GA01 Deliverable

M.U.P

massively underdeveloped project

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

## Updated A01

## 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 avarage software system will use 3-5GB of RAM.

## 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 |
| **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 |
| **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 transfering |
| **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 hardware emulation 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:** |

|  |
| --- |
| **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:** |

|  |
| --- |
| **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 a 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 then a predefined amount this will signal that the testing needs to be terminated. The wrapper then send a message to stop the testing up to the message handler in the data broker. |
| **Strategies/Tactics:** |

## Conceptual view description

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

### User Interface

The user interface that the tester will use to setup the test bench before the testing can begin. It will also be responsible for showing the final report of once the test is done.

### 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 know have knowledge of eachother 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 mention 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

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

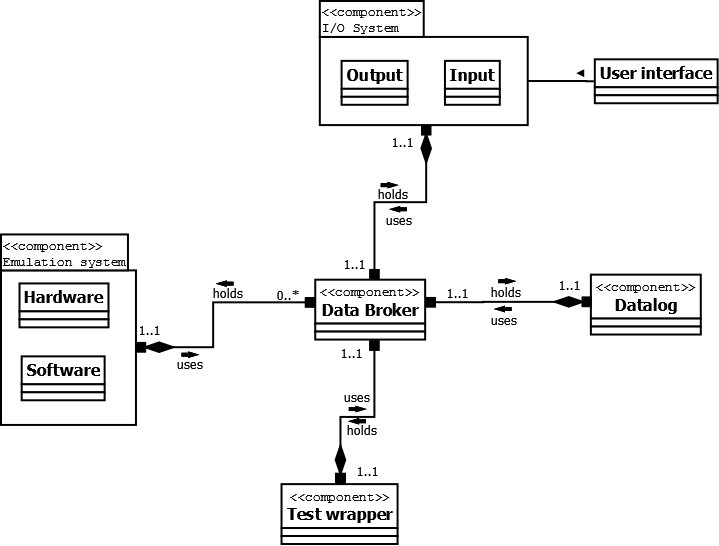
This component functions as a wrapper for the tested system as well as storing a checkpoint with the status of the tested system. With this the tested system can be restarted with minimal latency if it crashes. It will also send a message to the data broker to halt the testing until the tested system 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

This layer gathers the modules that emulate hardware and channel the data from software. We took the descision 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:

* *“Multiple software emulations issue”*
* *“Multiple hardware emulations issue”*

### Testing

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:

* *“Keeping the system running through a test crash”*
* *“Not crashing with tested system issue”*
* *“Multiple testing techniques issue”*

### Data Log

Here all the data from the testing is categorized and stored into a log-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:

* *“Creating report issue”*

### Data Channeling

Holds the data broker subsystem that handles transfering data between different parts of the system that do not have knowledge of eachother 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:

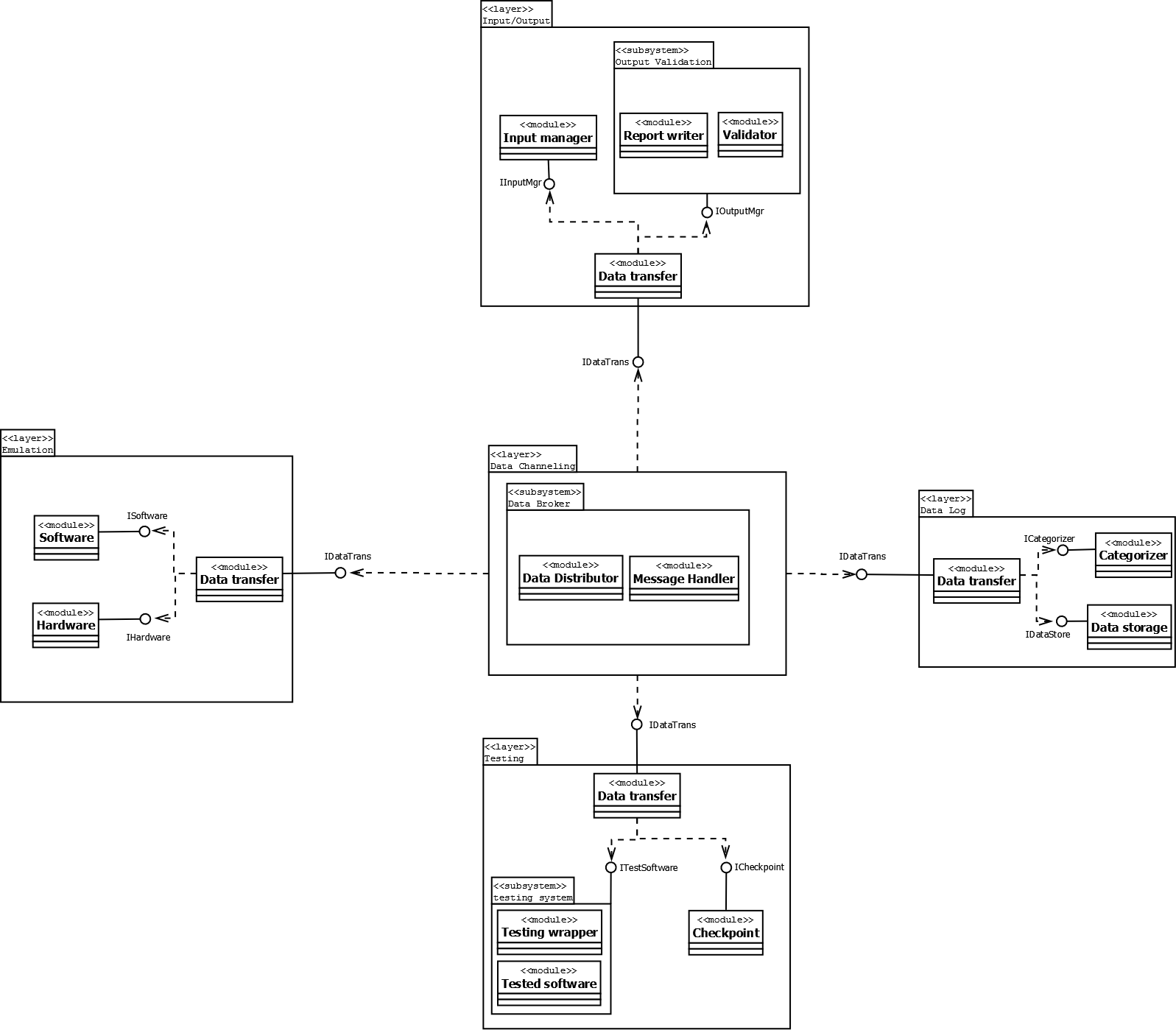
* *“Data transfer”*
* *“Not crashing with tested system issue”*
* *“Creating report issue”*

### Input/Output

This layer hold 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 wont 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:

* *“Multiple input issue”*
* *“Creating report issue”*

## Module view



## Execution view description

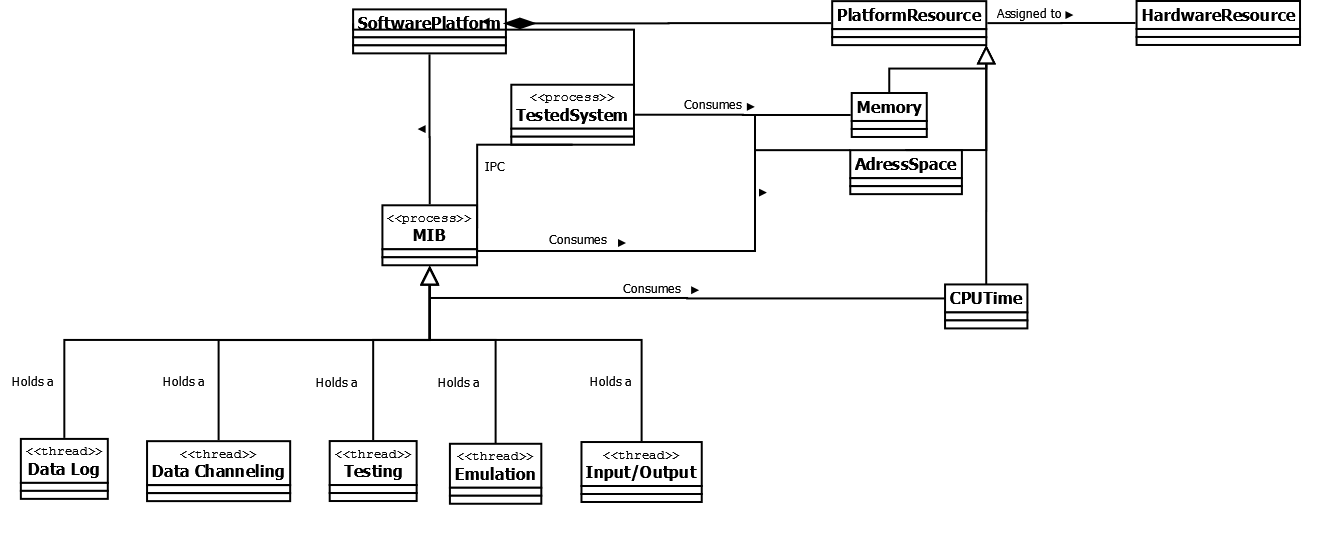
The software platform will house two processes. One for the MIB itself and one for the tested system. These will communicate via IPC(intraprocess communication). The MIB will have one thread for each major component in the system so that they can work independently. We also mapped which resources each 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 devided each thread-entity into single diagrams containing those modules that are coupled to that entity thus giving a better view that is not to 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.

We also have two separate entities for specific requirments on the MIB. One file-entity for the input manager since this module will require file handling. One Queue for the Data distrubutor so that it can handle the vast amounts of data that will be sent simultaneously over the MIB(From the data broker strategy).

## Execution view



#### Execution view entities

