The MIB will be maintained for 20-30 years.
- The MIB will be developed by a competent team with the required skills to implement it.
- The Development does not have a tight schedule nor any specific budget constraints.
- An average software system will use 3-5GB of RAM.

Chosen Changes for GA03

- Scripting
- Looped Scripting
- test -j
- Quis custodiet?

Motivation to not choose the other changes

"All thumbs" and *"Emulate Schmemulate"* are not chosen since our analysis is that they would have a impact too significant to refactor within the timeframe given by the deadline.

"Emulate Schmemulate" would require us to add Crash detection and checkpoints to the emulation component. This would break the current standard we have for separating different functionality in different components.

With All thumbs we felt that we lack the knowledge of mobile phones and touch screen for finding valid solutions to this change. With enough time to research the subject it could be solved but this does not fit the staff assumptions made by our team.

Analysis

There was no need to change to current factors, instead we added four new factors, one for each change. These are marked in the factor table with red. All the new factors are product factors.

To cope with the changes we have designed two new issue-cards with corresponding strategies and solutions. These are also marked with red like the new factors.

There will be some changes to the architecture to accommodate the new functionality. Thankfully we already designed the system to handle generating new input from output received for different parts of the MIB so most changes will be localized to the testing wrapper and the new scripting component.

A scripting component will be added along with the corresponding modules for the module and execution view. It will be connected to the input/output component to enable it to feed new input into the system as well as define new output validation rules.

Running parallel tests on the same instance will require the testing wrapper to keep track of several testing techniques at the same time. With our current model this should not be a problem as long as they both have unique identifiers so that the correct input/output channels can be maintained.

All other components of the system should be unaffected on a architectural level.

System analysis

Factors

Technological Factor	Flexibility and Changeability	Impact
T1: General-purpose hardware		
T1.1 Memory type		
The system should use at least DDR2 type memory	The requirements of the tested system can exceed the requirements of the MIB	Higher system costs
T1.2 Network bandwidth		
The system should use at least a network card which supports at least 1Gbps	The requirements of the tested system can exceed the requirements of the MIB	Higher system costs
T1.3 Processor requirement		
The system should use at least one single core processor with a clock of 2 Ghz	The requirements of the tested system can exceed the requirements of the MIB	Higher system costs
T3.1 Operating system		
The system should be able to run on Windows, Linux or MacOS	The system might not be needed to run on different operating systems	The system will not need to be platform independent

Product Factor	Flexibility and Changeability	Impact
P1: Functional features		
P1.1 Various Input		
Simulating various inputs types	None	None
P1.2 Identify and Validate		
Identify output data and compare it to expected output	None	None
P1.3 Emulate Hardware		
Emulate various different hardware applications	Hardware emulation could be removed	Components handling the emulation of hardware would have to be removed. Limited change to the rest of the system
P1.4 Emulate Software		
Emulate various different software applications, platforms and protocols.	Software emulation could be removed	Components handling the emulation of software would have to be removed. Limited change to the rest of the system
P1.5 Logging the test		

The system should log all the test data passing through the MIB	None	None
P1.6 Generate new input		
The MIB should generate new input from the received output	Script rules for generating new input during runtime	More complicated component for handling generating new input
P1.7 Run parallel tests		
It should be possible to run several tests in parallel	Limit the amount of parallel tests that can be done	Less strain on the system in terms of data transfer
P2: User interface		
P2.1 Adapting to new functionality		
The user interface should be possible to adapt to added functionality such as new input types etc.	Addition of new functions after the system has been completed	The user interface will have to be updated to reflect changes in functionality
P2.2 Monitor MIB during runtime		
It should be possible to monitor all test data during runtime	Limit the amount of available data that can be monitored	Less components are affected thus making changes easier
P2.3 Scriptable input		
The user should be able to script new input during runtime	The system could be required to handle several scripting languages	Several scripting components would be needed
P3: Performance		
P3.1 Handling high throughtput		
Handling a throughput of 75Mbps and peek 150Mbps	The required throughput could be made even higher than at this moment	Changes will have to be made in those components that handle data transferring
P4: Dependability		
P4.1 Graceful recovery		
Input and output components should gracefully recover if the tested system crashed	None	None
P4.2 Restarting from checkpoint		
Restarting the tested system from a certain point after a system crash	Latency for restarting the test could be increased to allow different kinds of recovery or the need for restarting from a certain point could be removed	Changes to this factor would only affect the component that is responsible for restarting the tested system
P5: Failure detection, reporting, recovery		
P5.1 Robust system		
Since the system should be automated it have to be robust	The system might not have to be able to run unsupervised	Crashes could be tolerated
P6: Service		
P6.1 Adding new input types		
Adding new input mechanisms	None	None
P6.2 Adding new hardware emulations		

Adding new hardware emulation mechanisms	None	None
P6.3 Adding new software emulations		
Adding new software emulation mechanisms	None	None
P6.4 Adding test types		
Adding new testing techniques	None	None

Issues

Name: Multiple input issue
Description: The system must support several types of input simulations as well as being able to add new types if the costumer requires it. **Factors:** P1.1.1 Simulate various input types P4.1.1 Adding new input mechanisms
Solution: Making the interface between the module handling input and the sub modules for different input types work the same no matter what input type it is.
Strategies/Tactics: "*Generalize the module*" from "*Software Architecture in Practice Second Edition*" Chapter 5.3 Len Bass, Paul Clements, Rick Kazman 2003. By ensuring that the input type does not affect the functionality of the methods in the different modules it will be possible to add several different kinds of input.

Name: Multiple hardware emulations issue
Description: The system must support several types of hardware emulations as well as being able to add new types if the costumer requires it. **Factors:** P1.1.3 Emulate various hardware devices P4.1.2 Adding new hardware emulations
Solution: Explore standards for hardware communications currently used or in development to support most hardware emulations without impacting the system.
Strategies/Tactics: "*Maintain semantic coherence*" and *"Anticipate expected changes"* from *"Software Architecture in Practice Second Edition"* Chapter 5.3 Len Bass, Paul Clements, Rick Kazman 2003 By ensuring that each module is strictly separated from the others a change in one of the hardware emulations will not interfere with the other emulations and it would thus enable the addition of new modules without too much fuss.

Name:
Multiple software emulations issue
Description: The system must support several types of software emulations as well as being able to add new types if the costumer requires it. **Factors:** P1.1.4 Emulate various software applications P4.1.3 Adding new software emulations
Solution: Explore standards for software communications currently used or in development to support most hardware emulations without impacting the system.
Strategies/Tactics: "*Maintain semantic coherence*" and "*Anticipate expected changes*" from "*Software Architecture in Practice Second Edition*" Chapter 5.3 Len Bass, Paul Clements, Rick Kazman 2003 By ensuring that the software emulations are strictly separated it will be possible to add new emulations without being hindered by already existing emulations.

Name:
Multiple testing techniques issue
Description: The system must support several types of testing techniques as well as being able to add new types if the costumer requires it. **Factors:** P4.1.4 Adding new testing techniques
Solution: Keeping the semantics of the testing modules coherent so that further testing techniques can be added with minimal changes to the current structure
Strategies/Tactics: "*Maintain semantic coherence*" and **"***Anticipate expected changes* "from *"Software Architecture in Practice Second Edition"* Chapter 5.3 Len Bass, Paul Clements, Rick Kazman 2003 By making sure that each testing technique is separated from the others the addition of new techniques would not disturb the already existing techniques.

Name:
Not crashing with tested system issue
Description: The MIB must not crash just because the tested system crashes. This means that the input must be stalled until the tested system is running again. **Factors:** P4.2.1 Reliable input and output components
Solution: If a crash occurs the wrapper around the tested system will send a message to the data broker stating that further testing must halt until the system is running again. This will stop further data transfers from crashing the rest of the system.
Strategies/Tactics: *"Crash detection"* Detect and store the number of crashes.

Name: Creating report issue
Description: The MIB needs to be able to create a report once the testing is done. This report must contain data from both the output component as well as the log for all the test data. **Factors:** P1.1.2 Identify and compare output P1.1.5 Logging all test data
Solution: There must be a connection between the component handling the output verification and the component that logs all the test data so that data can be sent between them to be combined into the final report at the test end. This will be handled by the data broker.
Strategies/Tactics: “Broker pattern” A Broker works as the postal service, sending messages between the different components and the system. This keeps the coupling low.

Name: Keeping the system running through a test crash
Description: The MIB needs to be able to standby further testing if the tested system crashes until it has been restarted from an earlier point. **Factors:** P4.2.2 Restart tested system on crash
Solution: A component wrapped around the tested system will record the state of the tested system at regular intervals. If a crash occurs the component will restart the system using the latest checkpoint as reference.
Strategies/Tactics: “*Checkpoint/Rollback* "from *"Software Architecture in Practice Second Edition"* Chapter 5.2 Len Bass, Paul Clements, Rick Kazman 2003 A checkpoint records the state of the system and will be loaded on the event of a crash.

Name: Running system on all standard computers with required performance
Description: The MIB needs to be able to run on all standard computers on the market which have enough performance to run both the MIB itself and the tested system. **Factors:** O3.1.1 Development platform
Solution: Making the MIB cross-platform compliant
Strategies/Tactics:

Name: Data transfer
Description: Data and messages need to be sent between several components that do not have knowledge of each other **Factors:** P3.1.1 Large throughput of data P1.1.5 Logging test data
Solutions: We will implement a central data broker that will handle transferring data between components.
Strategies/Tactics: "Broker pattern" A Broker works as the postal service, sending messages between the different components and the system. This keeps the coupling low.

Name: Multiple crash issue
Description: When the tested system crashes repeatedly so that the test can't continue the testing needs to be terminated **Factors:** P4.2 Restarting from checkpoint P5.1 Robust system
Solutions: The wrapper around the tested system will keep track of which checkpoint was the last to be used to restart the tested system. If the same checkpoint is used more than a predefined amount this will signal that the testing needs to be terminated. The wrapper then sends a message to stop the testing up to the message handler in the data broker.
Strategies/Tactics: *"Crash detection"* Detect and store the number of crashes.

Name: Communication issue
Description: How will the system be constructed and what will the different communication-paths be. **Factors:**
Solutions: The MIB itself will be a multithreaded process with each thread communicating using intrathread communication. This will enable each component, held by a separate thread, to run simultaneously.
Strategies/Tactics: "M*ultithreaded processing*" We will implement a multithreaded process for the MIB. Thus the MIB will be a process that runs its different functionality on separate threads.

Name: "Quis custodiet?" issue
Description: Apart from being run automatic the system needs to support a monitored mode where a user can view all the test data and script new input to the MIB. **Factors:** P2.2 Monitor MIB during runtime P2.3 Scriptable input
Solutions: A Separate scripting component will feed scripted input into the input component for further transfer down to the tested system. Sending testing data up to be viewed during runtime will be handled by the data broker. This will be implemented as a separate mode so that the MIB can run as both an automatic system and a new monitored system.
Strategies/Tactics: *"Finite-state Machine"* enables the system to run two separate settings.

Name: Parallel testing issue
Description: The MIB must be able to run separate tests of the same instance of the system which means keeping check on which data goes where. **Factors:** P1.7 Run parallel tests
Solutions: This will require having the testing wrapper keep track of several testing techniques on the same instance. They will need a unique identification so that the checkpoint module and the data broker know which data goes to which testing technique.
Strategies/Tactics: "*Identifier"* Keep a unique identifier tied to each data transfer.

Name: Toggle in monitored mode
Description: A tester should not be forced to view the testing data if all he wants to do is script new input into the MIB. This is so that the performance loss from monitoring the test can be skipped for scripting. **Factors:**
Solution: We will make it possible to toggle certain functionalities in the monitored mode on and off to give the tester a higher degree of control.
Strategies/Tactics: "*Function toggling*" adapt certain functions to be toggled on and off.

Motivations for strategies and alternatives

Checkpoint/Rollback

We choose the *Checkpoint/Rollback* strategy to solve the issue with recovering from a crash of the tested system. Other options would have been to use something similar to *Active redundancy* but running one or several redundant copies of the tested system with these copies being run slightly behind the primary tested system ready to be switched over to in case of a crash. A third option would be to simply run the tested system from the beginning again while feeding it the same input. The benefit of the modified Active redundancy option would have been very short time to recover from a crash of the tested system, however, this option would also have a large footprint when it comes to both memory and processing resources. On the other end of the spectrum we have a restart of the tested system which will most likely result in quite long time to recover from a crash with the benefit of taking up little extra memory or processing resources. The *Checkpoint/Rollback* strategy that we choose have a large footprint when it comes to memory resources but do not require as much processing power as the modified *Active Redundancy*. We choose this strategy since it allows for relatively quick recovery and only have a large impact on memory resources.

Broker pattern

The reason we choose the broker pattern was to ensure strong decoupling between our components due to the many factors we have that concern extendibility. One option to this pattern would have been direct coupling, however, this would have led to a very high coupling which would in turn have made extendibility much harder. In the choice between improved performance and a high level of changeability we chose the latter.

Multithreaded processing

Another option would have been to separate functionality into different processes. However, this would result in potentially higher latency in the throughput since we relinquish a certain degree of control over data transfer to the operating system.

Finite-state machine

There are many modern alternatives to using state machines claiming better functionality, performance and changeability. But in this MIB we will only have a few states with very small differences in-between. This will not result in any major loss in functionality, performance or changeability. On the other hand we feel that keeping it simple with a strategy that can take many forms and implementations is the better choice.

Identifier

An alternative would be to have a direct communication between every communicating component so that no external identification is needed. This not an option for a system that is highly dependent on changeability and low coupling.

Another alternative is to have the data broker setup temporary connections between components. As long as the connection is up you can send data through to the other component unhindered and multiple connections could be open over the broker at the same time. The reason for not doing this implementation is that we foresee more smaller transfers to different components then one long transfer to the same component constantly. Setting up communication paths each time a smaller transfer is done and then closing it immediately would probably be more expensive than putting

headers/identifiers on each data transfer and letting the data broker sort them out. A hybrid system using both strategies depending on the transfer being made would probably be the most efficient, however, this will require more evaluation that we do not have time for as of this moment. But it will be considered for future implementations.

Conceptual view description

Here follows short descriptions of all the components of our conceptual view and their link to the strategies we decided on.

Scripting

This is the component for handling scripting new input and output to the MIB. This component was added to adapt the system to the “Scripting” change given by the assignment. It is linked to the issue-card and strategies: “Quis custodiet?”, “Finite-state Machine”.

I/O

This component handles the different types of predefined input and output using two sub-components, one for input and one for output.

Input

This part of the component handles the predefined input. It uses different modules for each input-type that in turn have a common interface such as was decided by the strategy mentioned in the issue-card for *"Multiple input issue"*, strategy: **"***Generalize the module".*

Output

This part of the component takes care of validating the output from the tested system against the predefined output. Like the input component it will houses separate modules for each type of output validation and in turn have a common interface. It will also create the final report once the data log has been received, which is mentioned on the issue-card *"Creating report issue".*

Data Broker

This component is responsible for transferring data between the other modules. It houses two components, the data distributor and the message handler. The data distributor sends data between the components that do not have knowledge of each other making it a central part of the construction. The message handler takes care of telling the system to halt further testing in case of a crash in the tested system. It will also send a message to resume testing once the tested system has been restarted. The issues mentioned on the following issue-card are relevant for this component:

“Data transfer”, “Creating report issue”, “Not crashing with tested system issue”

Emulation

This Component holds the two separate components for handling software-interfaces and hardware emulation.

Software

This is the component for the different software-interfaces that will channel different types of software data. The issues mentioned in the issue-card *"Multiple software emulations issue* "will be handled here by maintaining semantic coherence between the interfaces and by exploring standards in software communications.

Hardware

This component does the emulation of the different hardware devices, one separate structure for each device. Like the software component mentioned above it will solve the issues of the issue-card *"Multiple hardware emulations issue"* by maintaining semantic coherence between components and by exploring standards for hardware communication.

Datalog component

This component receives all the data from the testing and categorizes it, then stores it so that it can later be made into the final report by the output-component which is mentioned by the issue-card *"Creating report issue".*

Test wrapper

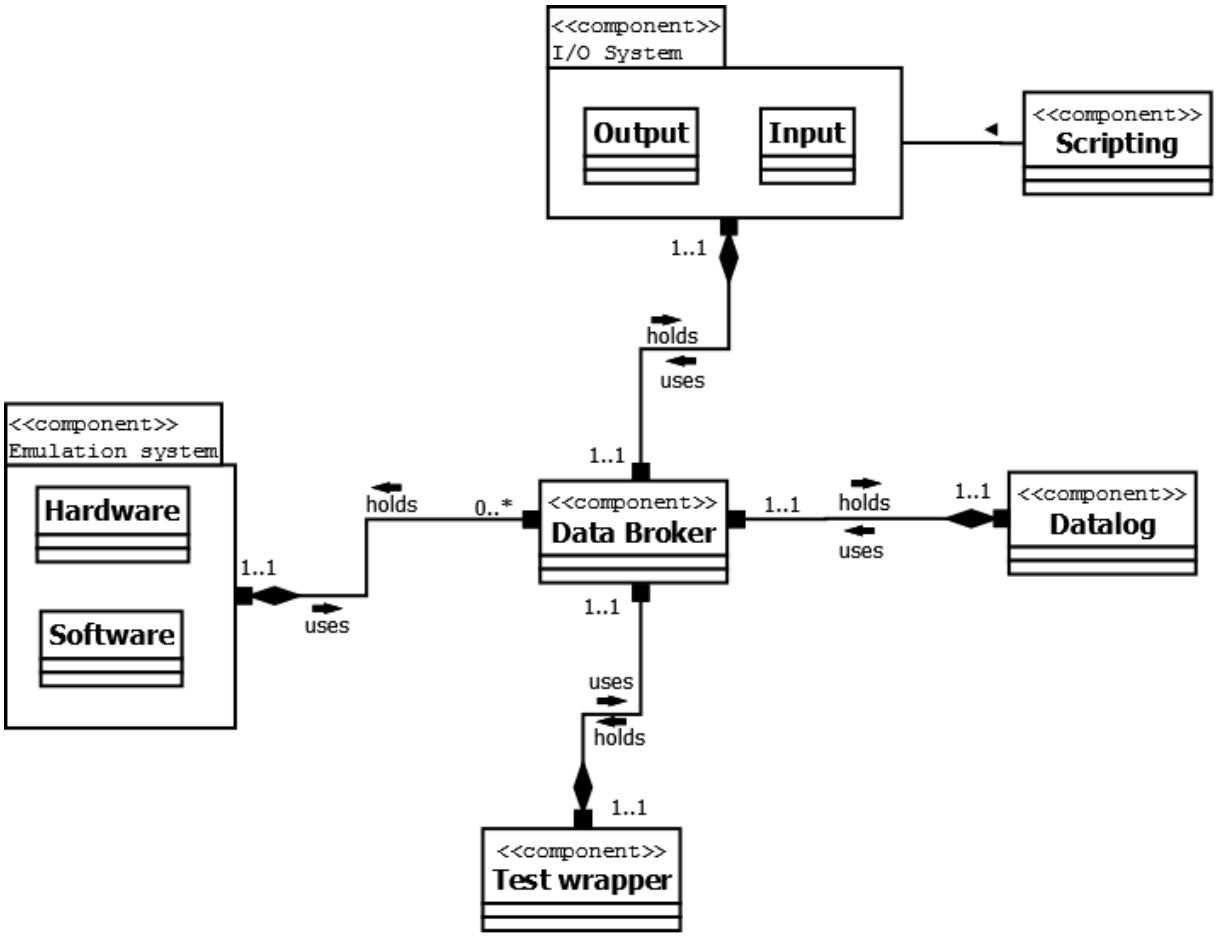
This component functions as a wrapper for the tested system as well as storing a checkpoint with the status of the tested system. With this the tested system can be restarted with minimal latency if it crashes. It will also send a message to the data broker to halt the testing until the tested system has restarted. This solves the issue mentioned in the issue-card *"Keeping the system running through a test crash"* by using the strategy mentioned in the same card,

Strategy: *"Checkpoint/Rollback"*.

The semantics in the wrapper will also be kept coherent so that new testing techniques can be added, thus solving the issue mention in the issue-card *"Multiple testing techniques issue" with its strategies:*

"*Maintain semantic coherence*" and "*Anticipate expected changes"*

Conceptual View



Module view description

Emulations

Based on component: Emulation system

This layer gathers the modules that emulate hardware and channel the data from software. We took the decision that these will benefit from ease of communication as well as similar data transfer to the rest of the system since an emulated software might be required to send data to software and vice versa. Data to the rest of the system is sent through the data transfer module. This layer and its modules are a product of the following issue-cards and strategies:

- *"Multiple software emulations issue, with strategy: Maintain semantic coherence and Anticipate expected changes"*
- *"Multiple hardware emulations issue, with strategy: Maintain semantic coherence and Anticipate expected changes"*

Testing

Based on component: Test wrapper

This layer holds the modules tightly connected with the actual system to be tested. When the tested system crashes the wrapper will need quick access to the checkpoint module so that the system can be restarted with minimal latency. Data to the rest of the system is sent through the data transfer module. This layer and its modules are a product of the following issue-cards and strategies:

- *"Keeping the system running through a test crash with strategy: Checkpoint/Rollback"*
- *"Not crashing with tested system issue, with strategy: Crash detection"*
- *"Multiple testing techniques issue, with strategy: Maintain semantic coherence and Anticipate expected changes"*

Data Log

Based on component: Datalog

Here all the data from the testing is categorized and stored into a storage-module until it can be sent up to the report module at the end of the test. Data to the rest of the system is sent through the data transfer module. This layer and its modules are a product of the following issue-cards and strategies:

- *"Creating report issue, with strategy: Broker pattern"*

Data Channeling

Based on component: Data Broker

Holds the data broker subsystem that handles transferring data between different parts of the system that do not have knowledge of each other as well as the message handler that takes care of informing the rest of the system in case of a crash in the tested system. This layer and its modules are a product of the following issue-cards and strategies:

- *"Data transfer, with strategy: Broker pattern"*
- *"Not crashing with tested system issue, with strategy: Crash detection"*

- *"Creating report issue, with strategy: Broker pattern"*

Input/Output

Based on component: I/O system

This layer holds the module for the Input managing as well as the subsystem for validating the output and writing the report. We put the report module and the validation module in the same subsystem since the output will not have to be validated until the end of the test for the report. This layer and its modules are a product of the following issue-cards and strategies:

- *"Multiple input issue, with strategy: Generalize the module"*
- *"Creating report issue, with strategy: Broker pattern"*

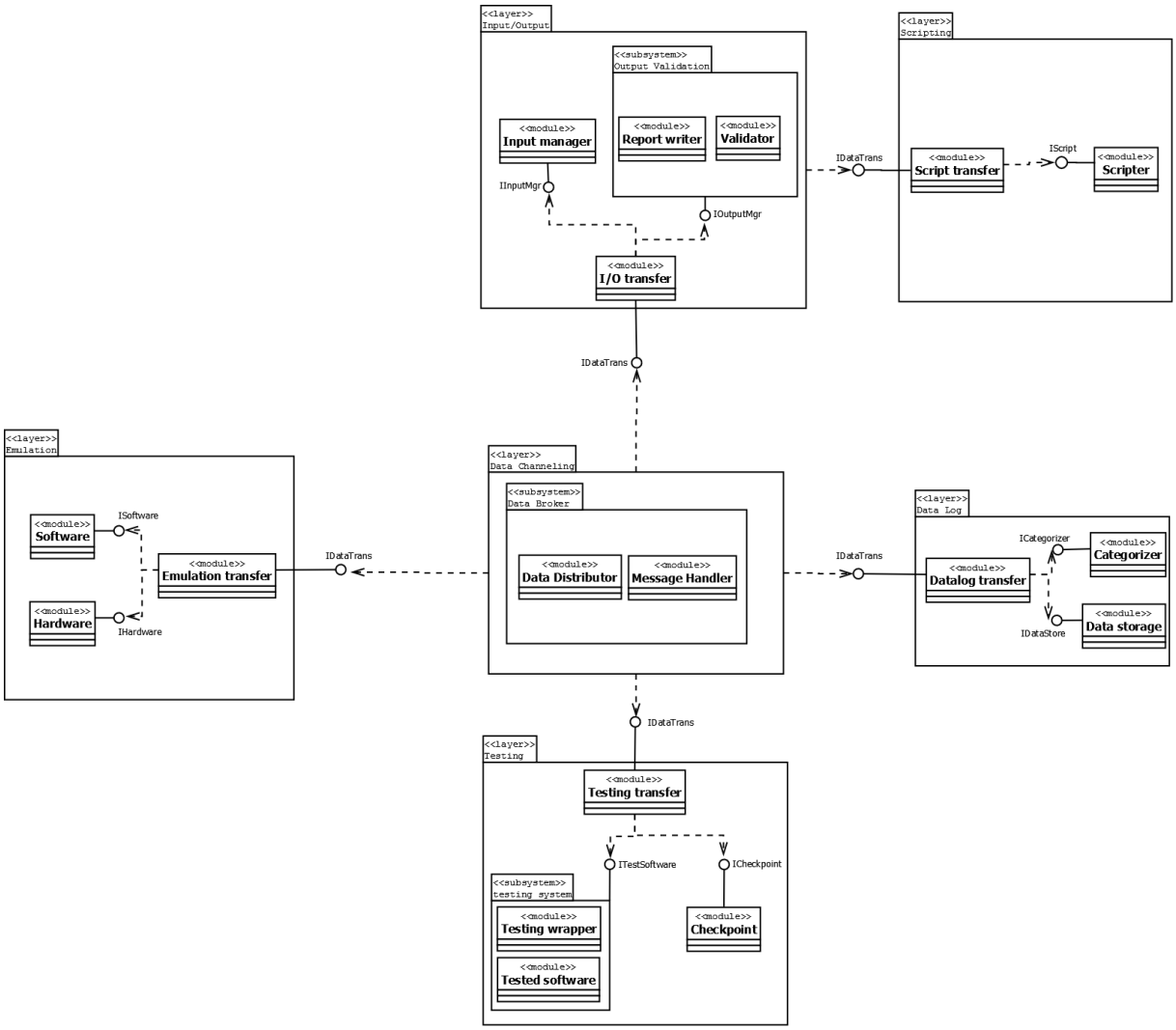
Scripting

Based on component: Scripting

This layer holds the module for scripting new input and output that will then be transferred to the I/O system. We made a layer around just one module to simplify adding more modules later if made necessary by changing factors. This layer and its module are a product of the following issue-cards and strategies:

- "Quis custodiet?, with strategy: Finite-state Machine strategy"

Module view


<<layer>>
Input/Output
<<subsystem>>
Output Validation
<<module>>
Input manager
<<module>>
Report writer
<<module>>
Validator
IInputMgr
IOutputMgr
<<module>>
I/O transfer
IDataTrans
<<layer>>
Scripting
<<module>>
Script transfer
IScript
<<module>>
Scripter
<<layer>>
Emulation
<<module>>
Software
ISoftware
<<module>>
Emulation transfer
<<module>>
Hardware
IHardware
IDataTrans
<<layer>>
Data Channeling
<<subsystem>>
Data Broker
<<module>>
Data Distributor
<<module>>
Message Handler
IDataTrans
<<layer>>
Data Log
<<module>>
Datalog transfer
ICategorizer
<<module>>
Categorizer
IDataStore
<<module>>
Data storage
IDataTrans
<<layer>>
Testing
<<module>>
Testing transfer
ITestSoftware
ICheckpoint
<<subsystem>>
testing system
<<module>>
Testing wrapper
<<module>>
Tested software
<<module>>
Checkpoint

Execution view description

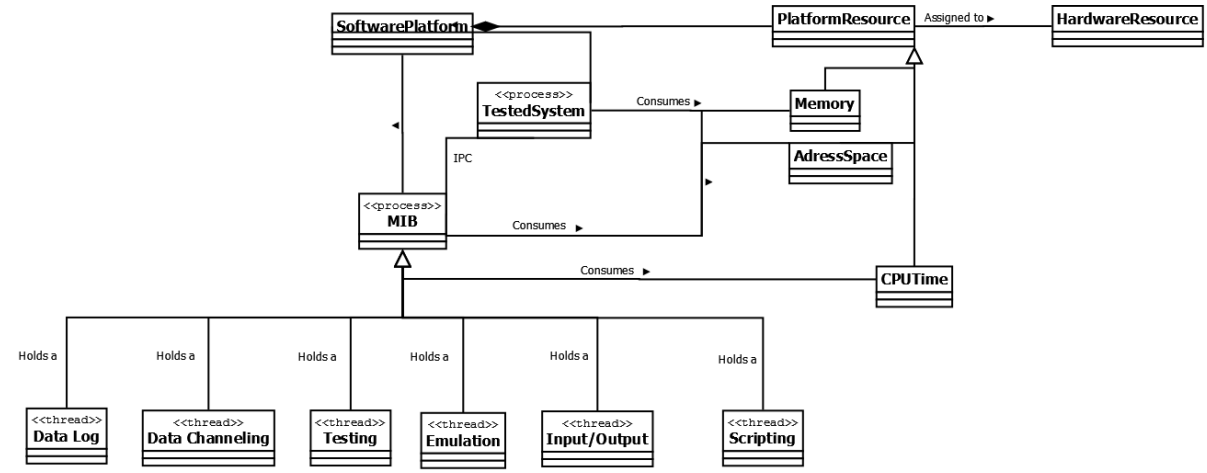
The software platform will house two processes, one for the MIB itself and one for the tested system. These will communicate via intraprocess communication, IPC. The MIB will be a multithreaded process that has one thread for each major component in the system so that they can work independently. These threads will communicate via intrathread communication, ITC. The reason for this design is tied to the choice of using a multithreaded process strategy. We also mapped which resources each entity needs. The processes consumes Memory and Address-space where as the threads consume CPU-time.

In the diagram we have first described this relationship as well as the relationship with the platform resources. After this we have divided each thread-entity into single diagrams containing those modules that are coupled to that entity thus giving a better view that is not too cluttered.

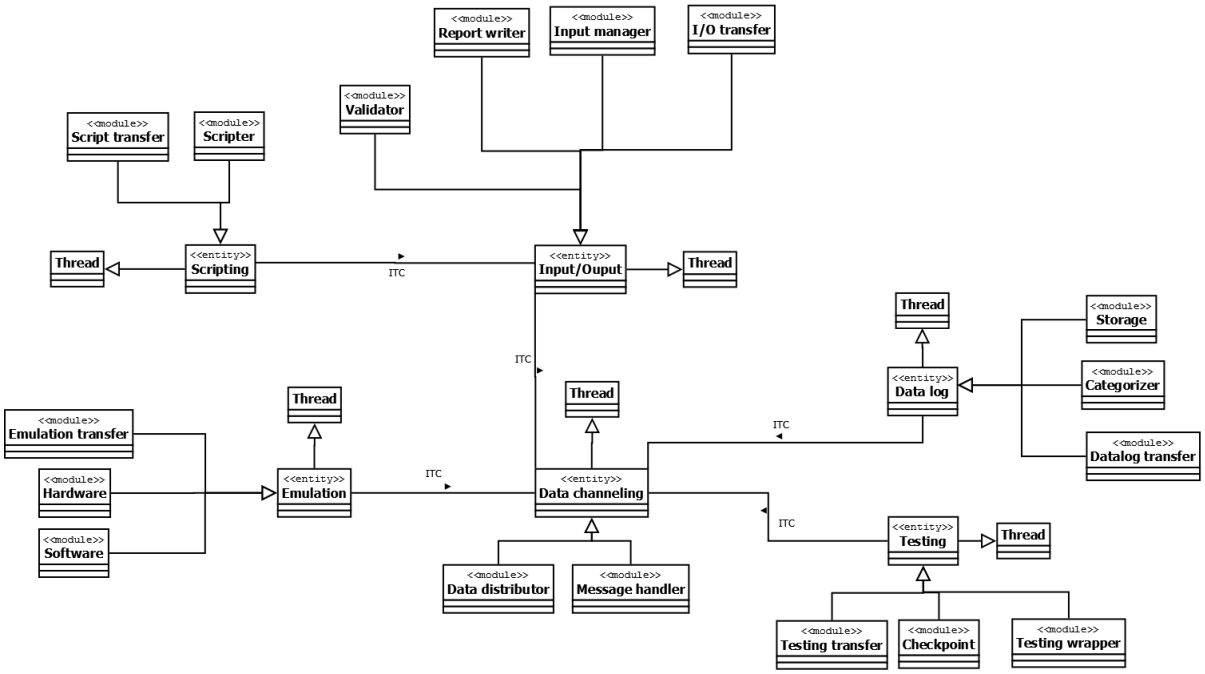
The division on the thread are based on the division already made in the conceptual and the module view. Components and modules that will work closely together are grouped on the same thread. There has been no new modules added, thus all modules in the execution view can be mapped directly to the same modules in the module view.

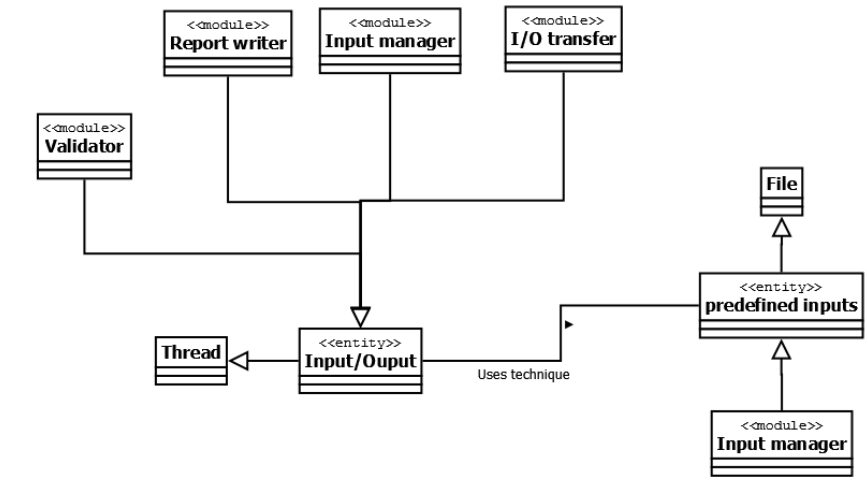
We also have two separate entities for specific requirements on the MIB. One file-entity for the input manager since this module will require file handling. As well as one queue for the "*Data distributor*" so that it can handle the vast amounts of data that will be sent simultaneously over the MIB, which comes from the data broker strategy.

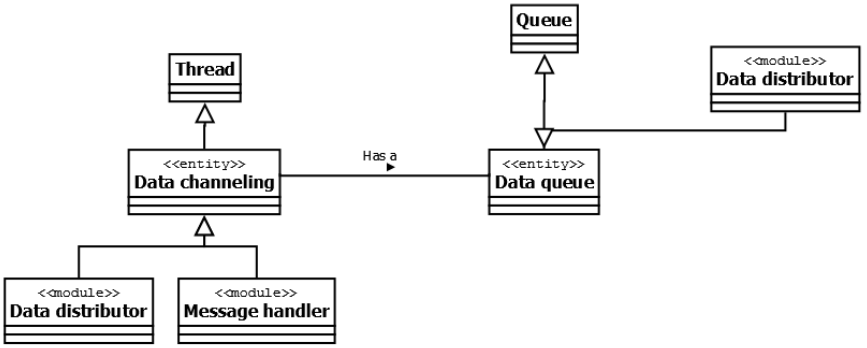
Execution view



Execution view entities




<<module>>
Report writer
<<module>>
Input manager
<<module>>
I/O transfer
<<module>>
Validator
File
<<entity>>
predefined inputs
Thread
<<entity>>
Input/Ouput
Uses technique
<<module>>
Input manager


Queue
Thread
<<module>>
Data distributor
<<entity>>
Data channeling
Has a
<<entity>>
Data queue
<<module>>
Data distributor
<<module>>
Message handler

Evaluation

Method of Evaluation

What method did we choose?

We chose a variant of the BTH 4-hour Architecture Evaluation method, BTH method.

Why did why choose this method?

We chose the BTH method because it was the only evaluation method we were familiar with and it was the only one that we found that did not require weeks of evaluation or huge teams of evaluators.

What is the BTH 4-hour Architecture method?

The "*BTH 4-hour Architecture method*" is a method developed at Blekinge Institute of Technology. The BTH method is very much like the "*Architecture Tradeoff Architecture Method*" and the "*Software Architecture Analysis Method*" with the main difference that the BTH method is much shorter. It revolves around creating different scenarios that the architecture should be able to handle, or explained why it does not in fact need to handle the certain scenario. The main purpose of the scenarios is to define the limits and requirements of the evaluated system and in some cases to develop strategies to counter eventual problems or drawbacks in the system.

When using the BTH method one should have an exterior part to ask the questions and help formulate scenarios. However since we were unable to procure another group for our evaluation we had to improvise and evaluate ourselves. Unfortunately since one is sometimes blind to one's own work we risked missing several possible scenarios that would have aided us greatly, and we did not gain the insights one may receive when collaborating with another group.

Other methods we considered

We considered the both "*Architecture Tradeoff Analysis Method*", ATAM, and "*Cost Benefit Analysis Method*", CBAM, methods. And the reason as to why we considered these alternatives, and only these, was because they were the only other evaluation methods described in the main course literature. There were other methods described during the lectures but we felt we did not know enough about them to take them into consideration.

Reasons as to why we did not choose the other methods

The reason why we did not choose the ATAM evaluation method was because it required resources we did not have, namely time and a larger team. With ATAM you practically start with a hiatus of two to three weeks after a day of evaluation, and after the hiatus it requires up to a week of continued work, and this was, and is, time that we do not have. And this alone made ATAM impossible for us to use, but beyond the time requirements ATAM also requires a large evaluation team and preferably stakeholders, persons we do not have available for obvious reasons.

The main reason as to why we did not choose to use CBAM was because it mainly concerns itself with budget and profits, the economical aspect of the whole project. And since we have been told not to concern ourselves with the organizational aspects of the project, and the fact that we do not have a budget nor a customer we felt that this method was highly irrelevant.

Quality Attributes Tested GA02

The three qualities attributes that we found most important to evaluate were *System dependability*, *Input types* and *Maintainability*.

We felt that the system dependability was the most important attribute to be tested at this point, probably will continue to be, as our system's purpose is to ensure the integrity and stability of other systems.

We chose evaluate input types since it is one of the major issues in the system description:

> "*The system shall be able to simulate various inputs, such as, but not limited to, file input, command line input, socket communication, and graphical input. It must be easy to incorporate new input mechanisms in the system.*"

With this in the description we felt that it was well worth more of our attention.

The maintainability attribute was chosen because we envision that the MIB will be used by their companies for many years to come and thus it is important to consider the maintainability aspects of our system.

Scenarios GA02

System dependability

The tested system uses as much as 10 gigabyte if primary memory

According to our research this is not a feasible quantity of required primary memory for any program running on a standard computer both presently and in the perceivable future. This scenario did however make us add an assumption about the memory requirements of the tested system.

The tested system crashes during testing

The testing wrapper will acquire the last saved state of the tested system from the checkpoint module and then restore the tested system from there and continue the testing. While this is happening the testing will be halted by the data broker until it receives the all clear from the testing wrapper.

The tested system crashes during testing and keeps on crashing at the same moment every time

As we discussed this scenario it became clear that we needed a new issue card, that we named "Multiple Crash Issue". The wrapper around the tested system will count the amount of crashes and map them to the checkpoints. If the tested system crashes a predetermined amount of times between two checkpoints the MIB will terminate the current test and note this in the test log.

Input types

Input stream from a blue-ray disc

Our input system is able to handle the streaming from a blue-ray disc as the maximum transfer rate of a blue-ray disc, 54 MB/s, is well below the capacity of any standard computer. We chose this scenario to get some idea of how high transfer rate a common storage device with high performance could have and how our system could handle this type of throughput.

Maintainability

The customer returns after buying the system and requests two new testing techniques

By maintaining good semantic coherence throughout the wrapping module as well as anticipating any expected changes in testing techniques. Thus changes will be kept to a minimum and localized to the testing wrapper.

The costumer returns after buying the system requests two new input types

By generalizing the input modules we make it simple to implement new input types and changes the impact on the rest on the system will be minimal. The goal is that implementing a new input type will only mean adding new code, not changing old one.

The test is finished and the output needs to be validated

When the testing is completed the data from the test will be transferred from the data log to the output validation system. Within the output validation system the output will be controlled and validated against predefined output in the validation module. The report module will then compile all the data together with the validated output into a final readable report.

Evaluation Conclusion GA02

We have found that, with our current knowledge of architecture structures and with the results of our self evaluation, our current structure upholds the quality requirements as set by the system description. As such we have found no doing an architecture transformation.

The new issue-card "Multiple crash issue" had already been considered in the first assignment but not properly documented, as such the views have not required any polishing because of this “new” issue card.

Choice of Method and Quality Attributes for GA03

We have chosen to adapt the same evaluation method as before to the design. It has worked well and we have yet to find another testing method that has an overwhelming amount pros over the cons of working with a method we are not familiar with.

The evaluation will focus on the new changes made to the MIB, not the old requirements already established at the beginning of the course. the new requirements from the changes are divided into the functionality that they describe. The reason for focusing on the new requirements is that the old ones have already been evaluated above and we do not feel a need for further analysis of these, instead we focus on the new ones that have not had a formal evaluation as of this time.

Scenarios GA03

Scripting

Customer asks us to implement a specific script-language for the input-scripting.

We had already foreseen this and thus chose to make our scripting into a whole layer in the module view so that we could easily extend it with functionality to support our own scripting languages. The changes from an existing language and our own should therefore be very local. The rest is all a matter of budget and time.

The customer wants 10 different scripting languages implemented.

This is not a realistic but doable. At an architectural level the scripting component should be able to hold a potentially indefinite number of languages, though we might have to point out the lack of gain for having so many instead of a selected few.

How deep will the looped scripting go? Will you be able to script just for input and output validation or will you be able to implement new rules for the other components as well?

It is not a requirement from the customer to be able to script anything besides the input and output-validation; as such it is nothing we will implement into our MIB since we do not work for free. However with the high changeability in our MIB it should be possible to some extent to implement such scripting in other components as well.

Parallel testing

Output validator receives output to be validated from two parallel tests on the same tested system, how does it tell them apart?

Each testing technique will have a unique identifier when it arrives. Thus the output validator can match this identifier with the corresponding identifier that has been set at the start of the test or during scripting. It can then log the different outputs using these identifiers as well.

We want to run 100 different test techniques in parallel.

The main concern with this scenario is the strain it would put upon the data broker for sending and receiving output/input to the different tests. We would probably have to limit the amount of parallel testing to a much smaller number so the system does not break. The number of techniques will also be dependent of the types of techniques to be run and how large their throughputs are. As such it is a difficult question to answer since different customers may want different testing techniques. the maximum number can probably not be defined until a prototype can be built.

While the MIB is running parallel testing techniques one of them crashes the tested system and the system is restarted from a checkpoint. What happens to the testing technique that did not crash the system?

All testing techniques will have to redo their testing from the last checkpoint. This will give some testing overhead but the alternative is a test that is less deterministic unless we can assume that the different techniques have no sustained impact on the tested system. The reason for this is that if a testing technique can change system variables, then the reason for the crash may not lie with the technique that triggered the crash, but instead with a previous technique that changed the system.

Monitoring the test

A tester starts the MIB in automatic mode and later wants hands on approach, thus wanting to switch into a monitoring mode.

There are two different modes for the MIB to switch between, automatic and monitored mode. The tasks given to the data broker will vary slightly depending on which mode is being run but there are no other changes to the rest of the system at this level.

We are testing a system with an extreme throughput of data and the tester wants to run in monitored mode.

This will only change the amount of data being transferred in real-time since the testing data can no longer be stored in the data log until the test is complete. Instead that data will have to be transferred to the output validator constantly so that a report can be made for the tester monitoring the system. Without hardware specifications and a working prototype it is hard to evaluate the effect of this increase in data.

A tester want to only script without looking at the test results in real-time

The monitored mode will have to be able to toggle different functionalities on and off to make it possible to script without getting the testing result. A separate win for this is that it takes away the performance issues that could arise from monitoring the test results from scripting. We will add a new strategy to adapt this decision.

Evaluation Conclusion GA03

From the scenarios we have found one new strategy that needs to be implemented. We cannot see this will have any effect on the different views and as such will only be added to the list of issue-cards we have. It could affect later stages but as of now the architectural structure remains the same. Here follows a short summary:

- It will be possible to implement our own scripting languages.
- Several different scripting languages can be implemented.
- The customer does not require us to script anything beside input and output-validation.
- Each testing technique will have a unique identifier.
- The performance issues with multiple testing techniques are hard to map without making a working prototype.
- If parallel testing is being done, all tests will have to redo their work from the last checkpoint if the tested system crashes.
- The MIB will have two different modes, one automatic and one monitored.
- In monitored mode it will have to be possible to toggle certain functionalities on and off depending on what the tester wants to do, the following strategy was created for solving this: "Toggle in monitored mode", it will be marked with yellow.