 THE RESEARCH LEADING TO THESE RESULTS HAS RECEIVED FUNDING FROM THE EUROPEAN UNION'S SEVENTH FRAMEWORK PROGRAM (FP7/2007-2013) FOR CRYSTAL – CRITICAL SYSTEM ENGINEERING ACCELERATION JOINT UNDERTAKING UNDER GRANT AGREEMENT N° 332830 AND FROM SPECIFIC NATIONAL PROGRAMS AND / OR FUNDING AUTHORITIES.

CRitical S**YST**em Engineering **A**cce**L**eration

Motion control of patient table and X-ray beam positioning
Use – Case Definition
D403.010

DOCUMENT INFORMATION

Project	CRYSTAL
Grant Agreement No.	ARTEMIS-2012-1-332830
Deliverable Title	Use – Case Definition
Deliverable No.	D403.010
Dissemination Level	CO
Confidentiality	R
Document Version	V1.0
Date	2013-10-31
Contact	Rogier Vermeulen
Organization	Philips Healthcare
Phone	+31402769495
E-Mail	rogier.vermeulen@philips.com

AUTHORS TABLE

Name	Company	E-Mail
Rogier Vermeulen	Philips Healthcare	rogier.vermeulen@philips.com

CHANGE HISTORY

Version	Date	Reason for Change	Pages Affected
1.0	2013-10-31	Initial version.	-

CONTENT

D403.010 I
1 INTRODUCTION 6
1.1 ROLE OF DELIVERABLE 6
1.2 RELATIONSHIP TO OTHER CRYSTAL DOCUMENTS 6
1.3 STRUCTURE OF THIS DOCUMENT 6
2 USE CASE CONTEXT 7
2.1 RATIONALES 7
2.2 THE GOAL OF USE CASE 4.3 8
2.3 *IN-THE-LOOP* SIMULATION 8
2.3.1 Model in-the-Loop simulation 9
2.3.2 Software in-the-Loop simulation 9
2.3.3 Processor in-the-Loop simulation 9
2.3.4 Hardware in-the-Loop simulation 9
2.4 CONTINUOUS INTEGRATION 10
3 USE CASE PROCESS DESCRIPTION 11
3.1 CURRENT DEVELOPMENT PROCESS 11
3.1.1 The "Implement and Test" cycle 12
3.1.2 System Verification 16
3.2 PROPOSED DEVELOPMENT PROCESS 17
3.2.1 The "Implement and test" cycle 17
3.2.2 System Verification 19
4 IDENTIFICATION OF ENGINEERING METHODS 20
5 TECHNICAL CASE STUDY: TESTING THE TABLE FORCE SENSOR 22
5.1 SPECIFICATION 22
6 TERMS, ABBREVIATIONS AND DEFINITIONS 25
7 REFERENCES 26
8 ANNEX I: DETAILED DESCRIPTIONS OF THE ENGINEERING METHODS 27
8.1 TEST WITH IN-THE-LOOP-SIMULATION 27
8.2 REPORT VERIFICATION RESULTS 28
9 ANNEX II: TECHNOLOGY BASE LINE & PROGRESS BEYOND 29

Content of Tables

Figure 2-1: *the V-model showing the process (left) and the documentation (right). Pictures are borrowed from internet sources and Mouz et. al. (1996,2000)* 7
Figure 2-2: In-the-loop simulation definitions....9
Figure 3-1: System Verification process in more detail. 16
Figure 3-2: Implement and Test cycle (proposed).....18
Figure 3-3: System Verification process (proposed). 19
Figure 4-1: identification of engineering methods (system verification). 21
Figure 5-1: Monitor Ceiling Suspension. 22

Content of Figures

Table 6-1: Terms, Abbreviations and Definitions 25

Content of Appendix

No table of contents entries found.

1 Introduction

1.1 Role of deliverable

This document has the following major purposes:

- Define of the overall use case, including a detailed description of the underlying development processes and the set of involved process activities and engineering methods
- Provide input to WP601 (IOS Development) required to derive specific IOS-related requirements
- Provide input to WP602 (Platform Builder) required to derive adequate meta models
- Establish the technology baseline with respect to the use-case, and the expected progress beyond (existing functionalities vs. functionalities that are expected to be developed in CRYSTAL)

1.2 Relationship to other CRYSTAL Documents

1.3 Structure of this document

2 Use case context

2.1 Rationales

Healthcare systems are subject to strict regulations from ISO, IEC and FDA regarding safety of operators and patients [Ref ISO/IEC/FDA norms]. A well-defined development process needs to be defined including harm and hazard analysis, risk management and extensive documentation for that purpose. The development process is typically following the 'traditional' V-model; Figure 1 (left) outlines this V-model while Figure 1(right) maps this onto the documentation.

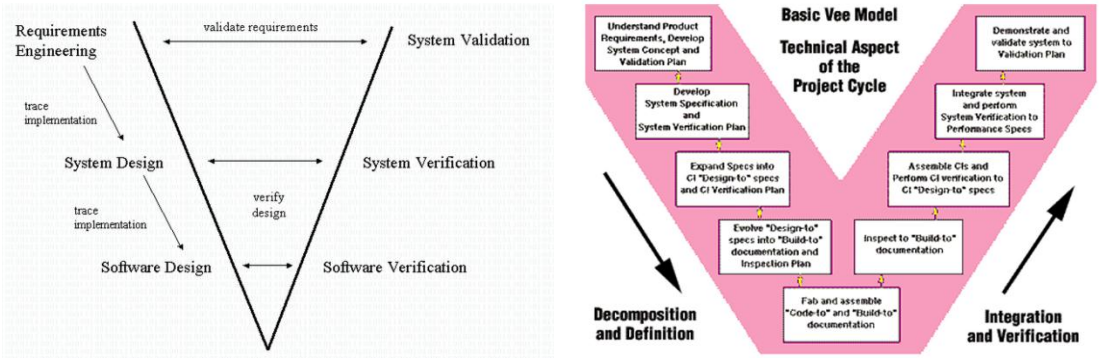


Figure 2-1: *the V-model showing the process (left) and the documentation (right). Pictures are borrowed from internet sources and Mouz et. al. (1996,2000)*

V-Model: Advantages of linearly following the V-model, in particular for safety, include the well-documented record and audit-trail of process and products, and the 'push-forward' nature of obtaining the final product, which fits engineers quite well. Among the downsides are a lack of incremental approaches, the late system integration and the extensive documentation (which must be updated upon every change and for every different member of a product family). A particular consequence of the late integration is that negative effects of safety measures on usability are observed only in a very late stage, or even only in the field. In practice this leads to much manual effort in producing documentation and defining tests.

New challenges: Safety-critical systems engineering faces also new challenges. The complexity of systems is ever increasing due to higher customer demands, more advanced functionality and integration with other medical equipment. System components, in particular, software components become COTS rather than proprietary and, since many safety aspects are software defined, new methods are needed for guaranteeing safety for component-based systems. In addition, systems have to be compliant with updated and new regulatory norms. Because of this, and because of error corrections and changing requirements, updates in the field have to be performed. Finally, in order to maintain a competitive edge, time-to-market must be kept as small as possible or at least predictable.

Improvements: Although current systems do satisfy the safety requirements, there is a need to improve on the following aspects:

1. The call-rate due to a mismatch between user needs and final implementation.
2. The development effort and lack of early feedback on extra-functional requirements.
3. High release effort due to late integration and manual testing.
4. Effort to show complete requirements traceability for regulatory affairs audits.

The goal of the CRYSTAL project is to improve these four metrics through a change in the engineering process but more importantly, in the tool support. At the same time these four are the respective drivers of the three use cases of Philips in the healthcare domain in CRYSTAL.

2.2 The goal of Use Case 4.3

Use Case 4.3 will target improvement items 2 and 3 and will focus on the part of the V-model as indicated in Figure 2-1. It's aim is to reduce development and test effort through the use of In-the-loop simulation and applying a Continuous Integration strategy. In the remainder of this chapter these techniques will be explained.

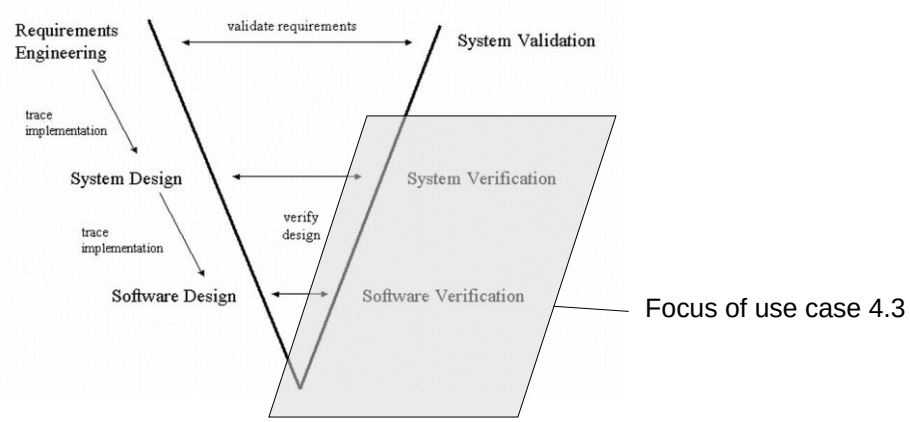


Figure 2-1: Development process scope of UC4.3.

2.3 *In-the-loop* simulation

Hardware-in-the-Loop simulation definition [Wikipedia]:

Hardware-in-the-loop (HIL) simulation is a technique that is used in the development and test of complex real-time embedded systems. HIL simulation provides an effective platform by adding the complexity of the plant under control to the test platform. The complexity of the plant under control is included in test and development by adding a mathematical representation of all related dynamic systems. These mathematical representations are referred to as the "plant simulation". The embedded system to be tested interacts with this plant simulation.

Next to HiL simulation a number of other simulation definitions exist (see figure below):

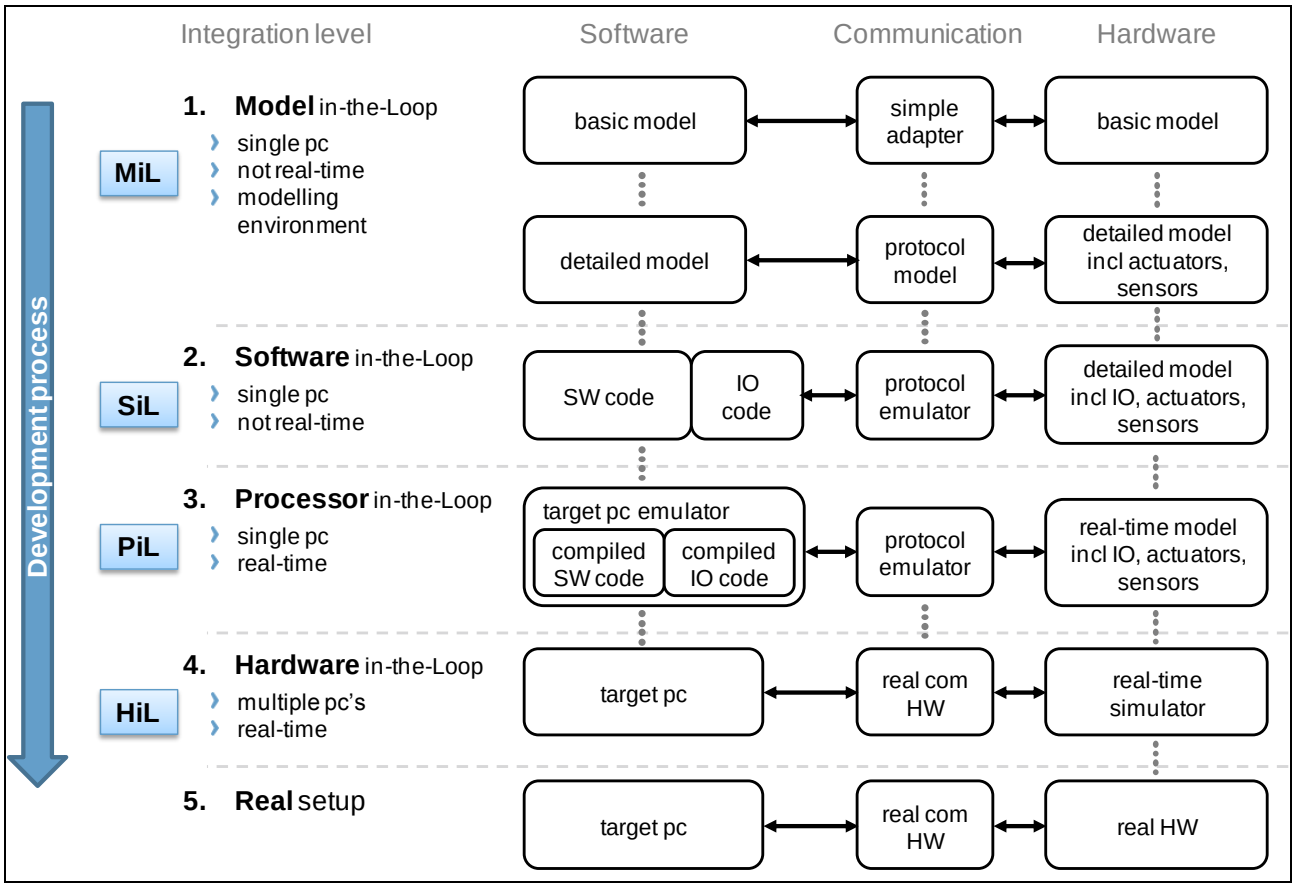


Figure 2-2: In-the-loop simulation definitions.

2.3.1 Model in-the-Loop simulation

Here models of software and or hardware can be simulated and give the designer feedback on the dynamical behavior of his design/architecture. A software model in combination with the actual hardware can be used for rapid prototyping (note that the model will typically not be real-time, but this need not be a problem when a part of the software is simulated which is non real-time).

2.3.2 Software in-the-Loop simulation

Here SW code is not run on the target hardware, but on a PC (non real-time) and executed together with a model of the hardware. This is very useful for implementation and debugging, but SW performance testing is not possible. And there are always problems that occur on the target hardware/OS and not in a PC simulation (and vice versa).

2.3.3 Processor in-the-Loop simulation

Here SW code is run on an emulation of the target machine (e.g. a VMWare session of the target OS), together with a model of the hardware. This is more representative than SiL; the code could now for instance be subject to real-time scheduling. Performance testing may still be a problem because of the emulation.

2.3.4 Hardware in-the-Loop simulation

Here SW code is run on the target PC/OS together with a real-time simulation of external hardware. The detail of the hardware model determines how much software testing can be done without using the actual hardware.

2.4 Continuous Integration

Continuous Integration definition [Wikipedia]:

Continuous integration (**CI**) is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day. It was first named and proposed as part of extreme programming (XP). Its main aim is to prevent integration problems, referred to as "integration hell" in early descriptions of XP. CI can be seen as an intensification of practices of periodic integration advocated by earlier published methods of incremental and iterative software development, such as the Booch method. CI isn't universally accepted as an improvement over frequent integration, so it is important to distinguish between the two as there is disagreement about the virtues of each.

CI was originally intended to be used in combination with automated unit tests written through the practices of test-driven development. Initially this was conceived of as running all unit tests and verifying they all passed before committing to the mainline. This helps avoid one developer's work in progress breaking another developer's copy. If necessary, partially complete features can be disabled before committing using feature toggles.

Later elaborations of the concept introduced build servers, which automatically run the unit tests periodically or even after every commit and report the results to the developers. The use of build servers (not necessarily running unit tests) had already been practised by some teams outside the XP community. Nowadays, many organisations have adopted CI without adopting all of XP.

In addition to automated unit tests, organisations using CI typically use a build server to implement *continuous* processes of applying quality control in general — small pieces of effort, applied frequently. In addition to running the unit and integration tests, such processes run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code and facilitate manual QA processes. This continuous application of quality control aims to improve the quality of software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control *after* completing all development. This is very similar to the original idea of integrating more frequently to make integration easier, only applied to QA processes.

In the same vein the practice of continuous delivery further extends CI by making sure the software checked in on the mainline is always in a state that can be deployed to users and makes the actual deployment process very rapid.

3 Use Case Process Description

In this chapter we describe the current development process and from an analysis of the bottlenecks in this process we derive a new development process.

3.1 Current development process

In this section the current development process for the lower right half of the V-model (Figure 2-1) is described in more detail and an analysis is done of the problems regarding development and test effort encountered there.

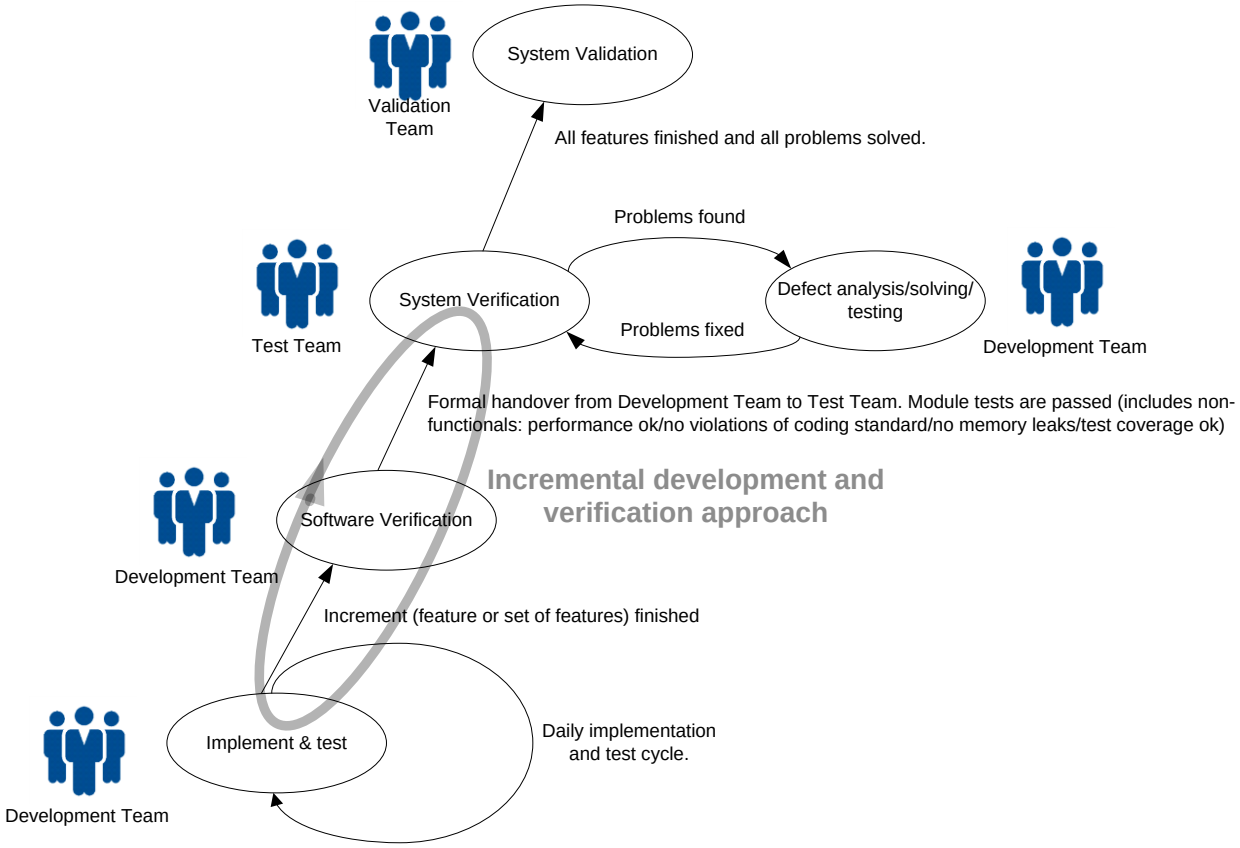


Figure 3-1: Right side of the V-model in more detail.

As can be seen in Figure an incremental way of working is employed. During each increment one or more features are implemented and tested. Each increment passes through the following stages:

- Implementation & test: This is in itself an iterative process where the development team implements the software in daily implementation and test cycles. The aim is to always have a working integrated subsystem (this is part of the Continuous Integration philosophy).
- Software verification: the development team provides evidence for the quality of the delivered software product. Module level verification reports have to be produced showing that all tests are passed.
- (Sub)System verification: the test team verifies that the subsystem requirements are met. Here requirements for the integrated (sub)system (mechanics/electronics/software) are tested. The test team executes the test cases developed for the new features and regression test cases. The testing done on this level is almost entirely manual. Problems found are fixed by the Development Team.

The verification of safety requirements is carried out by the Test Team and the Development Team together. The regression testing strategy for Safety Requirements is risk based, but usually a lot is retested.

For the process sketched above it holds that the later a problem is found the more costly is to fix it. When a developer finds a bug testing his SW update on his PC it may take 1 manhour to fix it. When a problem is found during System Verification a PR (Problem Report) must be made (by the tester), a developer has to do an invest (and document it in the PR), implement the solution (and document what he has done in the PR) and the tester verifies that the problem is fixed (and documents this in the PR). All these activities are coordinated by a CCB (Change Control Board). This process will atleast cost 8 to 16 manhours. When a serious problem is found in the field several man-months may be required to fix it an deploy the solution to the field. Initial quality is of the utmost importance given the increasing cost of finding and solving a defect later on in the V-model.

An important driver for reducing development and test effort is therefore improving initial quality.

3.1.1 The “Implement and Test” cycle

Initial quality is the responsibility of the Development Team and should be covered to a large extent in the “Implement and Test” cycle. In the figure below this cycle is shown in more detail (including the tooling used):

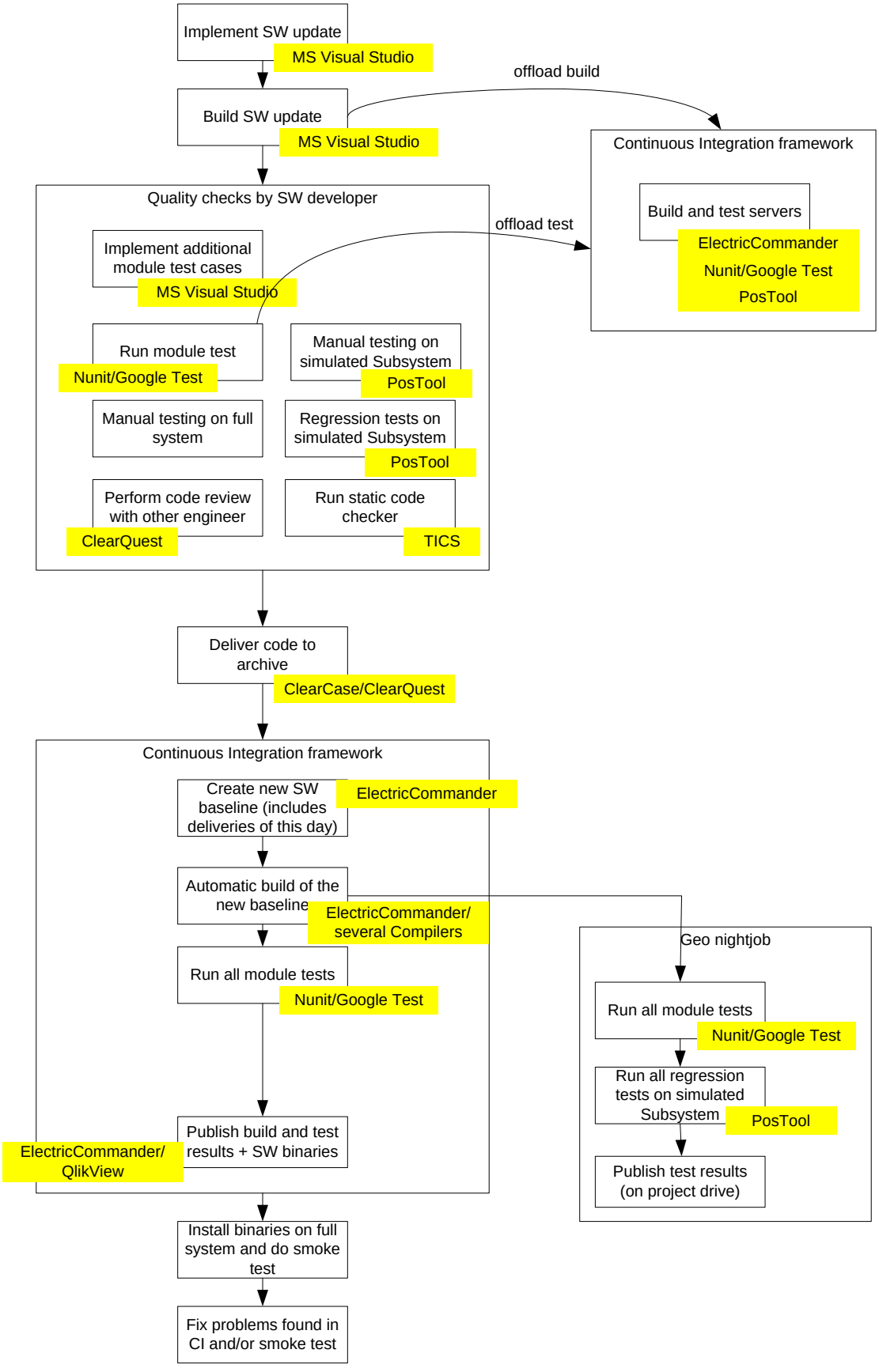


Figure 3-2: The daily implementation and test cycle in more detail.

3.1.1.1 Identification of bottlenecks and improvements

The table below lists the steps from the “Implement and test”-cycle, the known problems regarding effort and providing initial quality, recent improvements and possible future improvements. The improvements marked yellow (hardware-in-the-loop simulation) and green (continuous integration) are in the scope of Crystal UC4.3.

nr	Development step	problems	Recent improvements	Possible future improvements
1	Implement SW update	-	-	-
2	Build SW update	4 hours for full build	0,5 hr for full build (snapshot views, SSD disks, offloading to fast machine, making better use of multiple cores).	Incremental building. Further parallellization (Incredibuild tooling).
3	Implement additional test cases	High effort	-	Use coverage tooling to optimize testing.
4	Run module tests	Some tests take long (> 1 hr).	-	Use coverage tooling to optimize testing.
5	Manual testing on full system	Test systems are scarce (shared between projects).	-	Better simulation may reduce the need for test systems.
6	Manual testing on simulated system	Windows simulation not representative enough for the actual VxWorks target.	-	Provide simulation on (or of) VxWorks.
7		Simulation of hardware not representative enough.	Simulation of bodyguard sensor via 3D model information.	Further improvements of the quality of simulation (HiL): - model more sensors - model motor behavior - model electrical circuits and their failure modes - etc.
8	Regression tests on simulated subsystem	Windows simulation not representative enough for the actual VxWorks target.	-	See 6
9		Coverage is low (simulation of hardware not representative enough).		See 7.
10	Perform code review	Not always done.	-	Provide feedback to developer on deliveries not

				reviewed.
11	Static code checking	Slow.		Speed improvements. Less checking. Different tool.
12		Not always done.		Provide feedback to developer on violations in deliveries.
13	Deliver code to archive	-	-	-
14	Create new baseline	-	-	-
15	Automatic build of new baseline.	4 hours for full build	See above.	See 2.
16	Run all module tests.	6 hours for all tests	-	Use coverage tooling to optimize testing. Split up tests to run on different machines. Provide a fast partial test run and an extended full test run.
17		Tests are run in CI environment and in the Geo Nightjob environment (predecessor of CI)	-	Move also simulated subsystem tests to CI and get rid of old environment.
18	Run all regression tests on simulated subsystem	6-8 hours for all tests	-	Use coverage tooling to optimize testing. Split up tests to run on different machines.
19		Test are not executed from CI environment.		See 17.
20		Coverage is low (simulation of hardware not representative enough).	-	See 7.
21		Performance not regression tested (manual testing during SW Verification phase, about 1-2 man-weeks)	-	Provide simulation on (or of) VxWorks to automatically test performance. Deploy SW on target HW + VxWorks from the CI environment. Show performance trend via QlikView.
22	Publish build and test results + SW binaries	-	-	-
23	Publish test results (on project drive)	Duplication of 22.	-	See 17.
24	Install binaries on full test system and do smoke test	High effort (5 hrs/wk).	-	Do automatic deployment (+test) on test systems from CI environment.
25	Fix problems found in CI a/or	-	-	-

	smoke test			

Tabel 3-1 : bottlenecks and improvements.

3.1.2 System Verification

Below the current System Verification process is depicted.

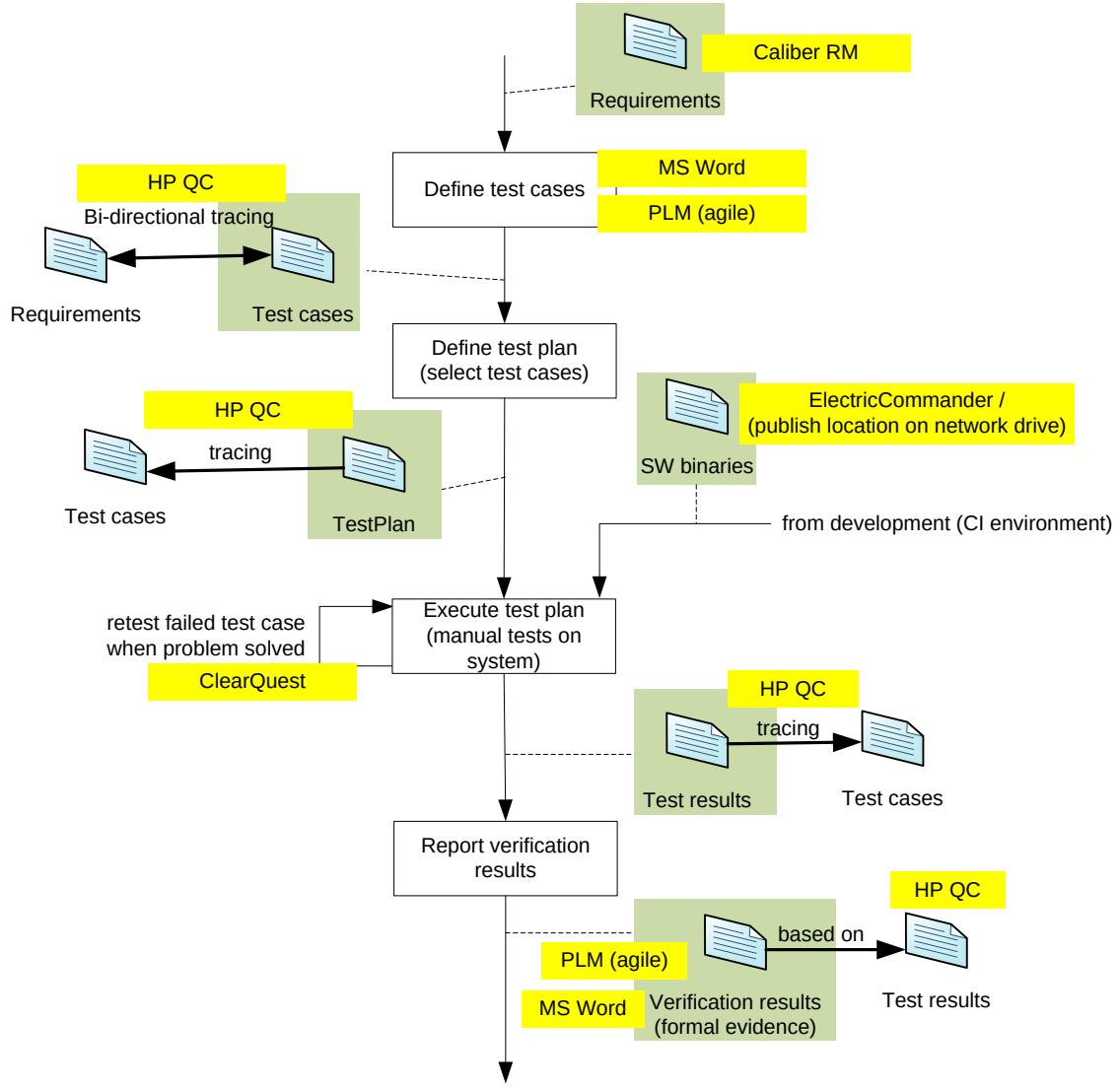


Figure 3-1: System Verification process in more detail.

1. *Define test cases*: Test cases are documented in HP QC. The test cases are traced to requirements, which are first imported from Caliber RM. From HP QC a word document is generated and stored in the documentation archive (PLM). It is the word document which is reviewed and which counts as evidence for the FDA.
2. *Define test plan*: For a project a selection of test cases to execute is made (risk based).
3. *Execute test plan*: Test cases are executed. These are manual test cases. Individual verification steps can be set to PASSED or FAILED in HP QC by the tester. In case of failure a PR (problem report) will be written in ClearQuest. When the defect handling process is completed and the problem is solved the tester will re-execute the test case.

4. *Report verification results*: Word documents containing the test results are created from the information in HP QC and stored in PLM as official evidence.

3.1.2.1 Identification of bottlenecks and improvements

nr	Development step	problems	Recent improvements	Possible future improvements
1	Import requirements	Proprietary import mechanism (possible maintenance trap).	-	Use OSLC link between CaliberRM and HP QC. This is part of another Crystal Use Case (4.1).
2	Define test cases	Requirement specification is complex. Much effort required to define test cases.	Simplification and unification of Bodyguard behaviour (not yet implemented).	Continue simplifying the requirement specification.
3	Define test plan	-	-	-
4	Execute test plan	Test systems are scarce (shared between projects). A lot of different HW configurations need to be tested.	-	Better simulation may reduce the need for test systems.
5		Lot of effort involved in manual testing.	-	Use automatic testing where possible. When combined with simulation, results from the CI nightjob can be used as test evidence.
6	Export test results	-	-	-

3.2 Proposed Development Process

3.2.1 The “Implement and test” cycle

Below the proposed “Implement and test” cycle is depicted, showing the improvements (indicated in blue) identified in section 3.1.1.1.

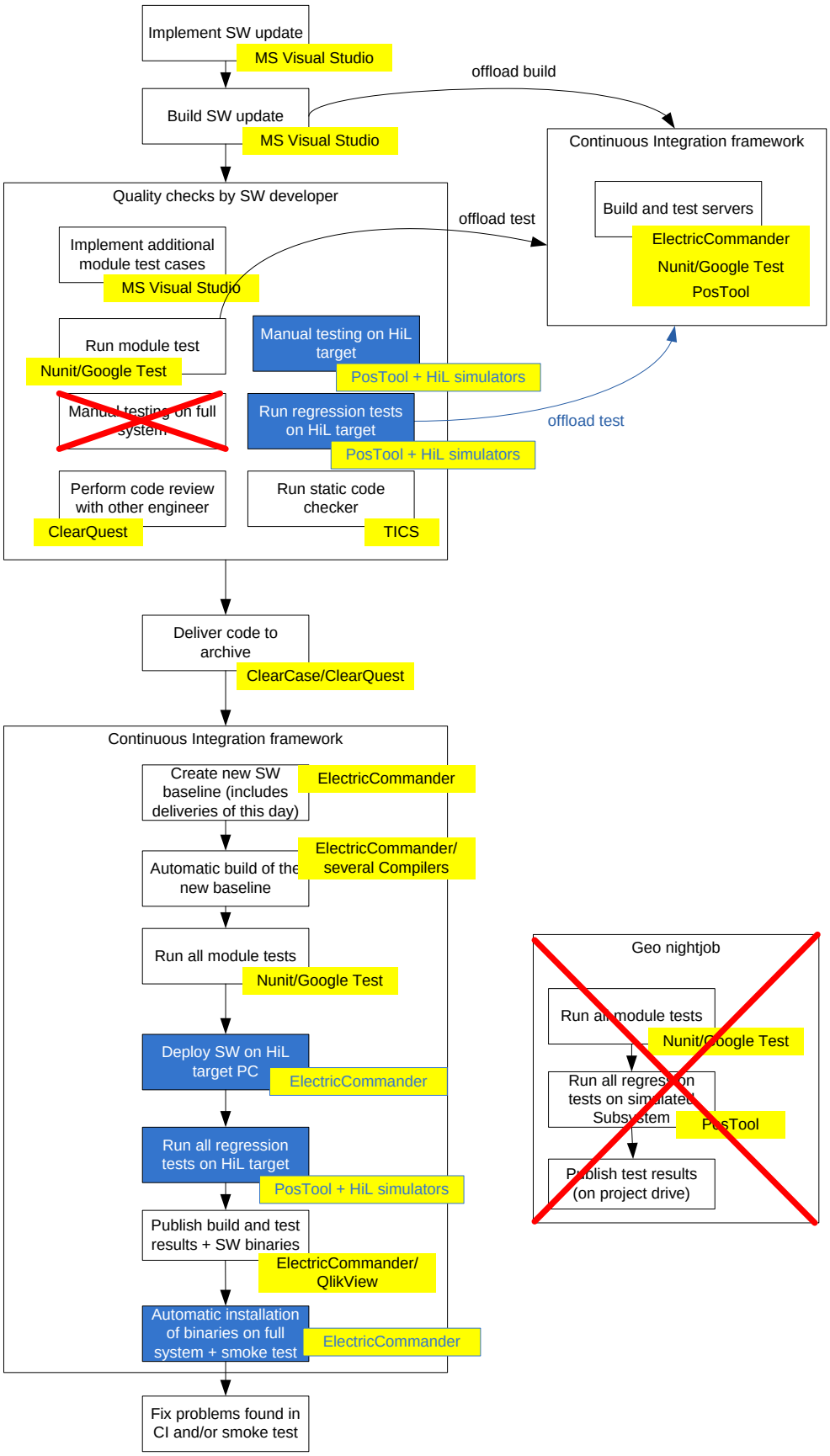


Figure 3-2: Implement and Test cycle (proposed).

3.2.2 System Verification

Below the proposed System Verification process is depicted, showing the improvements (indicated in blue) identified in section 3.1.2.1. The picture includes the currently used tooling and the proposed new tooling.

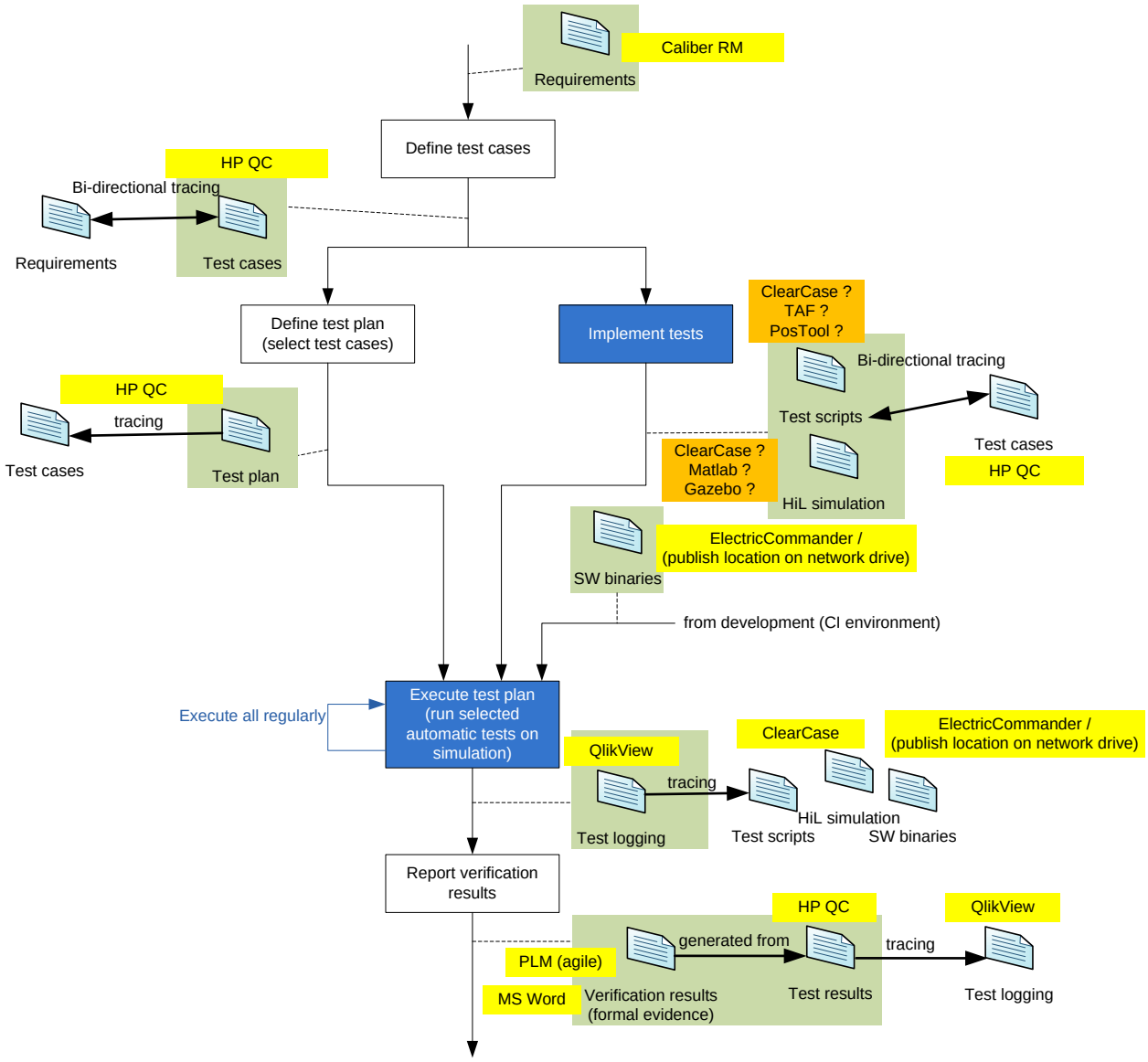


Figure 3-3: System Verification process (proposed).

Description of the changed process activities:

1. *Implement tests*: This is a new step. The test cases are implemented. This (possibly) requires scripting tools and HiL simulation tools. The resulting work products will also need to be stored in a database. For these new work products traceability and versioning is required (a change in a test case may invalidate the associated test script).
2. *Execute test plan*: The automatic (subsystem level) test cases are executed regularly in the Continuous Integration environment. The test results must be traceable to software baselines and test implementation versions (which must be traceable to test cases). The traceability should make it possible to obtain the “Verification results” from the “Test logging” automatically (in the Report Verification Results activity).

4 Identification of Engineering Methods

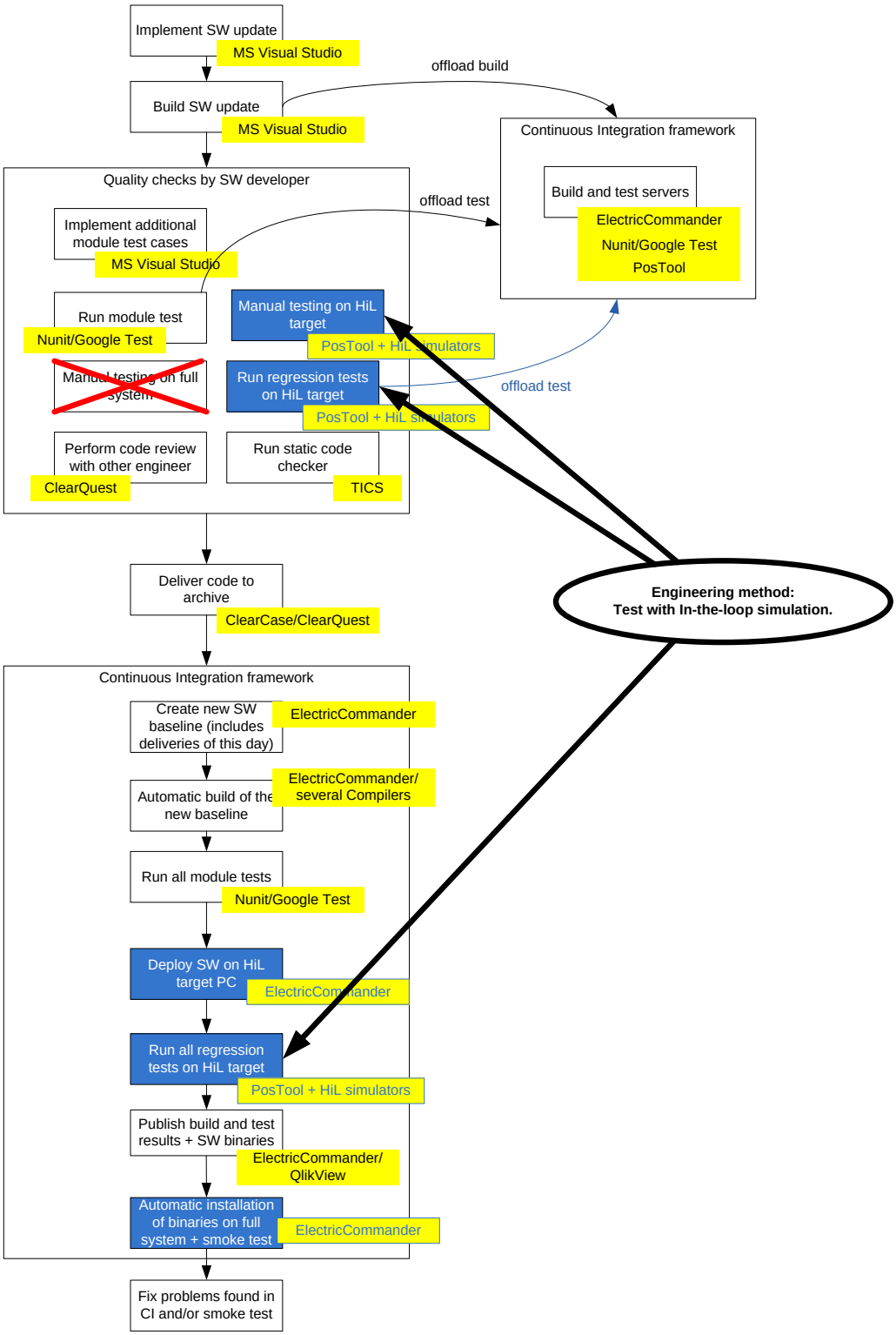


Figure 4-1: identification of engineering methods (implementation and test cycle).

We distinguish between manual and automatic testing engineering methods because they may require different tooling. For both behavioral modelling is needed, but for manual testing also graphical user interfaces are needed (Joystick control, 3D model view showing the movements of the system,...).

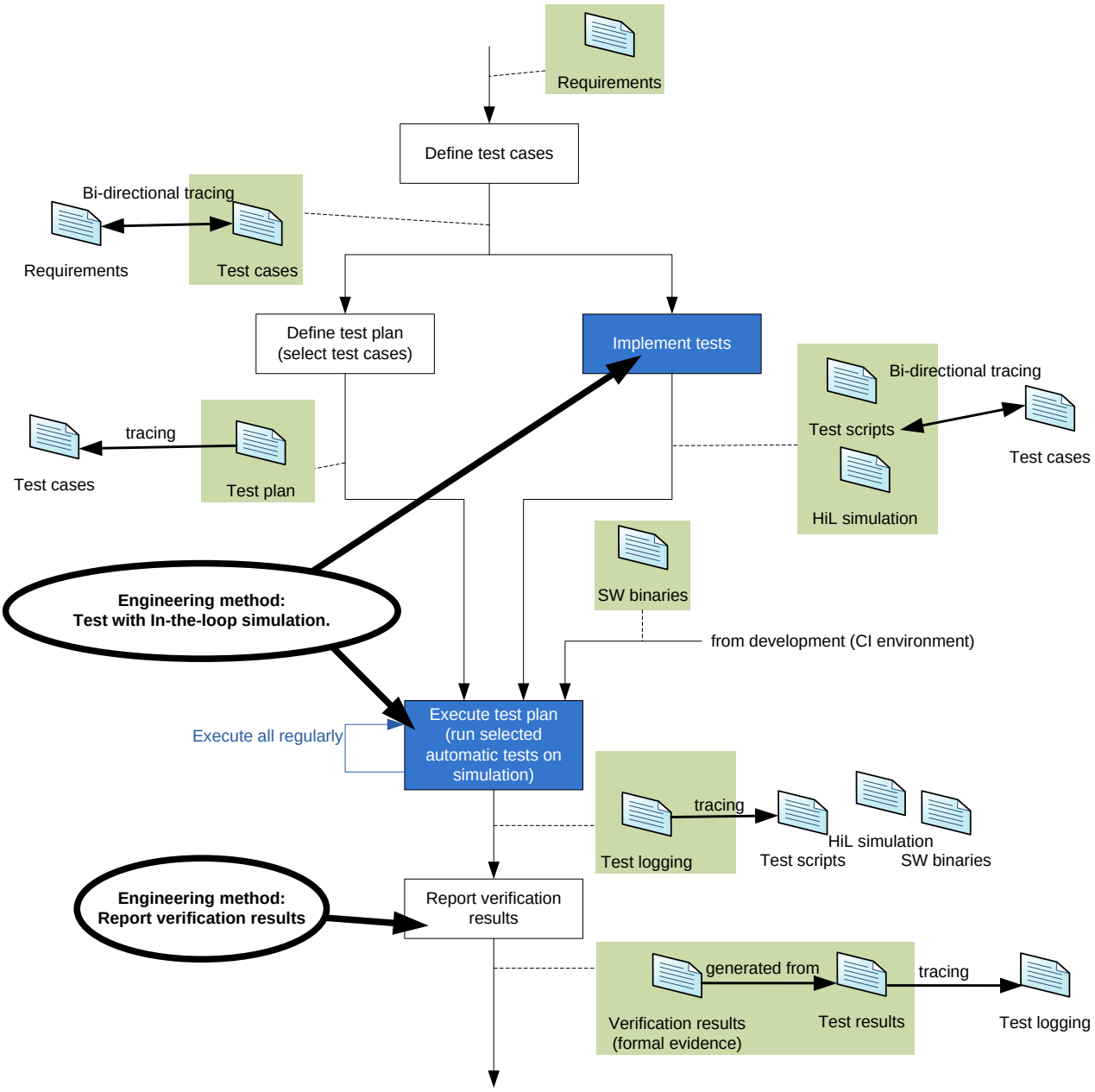


Figure 4-1: identification of engineering methods (system verification).

The *"Report Verification Results"* engineering method is about providing the correct test evidence. In this case this involves the collecting of results of automatic tests.

5 Technical case study: testing the Table Force Sensor

The table force sensor is a safety measure introduced to detect collisions between a patient and the monitor ceiling suspension (MCS). In order to facilitate automatic testing of this feature some form of simulation is required as collision forces from the environment are needed as input. This makes the force sensor a good candidate for HiL simulation testing.

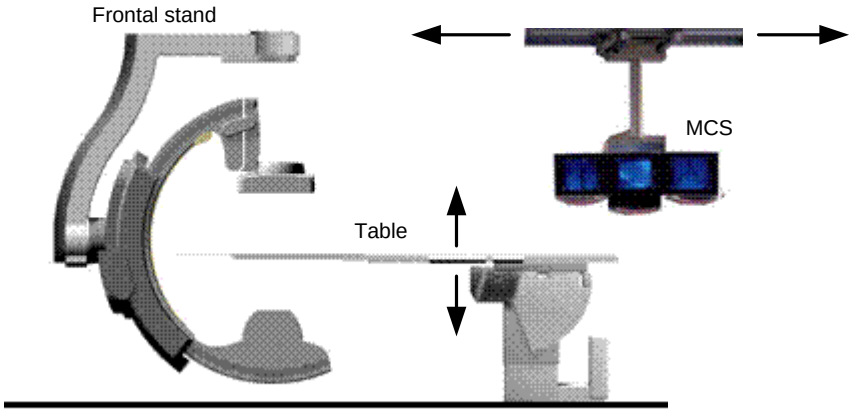
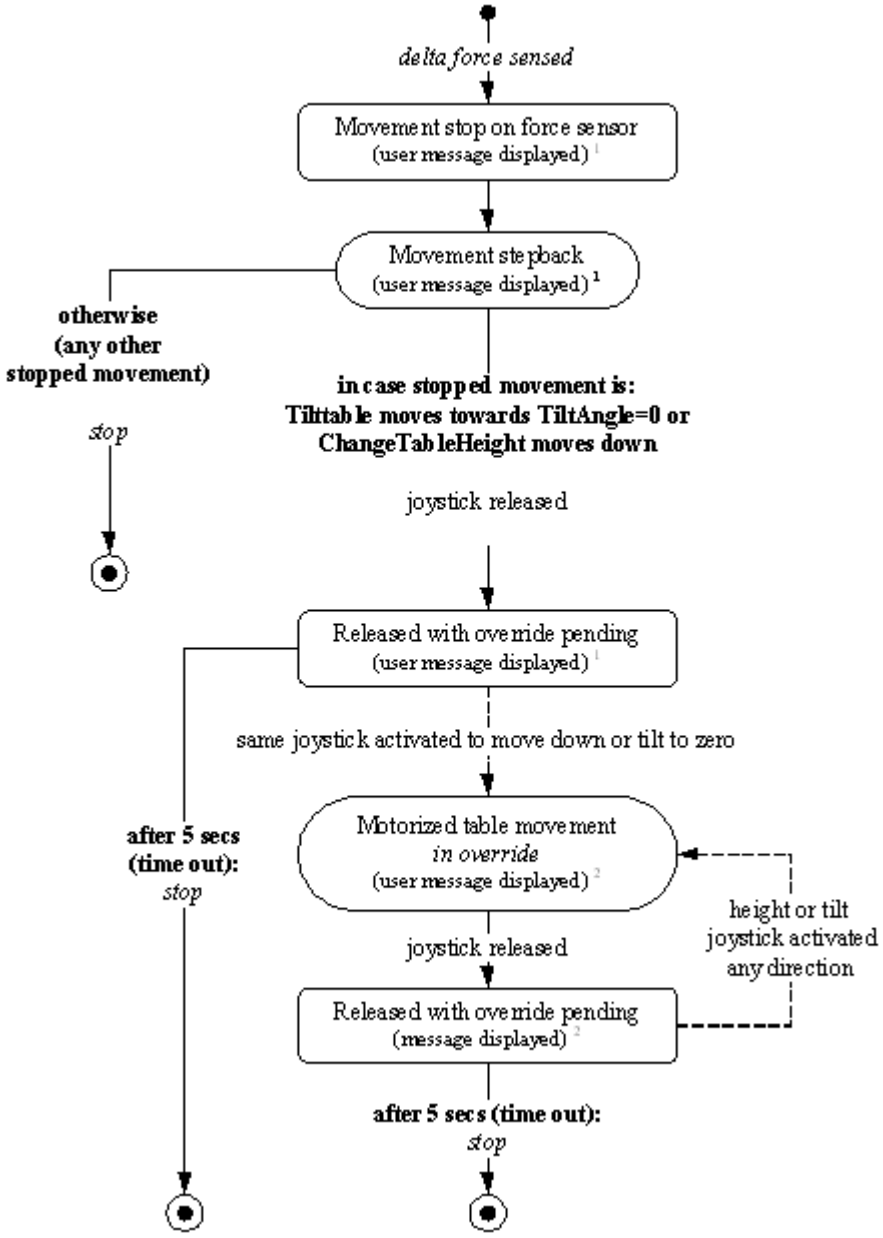


Figure 5-1: Monitor Ceiling Suspension.

Below is an excerpt from the requirement specification for the table force sensor behavior.

5.1 Specification

The patient table is equipped with a Force Sensor measuring a force vertically applied to the surface of the tabletop. Normally the measured force will be determined by the patient weight. The behavior is indicated in the following flow diagram:



When during the motorized movement the Force Sensor detects a collision force that exceeds the threshold, which in most cases will be below 350N but always below 450N:

- Movement Stepback function: The motorised movement moves as quickly and fast as possible in reverse direction during at least 0.5 sec;
- The UIMessage TABLE_COLLISION_ACTIVE is given to warn the user about the collision.

But for some movements, typically performed during CPR, it is required to continue uninterrupted by the Force Sensor:

- motorized table movement function: The stopped movement is performed but now in override of the force sensor according the flow diagram above .
- The UIMessage TABLE_COLLISION_OVERRIDE is given to warn the user about the collision.
- NOTE: To prevent interrupting the CPR no audible signal is given

When the main usecase movement is stopped by the force sensor it only can be continued in override when it is activated again, within the defined timeout interval, in the same movement direction. But when the movement is activated in override and the joystick is released it can be activated again in override, within the defined timeout interval, in each direction.

The override pending status couples the ChangeTableheight and TiltTable main usecases such that the override mode is combined for both usecases. When tilting in override and the joystick is released than within the defined time out interval the height can be changed in override and v.v.

6 Terms, Abbreviations and Definitions

Caliber RM	Caliber Requirements Management (tool).
ClearCase	Configuration Management (code archive).
ClearQuest	Change Control Management tool (defect handling)
ElectricCommander	Continuous Integration tool
Google Test	C++ unit test framework
HiL	Hardware-in-the-Loop
HP QC	HP Quality Centre
MiL	Model-in-the-Loop
NUnit	C# unit test framework
PiL	Processor-in-the-Loop
PLM/Agile	Product Lifecycle Management (documentation archive)
PosTool	Proprietary perl-based tool to manage the subsystem simulation (selecting configurations, starting/stopping the subsystem, selecting and starting tests, etc.).
QlikView	Dashboard tooling (provides reporting of build results/test results/quality checks.
SiL	Software-in-the-Loop
TAF	Test Automation Framework. Proprietary Excel and Visual Basic based tooling to perform tests on (sub)system level.
TICS	Static code checker from TIOBE (checks violations against the coding standard).

Table 6-1: Terms, Abbreviations and Definitions

7 References

Please add citations in this section.

[Author, Year]	Authors; *Title*; Publication data (document reference)

8 Annex I: Detailed Descriptions of the Engineering Methods

8.1 Test with In-the-loop-simulation

See excel sheet: EngineeringMethods-TestWithInTheLoopSimulation v1.0.xls (screendump below).

Engineering Method: UC43 - Test with In-The Loop-Simulation		
Purpose: detect software problems early, especially concerning relation with hardware		
Comments: related to heterogeneous simulation, but this method allows also other combinations		
Pre-Condition	Engineering Activity as Steps	Post-Condition
Availability of model(s) for the hardware (for different system configurations and with different level of detail) and control software or models of this software (also including different configurations). Models and control software are not defined in same language (e.g. Matlab, Dymola, POOSL)	1. The user installs the simulation environment(s) and software components on the appropriate resources. 2. The user selects simulation purpose (e.g., functional, real-time) and mode (e.g., manual, automated testing). 3. The user selects the machine configuration (e.g., component types, software version) to be simulated. The simulation environment presents a subset of the models that can be selected for the desired simulation (e.g., detailed models, fast high-level models). 4. The user selects the models to be used, the tool environment prepares appropriate glue / communication code that allows communication between hardware model and (model of) the software. 5. In case of automated testing, test scripts (including input data for the models) are downloaded from a database, taking into account the machine configuration. 6. The user starts the simulation; the simulation of the hardware model is synchronized with (the model of) the software. 7. During manual simulation: a 3D visualziation of the system is shown giving the user feedback on movements; the user can provide input (to the software; and via the simulation environment to the models) and inject faults; Giving input to the models should be user-friendly (e.g.: for testing an object distance sensor the user should not have to input distances, but rather place a foreign object in the 3D environment, to which distances can be calculated). 8. The simulation results (e.g., pass/fail) are automatically stored in a database.	File with results of the simulation, e.g, results of test cases. Insight in the correctnes of the software, including the impact of faults. Identification of problems and bottlenecks, especially concerning the combination of hardware and software. Database where simulation results of different configurations, software versions, etc, are collected.
Notes: The simulation environment takes care of glue code to connect the model (e.g., to simulate a certain communication protocol), and user windows to generate input or to inject faults.	Notes: research is needed to determine how to execute the physical model in combination with (a model of) the software. Preferably, a single simulation environment will be selected	Notes:

Artefacts provided as input of the activity		Artefacts produced during of the activity		Artefacts which are the result of the activity	
Name		Name		Name	
Generic Type: (Tool or language independend type)		Generic Type: (Tool or language independend type)		Generic Type: (Tool or language independend type)	
Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)	
Description:		Description:		Description:	
Name		Name		Name	
Generic Type: (Tool or language independend type)		Generic Type: (Tool or language independend type)		Generic Type: (Tool or language independend type)	
Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)	
Description:		Description:		Description:	
Name		Name		Name	
Generic Type: (Tool or language independend type)		Generic Type: (Tool or language independend type)		Generic Type: (Tool or language independend type)	
Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)	
Description:		Description:		Description:	

8.2 Report verification results

See excel sheet: Engineeringmethods-ReportVerificationResults v1.0.xls (screendump below).

Engineering Method: UC43 - Report Verification Results					
Purpose: provide test evidence					
Comments:					
Pre-Condition		Engineering Activity as Steps		Post-Condition	
* test cases present in test management tool. * test plan present in test management tool. * test logging available in some database (e.g. dashboard)		* Select the appropriate test logging; this will typically involve things like: * Selecting a project. * Selecting a SW baseline. * ... * Copy the test logging to the test management tool and link it to the test cases. * Generate verification report from the test result data.		The results (PASSED/FAILED) and the detailed logging of the executed test cases (test plan) are stored in the test management tool and is linked to the associated test cases. A verification report of the executed test plan is present in the document archive. This is the official evidence, which is authorized by the responsible person.	
Notes:		Notes: copying the test logging could be an import action in the test management tool, an export action in the test logging database tool (dashboard), or an import-export action in a separate tool.		Notes:	
Artefacts provided as input of the activity		Artefacts produced during of the activity		Artefacts which are the result of the activity	
Name	Test cases	Name		Name	Verification result
Generic Type: (Tool or language independend type)	Textual descriptions	Generic Type: (Tool or language independend type)		Generic Type: (Tool or language independend type)	PASSED/FAILED
Shared Properties: (Information to be shared in interaction between steps)	identifications, including some version control	Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)	identification (linked to versioned test case)
Description: textual specification of a test		Description:		Description:	
Name	Test plan	Name		Name	Verification logging
Generic Type: (Tool or language independend type)	set of test cases	Generic Type: (Tool or language independend type)		Generic Type: (Tool or language independend type)	log files
Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)	identification (linked to versioned test case)
Description: a set of test cases (to be executed).		Description:		Description: files containing detailed logging of test execution.	
Name	Test logging	Name		Name	Verification report
Generic Type: (Tool or language independend type)		Generic Type: (Tool or language independend type)		Generic Type: (Tool or language independend type)	document
Shared Properties: (Information to be shared in interaction between steps)	Must be traceable to test cases	Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)	
Description: the output of test execution, including PASSED/FAILED result, software log files.		Description:		Description: document describing the results of the tests executed (test plan).	

9 Annex II: Technology Base Line & Progress Beyond

This information will be collected globally, and the respective part will be inserted here. Basically it could be something like a table with a row for each engineering method and a column for the current functionality, which is the technology baseline (e.g., “data has to be transferred by hand”), and a column for the expected progress in CRYSTAL (e.g., to be implemented in CRYSYTAL / “future work”).

The exact content of this section will be defined in the next technical Board Meeting.