CRYSTAL

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

Use Case Development Report
UC403 Motion control of patient table and X-ray beam positioning

D403.901

DOCUMENT INFORMATION

Project	CRYSTAL
Grant Agreement No.	ARTEMIS-2012-1-332830
Deliverable Title	Use Case Development Report UC403 Motion control of patient table and X-ray beam positioning
Deliverable No.	D403.901
Dissemination Level	CO
Nature	R
Document Version	V1.00
Date	2014-04-30
Contact	R. Vermeulen
Organization	Philips
Phone	+31 402769495
E-Mail	rogier.vermeulen@philips.com

AUTHORS TABLE

Name	Company	E-Mail
E. Korff de Gidts	Philips	eric.korff.de.gidts@philips.com
R. Vermeulen	Philips	rogier.vermeulen@philips.com
P. Tielemans	Philips	paul.tielemans@philips.com
R. Albers	Philips	r.albers@philips.com
H. Schouten	TNO	hanno.schouten@tno.nl
R. Verhoeven	TU/e	P.H.F.M.Verhoeven@tue.nl

CHANGE HISTORY

Version	Date	Reason for Change	Pages Affected
V1.00	2014-04-29	Approved version	

CONTENT

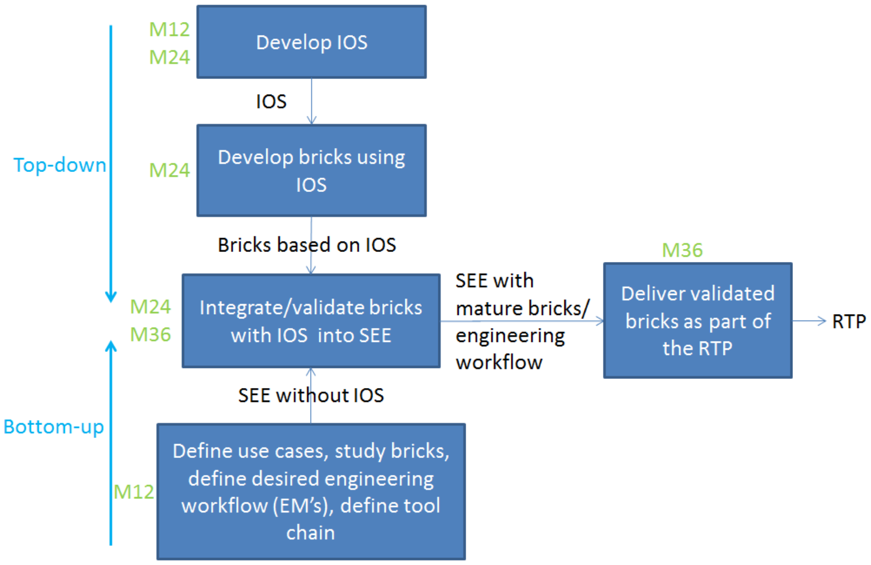
Content of Figures

Content of Tables

1 Introduction

1.1 Role of deliverable

The intention of this Use Case Development Report is to provide an (annual) overview on the status of the development of the Interoperability Specification (IOS), its engineering methods, engineering environment and improvement activities related to the development of Use Case 4.3 Motion control of patient table and X-ray beam positioning. As depicted in the figure below, its content will vary over time, in line with the phase of the Crystal project it is reporting upon.

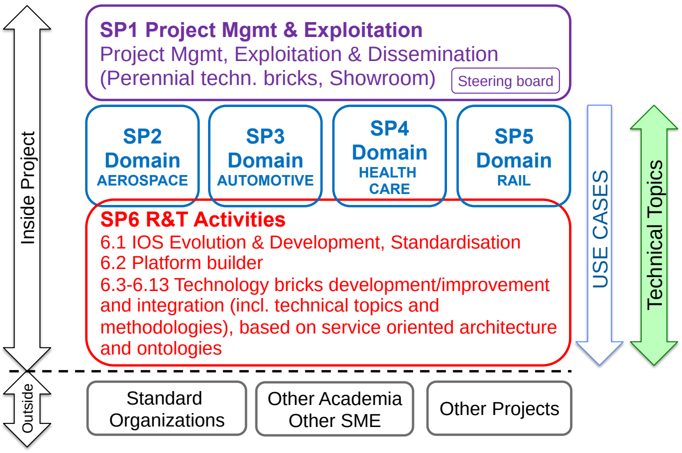


M12
M24
Develop IOS
IOS
Top-down
M24
Develop bricks using IOS
Bricks based on IOS
M36
M24
M36
Integrate/validate bricks with IOS into SEE
SEE with mature bricks/ engineering workflow
Deliver validated bricks as part of the RTP
RTP
SEE without IOS
Bottom-up
M12
Define use cases, study bricks, define desired engineering workflow (EM's), define tool chain

Figure 1 Crystal timeline

1.2 Relationship to other CRYSTAL Documents

The figure below provides a general overview of the internal structure of the Crystal project. This work package is part of the Healthcare domain (SP4). Its information and reports are input for WP6.



Inside Project
Outside
SP1 Project Mgmt & Exploitation
Project Mgmt, Exploitation & Dissemination
(Perennial techn. bricks, Showroom)
Steering board
SP2 Domain AEROSPACE
SP3 Domain AUTOMOTIVE
SP4 Domain HEALTH CARE
SP5 Domain RAIL
SP6 R&T Activities
6.1 IOS Evolution & Development, Standardisation
6.2 Platform builder
6.3-6.13 Technology bricks development/improvement and integration (incl. technical topics and methodologies), based on service oriented architecture and ontologies
USE CASES
Technical Topics
Standard Organizations
Other Academia Other SME
Other Projects

Figure 2 Crystal project structure

This document is closely related to the [1] Use Case Definition Report for Use Case 4.3 Motion control of patient table and X-ray beam positioning. Where the *Use Case Definition Report* elaborates on the technical details and the decision making process, the *Use Case Development Report* is used to provide a condensed overview of the planned and scheduled improvement activities, with an Executive summary on the description of work and its conclusions.

1.3 Structure of this document

The structure of the document is as follows:

- *Section 2* provides a high-level presentation of the use case, the product developed, and the problem space as perceived from the end user perspective. It provides a refinement of the use case concerning a robotic positioning system and it lists the requirements concerning interoperable tooling.
- *Section 3* describes the development activities related to the engineering workflow for this work package. It describes the initiatives started, and the envisioned engineering workflow, planned to be available at the M36 milestone. It highlights the engineering methods associated with this work package and concludes this section with a list of artefacts relevant for this work package.
- *Section 4* discusses the Systems Engineering Environment and the improvements made here. It also provides a description of the tool chain and its artefacts.
- *Session 5* provides a brief description on the content of the demonstrator prepared. *Session 6* elaborates on the lessons learned, both within the work package, from other industry partners, or cross-domain.
- *Appendix A* provides the Engineering Method descriptions.
- *Appendix B* provides tool descriptions for the tool chain
- *Appendix B* captures the interaction diagrams for the build, integration and test environment.
- *Appendix C* provides an integral version of the Use Case Definition Report of Use Case 4.3.

2 Use Case 4.3 Motion control of patient table and X-ray beam positioning

This paragraph provides a high-level presentation of the use case, the product developed, and the problem space as perceived from the end user perspective. It provides a refinement of the use case concerning a robotic positioning system and it lists the requirements concerning interoperable tooling.
More detailed information on Use Case 4.3 is available in the [1] Use Case Definition Report of WP4.3.

2.1 Introduction

The use cases of Philips Healthcare concern the control part of an interventional X-ray system. It is important to be able to incorporate medical innovations quickly in such systems. This is challenging, because high quality standards have to be met and a trade-off has to be made between usability and safety. To meet these goals, Philips investigates a migration towards a new ***component-based architecture*** and an improved ***model-based iterative development process***, supported by ***interoperable tooling***. The three use cases concern different layers of control of an interventional X-ray system.

The development is used to investigate interoperable tooling for:

- Component-based development
- Multi-disciplinary modelling and simulation, supporting continuous integration
- Code generation from models
- Real-time behaviour and performance analysis
- Test and integration

The primary objective of this Use Case is to achieve the desired speed profiles and positions of the robotic positioning system. The Use Case is focusing on mixed physics/data-based modelling and simulation of sensorial and mechanical uncertainties in robotic positioning systems and on how these uncertainties translate to the performance of the systems in their environment and therefore on the safety of the complete system.

2.2 Medical context

Image-guided interventions and therapy demand for an eased workflow with regards to manoeuvring patient examination table and stands. The integration with various components into the OR and Cathlab makes safe positioning of the X-ray system challenging. As an example, see the figure below where a Hybrid OR room is shown, full of equipment.

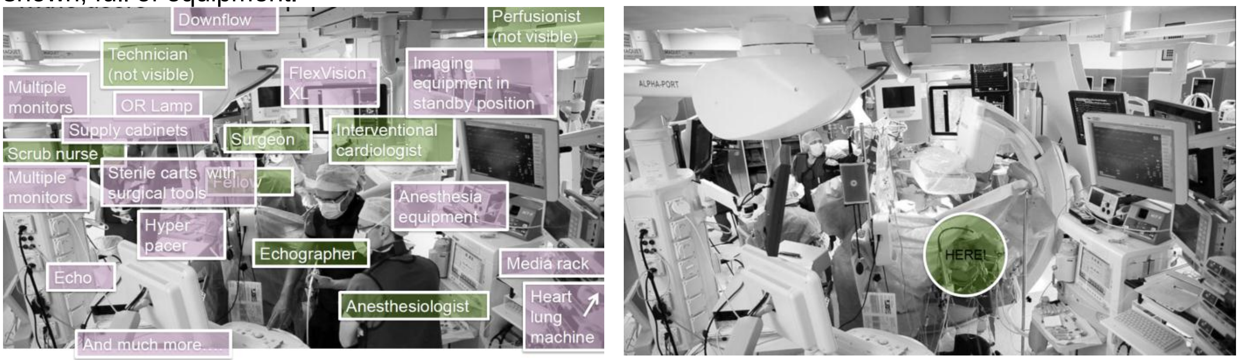


Figure 3: Hybrid OR with (left) all equipment and (right) the position of the patient in the room.

Ideally interventional X-ray camera’s would be small and light, enabling easy control, not restricting in anyway the doctor in doing clinical procedures. Unfortunately this is not the case; in real life we have a heavy camera (consisting of tube, collimator and flat detector, etc.) which needs heavy and large mechanics. Moreover we need a large patient examination table to support the desired position of our patient.

The mechanical degrees of freedom

The mechanics part of the X-ray system contains a patient examination table to support the patient and a stand that carries the X-ray tube and detector. Its user interfaces allow the clinical staff to move and position the patient and the X-ray beam in order to generate images or series of images from medically appropriate projections.

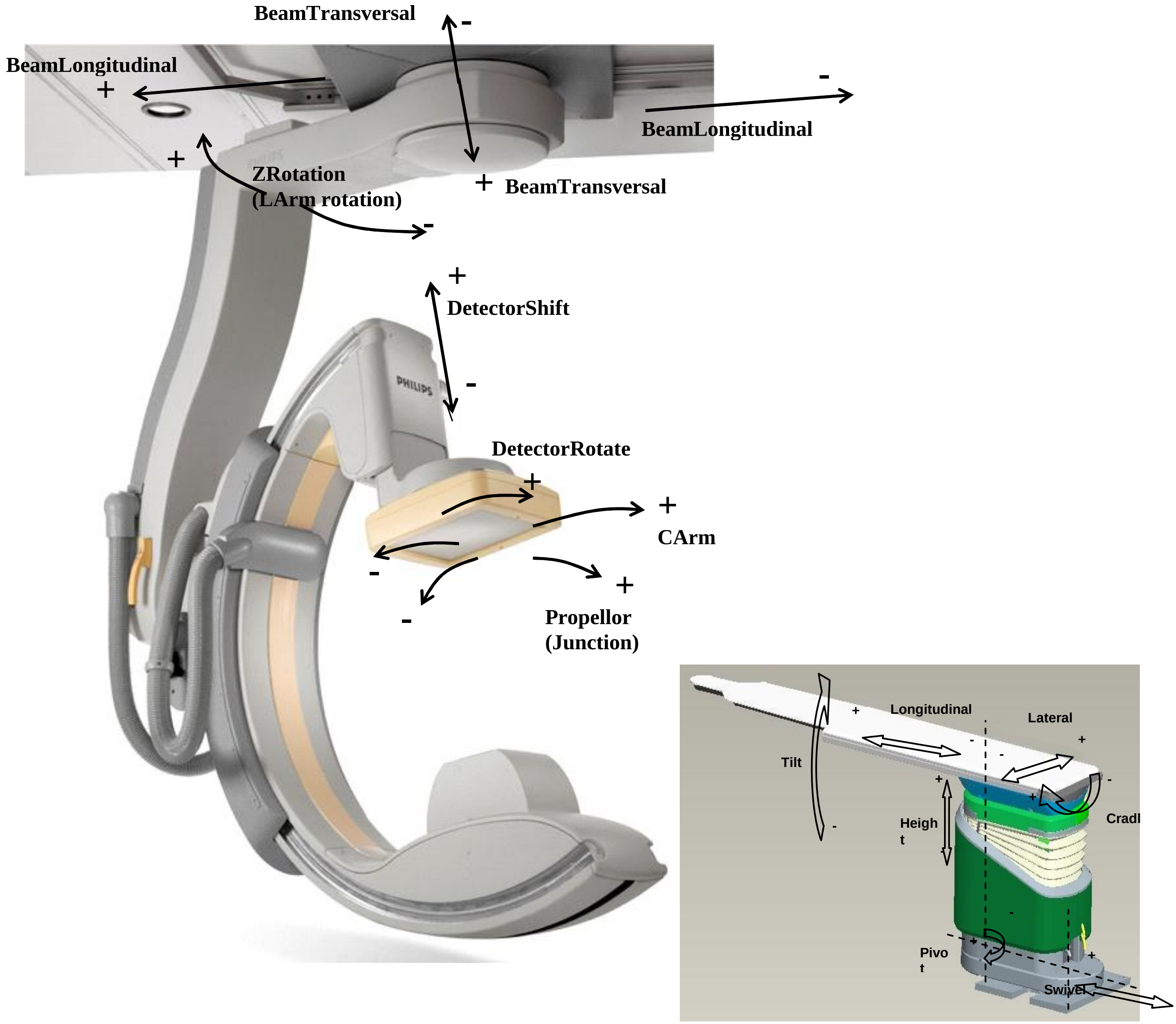


Figure 4: A sample product configuration and its axis of movement; a table and ceiling suspended stand.
A sign indicate whether the position increases (+) or decreases (-) with a movement in the arrowed direction.

Patient positioning coordinate systems[1]

The room, patient support, patient, X-ray and detector each have their own co-ordinate system. In this way, the relative positions of them can be defined. The following pictures show these co-ordinate systems.

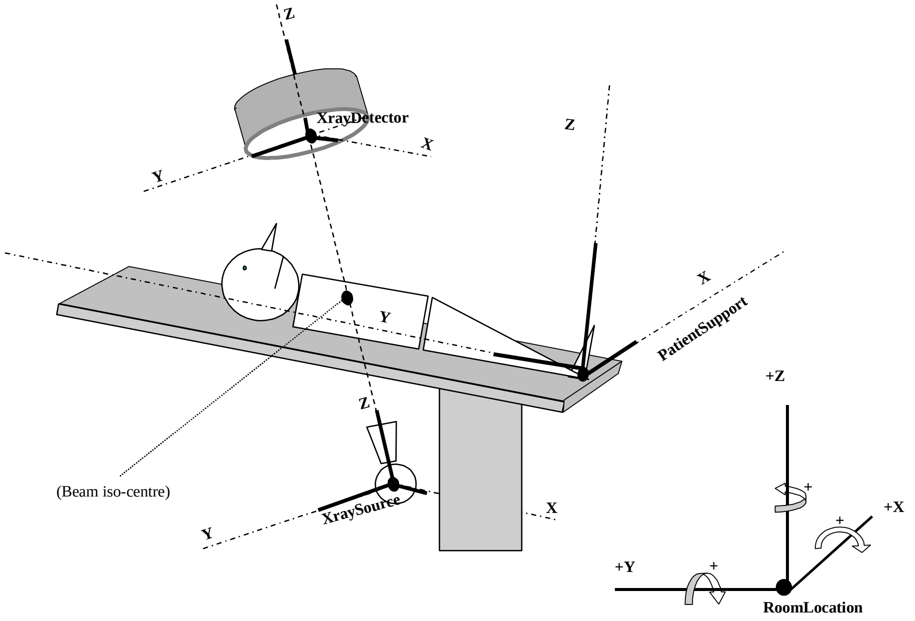


Figure 5: Patient positioning coordinate systems (1)

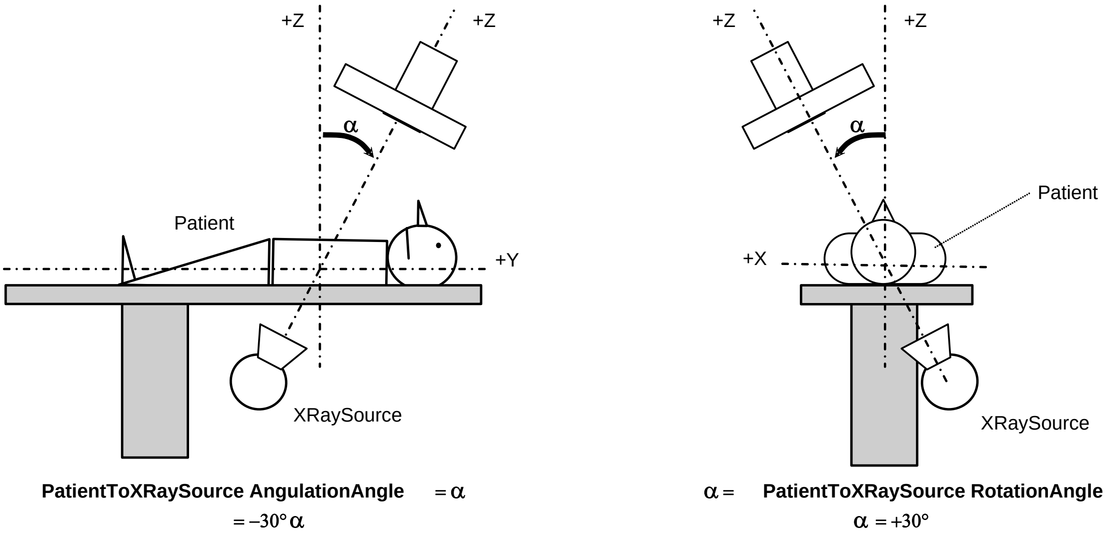

+Z
+Z
α
Patient
+Y
XRaySource
PatientToXRaySource AngulationAngle = α
= –30° α
+Z
+Z
α
Patient
+X
XRaySource
α = PatientToXRaySource RotationAngle
α = +30°

Figure 6: Patient positioning coordinate systems (2)

[1] Patient positioning coordinate system details are purposely omitted.

Patient accessibility

To improve the system on flexibility in patient setup, the detector can be rotated. Detector rotation enables several features: Instead of controlling “stand oriented” movements the system enables control of “patient oriented” movements. At the same time the stand will not obstruct the doctor in his work and enable better patient accessibility. Moreover, flexibility of positioning the mechatronic is improved by extending the set of positions it can be used in.

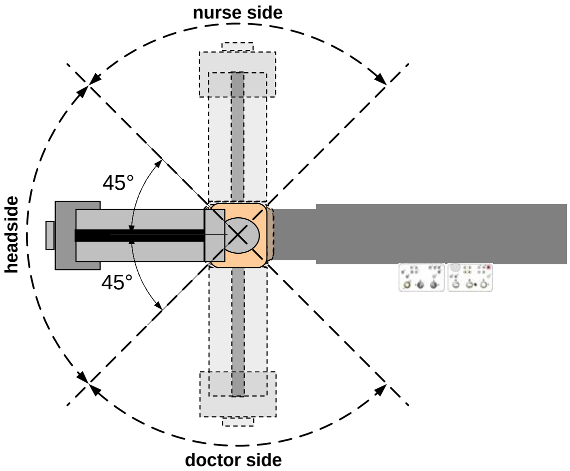


Figure 7: patient accessibility is improved by detector rotation and larger Z-rotation angle.

It’s desired to decouple the clinical user interface from the basic axes movements. Detector and collimator are aligned with lines of constant rotation and angulation (see globe).

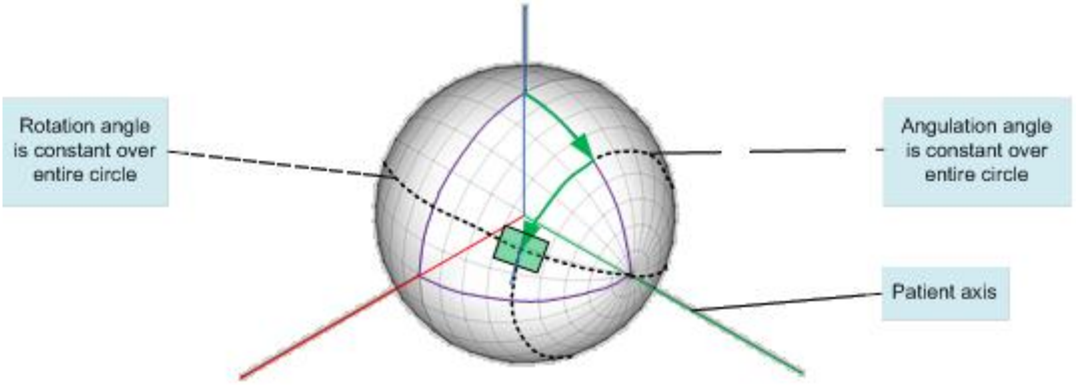

Rotation angle is constant over entire circle
Angulation angle is constant over entire circle
Patient axis

Figure 8: Patient oriented movements: joystick controls angulation and rotation angle of image beam.

Several challenges appear at the horizon when designing a system where there is no one-to-one relation between user interface buttons/joysticks and the basic movement axes:

(1) Limitations of patient oriented movements by hardware restrictions. Philips wants to gain insight in and demonstrate the possibilities and limitations of the system to applicants or stake holders.

(2) Multiple scenarios where detector or tube might collide with the patient examination table stand. The physical movement ranges of all individual axes are not limited such that no collisions can occur. In order to prevent collisions path guarding software checks for impending collisions. This way a certain clearing distance is taken into account. From an end users point of view the clearing distance should be as small as possible in order to get an optimal view on a patient. From machine point of view the clearing distance should be as large as possible in order to guarantee NO collisions in any circumstance. Due to uncertainty of all individual axes positions there is uncertainty of the position where a collision might occur. Philips want to gain insight in to determine how the smallest possible clearing distance between C-arc (with collimator and detector) and patient examination table stand can be achieved given axes accuracies in multiple scenarios.

2.3 Challenges at M0

Various trends and drivers challenge the current engineering workflow, including:

- Increased design complexity due to higher demands on flexibility in the clinical room layout, clinical requests for patient oriented movements, and increased integration with other medical equipment.
- Increased variability triggered by efforts to adapt the same product platform for a broader audience (e.g. in intended use, value segment, or notified bodies)
- Reduce time to market.

While the current way of working is rather document driven and code-centric, various opportunities for improvement in the engineering workflow are identified:

- Information is stored at various places and thus hard to find.
- Information is typically replicated in the various tools used and thus hard to keep consistent.
- Requirements are predominantly natural language based which may not be the most effective means of communication, makes it hard to determine quality levels, and hampers re-use throughout the engineering workflow. Frequently ambiguous, error-prone, or misinterpreted.
- Models aren't re-used throughout the engineering workflow, thus it is hard to preserve the overall consistency in requirements, design, implementation, and verification/validation.
- Model engineering is done manually while tools can assist in establishing a level of abstraction.
- Maintaining the traceability of information is a manual exercise which is hard to keep consistent across the various tool environments.

Models are recognized as a means to counter complexity by raising the level of abstraction:

✓ As requirements aid by defining the desired product behaviour (e.g. behaviour models)
✓ As design aid by defining the actual product behaviour (e.g. architectural / structural models)
✓ As implementation aid via code generation.
✓ As verification aid by predicting product behaviour (e.g. emulation or simulation models)
✓ As validation aid by providing early clinical feedback on the product behaviour.

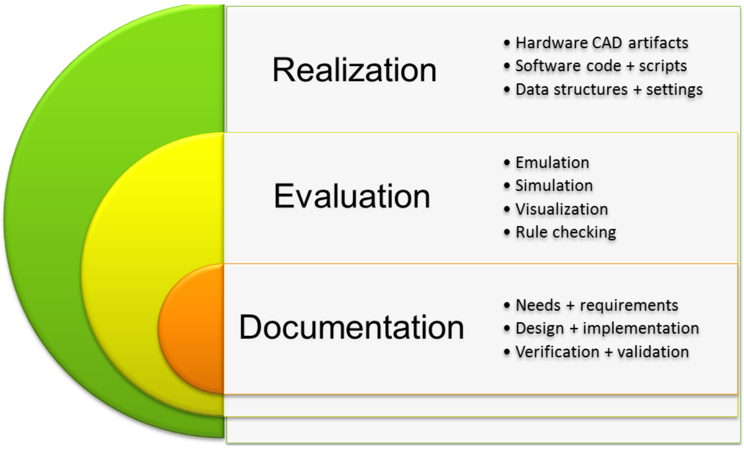


Figure 9 Evolutionary path towards model re-use throughout the engineering workflow

Re-use of models throughout the engineering workflow is perceived a strategic and breakthrough systems engineering innovation. While it aligns closely to most of the challenges and positively affects the concerns on information scattering, replication, consistency, quality levels, re-use, ambiguity, or tractability, it doesn't come for free.

This breakthrough innovation challenges the organizations capabilities, especially in the area of model engineering, model-based development, and model-based simulation and verification and poses new requirements towards its processes, tools, and engineers. Other factors to consider while re-using models are the potential impact on product safety (as depicted in Figure 9 above) or the chance for detecting or the cost for resolving model deficiencies.

3 Engineering workflow

This paragraph describes the development activities related to the engineering workflow for this work package. It describes the initiatives started, and the envisioned engineering workflow, planned to be available at the M36 milestone. It highlights the engineering methods associated with this work package and concludes this section with a list of artefacts relevant for this work package.

3.1 Engineering workflow at M0

WP4.3 targets improvements that focus on the part of the V-model indicated in Figure 10. Figure 11 zooms in on this focus; it highlights the engineering workflow as was the current way of working prior to Crystal.

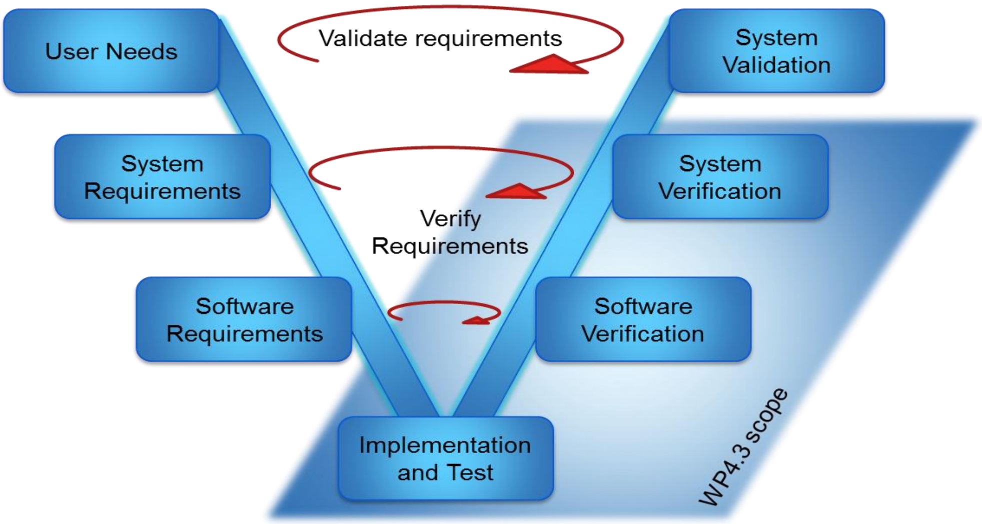


Figure 10 Scope of Crystal Work Package 4.3

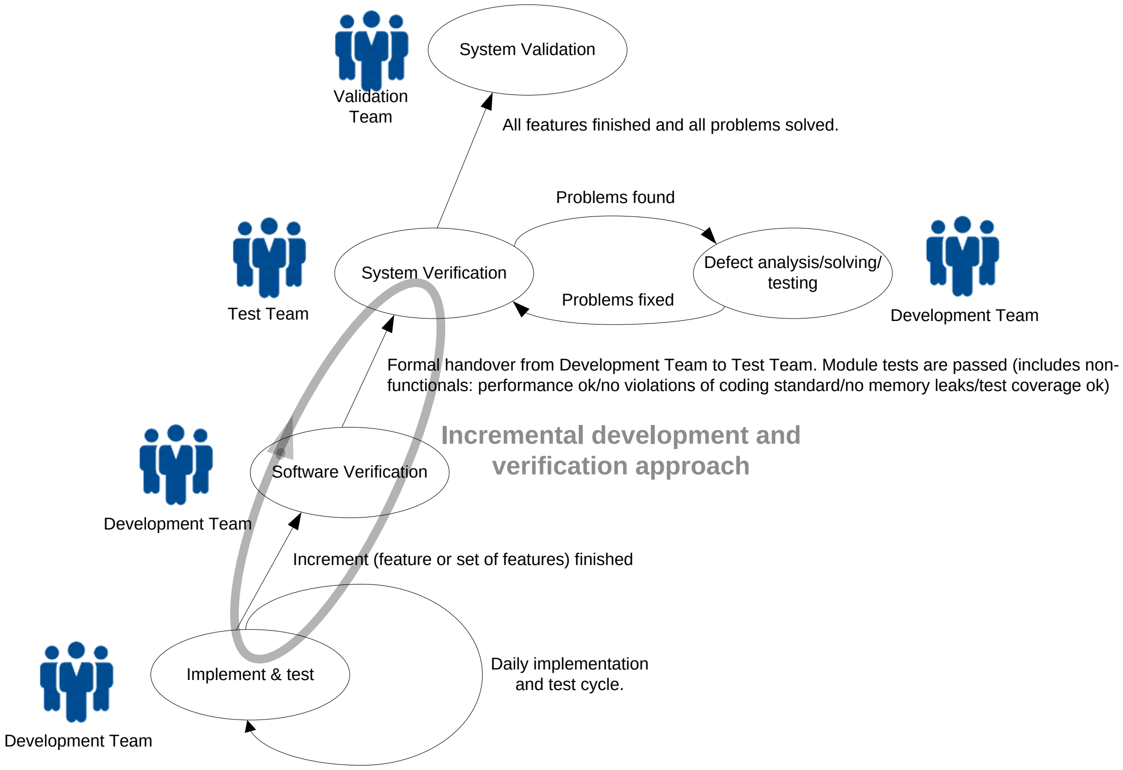

Validation Team
System Validation
All features finished and all problems solved.
Problems found
Test Team
System Verification
Problems fixed
Defect analysis/solving/ testing
Development Team
Formal handover from Development Team to Test Team. Module tests are passed (includes non-functionals: performance ok/no violations of coding standard/no memory leaks/test coverage ok)
Incremental development and verification approach
Development Team
Software Verification
Increment (feature or set of features) finished
Implement & test
Daily implementation and test cycle.
Development Team

Figure 11 Engineering workflow of Use Case 4.3 at M0

An incremental way of working is employed as is described in Figure 11 Engineering workflow of Use Case 4.3 at M0. 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 it is to fix it. When a developer finds a bug testing his SW update on his PC it may take 1 man-hour 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 investigation (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). When customers indicate that needs aren’t fully met, even the product architecture may be affected. All these activities are coordinated by a CCB (Change Control Board). This process will at least cost 8 to 16 man-hours. 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.

3.2 Initiatives started

In order to reach our improvement goals, initiatives were initiated within WP4.3 to explore the currently available solutions – as offered by the technology providers – for their fitness of use.

With regards to the development of the individual models, various opportunities were recognized. A few were selected for their level of complexity (not to simple, but also not too complex to start with) and their alignment with the short-term engineering needs:

- Motion Control Interface behaviour
- Table height model; hardware emulation model for single axis
- Table force sensor model; hardware emulation model for force sensor
- Mechanical model of the patient table and stand, as used for the 2D visualizer

The engineering task at hand offered a context for exploring model based engineering and the opportunity to practice with the model engineering cycle in a rather condensed setting. While the simulation environment is in its infant state, most activities delivered a rather stand-alone and isolated environment for creating and exploring these models.

The subsequent adaptations to the product and its associated models are graphically depicted below;

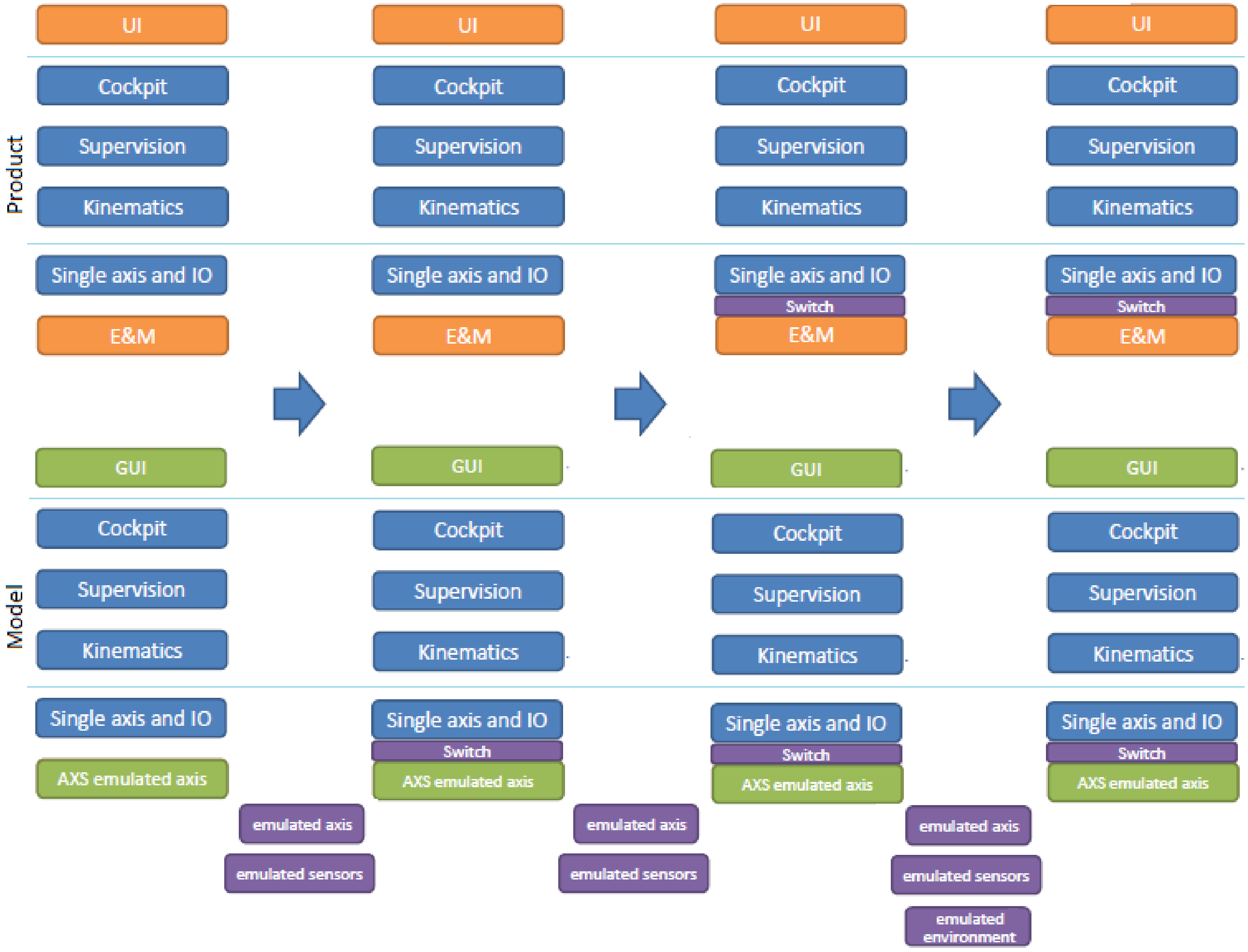


Figure 12 Modelling environment changes over time

The Switch is realized in *Activity A7: The actual implementation of Communication Abstraction Layer*. The emulated axis and sensors by *Activity A5: The Matlab Modelling for the table force sensor*. The emulated environment is the combined end result of the activities in *Modelling Infrastructure* column in Table 1.

The initiatives were split along the following themes;

a) Individual models and the ecosystem architecture.
The activities at this level are primarily concerned with the content of an individual model, and its associated quality attributes as defined under ISO 25010. Its ultimate goal; develop an ecosystem of models that together cover all critical to satisfaction/quality aspects of a product family. The models ability for re-use throughout the engineering workflow is another example of a factor to take into consideration while defining the scope and context of individual models.

b) Model engineering infrastructure.
Activities related to the simulation and visualization architecture and its underlying components, enabling the extraction, distribution, and processing of data that facilitates in the development and verification of model content. The tools in this category may exhibit product specific behaviour.

c) Optimizing the engineering workflow.
Emphasis is on the interoperability between tools and the smooth exchange of engineering artefacts and associated meta-data while heading towards an environment that supports continues build, integration, and testing. The tools are typically generic in nature and don't exhibit product specific behaviour.

d) Institutionalize changes to the Systems Engineering Environment
Change management activities required for embedding alternative tools, technologies, or processes in the standing R&D organization, thus ensuring on its sustained adaptation within the domain or industry. It includes activities required for technology demonstrators and such.

The table below provides an overview of the main theme behind the initiatives started in the M12 time frame.

	Activity	Individual models	Modelling infrastructure	Workflow optimization	Change management
A1	Model-to-code transformation for the Motion Control Interface	■		■	
A2	IBM Rational Team Concert (RTC) pilot		■	■	
A3	Evaluating the organizational needs and potential benefits of using GAZEBO and OROCOS		■		
A4	Evaluation and considerations related to the levels of integration		■		
A5	The Matlab Modelling for the table force sensor	■			
A6	The definition of a simulation architecture and integration plan		■	■	
A7	The actual implementation of Communication Abstraction Layer		■		
A8	M12 Demonstrator: Integrated demo WP4.1 + WP4.3				■

Table 1 Main theme behind the initiatives for WP4.3

The following sub-paragraphs provide an executive summary of these initiatives, irrespectively on the study outcome. The demonstrator (A8) is further discussed in Demonstrator section (§5.1).

3.2.1 Activity A1: Model-to-code transformation for the Motion Control Interface

Description of work

This activity is based on a small-scale study using IBM Rhapsody, executed before CRYSTAL. This previous study showed positive results of the methodology behind IBM Rhapsody. It also showed a potential increase of software development speed when using UML modeling and, with that, code generation and document generation and a potential decrease in late found defects. The goal of this follow up activity within CRYSTAL is to investigate whether the use of the modelling tool IBM Rhapsody during the software development steps really has a positive effect on development speed and/or initial product quality.

The stake holders for this activity are Philips, and IBM.

As shown in the picture below, in the traditional way of software development the steps to go from requirements to code are all done manual. During each of these manual steps potentially “translation” bugs are introduced. These bugs are only found during unit/system testing when the product is ready. Preferably these bugs should be avoided or they should be found in an earlier stage in the development process.

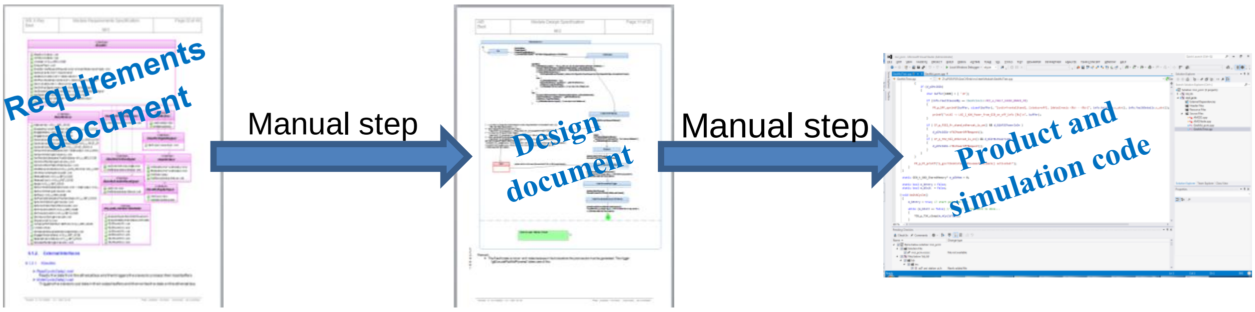


Figure 13 Transforming abstract requirements into executable code

Using IBM Rhapsody as a modelling tool can potentially reduce the number of bugs and also speed up the development. As the picture below shows, the model in IBM Rhapsody is the single source for requirements/design documentation and product/simulation code. This process does not contain manual steps that can introduce such “translation” bugs. The types of models in IBM Rhapsody are mainly object-oriented designs (and state diagrams) solely used within software development.

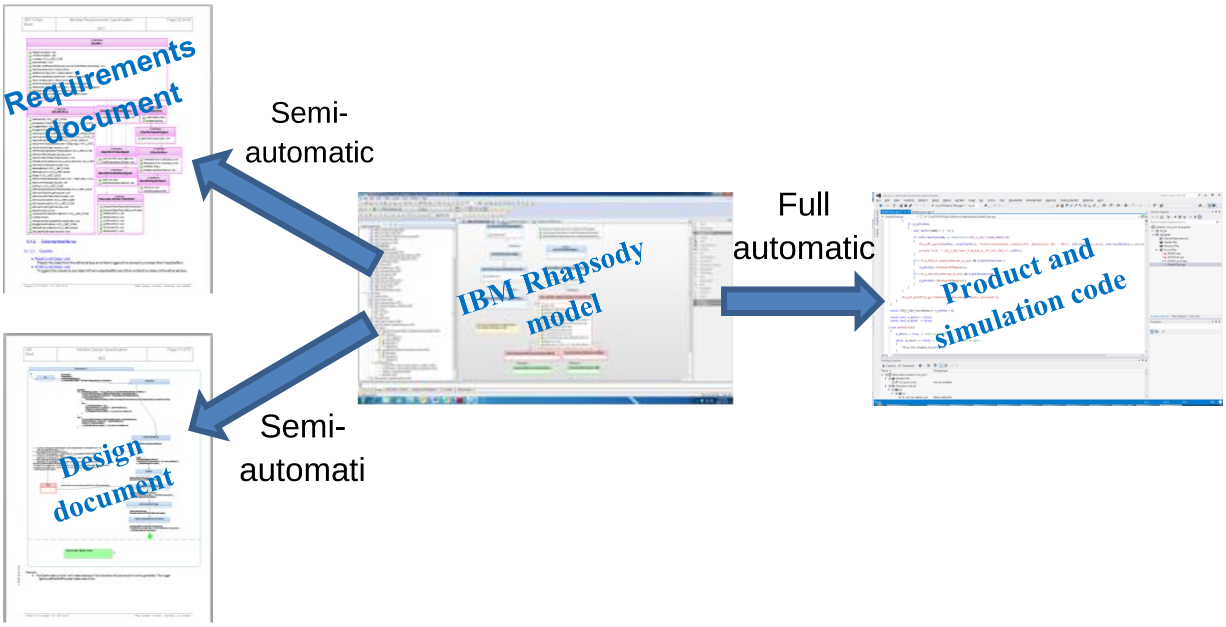


Figure 14 Transforming models into abstract requirements and executable code

During this activity we used IBM Rhapsody for one new module called MCI (big part of “single axes and IO”). The main reason why this layer of software was chosen is because all engineering steps were needed: i.e. requirements, design, implementation and test. Besides that the module can very well be design in an object-oriented way, the size of the module is big enough to really get representative feedback.

For this module we created a total of 26 object model diagrams in IBM Rhapsody, all with multiple classes in them. Figure 15 shows the complexity of a representative objects models diagram. Together with the sequence (see inlay), deployment and component diagrams they constitute the basis for generating over 150 source code-files (> 30kloc).

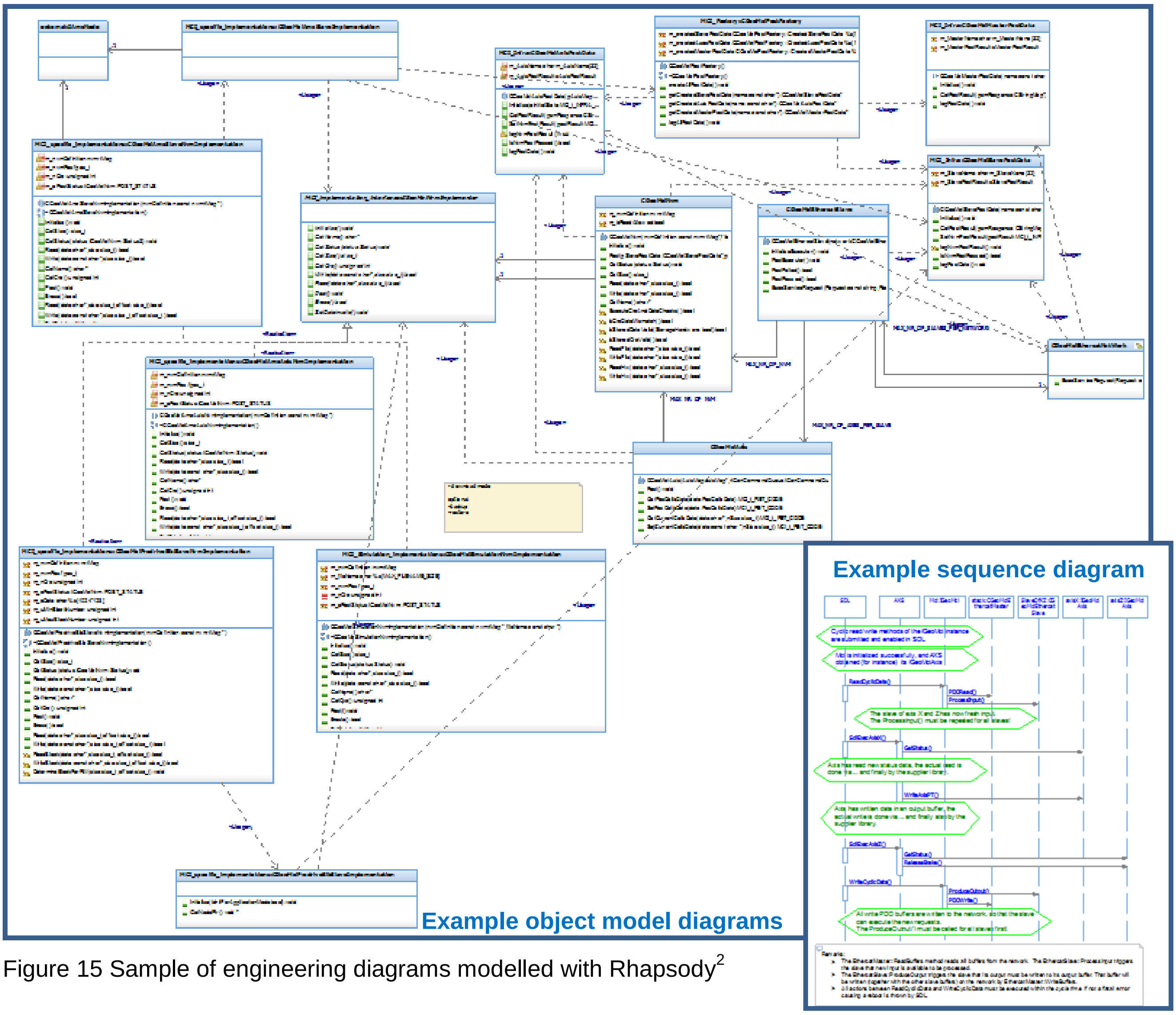
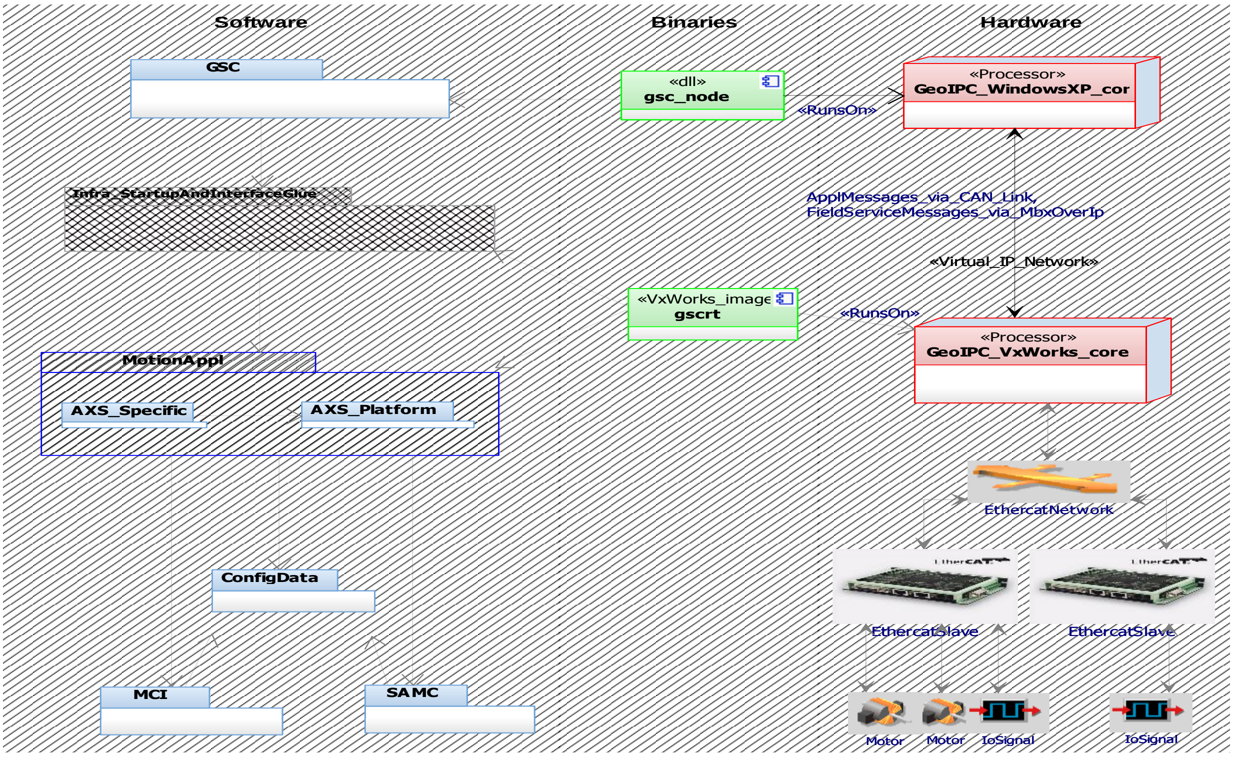


Figure 15 Sample of engineering diagrams modelled with Rhapsody[2]

During the full development cycle from requirements until first systems in the field, IBM Rhapsody has been used. During this development, software designers worked in parallel on this same module, similar as they would do in traditional software development. The IBM Rhapsody models are archived in IBM ClearCase linked to IBM ClearQuest. The file structure of the IBM Rhapsody files was chosen in such a way that enabled parallel working. During each product build from the IBM Rhapsody models code is generated from which the software product is compiled.

[2] The engineering diagram is intentionally blurred



The Rhapsody pilot revealed the following:

- The methodology behind IBM Rhapsody is good for reusing the model towards requirement, design and implementation. It provides an increase in overview of your object-oriented design and its dependencies. A drawback of it is the limited level of abstraction of this methodology. It only brings a level of abstraction by visualizing your object-oriented design, but all detailed coding of methods still needs to be done. It also means that the models can only be used by software. People that don't have a background in software engineering have issues interpreting object-oriented models expressed in UML.
- The tool has a steep learning curve, making it hard for new people to join such software development. Several issues were found with the tool itself, limiting the speed of development. Together with IBM we looked at the issues, where IBM concluded that some of these issues could be solved by changing settings in the tool (see planned future work). Having a detailed overview can be hard in IBM Rhapsody. This for instance is caused by the way the manually created code is embedded in the overview.
- In comparison to other newly traditionally developed modules, there was no noticeable difference in the number of product defects (bugs). Neither did we see any difference in development speed. Because of the issues in the tool, the development speed was even much less in several moments in time.

Weighting the pros and cons, in its current state it isn't perceived a game-changing breakthrough technology.

Although we believe that IBM Rhapsody can eventually bring some increase of development speed, the abstraction level is too small to really bring a major quality or development speed advantage. To benefit more from improvement on quality or development speed towards the future we will look more into the more abstract ways of modelling. Since more abstract models can more easily be understood by non-software people, the amount of reuse for these more abstract models is believed to be bigger as well.

Current status

The current status is that IBM Rhapsody output is part of a product. For now we will keep on using Rhapsody for developing this single model, but we will not extend the usage of Rhapsody.

Future work

Besides that IBM Rhapsody will be used for further development of MCI (the module described in this activity), we will together with IBM further check the issues found.

3.2.2 Activity A2: IBM Rational Team Concert (RTC) pilot

Description of work

The goal of this pilot is to determine whether Rational Team Concert (RTC), as part of the Jazz platform of IBM (which is based on OSLC), is able to solve the workflow and engineering requirements and defined TAPB1 decision fit with the needs of the local R&D organization. Several pilots have been run in different R&D teams, as depicted in Table 2.

For each pilot, the stake holders are:

- Cluster Management team
- Pilot team
- Configuration Management team
- TA-Case (local IT)
- IBM was involved during the course of the pilots and supported technical questions.

The following activities have been executed by the pilots:

Step 1: Preparation phase

- Management agreement
- Define requirements from software perspective
- Define Change Control flow
- Setup archives/streams
- Install RTC clients on development PC's.

Step 2: Execution phase

- Use RTC as main Source/Configuration/Change management tool (Table 3)
- Use RTC as SCRUM planning tool (Table 4)
- Define/adapt practices for tool usage

Step 3: Retrospective phase

- Check fulfilment requirements
- Present results to cluster management team
- Define next steps for pilot team (continue RTC / fall back to CC/CQ)
- Present results within iXR together with Configuration Management team & other pilot teams

Table 3 and Table 4 provide an overview of the requirements posed to RTC while doing the evaluation. The outcome differs per R&D cluster, is summarized below:

R&D team	Team 1	Team 2	Team 3
Outcome	Pilot successful New projects within this cluster will start with RTC	Pilot successful Migration will be initiated within the running project once the Release-For-Verification milestone has been passed.	Pilot successful New projects within this cluster will start with RTC

Table 2 R&D teams with IBM Rational Team Concert pilots

Observed shortcoming:

1. On Merge conflicts:
   User interaction is required for all merged files, even the trivial ones. Also, integration issues were found when using an alternative 3rd party compare editor (e.g. Beyond Compare).
2. On Revert changes:
   While the file was already merged, RTC doesn’t know which one of the 2 previous files it should roll-back to. Therefore it asks the user to open the file's history view and select the file version to which it needs to be reverted.
3. On multiple history panels:
   When you open the history of a baseline or file, you'll loose the history panel of the previous item you were checking. This is unwanted behaviour as one should be able to show a new history panel.
4. On locking streams and/or deliverables:
   As workaround one can go for the option of not accepting all change sets, or one can change the ownership of the Stream to a Project-/Team Area with no members, but this is not really elegant.
5. In combination with Microsoft - Visual Studio:
   The 'Pending Changes' panel does sometimes not show Unresolved items automatically. Also, when deleting a pending patch from the Visual Studio client, it re-appears the next time, so you need to use the RTC Eclipse Client to delete it.

Conclusion: the outcome of the pilot is positively received by the different development teams but both the source code archive functionality as the planning functionality of RTC offer opportunities for improvements.

Current status

Alongside ClearCase, RTC has been integrated with ElectricCommander and is now an integral part of the build and test framework used by the Configuration Management Team. On top of the pilot teams, various engineering teams previously working with ClearCase, have switched over to using RTC. Especially the teams used to using more lightweight source control tools (e.g. Subversion, GitHub) are still hesitated in moving over to the more heavyweight RTC.

Future work

Evaluate the impact on the remaining R&D teams to switch over to RTC and in parallel evaluate alternative solutions against the same set of requirements.

	Requirement	MoSCoW	Fulfilment
1	As a developer I want a simpler workflow for my daily work (Check-in, Rebase/Accept) compared to ClearCase.	M	++
2	As a developer I want to be able to show the history of a configuration item in a date-sorted list.	S	++
3	As a team we want our build triggered by a source code delivery.	M	+
4	As a developer I want the performance of the RTC tool to be at least equal to the ClearCase tool for daily work (check-in, Rebase/Accept)	M	-+
5	As a developer I want integration between RTC and my development environment so that I can perform main tasks directly from that environment (Visual Studio, Windows Explorer)	S	-+
6	As a team I want a dashboard that can show all critical identifiers of the project and product during development in a user friendly way.	C	+
7	As a developer I want to have an “easy-to-use” configuration management tool	M	+
8	As a developer I’d like to have a merge tool that automatically merges trivial conflicts.	M	+
9	As a developer I want to be able to rename/move configuration items but keeping its history.	M	-+
10	As a team we would like to have an Issue tracking system to manage our issues found during development	M	++
11	As a developer I want to be able to work from home without restrictions.	M	++
12	As an organization I want distributed development teams to work together in a single archive with accurate performance.	M	Not tested
Very poor: -- Poor: - Neutral: -+ Good: + Very good: ++			

Table 3 RTC fulfilment score on engineering requirements posed towards a source code archive

	Requirement	MoSCoW	Fulfilment
1	As a scrum master I want an easy to use tool to schedule releases and sprints	M	-+
2	As a product owner I want to have a tool which helps me maintaining my release and sprint backlogs including prioritization.	M	+
3	As a team we want to visualize the status of our current sprint in a burn down chart.	M	-
4	As a team we want a dashboard that shows all critical project aspects in a single view.	M	-
5	As a team we want to subdivide user stories in smaller user stories or tasks and move stories to next sprint.	M	-+
6	As a team we want to have a task board which can be used during the stand-up to update the tasks of the current sprint.	M	+
7	As a team we want to have trend information on our velocity.	S	-+
8	As a team we want to have a release burn-down/burn-up which shows the added value of the current release.	M	-+
Very poor: -- Poor: - Neutral: -+ Good: + Very good: ++			

Table 4 RTC fulfilment score on engineering requirements posed to a SCRUM planning tool

3.2.3 Activity A3: Evaluating organizational needs & potentials GAZEBO / OROCOS

Description of work

TU/e possesses valuable knowledge on simulation due to the efforts of its Robotics research team, namely on two tools that can provide good contributions to this project, Gazebo and Orocos. There is a common interest in the research and study of these tools, between the Crystal Project, the @Home Service Robot team and the Middle Size League robotics soccer team, where the last two are part of the Robotics Research Team. The stake holders for this activity are TU/e and Philips.

Gazebo (connection to B4.10) is a simulation engine for physical objects moving through 3D space, used for simulation of groups of robots. The benefits of the use of this tool range from realistic physics simulation, sensor simulation and graphical rendering engine, all interchangeable due to its plugin structure

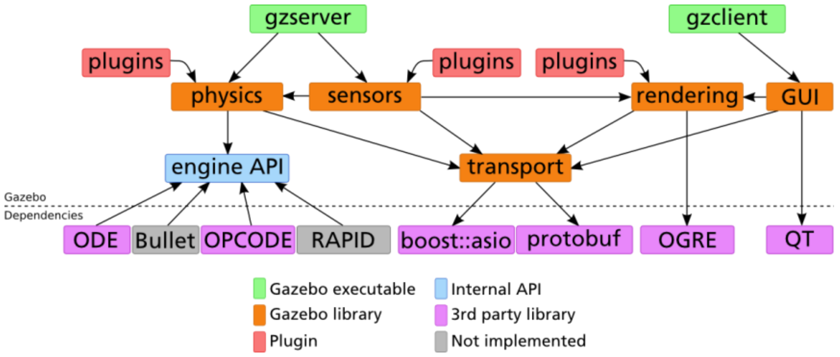


Figure 16 Overview of the Gazebo architecture

Orocos (connection to B4.11) is a tool used to create real-time robotics applications, with a component based structure. It consists of a set of libraries, a run-time system, and tools that enable faster development and higher quality of models. The use of these tools in the Crystal Project, more specifically in WP 4.3 for HiL simulation, enable realistic real-time simulations, a powerful rendering engine for improved visualization and graphical user interfaces in the form of plugins that can be used to control the simulation, for example performing manual fault injection.

Current status

Concrete applications of the tools Gazebo and Orocos are dependent on several factors, such as the software ecosystem used in the X-Ray system, and the implementation structure of the Simulation Layer. Since some implementation details are yet to be defined due to the on-going development, the exploration of these tools has been made in a different environment, namely on the @Home service robot.

Gazebo has been used for simulation of physical objects in virtual scenarios, enabling the testing of different navigation algorithms in a way that is safe for the robot. Orocos is used for the creation of a component-based structure, enabling real-time operation of controllers using its Real-Time toolkit.

The connection of these two tools requires a specific middleware, ROS (http://www.ros.org/). Using this middleware on the current software environment of Philips is not an option since it does not support real-time operating systems such as VxWorks, therefore a plugin is being developed by other researchers to perform this connection directly (https://github.com/jhu-lcsr/rtt_gazebo). A study is being made on the viability of using this plugin to connect the two tools since right now it is still in an early stage and unreliable.

Future work

In the current Simulation Layer, Matlab Simulink is used to perform code generation using models provided by TNO on some of the components of the X-Ray System (connection to Activity A5). It is important to study which tool, Matlab or Orocos, provides the best performance and accuracy when simulating a large number of machine components, such as motors, sensors, etc, in the current scenario, and whether the benefits out weight the use of Matlab at Philips.

As for the visualization of these results, further study should be made on whether it is necessary to use the complete suite of tools offered by Gazebo, or just the rendering engine Ogre, as other work-packages (connection to Engineering method: Verify SW Architecture Design) already use this rendering engine incorporated in the Xposer tool. Furthermore the study on the plugin to connect the two tools is going to be continued to determine if this connection is indeed possible and reliable.

3.2.4 Activity A4: Evaluation and considerations related to the levels of integration

Description of work

In order to reduce the need for testing on the real system and to reduce the risk of issues during these tests, model-based testing of control software could benefit Philips' software development process (see use-case description). Since this was not the case yet, people from Philips, ESI and TNO have had several discussions on how this process could look like in the use-case context. Furthermore, the place of the existing tooling (POOSL modelling by ESI, Matlab modelling by TNO) in this development process needed to be defined.

The stake holders for this activity are Philips, TNO, and ESI.

Several open and creative discussions were held to come to a common understanding. Re-use of Models for mechatronics control software was rather unknown at Philips. In the automotive and aerospace domain, this is however a common development process. Using the available experience at TNO from the automotive domain, different integration levels for model-based software development were introduced, as is depicted in the figure below.

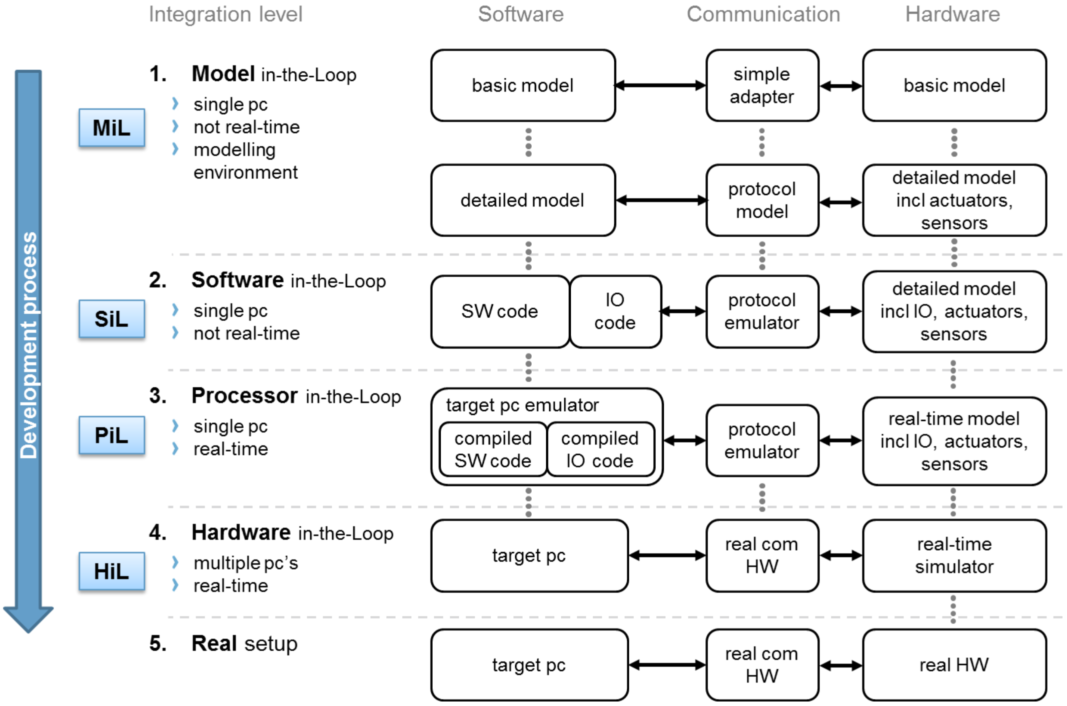


Figure 17 Integration levels for model-based software development

The left part of this figure shortly summarizes the characteristics of the integration levels. The right part of the figure shows a schematic view on the simulation scheme for every integration level. During this process different models of both software (e.g. using POOSL) and hardware (e.g. using Matlab) are used throughout the process. This shows that, as the simulation model of the software evolves towards the real software code, the hardware simulation model also evolves towards more realistic and complete model.

The idea is that the risk of issues when testing on the real machine will be minimized by applying model-based testing on different integration levels. Problems can then be detected much earlier in the development process. This will facilitate a 'first-time-right' approach. For a description of the levels, please see Appendix D, section 2.3.

Current status

This activity is finished.

Future work

The concepts are continued via the Engineering Method *UC403_TestWithInTheLoopSimulation_001.*

3.2.5 Activity A5: The Matlab Modelling for the table force sensor.

Description of work

In order to demonstrate a model-based software development process, a technical case study was introduced called 'Table force sensor'. Here, the hardware considered is one of the tables, including a force sensor. Currently, no dynamic model of such table is available, which can be used for a model-in-the-loop or software-in-the-loop demonstrations. Therefore a basic model of this table is developed, containing the relevant dynamics for this case study. Furthermore this model will be set up in alignment with the work done in WP606 and WP611, so it can also be used as a demonstrator for these developments.

The stake holders for this activity are Philips, TNO, and TU/e.

For this first case study, a dynamic model of the vertical motion of the table is needed, including the output of the force sensor. Using several design documents of this table, a simple model of the table mechanics regarding the vertical motion has been set up. This model receives height set points from the control software and outputs the 'real' table height. Furthermore it contains a model of the force sensor, which outputs a measured force. Figure 18 shows the top-level of the model.

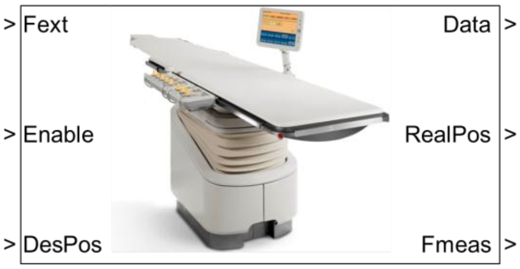


Figure 18 Abstract model

The validity of the model is shown by measuring inputs given to the real table and replaying them on the simulated table. The resulting behaviour of the simulated table in terms of position, velocity and motor current is then compared to the behaviour of the real table. Most parameter values are taken from the design documents, while the friction in the simulated power-train is tuned to match the measured behaviour.

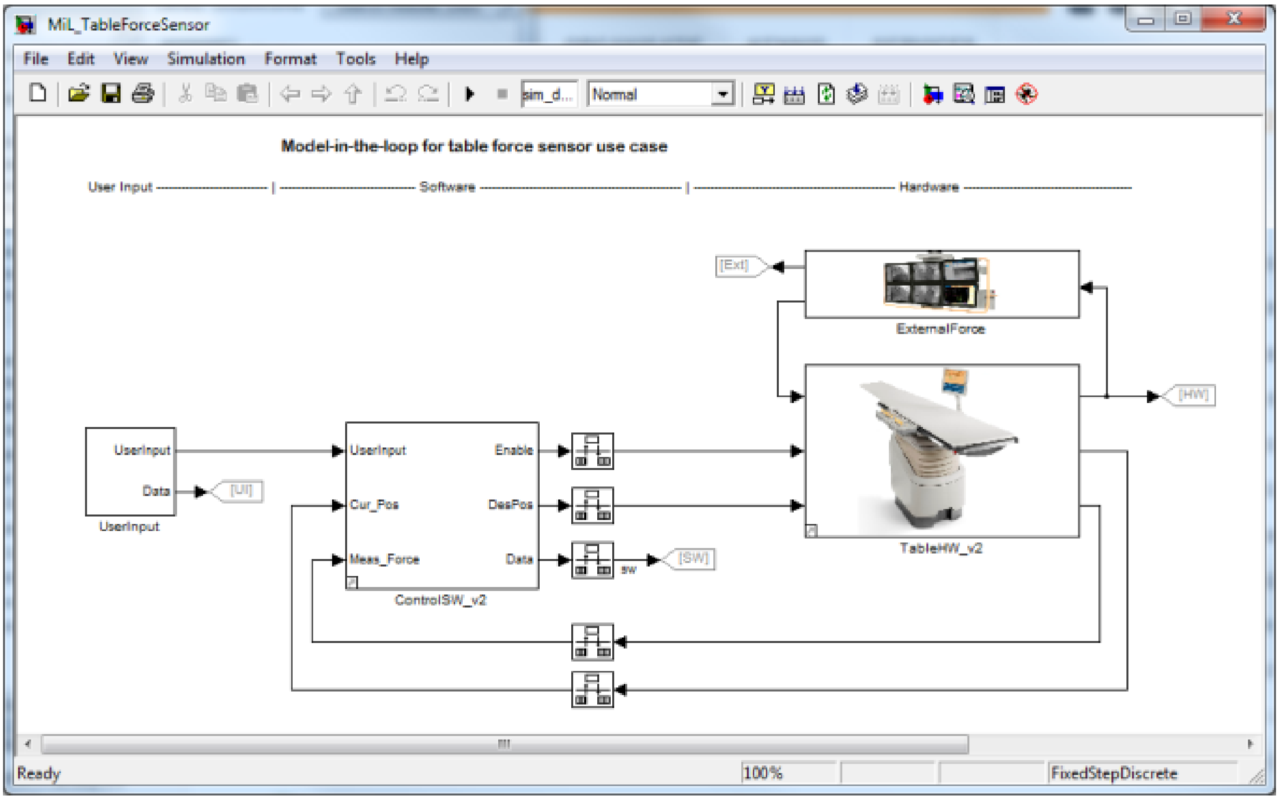


Figure 19 Matlab Modelling of table force sensor

To demonstrate a Model in-the-Loop test, additional models are needed, as is shown in Figure 19:

- Model the environmental influences.

This part will represent the external force acting on the table, for example a person pushing on the table, or a collision force. This model, located above the table model, closely interacts with the mechatronic hardware model of the table.

- Model the control software.
  This part is needed to control the (virtual) table, and is located to the left of the table model. Currently, this model is derived from the specification documentation of the control software. It can be seen as a first model variant of the desired software.
- Model the user interactions
  A simple model is used to simulate user input given to the control software, indicated by the leftmost component in the figure.

The developed models are used to simulate the procedure ‘Override.UserAction’, which is used as proof of concept for the use of models and simulations early in the engineering workflow. The system switches to override mode when a clinical user re-asserts a movement that has been stopped due to collision detection.

Current status

Together with the TU/e (Mechanical Engineering), a software-in-the-loop demo is prepared. Here, the developed table model will be connected to the real control software in a (semi) real-time environment.

Future work

Expand simulator complexity and include more hardware with multiple degrees-of-freedom and sensors, and include models representing the clinical context of the products’ intended use.

3.2.6 Activity A6: The definition of a simulation architecture and integration plan

Description of work

To support the use of models in the suggested Software-, Processor-, and Hardware in-the-Loop integration levels from activity A4, a simulation architecture is required where the different models can be connected to the software that is used to control the target product. The main purpose of the simulation architecture is to divide the target product into two parts, one part consists of artefacts that will appear in the final product (like software, target PCs and communication infrastructure), while the other part consists of artefacts that are used during the engineering process (models, simulators, emulators).

The stake holders for this activity are Philips, TNO, and TU/e.

On the boundary between these two parts, an interface exists that connects the different artefacts.

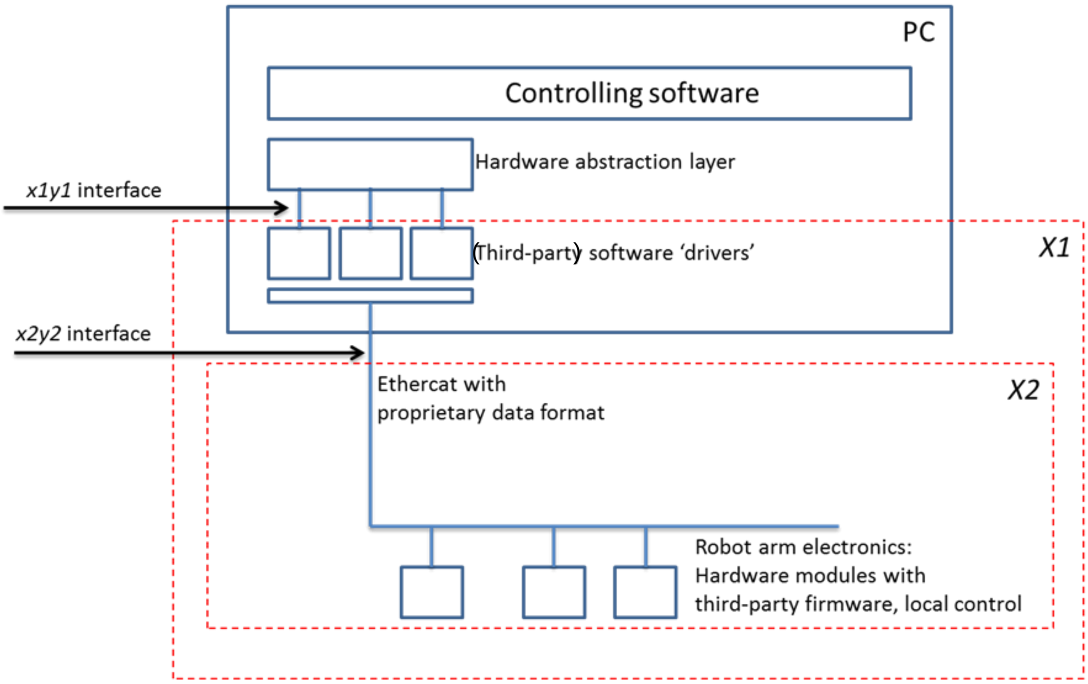


Figure 20 Alternative locations for interface cuts

In Figure 20, a schematic representation of 'X-in-the-loop' simulation is presented, where two possible cuts are shown in the form of parts X1 and X2. For part X2, the interface between the target software and the simulation (x2y2) would be the EtherCAT communication. Although that interface would be very precise and well specified (based on the EtherCAT standard), it has a number of drawbacks:

- The simulation software would have to be executed on a separate PC with the proper EtherCAT interface. For software developers, it would require additional hardware.
- It would have to simulate the behaviour of the actual hardware very precisely, in order to behave as expected by the third-party software drivers.
- The third-party software drivers could use proprietary packet payloads, which would have to be processed by the simulation software.
- The timing requirements for an EtherCAT slave are very strict, which would result in strict real-time requirements for all simulations.

For part X1, the interface x1y1 would consist of the API that is provided by the (third-party) software drivers of the EtherCAT hardware (and possibly other hardware).

Although these APIs are non-standard, which is a serious disadvantage, using that interface for connecting to a simulation has a number of advantages:

- The actual EtherCAT hardware is no longer required by software developers.
- The API is also the boundary between local software and third-party software, such that all local software can be tested based on simulations.
- When the simulation provides the same API, there is no longer a platform dependency due to the software driver implementation. It would be possible to test the functional correctness on a different platform.

Based on interface x1y1, a simulation architecture is required where the different models can be integrated, like the table force sensor model from activity A5. Based on the level of details of the model, the architecture can be used for SiL, PiL or HiL simulations.

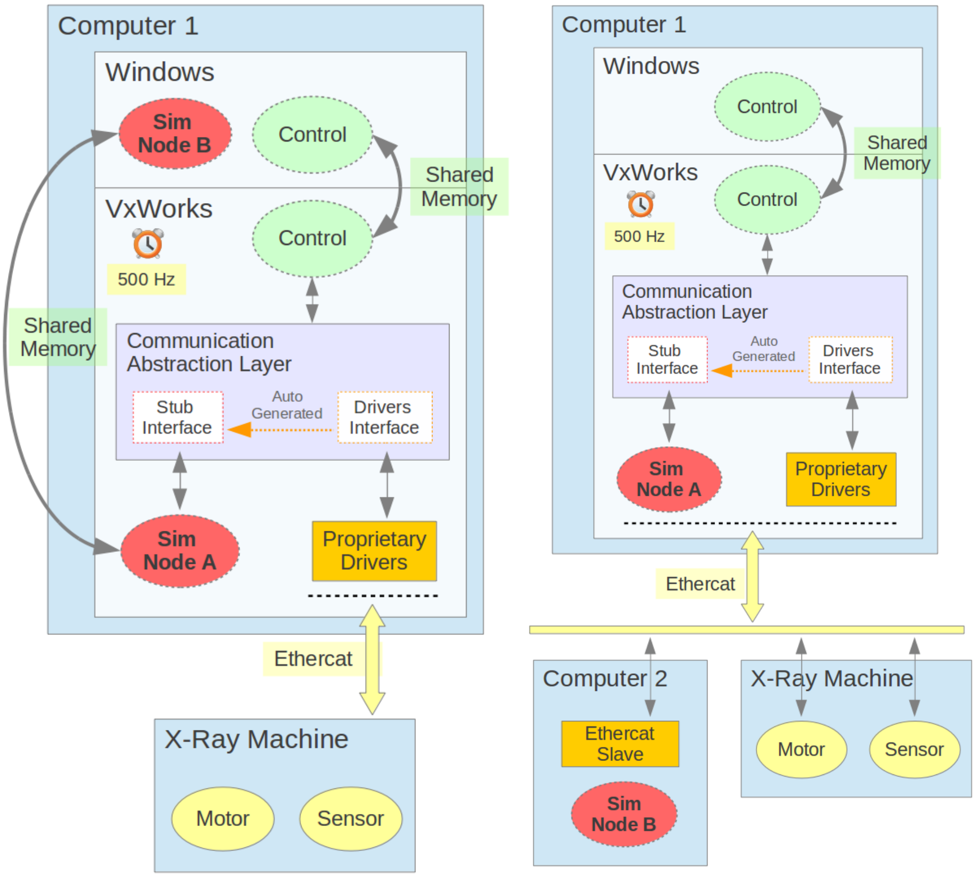


Figure 21 Alternative configurations; left: hosted at the target platform and right: hosted externally

The simulation architecture will include a Communication Abstraction Layer that replaces or intercepts the communication with the actual drivers, such that control software can be used unmodified. Based on the selected configuration (selected by a developer, a test script or the continuous integration), the simulation framework would connect different simulation models to the Communicatioin Abstraction Layer, such that certain functionality can be evaluated. Figure 21 shows two possible configurations:

- In the left configuration, the target control software communicates with a simulation module running under VxWorks (Sim Node A), which in turn communicates with a simulation module that runs under Windows (Sim Node B). For a developer, Sim Node A could simulate the behavior of the actuators and sensors, while Sim Node B could provide a user interface to apply manual inputs or failures.
- In the other configuration, an extensive simulation (Sim Node B) is running on a separate computer, such that the resource load on the target PC (Computer 1) is similar to the resource load on the final product. In this case, Sim Node A would emulate the resource usage of the proprietary driver and pass the information to the second computer through EtherCAT. The second computer can dedicate all its resources to the simulation.

In both cases, the simulation framework would provide the modules to communicate between Sim Node A and Sim Node B, for example based on shared memory, inter-process communication or EtherCAT traffic. In particular, both the control software and the simulation model should be unaware of the deployment infrastructure.

Figure 21 also shows a connection through EtherCAT with the hardware of the X-Ray machine. For SiL and PiL simulation, such a connection would not be required. For HiL simulation, where real-time behaviour is required, the connection to actual hardware would be useful, since it can reduce the complexity of the simulation models. For example, a mixed reality simulation with a real table and a simulated C arc would not require a detailed model of the table. A mixed reality simulation is also useful for testing the behavior of the control software under different hardware failures, which would currently require manual testing.

In Figure 21, the Communication Abstraction Layer is placed within VxWorks, since the communication with the EtherCAT hardware takes place under VxWorks. However, for SiL simulations, the target processor (and VxWorks operating system) would not be required. If all the control software is portable between VxWorks and Windows, it is possible to use the simulation framework and the Communication Abstraction Layer without VxWorks.

In principle, the simulation framework can be independent with respect to the tools that are used to create the model, as long as it is possible to make connections between the I/Os of the models and the EtherCAT elements accessed by the control software. The simulation itself should provide a method to time-wise step through the simulation, such that it can be connected to the real-time control loop.

The simulation framework will create an instance of the simulated hardware based on a configuration file, which would indicate which hardware is real, which hardware is simulated and which simulation model is responsible for which part of the hardware. As long as the models are independent (like the EtherCAT fieldbus, the table or the C arc), the simulation framework should be able to combine the simulations without any modifications. When there are dependencies between models (like a force sensor on a table and different actuators of that table), the framework would not be able to merge the models automatically in a sensible way.

Current status

The success of the suggested simulation architecture depends on the feasibility of the Communication Abstraction Layer, since it is essential in the connection between the control software and the simulation models. Especially when mixed reality simulation is required, the combination of simulation models and access to actual hardware requires a layer that is able to intercept the communication with the drivers of the hardware. The work on the Communication Abstraction Layer is done in activity A7.

The requirements of the SiL, PiL and HiL simulations are translated into requirements for the simulation framework. In particular, the different platforms, use cases, and simulation tools are investigated.

The integration of the simulation framework within the existing workflow is under investigation. Currently, the integration is a manual exercise to determine what is required with respect to additional code, conversions and training.

Future work

The simulation framework can make use of different communication modules, depending on the type of simulation and the type of installation. Initially, the simulation model is executed within the context of the Communication Abstraction Layer. When more advanced models become available, the simulation model could be executed in a different process, under a different operation system or on a different PC. The simulation framework should provide modules to enable transparent use of these alternative modes of operation.

As a first exercise, the simulation framework is constructed by hand, although code generation is used where possible. In the future, large parts of the framework should be generated automatically, such that changes in drivers or hardware configurations can be taken into account quickly.

It is expected that the simulation framework would help in the efficiency of the workflow. To evaluate this claim, the current workflow will be compared with the new workflow to determine the actual improvement for a number of usage scenarios. In parallel, Philips and IBM will be integrating the simulation architecture and its artefacts in the overall systems engineering tool chain. This activity will pose new technology constraints to the current simulation architecture.

Although the framework helps in system development and testing, it would be useful when the simulations are accurate enough to provide evidence for the certification process. However, the simulations themselves might have to be certified (or sufficient evidence has to be provided with respect to the accuracy).

3.2.7 Activity A7: Actual implementation of the Communication Abstraction Layer

Description of work

In activity A6, the simulation architecture is defined. One important part of the simulation framework is the Communication Abstraction Layer (CAL), which provides access to the simulation models based on the hardware driver API that is used by the actual control software.

The stake holders for this activity are Philips, TNO, and TU/e.

In the target product, the API is used to control the actual EtherCAT hardware, which might be provided by different third-party suppliers. Since the related software drivers can be closed-source libraries, it is in general not possible to modify the library to intercept all operations in order to attach the simulation models.

Due to product certification and testing effort, modification to the control software or the hardware abstraction layer should be avoided, since differences in implementations would introduce additional testing and could introduce new sources of errors and inconsistencies. Therefore, the CAL will be based on the software engineering Interceptor pattern. However, due to the closed-source third-party software driver and the required transparency towards the end-user application, there is no clear location where the Interceptor pattern can be applied directly. To overcome this problem, the interface definitions of the EtherCAT driver are used to generate template code that can be extended with the functionality provided by the CAL.

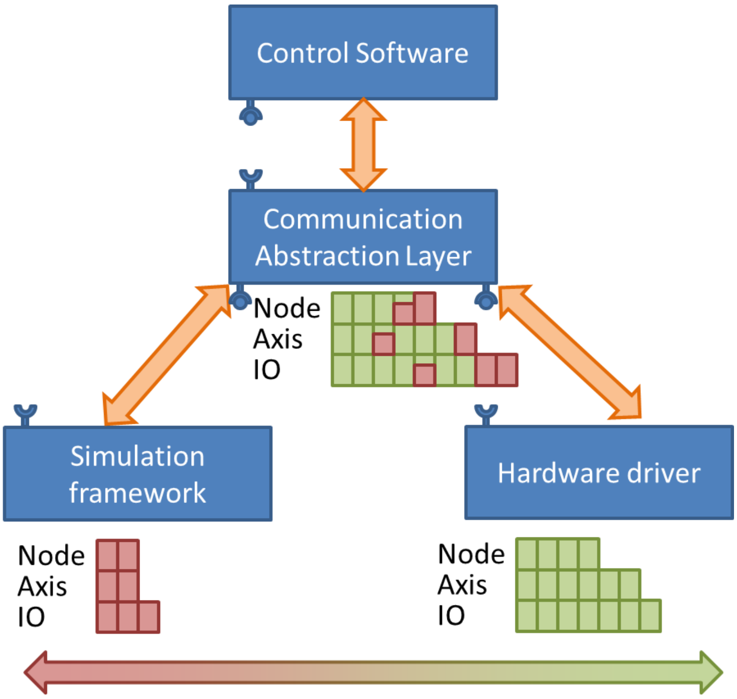


Figure 22 Context of Communication Abstraction Layer

The CAL will be used as indicated in Figure 22. At the top, the *Control Software* expects a hardware driver with a specific interface. There are three elements that provide that interface:

- The *hardware driver* will communicate with the actual EtherCAT hardware. This driver is only available under VxWorks and provides access to objects that are directly linked to hardware elements, such as *Node*, *Axis* and *IO*.
- The *Simulation framework* emulates the behaviour of the hardware driver and provides access to similar objects. The objects are created based on a configuration file and simulation models are attached to these objects, such that the control software can interact with the I/O of the model.
- The *Communication Abstraction Layer* provides the interceptor and mixed reality functionality. Internally, the CAL will use the *hardware driver* or the *simulation framework* (or both) to provide the actual functionality of the hardware (that is, it just ensures that it intercepts all calls). Depending on the configuration, the CAL could monitor the interaction with the hardware driver, combine real hardware elements with simulated elements, or provide access to the simulation framework.

Since all three elements provide the same interface, the control software can remain the same.

The driver for the EtherCAT hardware provides an interface with multiple C++ classes, singleton (factory) objects and call back methods for eventing. In order to intercept all method invocations on those objects, the CAL has to provide separate (wrapper) objects for all references that are passed over the interface. As a result, the CAL introduces some overhead w.r.t. memory and processor usage. The precise overhead depends on the usage scenario.

For simulations that are accurate w.r.t. the performance of the system (that is, resource usage for memory, processor, network, etc), the intercept functionality of the CAL can be used to collect detailed information about the performance of the hardware driver. In addition, the CAL can collect traces of the interactions with the hardware driver, such that performance problems can be analysed.

Current status

For the selected use case, Philips and TNO provided a number of artefacts:

- The interface definitions of several drivers for hardware from different vendors.
- An example implementation of a simple (non-real-time) simulation, which provides the interface of one of the drivers.
- A test application for the interface of that same driver.
- Code generated from the MatLab model constructed within activity A5.

Based on the interface that is used by the example implementation and test application, template code was generated for the CAL. The template code was extended to ensure that all interactions with the original driver are intercepted.

Both the CAL and the example implementation are compiled into separate libraries. The test application is compiled and linked with or without the CAL library, depending on the selected configuration. In both cases, the test application returns the same output (when the output of the CAL library is ignored).

In an additional test, a preliminary version of a simulation framework library is created. When the test application is linked with all three libraries, the output is again the same (when the output of the CAL and simulation framework library is ignored).

Future work

Currently, many steps in the process of constructing the CAL are manual operations, where code has to be adjusted due to better insights in required functionality. Such manual modifications are time consuming and error prone. Furthermore, when the interface of the driver changes (or when new hardware suppliers are used), some of the work has to be repeated. Once it is clear how the code can be generated automatically, the procedure should be automated as much as possible. To improve this process, it might be useful to include additional information in the interface definition files, such that the code generation can be improved.

The CAL is currently based on only one interface of a hardware driver, as provided by a 3rd party vendor, although that is also the most complex interface. To support the complete system, the hardware drivers of other vendors should be supported by CAL as well. To support multiple vendors of EtherCAT hardware, the drivers of those vendors should also be include in the CAL.

Older X-ray machines are based on other communication technologies (like CAN). For those systems, it might be useful to provide a similar CAL.

3.3 Engineering workflow at M12

The figure below highlights the engineering workflow as is the current way of working.

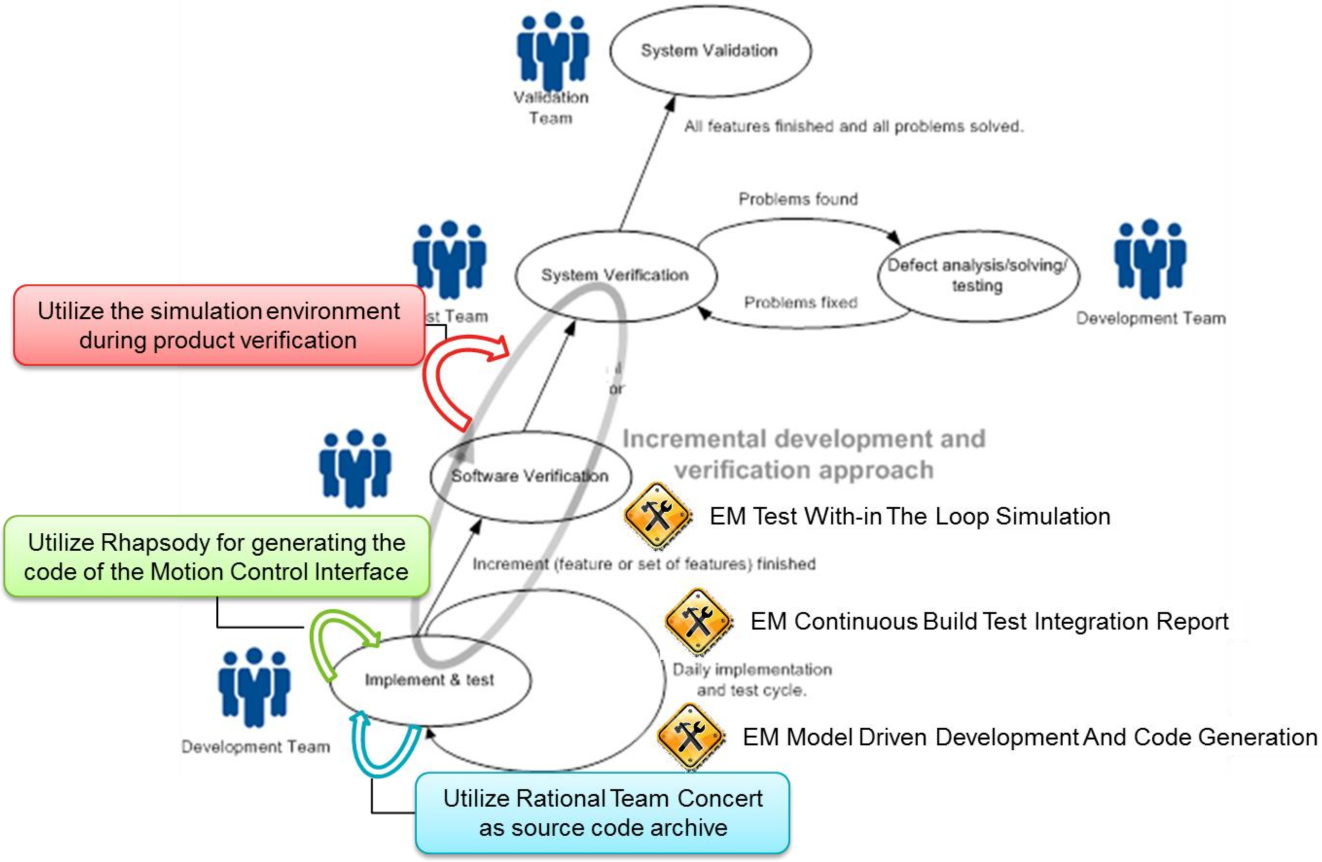

Validation Team
System Validation
All features finished and all problems solved.
Problems found
System Verification
Defect analysis/solving/ testing
Problems fixed
Development Team
Utilize the simulation environment during product verification
st Team
Incremental development and verification approach
Software Verification
EM Test With-in The Loop Simulation
Utilize Rhapsody for generating the code of the Motion Control Interface
Increment (feature or set of features) finished
EM Continuous Build Test Integration Report
Implement & test
Daily implementation and test cycle.
Development Team
EM Model Driven Development And Code Generation
Utilize Rational Team Concert as source code archive

Figure 23 Engineering workflow at M12

The initiatives started have impacted the engineering workflow threefold;

- The Motion Control Interface is now implemented via code generated from object oriented models expressed in UML, with over 30k lines of code.
- Various engineering teams now utilize Rational Team Concert as source code archive, which is a prerequisite for improving the tool interoperability using OSLC.
- The modelling infrastructure improvements allow for more flexibility during product integration and verification.

3.4 Engineering Methods

Engineering Methods provide a technical description of activities and scenarios which make up the overall use case from an end user perspective. They describe the general problem and workflow and the envisioned solutions. The Engineering Methods are defined by the Use Case Owners.

The Use Case is partitioned in the following Engineering Methods:

UC403_ContinuousBuildTestIntegrationReport_001
UC403_TestWithInTheLoopSimulation_001
UC403_ModelDrivenDevelopmentAndCodeGeneration_001

Appendix A Engineering Methods provide a detailed description of the Engineering Methods. More detailed information is available in the “Technical Management” section “Engineering Methods” in the Crystal project archive.

3.5 Envisioned Engineering Workflow

The figure below highlights the engineering workflow as is currently foreseen for M36.

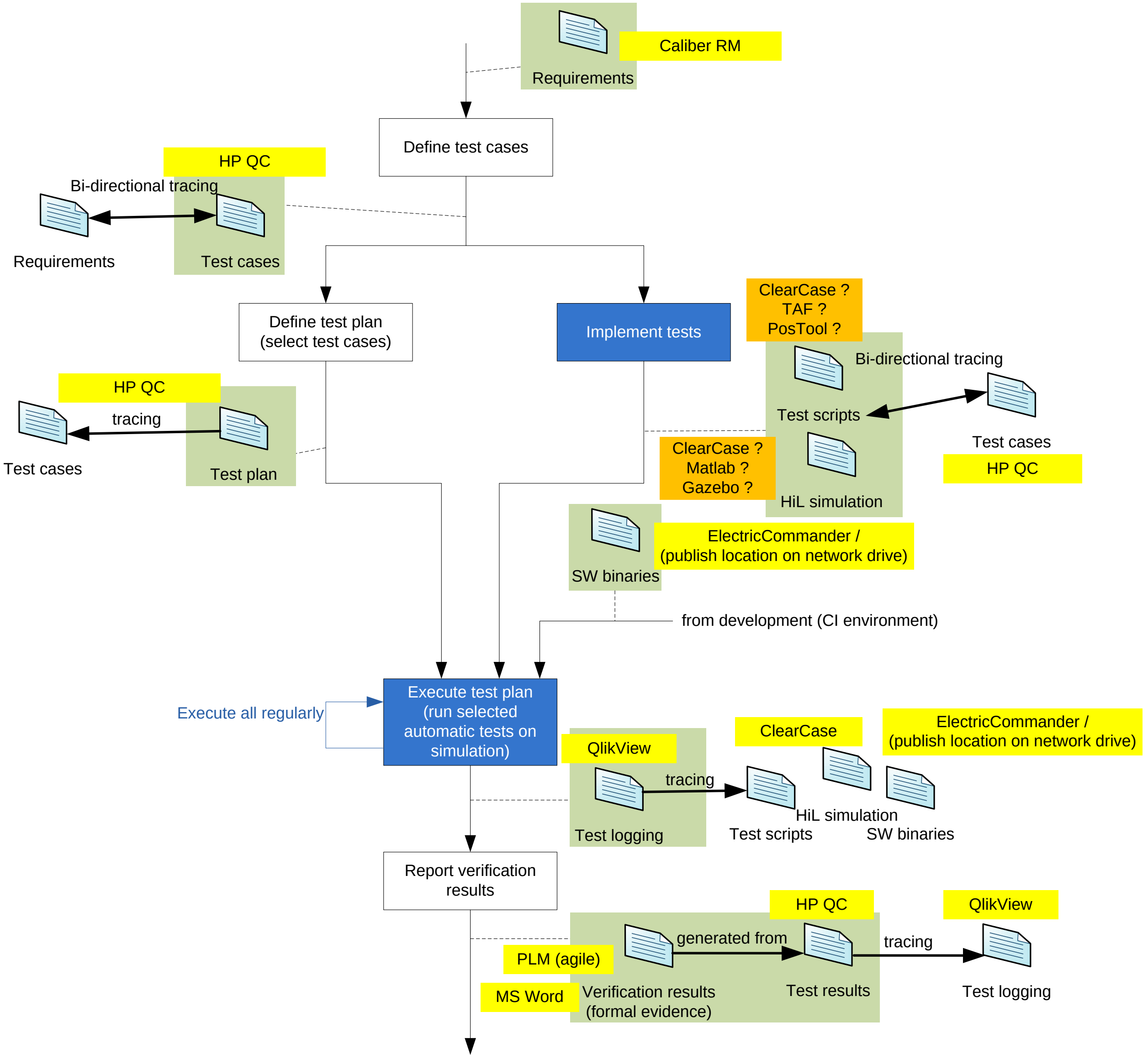


Figure 24 Proposed system verification process

Description of the changed process activities:

- 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).
- 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 Building SEE

4.1 Introduction

The basic view behind a System Engineering Environment supporting Model Driven Developments is based on the following points:

- The selection of the requirements language determines the tools to create the model;
- The output of the modelling activities can be used:
  - in a product;
  - In a visualisation tool to show the contents to the other stakeholders: in a demonstration, 3D/2D presentation of the system and its behaviour, etc.
  - As an input for test cases.
- The actual creation of a model shall be under version/configuration control, by a suited requirements management tool.
- For interoperability purposes the models shall be provided of annotation mechanism.

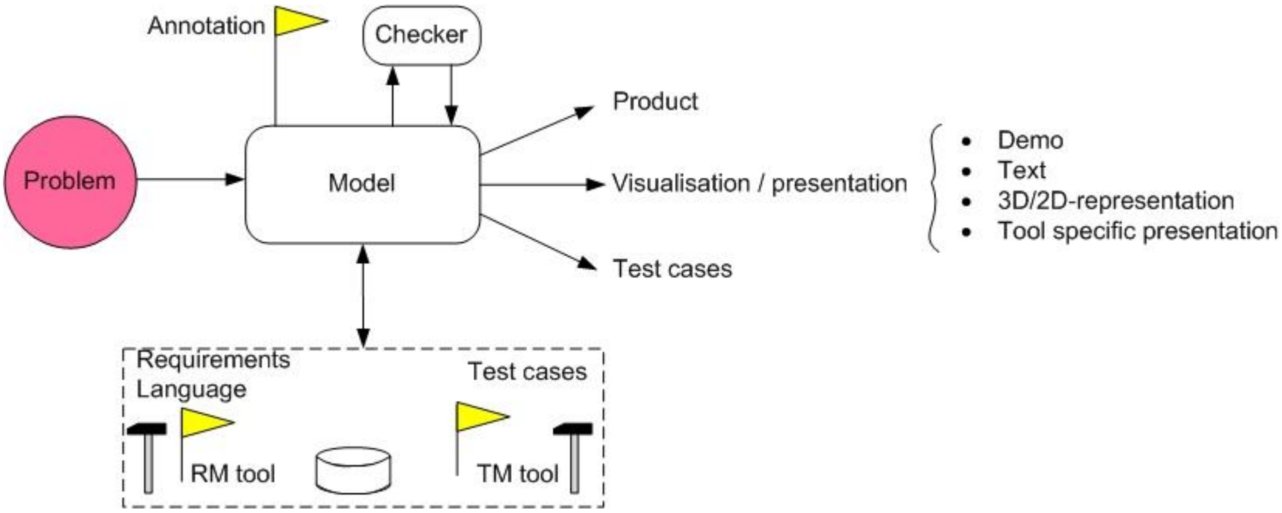


Figure 25: basic concept of a model in its system engineering environment.

4.2 SEE at M0

The System Engineering Environment at M0 shows a number of standalone environments for modelling, simulation and visualisation:

- Simulink and Matlab;
- UML and Rhapsody;
- Dedicated modelling languages and tools for visualisation.

Requirement specifications are mainly based on natural languages, processed in Word, managed in Agile.

Code (generated by means of Visual Studio) is stored and managed in Clearcase, test cases in ClearQuest.

Visualisation of requirements and/or design aspects is performed in dedicated standalone simulation environments.

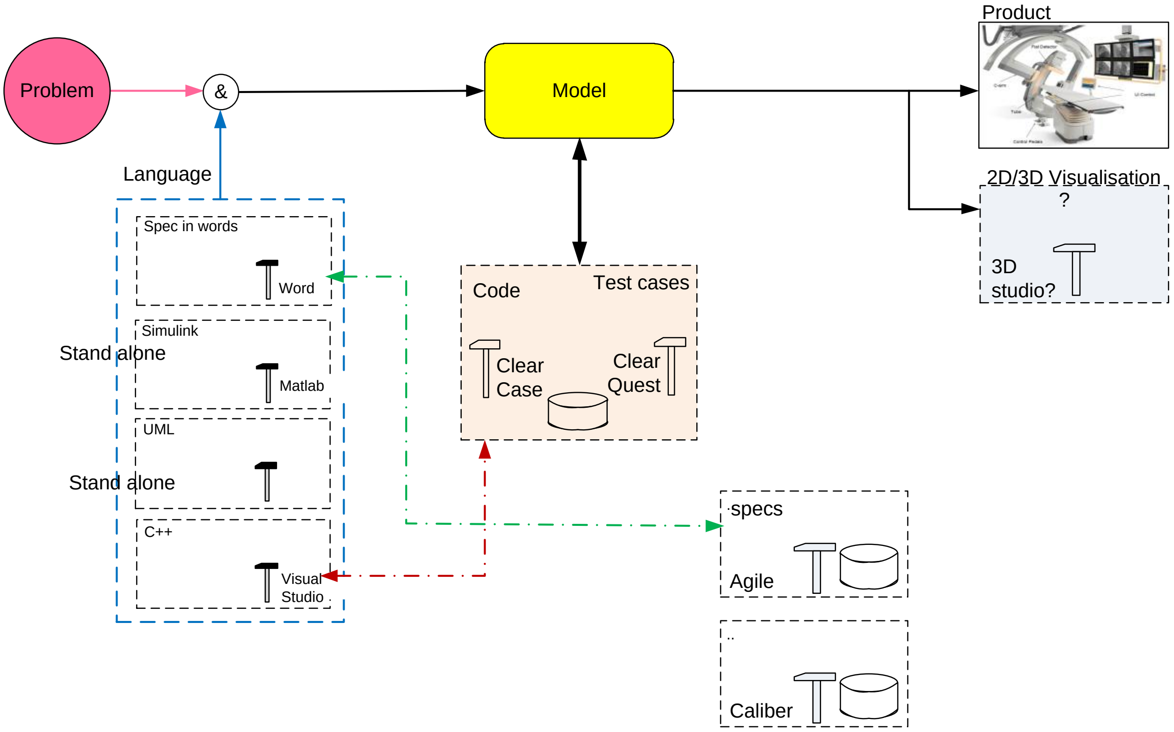


Figure 26: System Engineering Environment at M0 shows a number of standalone environments for modelling, simulation and visualisation.

4.3 SEE at M12

The figure below provides an overview of the Systems Engineering Environment that is currently in place at M12.

The SEE at M12 is characterised by the following points:

- Introduction of Simulink/MatLab (activity A4, A5, A6 & A7), UML/Rhapsody (activity A1), C++/VisualStudio, in an integrated simulation & test environment with ClearCase.
- Experiment with IBM RTC as an environment for continuous build, test and integration (activity A2).
- Experiment with Gazebo / Orocos as tooling for a simulation engine for physical objects and real-time robotics application (activity A3).
- Introduction of Caliber for requirements management in an integrated environment with Agile and Word as an editor for specifications in natural language.
- Experiment with the introduction of DSL with XText/Eclipse visualisation (see D401_901).
- Infrastructure to early visual verification visualize using 3D virtual reality viewer (see D401_901).
- Introduction of Simulink/Matlab models with Xposer visualisation for analysis of design concepts (see D401_901).
- Introduction of an early verification of mechatronics design concepts using Matlab and 3D viewer (see D401_901)
- Experiment with an early verification of software design concepts using POOSL (see D401_901).
- Coupling requirements to verification test cases using HPQC (see D401_901).

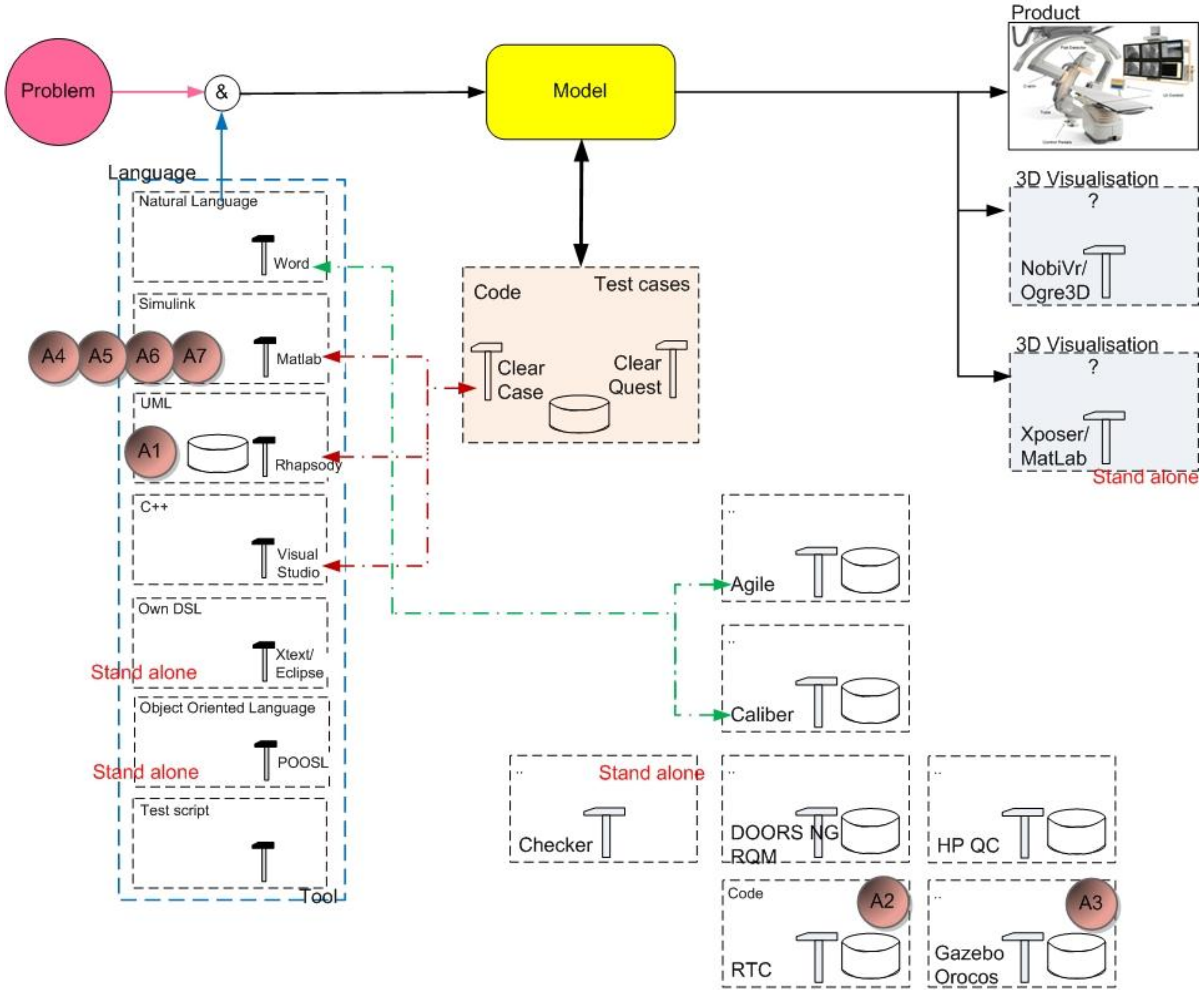


Figure 27 Overview of the Systems Engineering Environment at M12

The core of the simulation environment remains running on a developer PC, as is depicted in Figure 27, but now state information is made available to the outside world using the Communication Abstraction Layer and a visualization interface. This ability is used to bring in the stand/table state visualizer Xposer (see Appendix B, B.1.19). The current setup is real-time only; time is not under simulation control, hence the omission of a solver stub.

The simulation environment is at an infant state as (1) multiple version of Xposer exist e.g. with or without collision detection models, (2) the test environment is build compile time and can’t be arranged dynamically at run-time, and (3) due to the current product design, its rather hard to isolate the model algorithms in it.

Engineering the simulation environment offers an opportunity to apply model based engineering:

- Model scheduling and partitioning capabilities related to the processing of state information intercepted on stubbed interfaces (US2.01 Performance and scheduling analysis)
- Model and simulation of the simulator subassemblies (US2.07 Component-based development)
- Multi-physics models on physical, state, and behaviour (US2.08 Multi-physics modelling and simulation)

Improving the SEE simulation environment will be continued in the M24 time frame.

4.4 Tool chain description

Appendix B Tool chain description provides a short introduction on the individual engineering tools (Brick) as mentioned in the Systems Engineering Environment at M0 and M12 and the associated engineering artefacts they process.

5 Demonstrator descriptions

5.1 Activity A8: M12 Integrated demonstrator WP4.1 + WP4.3

Description of work

This paragraph provides a brief description of the integrated demonstrator prepared for WP4.1 and WP4.3.

The figure below indicates the demonstrator setup and the interaction between the various tools in the tool chain, as described in §4.4.

The demonstrator shows a first approach in integrating effort of WP4.1 and WP4.3 for the current stage of development towards the Use Case objectives.

It covers Component based development, Multi-disciplinary modelling and simulation, Code generation from models, and is an enabler for Real-time behaviour and performance analysis.

The stake holders for this activity are Philips, TNO, and TU/e.

.

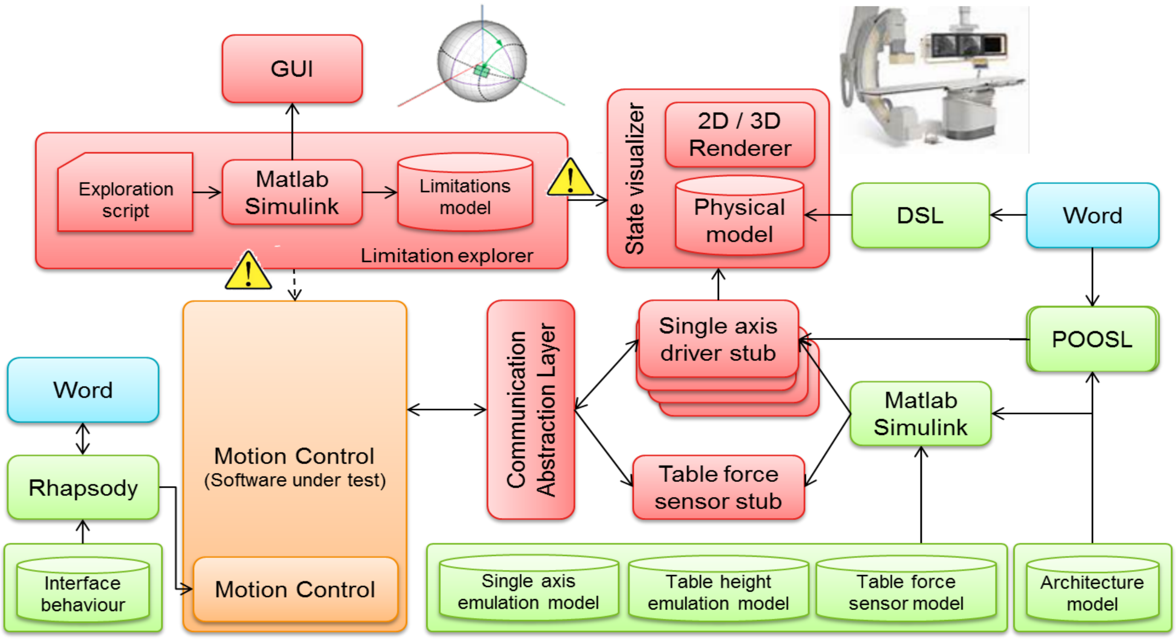


Figure 28 Overview of the demonstrator setup for the integrated demo for WP4.1 and WP4.3

The tools and models in green are used for model-based design and architecture. The tools and models in red are part of the systems simulation environment, while blue refers to elementary engineering tools, as in this case word.

The demonstrator shows the integrated multi-axis patient-oriented movement use case where a speed limitations model is reused and visualized for a predefined angulation and rotation angle of the stand, which constraints this medical use case.

A model-based approach is followed where models from various disciplines are combined and integrated. The demonstrator utilizes two UI layers:

- The interactive part controls the simulation environment
- The state visualizer that provides feedback on the stand/table positions.

Driver stubs emulate interface behaviour of virtualized hardware.

POOSL with Eclipse allows for the rapid prototyping of the motion control architecture. Rhapsody is used for generating parts of the single axis I/O control.

The explanation marks highlight the important integration points between the separated activities as explained in more detail throughout this document.

This demonstrator forms the basis for the future integration and reuse of models in the entire development process. In the M12 review demo, we will highlight more details about the individual development steps.

6 Conclusion and way ahead

6.1 Evaluation

As an answer to an increased design complexity due to higher demands on flexibility in the clinical room layout together with an increased variability triggered by efforts to adapt the same product platform for a broader audience, we have investigated the use of modelling in WP4.3.

At the same time, early verification of system concepts and reuse of modelling effort in the engineering flow is needed for creating acceptable time-to-market for safety critical system engineering products.

In the first twelve months of the CRYSTAL project, we started with using IBM Rational Rhapsody (A1) and MatLab Simulink (A5) as modelling tools for both raising the abstraction level and automatic code generation. Although both tools and mythologies are considered to have added value, we concluded that Rhapsody bring less abstraction compared to MatLab for mechatronics challenges.

Amongst others we investigated the added value of new tools on software configuration management and visualization of simulation models. As the first approach on modelling with MatLab was not connected to the mechatronics software platform, initiatives were started to define a simulation architecture and integration plan (A6). This has led to a first implementation of a Communication Abstraction Layer to be able to integrate the simulation model from MatLab into the mechatronics software platform (A7).

With Activity A8, we integrated a first simple motion control and environment emulator of WP4.3 in the complex use case study of WP4.1. This integrated demonstrator gains the insight that real-time related requirements needs to be added to the interoperability tool specification (IOS).

The outcome of this activity paves the way for using and reusing models during several engineering methods of the development process.

6.2 Planned future work

In the M12-M36 timeframe, we plan to extend the current approach as explained in the document, increasing the amount of tool interoperability for more complex models as depicted in Figure 30. In the figure below, an outlook is given how this would look like. Central in this approach is the extension of the motion control models and environment models with more complex system behaviour as outlined in the use case development report, see Appendix D High level description of use case and context. On top of that the focus will continue to be on integration with the other work packages and real-time aspects of the IOS in close cooperation with WP6 partners.

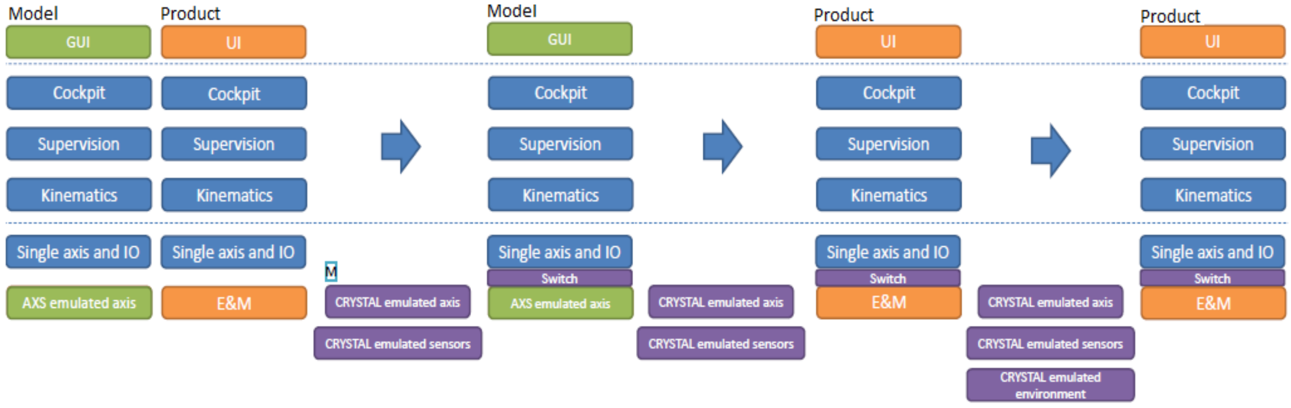


Figure 29: Future work on tool integration and reuse of models

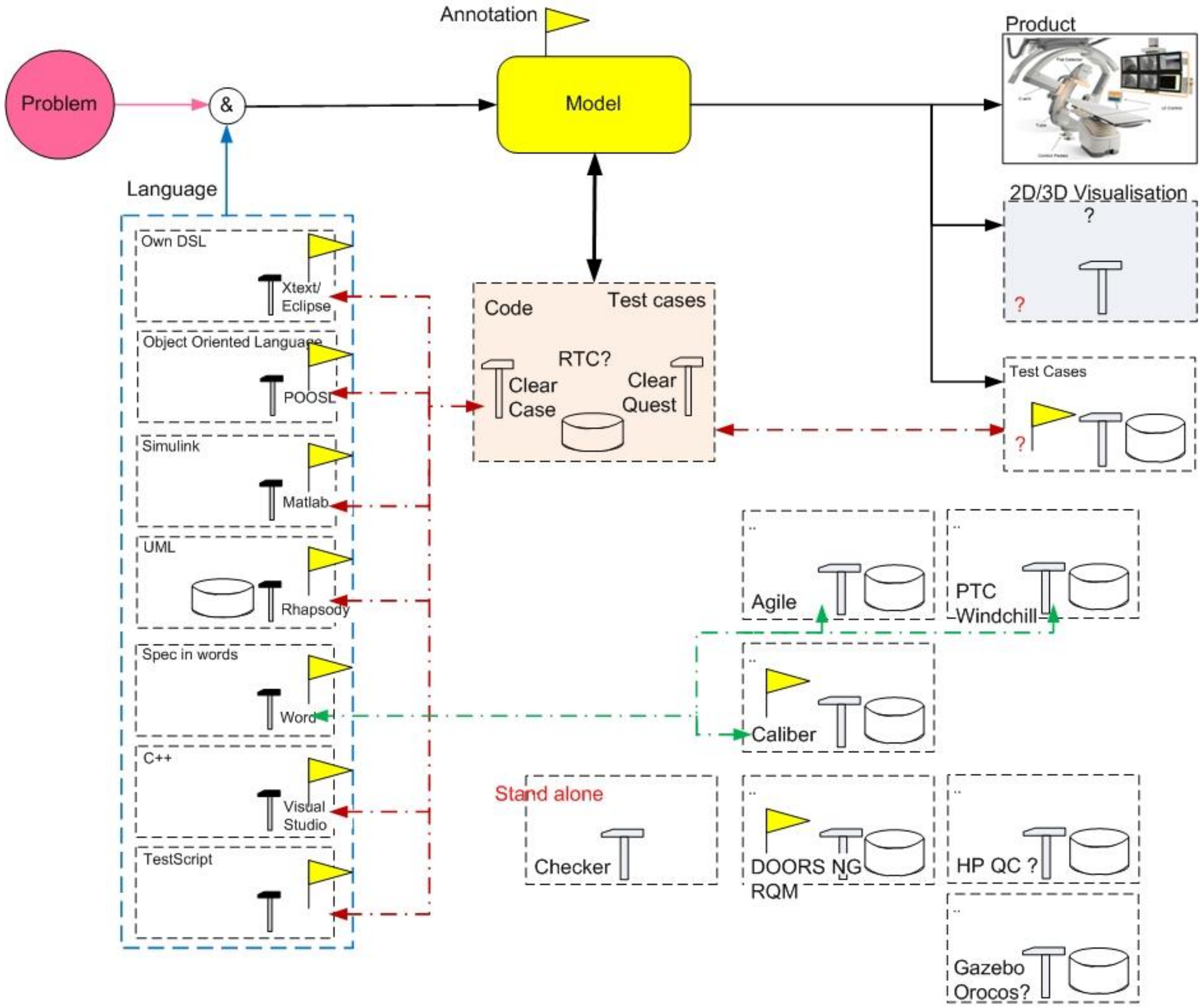


Figure 30 Overview of the future system engineering environment with annotation system.

7 References

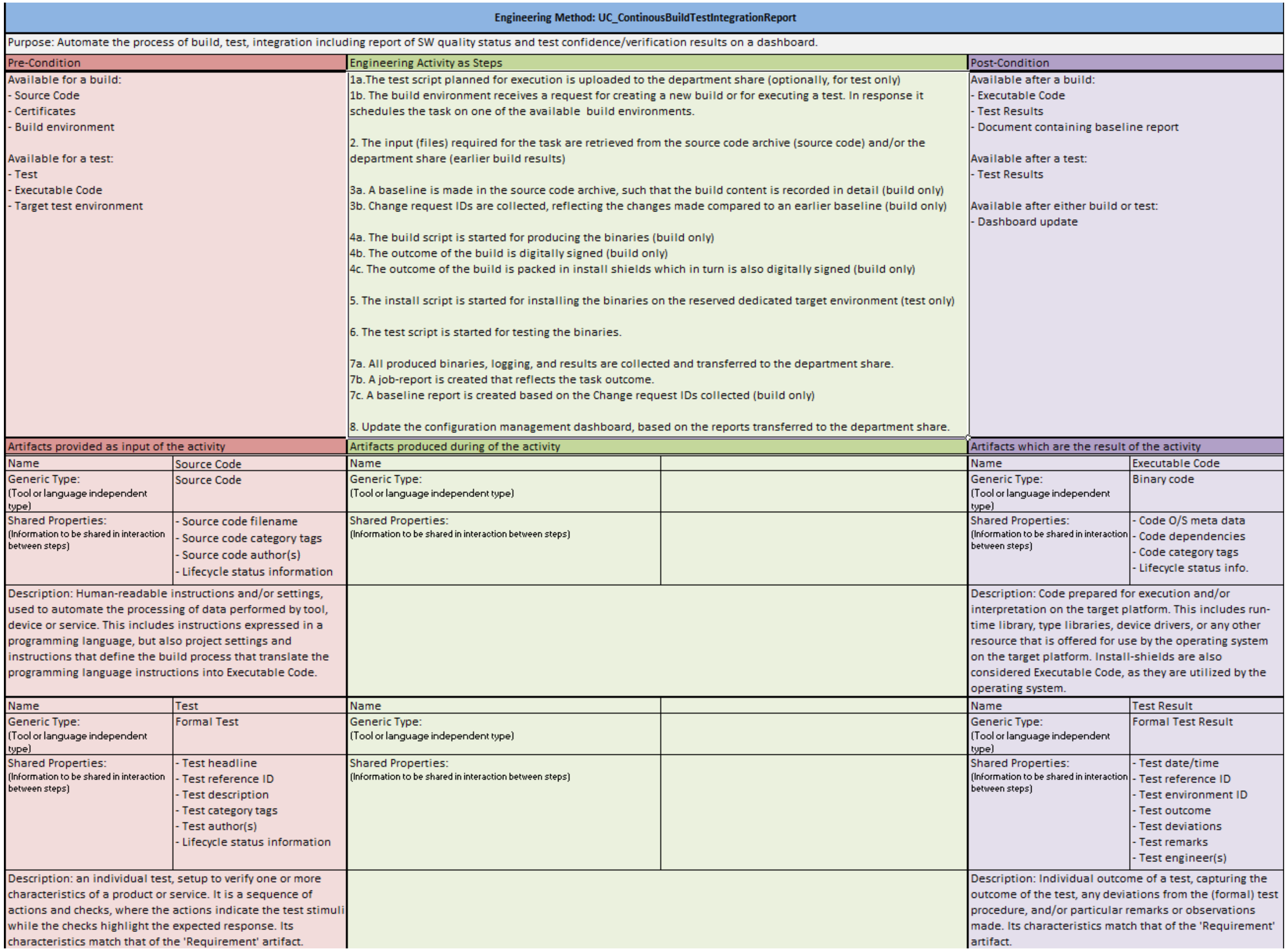
This paragraph provides additional details on the documents and reports referenced within this report. Please recall that this report provides the Executive summary on the description of work and its conclusions on a given activity; supporting reports are available on request.

[1]	CRYSTAL Deliverable D403.010 Use Case Definition Report for Use Case 4.3 Motion control of patient table and X-ray beam positioning. Update of the Use Case Definition can be found in Appendix D
[2]	ARTEMIS Project No. 332830 : Project Proposal Critical System Engineering Acceleration

Table 5 References

Appendix A Engineering Methods

A.1.1 EM UC403_ContinuousBuildTestIntegrationReport_001



Engineering Method: UC_ContinousBuildTestIntegrationReport

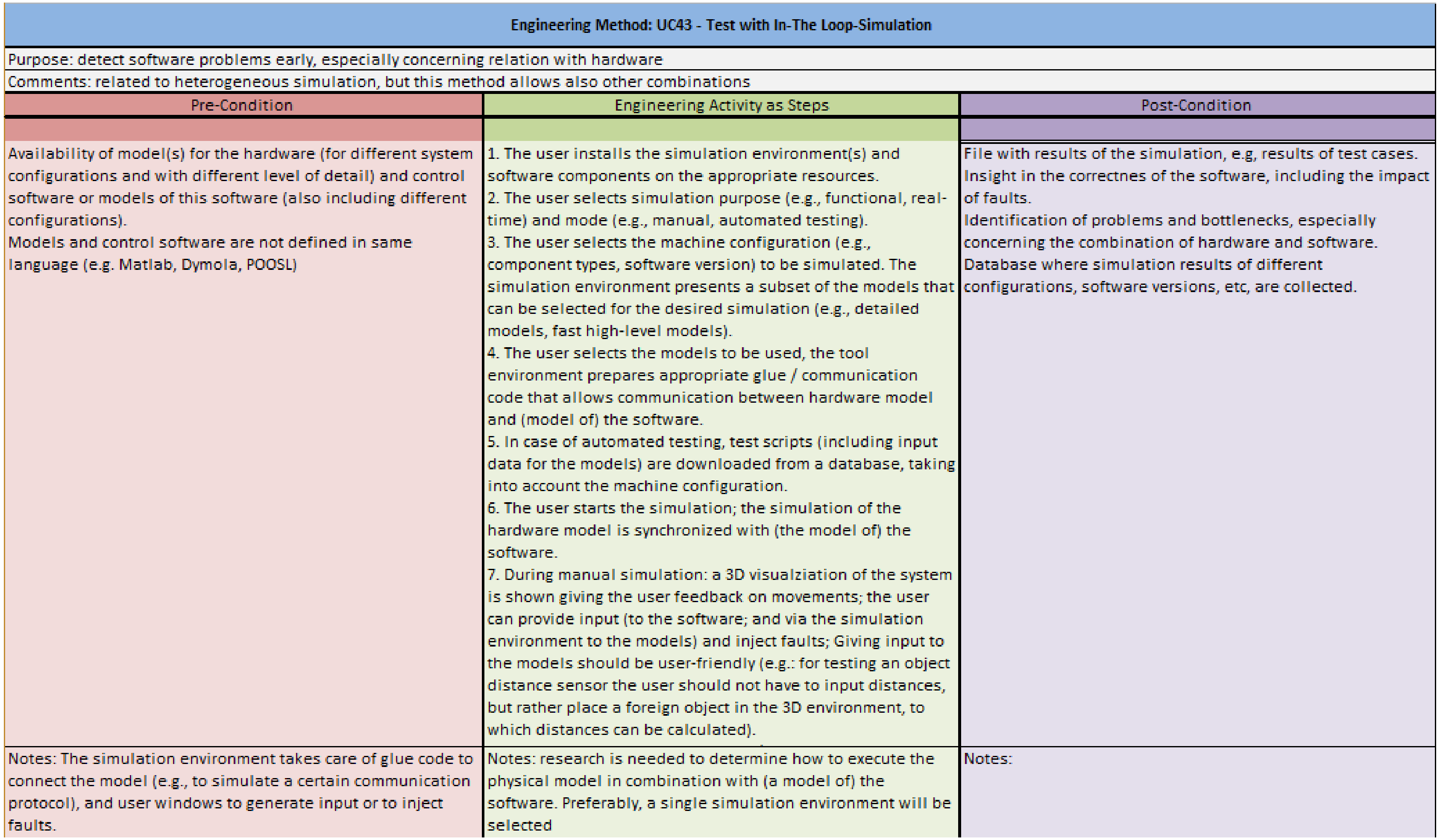
Purpose: Automate the process of build, test, integration including report of SW quality status and test confidence/verification results on a dashboard.

Pre-Condition	Engineering Activity as Steps	Post-Condition
Available for a build: - Source Code - Certificates - Build environment Available for a test: - Test - Executable Code - Target test environment	1a.The test script planned for execution is uploaded to the department share (optionally, for test only) 1b. The build environment receives a request for creating a new build or for executing a test. In response it schedules the task on one of the available build environments. 2. The input (files) required for the task are retrieved from the source code archive (source code) and/or the department share (earlier build results) 3a. A baseline is made in the source code archive, such that the build content is recorded in detail (build only) 3b. Change request IDs are collected, reflecting the changes made compared to an earlier baseline (build only) 4a. The build script is started for producing the binaries (build only) 4b. The outcome of the build is digitally signed (build only) 4c. The outcome of the build is packed in install shields which in turn is also digitally signed (build only) 5. The install script is started for installing the binaries on the reserved dedicated target environment (test only) 6. The test script is started for testing the binaries. 7a. All produced binaries, logging, and results are collected and transferred to the department share. 7b. A job-report is created that reflects the task outcome. 7c. A baseline report is created based on the Change request IDs collected (build only) 8. Update the configuration management dashboard, based on the reports transferred to the department share.	Available after a build: - Executable Code - Test Results - Document containing baseline report Available after a test: - Test Results Available after either build or test: - Dashboard update

Artifacts provided as input of the activity		Artifacts produced during of the activity		Artifacts which are the result of the activity	
Name	Source Code	Name		Name	Executable Code
Generic Type: (Tool or language independent type)	Source Code	Generic Type: (Tool or language independent type)		Generic Type: (Tool or language independent type)	Binary code
Shared Properties: (Information to be shared in interaction between steps)	- Source code filename - Source code category tags - Source code author(s) - Lifecycle status information	Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)	- Code O/S meta data - Code dependencies - Code category tags - Lifecycle status info.
Description: Human-readable instructions and/or settings, used to automate the processing of data performed by tool, device or service. This includes instructions expressed in a programming language, but also project settings and instructions that define the build process that translate the programming language instructions into Executable Code.				Description: Code prepared for execution and/or interpretation on the target platform. This includes run-time library, type libraries, device drivers, or any other resource that is offered for use by the operating system on the target platform. Install-shields are also considered Executable Code, as they are utilized by the operating system.	
Name	Test	Name		Name	Test Result
Generic Type: (Tool or language independent type)	Formal Test	Generic Type: (Tool or language independent type)		Generic Type: (Tool or language independent type)	Formal Test Result
Shared Properties: (Information to be shared in interaction between steps)	- Test headline - Test reference ID - Test description - Test category tags - Test author(s) - Lifecycle status information	Shared Properties: (Information to be shared in interaction between steps)		Shared Properties: (Information to be shared in interaction between steps)	- Test date/time - Test reference ID - Test environment ID - Test outcome - Test deviations - Test remarks - Test engineer(s)
Description: an individual test, setup to verify one or more characteristics of a product or service. It is a sequence of actions and checks, where the actions indicate the test stimuli while the checks highlight the expected response. Its characteristics match that of the 'Requirement' artifact.				Description: Individual outcome of a test, capturing the outcome of the test, any deviations from the (formal) test procedure, and/or particular remarks or observations made. Its characteristics match that of the 'Requirement' artifact.	

Figure 31 Engineering Method ContinuousBuildTestIntegrationReport

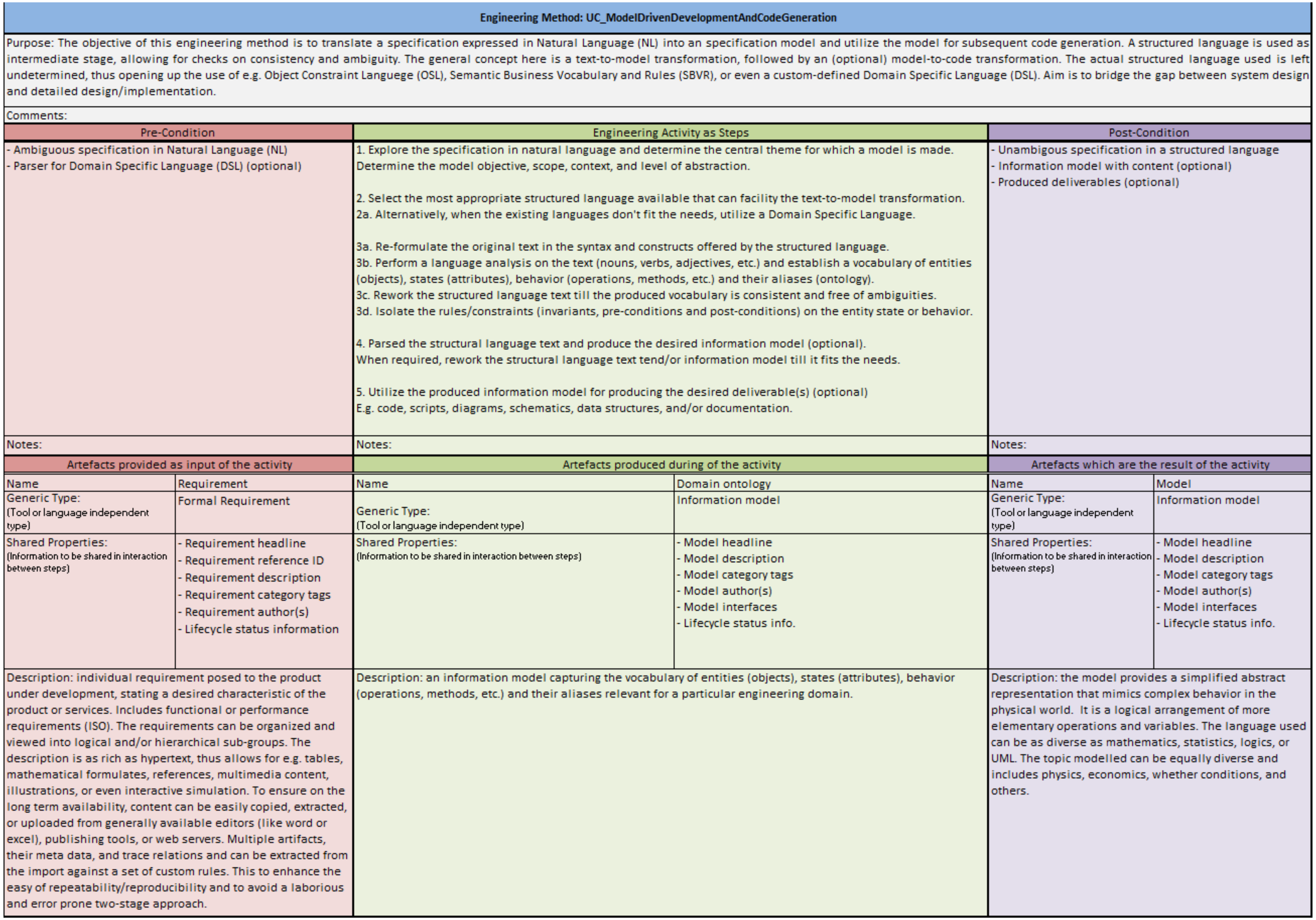
A.1.2 EM UC403_TestWithInTheLoopSimulation_001



<table>
<tr><th colspan="3">Engineering Method: UC43 - Test with In-The Loop-Simulation</th></tr>
<tr><td colspan="3">Purpose: detect software problems early, especially concerning relation with hardware</td></tr>
<tr><td colspan="3">Comments: related to heterogeneous simulation, but this method allows also other combinations</td></tr>
<tr><th>Pre-Condition</th><th>Engineering Activity as Steps</th><th>Post-Condition</th></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>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)</td><td>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).</td><td>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.</td></tr>
<tr><td>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.</td><td>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</td><td>Notes:</td></tr>
</table>

Figure 32 Engineering Method TestWithInTheLoopSimulation

A.1.3 EM UC403_ModelDrivenDevelopmentAndCodeGeneration_001



Engineering Method: UC_ModelDrivenDevelopmentAndCodeGeneration

Purpose: The objective of this engineering method is to translate a specification expressed in Natural Language (NL) into an specification model and utilize the model for subsequent code generation. A structured language is used as intermediate stage, allowing for checks on consistency and ambiguity. The general concept here is a text-to-model transformation, followed by an (optional) model-to-code transformation. The actual structured language used is left undetermined, thus opening up the use of e.g. Object Constraint Languege (OSL), Semantic Business Vocabulary and Rules (SBVR), or even a custom-defined Domain Specific Language (DSL). Aim is to bridge the gap between system design and detailed design/implementation.

Comments:

Pre-Condition	Engineering Activity as Steps	Post-Condition
- Ambiguous specification in Natural Language (NL) - Parser for Domain Specific Language (DSL) (optional)	1. Explore the specification in natural language and determine the central theme for which a model is made. Determine the model objective, scope, context, and level of abstraction. 2. Select the most appropriate structured language available that can facility the text-to-model transformation. 2a. Alternatively, when the existing languages don't fit the needs, utilize a Domain Specific Language. 3a. Re-formulate the original text in the syntax and constructs offered by the structured language. 3b. Perform a language analysis on the text (nouns, verbs, adjectives, etc.) and establish a vocabulary of entities (objects), states (attributes), behavior (operations, methods, etc.) and their aliases (ontology). 3c. Rework the structured language text till the produced vocabulary is consistent and free of ambiguities. 3d. Isolate the rules/constraints (invariants, pre-conditions and post-conditions) on the entity state or behavior. 4. Parsed the structural language text and produce the desired information model (optional). When required, rework the structural language text tend/or information model till it fits the needs. 5. Utilize the produced information model for producing the desired deliverable(s) (optional) E.g. code, scripts, diagrams, schematics, data structures, and/or documentation.	- Unambigous specification in a structured language - Information model with content (optional) - Produced deliverables (optional)
Notes:	Notes:	Notes:

Artefacts provided as input of the activity		Artefacts produced during of the activity		Artefacts which are the result of the activity	
Name	Requirement	Name	Domain ontology	Name	Model
Generic Type: (Tool or language independent type)	Formal Requirement	Generic Type: (Tool or language independent type)	Information model	Generic Type: (Tool or language independent type)	Information model
Shared Properties: (Information to be shared in interaction between steps)	- Requirement headline - Requirement reference ID - Requirement description - Requirement category tags - Requirement author(s) - Lifecycle status information	Shared Properties: (Information to be shared in interaction between steps)	- Model headline - Model description - Model category tags - Model author(s) - Model interfaces - Lifecycle status info.	Shared Properties: (Information to be shared in interaction between steps)	- Model headline - Model description - Model category tags - Model author(s) - Model interfaces - Lifecycle status info.
Description: individual requirement posed to the product under development, stating a desired characteristic of the product or services. Includes functional or performance requirements (ISO). The requirements can be organized and viewed into logical and/or hierarchical sub-groups. The description is as rich as hypertext, thus allows for e.g. tables, mathematical formulates, references, multimedia content, illustrations, or even interactive simulation. To ensure on the long term availability, content can be easily copied, extracted, or uploaded from generally available editors (like word or excel), publishing tools, or web servers. Multiple artifacts, their meta data, and trace relations and can be extracted from the import against a set of custom rules. This to enhance the easy of repeatability/reproducibility and to avoid a laborious and error prone two-stage approach.		Description: an information model capturing the vocabulary of entities (objects), states (attributes), behavior (operations, methods, etc.) and their aliases relevant for a particular engineering domain.		Description: the model provides a simplified abstract representation that mimics complex behavior in the physical world. It is a logical arrangement of more elementary operations and variables. The language used can be as diverse as mathematics, statistics, logics, or UML. The topic modelled can be equally diverse and includes physics, economics, whether conditions, and others.	

Figure 33 Engineering Method ModelDrivenDevelopmentAndCodeGeneration

A.1.4 Artefacts

The figure below depicts an UML representation of the engineering artefacts required for this Use Case. The sub-paragraphs elaborate further on the artefacts mentioned.

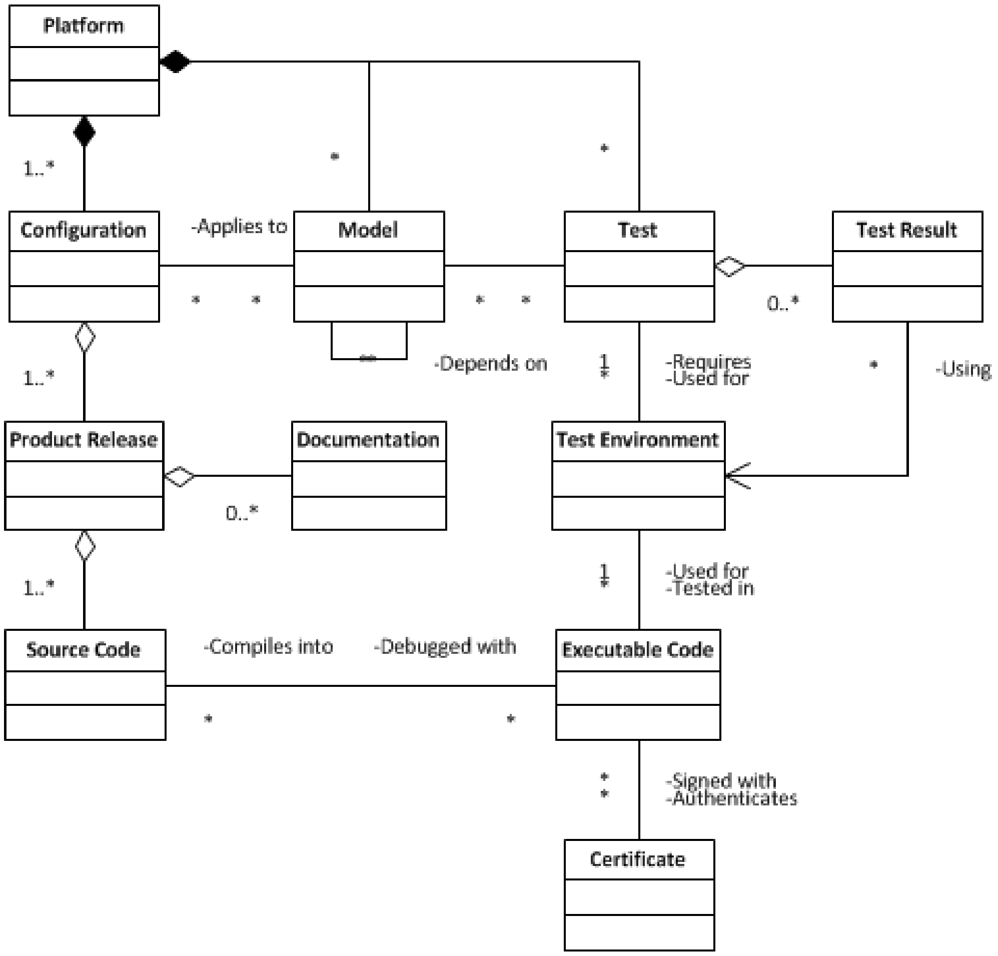


Figure 34 Artefact UML diagram

A.1.4.1 Artefact – Certificate

Chain of digital certificates and certificate revocation lists, as provided by a Public Key Infrastructure (PKI). It allows developers to include information about themselves and their code within their programs. Certificate trees allow for the verification of the authenticity of a code provider or the integrity of the code itself.

Shared properties:
- As per X509 standard

A.1.4.2 Artefact – Executable Code

Code prepared for execution and/or interpretation on the target platform. This includes run-time library, type libraries, device drivers, or any other resource that is offered for use by the operating system on the target platform. Install-shields are also considered Executable Code, as they are utilized by the operating system.

Shared properties:
- Executable code O/S meta data
- Executable code dependencies
- Executable code category tags
- Lifecycle status information

A.1.4.3 Artefact – Document

Individual document or report, consisting out of one or more files in formats supported by the engineering environment. Information is subject to access and change control.

Shared properties:
- Document title
- Document type
- Document reference ID
- Document author(s)
- Lifecycle status information

A.1.4.4 Artefact – Model

The model provides a simplified abstract representation that mimics complex behaviour in the physical world. It is a logical arrangement of more elementary operations and variables. The language used can be as diverse as mathematics, statistics, logics, or UML. The topic modelled can be equally diverse and includes physics, economics, weather conditions, and others.

A model is a composition of:
- Components that perform some predefined type of (logical) operation.
- Interconnections that build the graph how the components interact with one another.
- Meta data that defines or tailors the component or interconnection behaviour.

Shared properties:
- Model headline
- Model description
- Model category tags
- Model author(s)
- Model interfaces
- Lifecycle status information

A.1.4.5 Artefact – Source Code

Human-readable instructions and/or settings, used to automate the processing of data performed by a tool, device or service. This includes instructions expressed in a programming language, but also project settings and instructions that define the build process that translate the programming language instructions into Executable Code.

Shared properties:
- Source code filename
- Source code category tags
- Source code author(s)
- Lifecycle status information

A.1.4.6 Artefact – Test

An individual test, set up to verify one or more characteristics of a product or service. It is a sequence of actions and checks, where the actions indicate the test stimuli while the checks highlight the expected response. Its characteristics match that of the 'Requirement' artefact.

For the ease of automated regression testing, an action and/or check may refer to one or more test scripts that automate (part of the) procedure. Custom rule sets utilizing meta data allow for the sequencing of tests into a test run, scheduled for a particular test environment (while assuming that the test environment is suited for that particular test).

Shared properties:
- Test headline
- Test reference ID
- Test description
- Test category tags
- Test author(s)
- Lifecycle status information

A.1.4.7 Artefact – Test Environment

Inventory of measuring instruments and software that make up the test environment. For instruments it includes details as vendor, model, type, serial number, and last calibration date. For software and operating systems, it includes details like vendor, application, version, release date, and installed patches. For custom build tools and settings, it includes baseline information of the builds. Basically the Test Environment artefact ensures on the repeatability and reproducibility of test results.

Shared properties:
- Utility headline
- Utility reference ID
- Utility description
- Utility category tags
- Lifecycle status information

A.1.4.8 Artefact – Test Result

Individual outcome of a test, capturing the outcome of the test, any deviations from the (formal) test procedure, and/or particular remarks or observations made. Its characteristics match that of the 'Requirement' artefact.

Shared properties:
- Test date/time
- Test reference ID
- Test environment ID
- Test outcome
- Test deviations
- Test remarks
- Test engineer(s)

Appendix B Tool chain description

B.1.1 Borland – CaliberRM

CaliberRM is a requirements management tool to enable requirements communication, collaboration, and verification. It enables teams to fully define, manage and communicate changing requirements. Changes to requirement data are recorded and stored in the database, providing reliable and up-to-date information for effective requirements-based application development and testing.

Artifacts:
- In/Out: Requirements

B.1.2 Eclipse.org – Xtend (B4.04)

Xtend is a statically-typed programming language which translates to comprehensible Java source code. Syntactically and semantically Xtend has its roots in the Java programming language but improves on many aspects. Xtend is much more concise, readable and expressive than Java. Xtend's small library is just a thin layer that provides useful utilities and extensions on top of the Java Development Kit (JDK). The compiled output is readable and pretty-printed, and tends to run as fast as the equivalent handwritten Java code.

Artifacts:
- In/Out: Source Code

B.1.3 Eclipse.org – Xtext (B4.04)

Xtext is a framework for development of programming languages and domain specific languages. It covers all aspects of a complete language infrastructure, from parsers, over linker, compiler or interpreter to Eclipse IDE integration. It provides a set of domain-specific languages and APIs to describe the different aspects of your programming language. Based on that information it gives a full implementation of that language running on the JVM. The compiler components of the language are independent of Eclipse or OSGi and can be used in any Java environment. They include such things as the parser, the type-safe abstract syntax tree (AST), the serializer and code formatter, the scoping framework and the linking, compiler checks and static analysis a.k.a. validation and last but not least a code generator or interpreter. These runtime components integrate with and are based on the Eclipse Modelling Framework (EMF), which effectively allows the use of Xtext together with other EMF frameworks like for instance the Graphical Modelling Project GMF.

Artifacts:
- In: Model (of DSL language)
- In: Source Code
- Out: Executable Code (of DSL parser)

B.1.4 Electric Cloud – ElectricCommander

ElectricCommander automates and accelerates software delivery using an engine that unites production processes and their supporting IT resources. These processes typically include building, testing, releasing, and deploying software. It provides complete IT resource management through support for physical, virtual, and public/private cloud infrastructure. The platform provides hundreds of integrations to industry tools like compilers, test systems, code coverage tools, infrastructure platforms, etc.

Artifacts:
- In: Build Request
- Out: Build Result
- In: Test Request
- Out: Test Result

B.1.5 Google – Google Test

Google's framework for writing C++ tests on a variety of platforms (Linux, Mac OS X, Windows, Cygwin, Windows CE, and Symbian). It is based on the xUnit architecture and supports automatic test discovery, a rich set of assertions, user-defined assertions, death tests, fatal and non-fatal failures, value- and type-parameterized tests, various options for running the tests, and XML test report generation.

Artifacts:

- In: Test
- Out: Test Result

B.1.6 HP – Quality Center (B4.12)

HP Quality Center will be used by this brick in the context of requirement management, test case management and traceability, test case execution, creating of report information, SW release management. HP Quality Center should be integrated to the interoperability standard (IOS) in a corresponding manner.

Artifacts:

- In: Requirements
- In/Out: Tests
- In/Out: Test Results
- In/Out: Test Environment Details

B.1.7 IBM – Rational ClearCase

ClearCase is a software configuration management solution that provides version control, workspace management, parallel development support, and build auditing. You can integrate Rational ClearCase with other IBM solutions, including IBM Rational Team Concert, IBM Rational ClearQuest, IBM Rational Asset Manager, and IBM Rational Application Developer for WebSphere Software. Rational ClearCase scales to any size team from small workgroup to large, geographically distributed teams.

Artifacts:

- In/Out: Source Code
- In/Out: Executable Code
- In/Out: Change Request
- Out: Document (Baseline report)

B.1.8 IBM – Rational ClearQuest (B3.87)

A Change Request system, which controls the flow of information w.r.t. to any external or internal change requests after a freeze of requirements. This is essentially also a partial requirements database – however the information incoming from here needs to be transferred into the ReqPro database with data integrity and all attributes intact. Currently there is no common interface so IOS should be investigated.

Artifacts:

- In/Out: Change Request

B.1.9 IBM – Rational DOORS Next Generation (B2.16)

IBM Rational DOORS is a widely adopted product, system and software requirements management tool to enable requirements communication, collaboration, and verification. It is optimized for the needs of complex and embedded systems development, and is a candidate for prototyping of an IOS interface.

Artifacts:

- In/Out: Requirements

B.1.10 IBM – Rational Quality Manager

Quality Manager is a web-based centralized test management environment that provides a collaborative and customizable solution for test planning, workflow control, tracking and metrics reporting. It acts as collaborative hub for business-driven software and systems quality across virtually any platform and type of testing. This software helps teams share information seamlessly, use automation to accelerate project schedules and report on metrics for informed release decisions.

Artifacts:

- In: Requirements
- In/Out: Tests
- In/Out: Test Results
- In/Out: Test Environment Details

B.1.11 IBM – Rational Rhapsody (B2.10)

IBM Rational Rhapsody is a widely adopted family of tools targeting towards visual, model-driven development for systems and software applications. It provides collaborative design and development for systems engineers and software developers creating real-time or embedded systems and software, with support for dependable systems including safety, security and reliability.

Artifacts:

- In/Out: Models (on structure and behavior)
- In/Out: Source Code

B.1.12 IBM – Rational Team Concert (B2.19)

IBM Rational Team Concert is a widely adopted systems and software lifecycle management solution that enables real-time, contextual collaboration for distributed teams. It includes agile, formal and hybrid planning and reporting, with support for the automation of complex and embedded systems development and powerful collaborative change management capabilities. RTC is a candidate to be considered for prototyping of an IOS interface.

Artifacts:

- In/Out: Source Code
- In/Out: Executable Code
- In/Out: Change Request

B.1.13 Mathworks – Matlab/Simulink (B3.46)

Simulink is a popular dynamic systems modeler with a broad scope of features, rich ecosystem and wide use. Simulink is found in many domains and is the source for complex model-based tool-chains in software development for embedded systems. It is the de-facto industry standard in simulation model development.
The modelling language of Matlab is a proprietary 4th generation programming language developed by MathWorks Inc. The Simulink add-on uses a data flow graphical programming language.

Artifacts:

- In: Models (on data flow in dynamic systems)
- In: Test (signal data and the simulation script)
- Out: Source Code
- Out: Test Results

B.1.14 Microsoft – Authenticode

Microsoft Authenticode is a framework of tools that allow developers to include information about themselves and their code with their programs through the use of digital signatures. It allows for a verification of the authenticity of the code provider or the integrity of the code itself.

Artifacts:

- In: Executable Code
- In: Certificate tree
- Out: Executable Code (signed and e.g. packed in an install shield)

B.1.15 Microsoft – Visual Studio

Visual Studio is a comprehensive collection of tools and services for developing applications that target the desktop, the web, devices, and the cloud. It provides an integrated development environment (IDE) and collaboration environment that welcomes connection with other development tools, such as Eclipse and Xcode. It leverages Visual Studio's state-of-the-art development environment for .NET languages, HTML/JavaScript, and C++ for teams working across multiple platforms.

Artifacts:

- In: Model (optional: via plugins)
- In: Source Code
- Out: Executable Code (e.g. packed in an install shield)

B.1.16 NUnit.org – NUnit

NUnit is a unit-testing framework for all .Net languages. It is part of the xUnit based unit testing tool for Microsoft .NET. NUnit has two different ways to run your tests. The console runner, nunit-console.exe, is the fastest to launch, but is not interactive. The gui runner, nunit.exe, is a Windows Forms application that allows you to work selectively with your tests and provides graphical feedback

Artifacts:
- In: Source Code
- In: Test
- Out: Test Result

B.1.17 OSRF – GAZEBO (B4.10)

Gazebo is a multi-robot simulator for outdoor environments developed and maintained by the Open Source Robotics Foundation (OSRF). It is capable of simulating a population of robots, sensors and objects, but does so in a three-dimensional world. It generates both realistic sensor feedback and physically plausible interactions between objects (it includes an accurate simulation of rigid-body physics).

Artifacts:
- N.a.

B.1.18 Philips – CaliberToQC

CaliberToQC is a proprietary tool developed by Philips for exporting requirements engineered in Borland CaliberRM and importing these requirements into HP Quality Center. The tool first ensures on the consistency of the information structures and Meta data in both environments, and report any inconsistencies found. These inconsistencies are to be resolved manually. Once the pre-condition is met, it will perform the actual data exchange.

Artifacts:
- In/Out: Requirements

B.1.19 Philips – Xposer

Xposer is a proprietary tool developed by Philips for visualizing and simulating stand/table movements. It utilizes Ogre for 3D visualization and rendering, and QT for the modelling user interface. The physical configuration is described using XML. OGRE (Object-Oriented Graphics Rendering Engine) is a scene-oriented, flexible 3D engine written in C++ designed to make it easier and more intuitive for developers to produce applications utilising hardware-accelerated 3D graphics. The class library abstracts all the details of using the underlying system libraries like Direct3D and OpenGL and provides an interface based on world objects and other intuitive classes. QT provides different approaches for developers and designers to create application user interfaces and gives the freedom to select the best workflow and UI approach for the development purposes.

Artifacts:
- In: Model (of physical stand/table, with its axes and degrees of freedom)
- In: Test (optional: a user interaction scenario)
- Out: Test Result (optional: simulation movie of stand/table movement)

B.1.20 QlickTech – Qlikview (B4.13)

The Qlikview Business Discovery Platform is a Business Intelligence tool that provides a flexible and dynamic way to present information to support innovative and collaborative decision making. Qlikview supports easy creation of dashboards, dynamic data representation and powerful data analysis from multiple angles like functional disciplines and organizational hierarchy. Qlikview is used within Philips Healthcare to provide dashboards and analysis views for different disciplines to support the development and system engineering processes and improve insight in those processes.

Artifacts:
- In: Build Results
- In: Test Results
- Out: CM dashboard

B.1.21 SourceWorks – OROCOS (B4.11)

Orocos is a tool chain, dedicated to Open RObot COntrol Software development for Model-Driven Engineering. It is fully component-based and multi-vendor. This tool helps creating real-time robotics applications using modular, run-time configurable software components.

Artifacts:

- In: Model (of kinematics and dynamics)
- In: Source Code
- Out: Executable Code (e.g. packed in an install shield)

B.1.22 VMware – Workstation

VMware software provides an environment with virtualized hardware, software, and services. The heart of virtualization is the “virtual machine” (VM), a tightly isolated software container with an operating system and application inside. Because each virtual machine is completely separate and independent, many of them can run simultaneously on a single computer. A thin layer of software called a hypervisor decouples the virtual machines from the host and dynamically allocates computing resources to each virtual machine as needed.

Artifacts:

- In: Executable Code (content of Integrated Development Environment)
- Out: Executable Code (VMware image of build or development environment)

Appendix C Interaction diagrams EM Continuous Build Test Integration

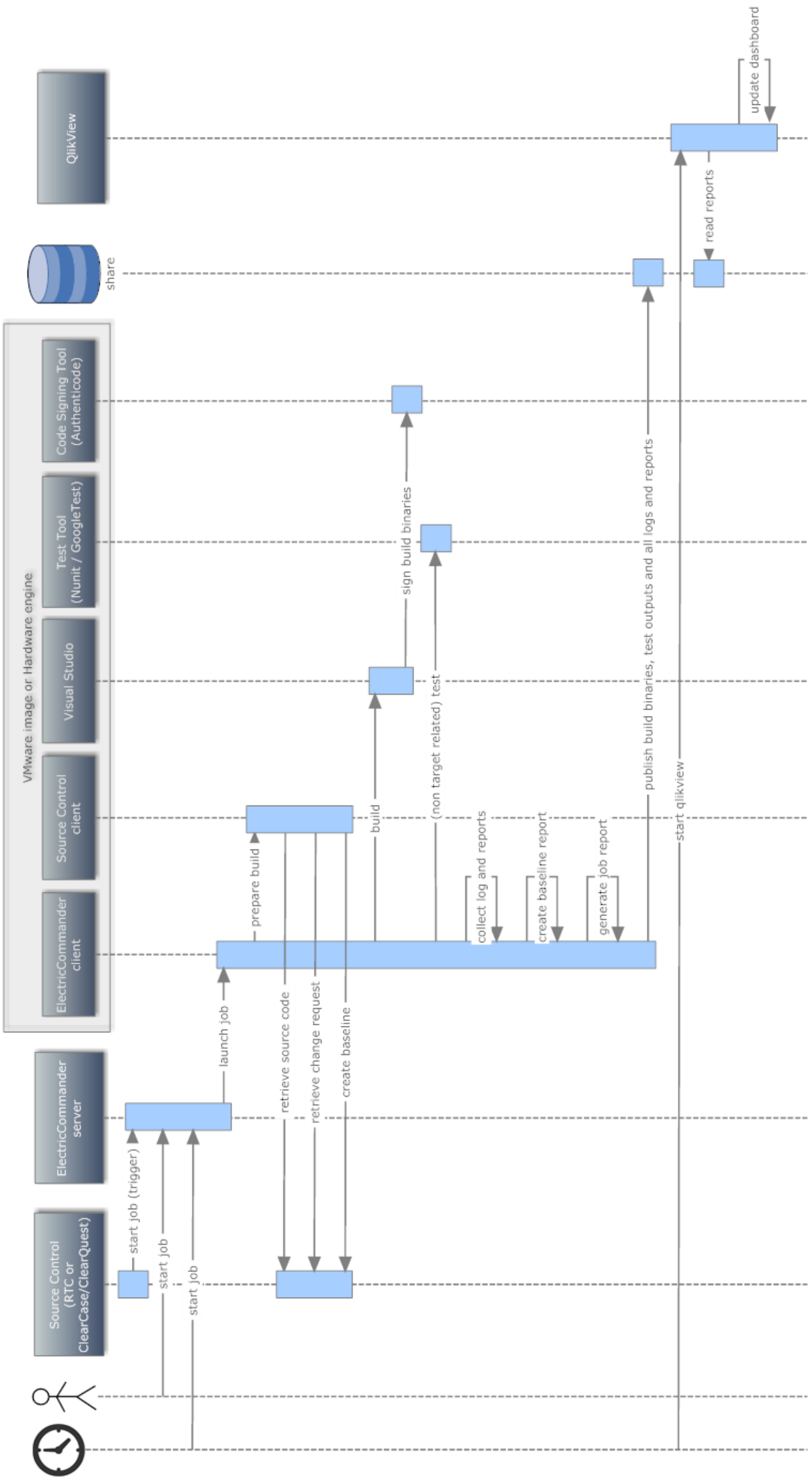


Figure 35 Interaction diagram of tool chain for building binaries

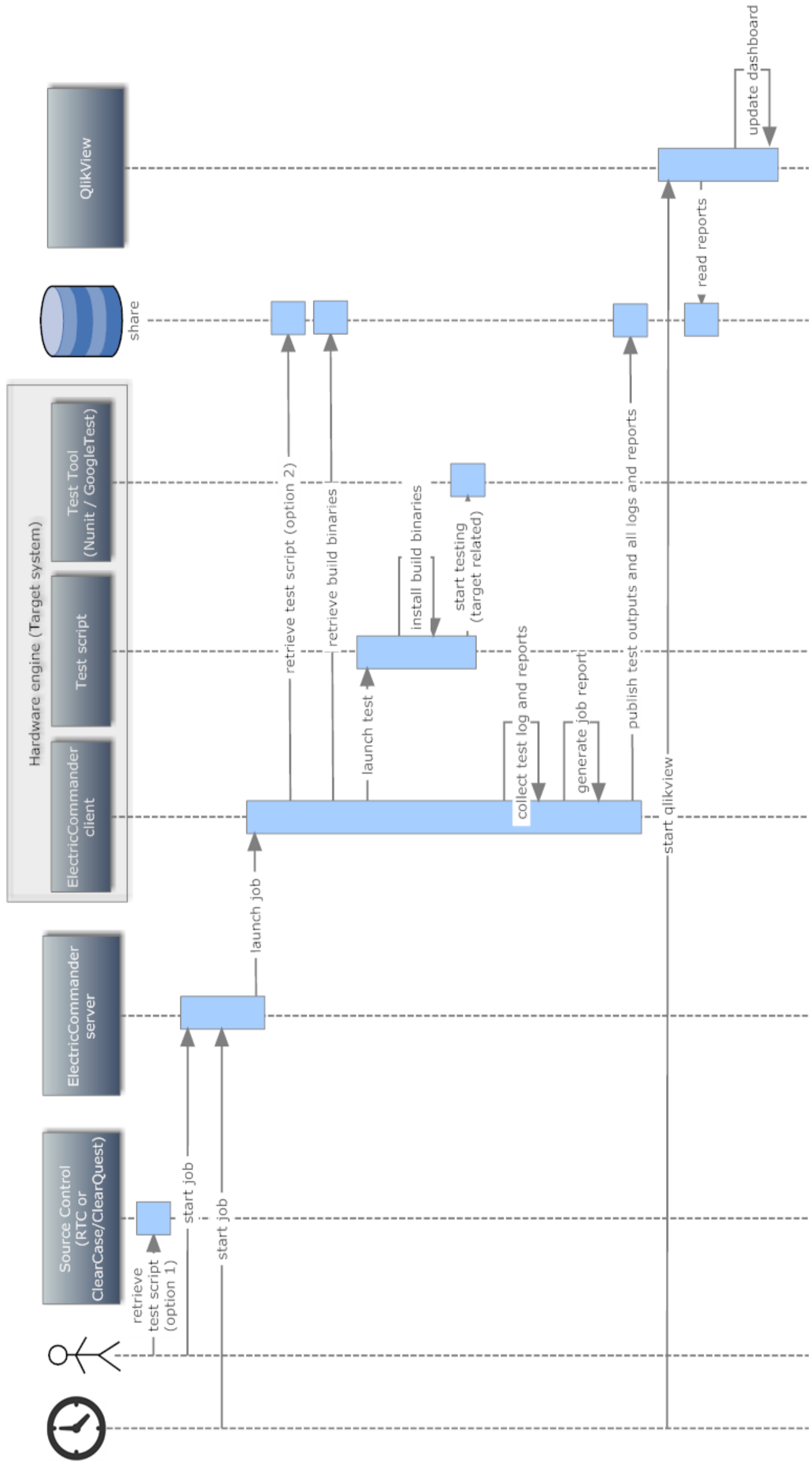

QlikView
share
Hardware engine (Target system)
Test Tool (Nunit / GoogleTest)
Test script
ElectricCommander client
ElectricCommander server
Source Control (RTC or ClearCase/ClearQuest)
retrieve test script (option 1)
start job
start job
launch job
retrieve test script (option 2)
retrieve build binaries
launch test
install build binaries
start testing (target related)
collect test log and reports
generate job report
publish test outputs and all logs and reports
start qlikview
read reports
update dashboard

Figure 36 Interaction diagram of tool chain for testing on dedicated target platform

Appendix D High level description of use case and context

D.1 Use case context

D.1.1 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.

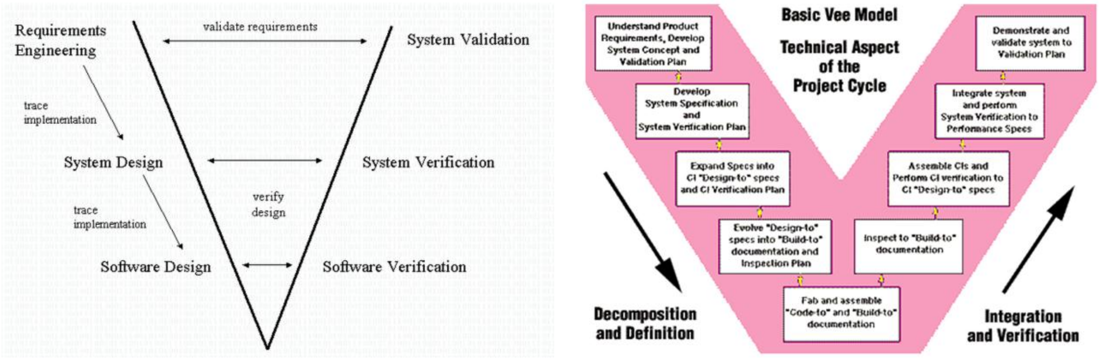

Requirements Engineering
validate requirements
System Validation
trace implementation
System Design
System Verification
verify design
trace implementation
Software Design
Software Verification
Basic Vee Model
Technical Aspect of the Project Cycle
Understand Product Requirements, Develop System Concept and Validation Plan
Develop System Specification and System Verification Plan
Expand Specs into CI "Design-to" specs and CI Verification Plan
Evolve "Design-to" specs into "Build-to" documentation and Inspection Plan
Fab and assemble "Code-to" and "Build-to" documentation
Inspect to "Build-to" documentation
Assemble CIs and Perform CI verification to CI "Design-to" specs
Integrate system and perform System Verification to Performance Specs
Demonstrate and validate system to Validation Plan
Decomposition and Definition
Integration and Verification

Figure D-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 compliaint 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.

D.1.2 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 D-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.

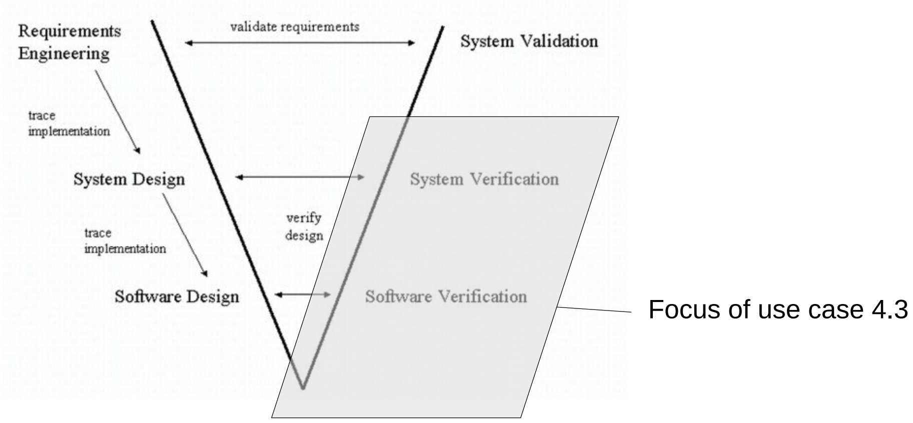


Figure D-1: Development process scope of UC4.3.

D.1.3 *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):

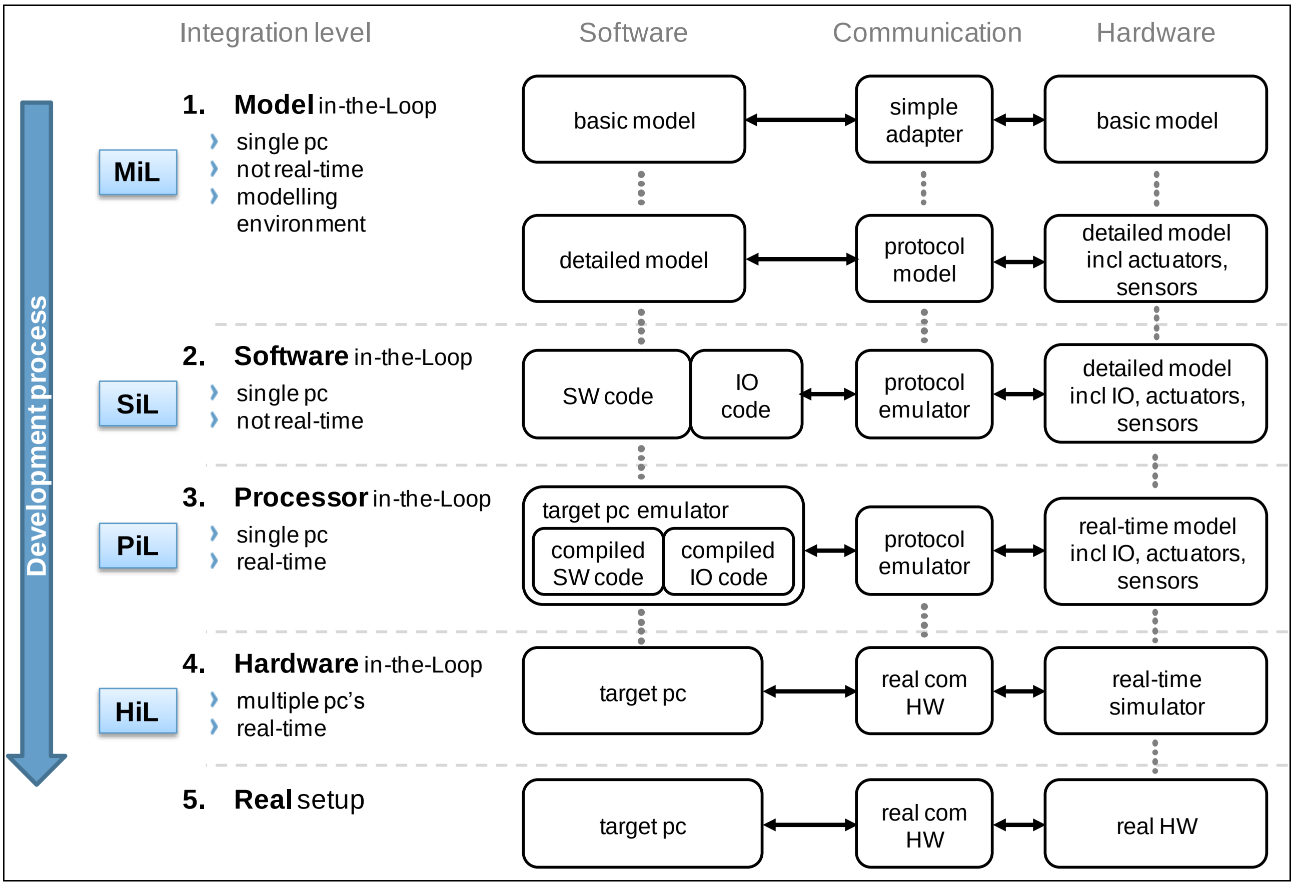


Figure D-2: In-the-loop simulation definitions.

D.1.3.1 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).

D.1.3.2 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).

D.1.3.3 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.

D.1.3.4 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.

D.1.4 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.

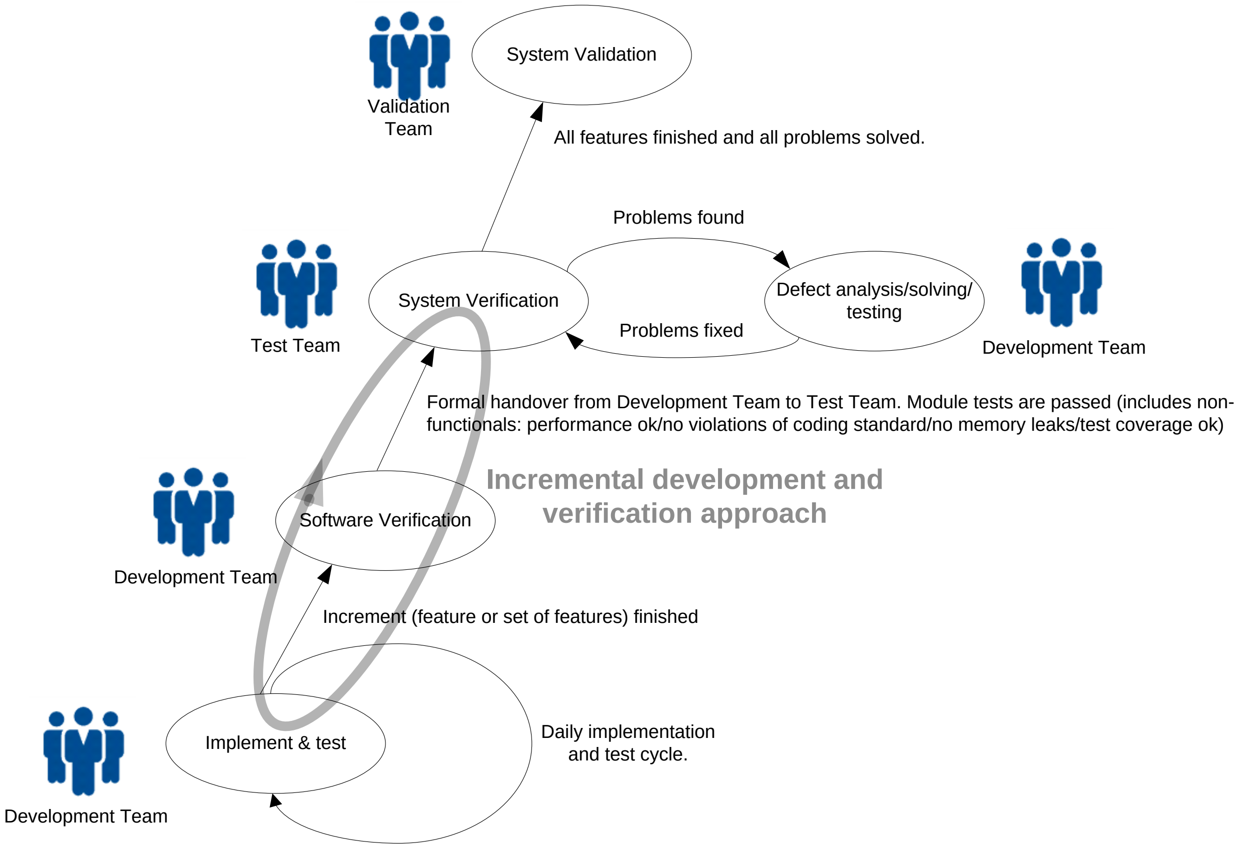
In the same vein the practice of continuous delivery further extends CI by making sure the software checked in on the mainline is always in a state that can be deployed to users and makes the actual deployment process very rapid.

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.

D.2 Use Case Process Description

D.2.1 *Current* development process

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



Validation Team
System Validation
All features finished and all problems solved.
Problems found
Test Team
System Verification
Defect analysis/solving/ testing
Problems fixed
Development Team
Formal handover from Development Team to Test Team. Module tests are passed (includes non-functionals: performance ok/no violations of coding standard/no memory leaks/test coverage ok)
Incremental development and verification approach
Development Team
Software Verification
Increment (feature or set of features) finished
Development Team
Implement & test
Daily implementation and test cycle.

Figure D-2: 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.

D.2.1.1 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):

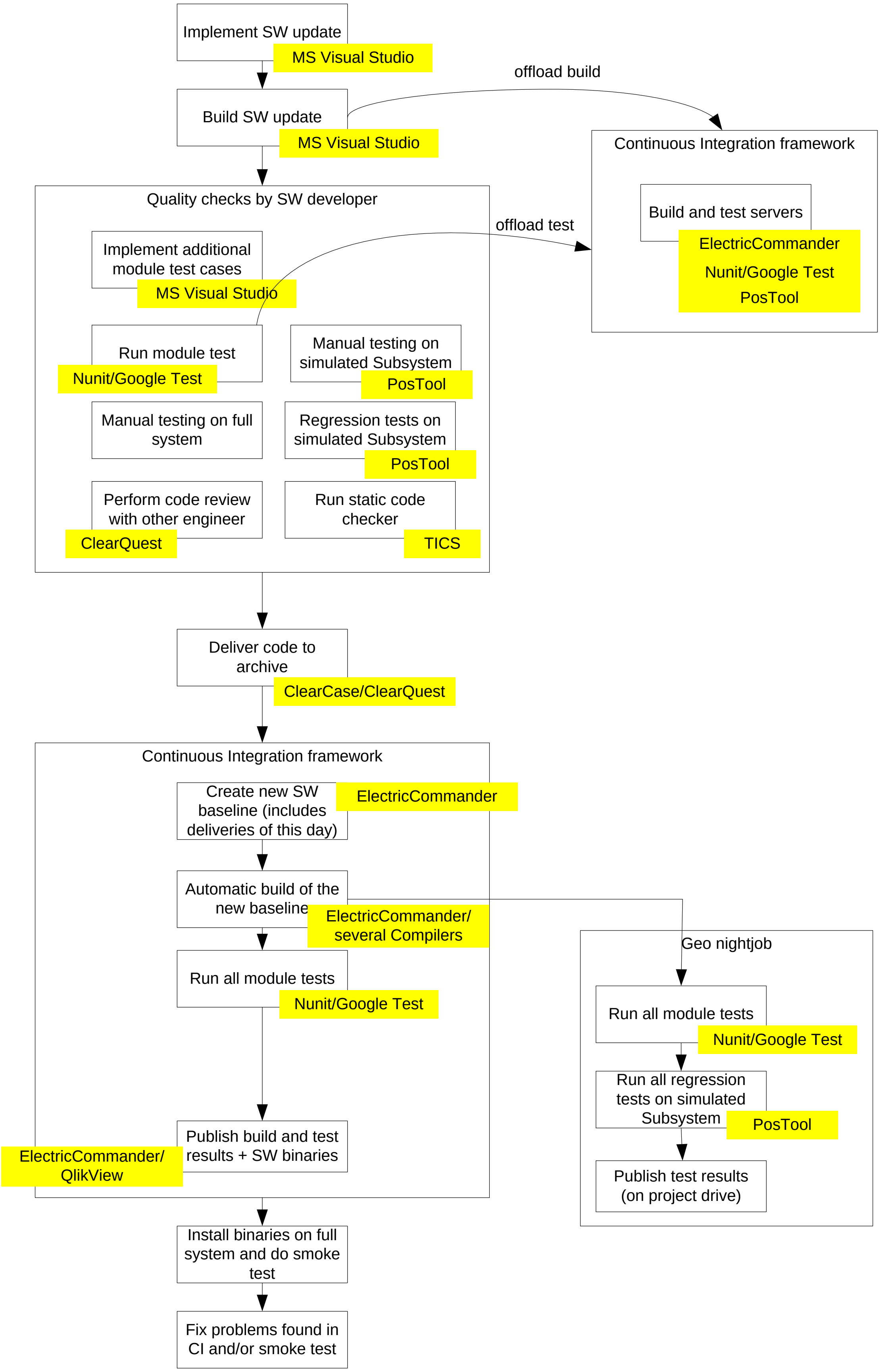

Implement SW update
MS Visual Studio
offload build
Build SW update
MS Visual Studio
Continuous Integration framework
Build and test servers
ElectricCommander
Nunit/Google Test
PosTool
Quality checks by SW developer
offload test
Implement additional module test cases
MS Visual Studio
Run module test
Nunit/Google Test
Manual testing on simulated Subsystem
PosTool
Manual testing on full system
Regression tests on simulated Subsystem
PosTool
Perform code review with other engineer
ClearQuest
Run static code checker
TICS
Deliver code to archive
ClearCase/ClearQuest
Continuous Integration framework
Create new SW baseline (includes deliveries of this day)
ElectricCommander
Automatic build of the new baseline
ElectricCommander/ several Compilers
Run all module tests
Nunit/Google Test
Publish build and test results + SW binaries
ElectricCommander/ QlikView
Geo nightjob
Run all module tests
Nunit/Google Test
Run all regression tests on simulated Subsystem
PosTool
Publish test results (on project drive)
Install binaries on full system and do smoke test
Fix problems found in CI and/or smoke test

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

D.2.1.2.1 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 RTOS target.	-	Provide simulation on (or of) RTOS.
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 RTOS 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) RTOS to automatically test performance. Deploy SW on target HW + RTOS 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 D-1 : bottlenecks and improvements.

D.2.2.1 System Verification

Below the current System Verification process is depicted.

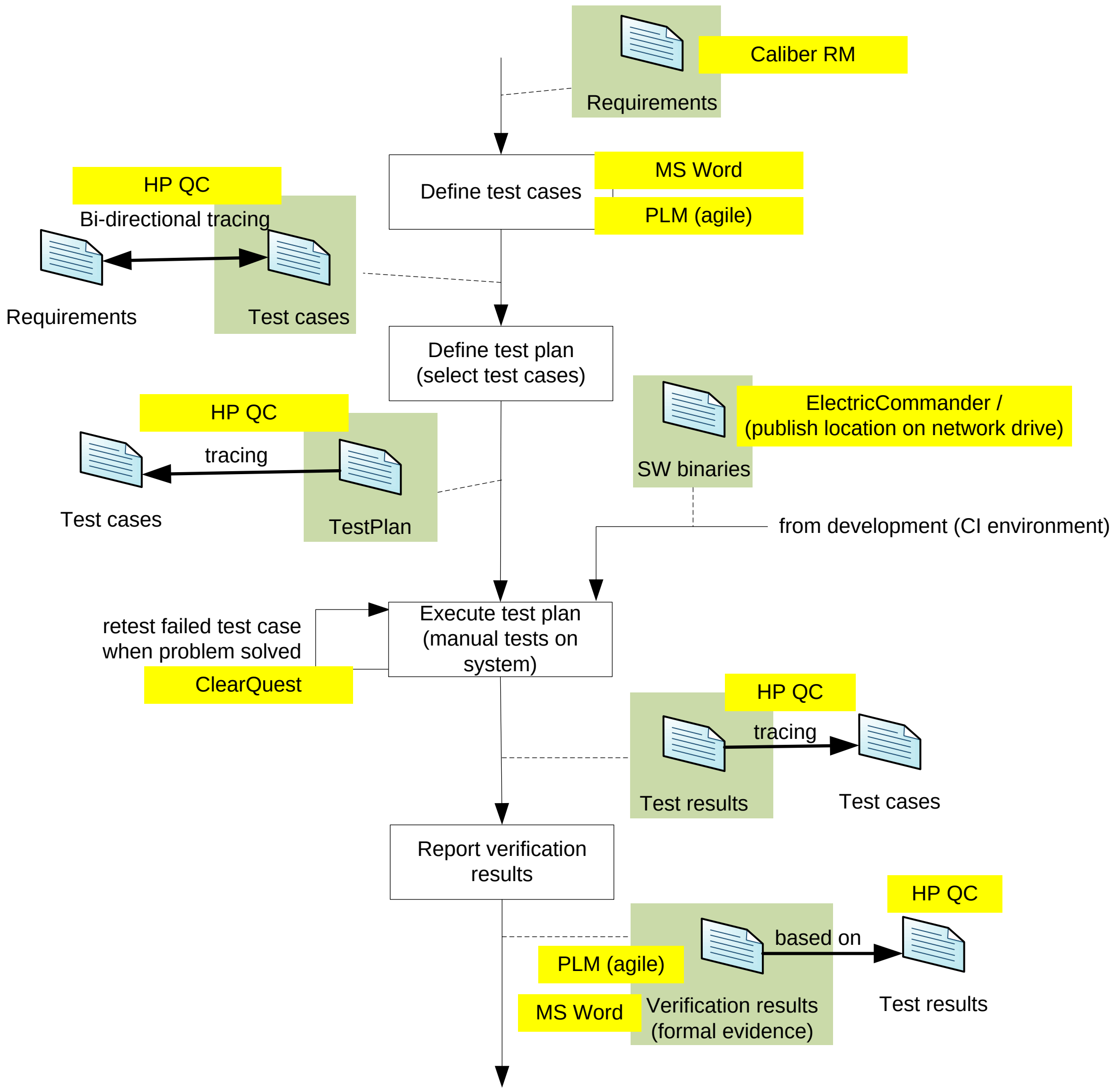


Figure D-3: System Verification process in more detail.

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.

D.2.1.2.1 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	-	-	-

D.2.2 Proposed Development Process

D.2.2.1 The “Implement and test” cycle

Below the proposed “Implement and test” cycle is depicted, showing the improvements (indicated in blue) identified in section D.2.1.2.1.

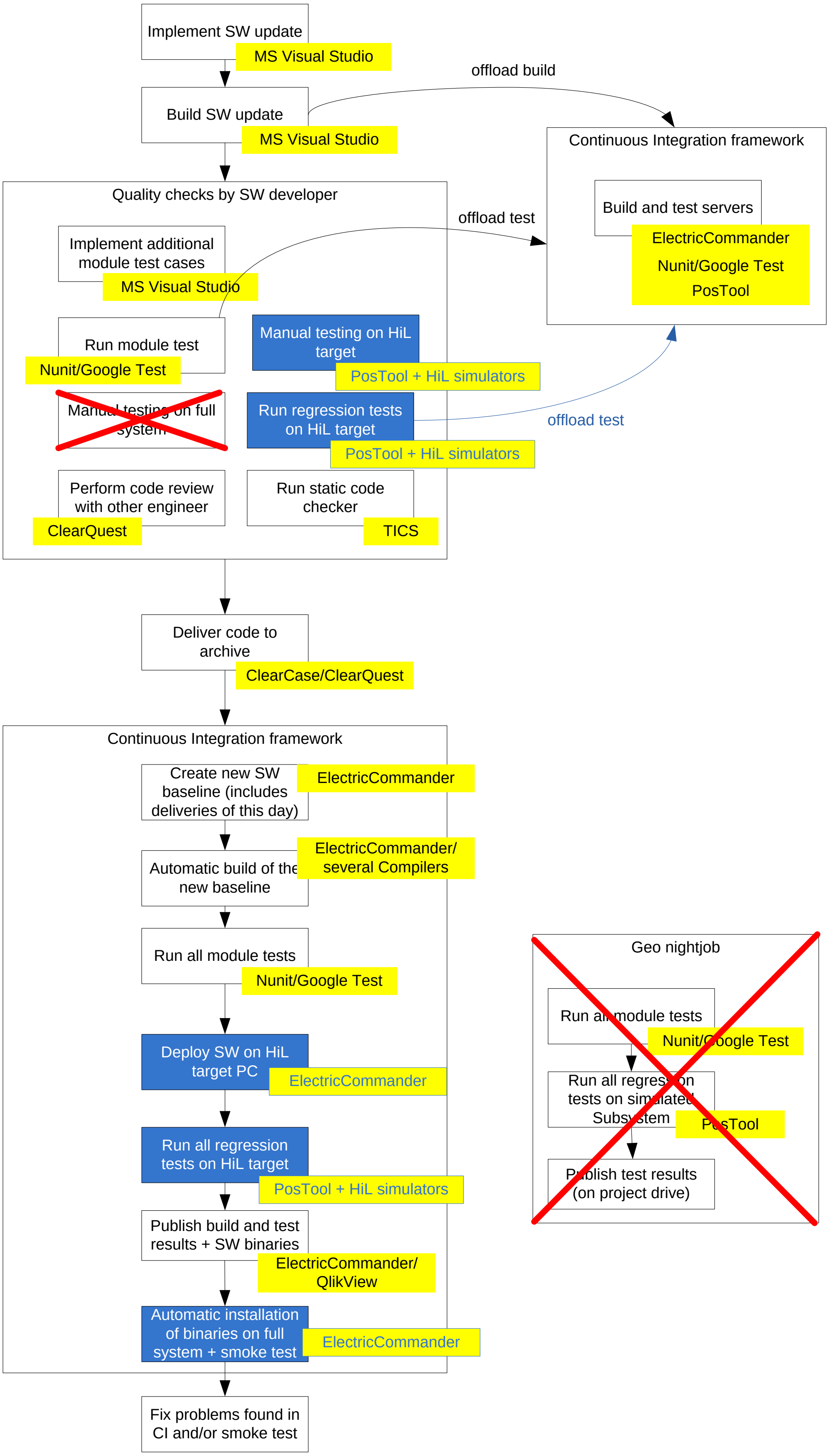

Implement SW update
MS Visual Studio
offload build
Build SW update
MS Visual Studio
Continuous Integration framework
Build and test servers
ElectricCommander
Nunit/Google Test
PosTool
Quality checks by SW developer
offload test
Implement additional module test cases
MS Visual Studio
Run module test
Nunit/Google Test
Manual testing on HiL target
PosTool + HiL simulators
~~Manual testing on full system~~
Run regression tests on HiL target
PosTool + HiL simulators
offload test
Perform code review with other engineer
ClearQuest
Run static code checker
TICS
Deliver code to archive
ClearCase/ClearQuest
Continuous Integration framework
Create new SW baseline (includes deliveries of this day)
ElectricCommander
Automatic build of the new baseline
ElectricCommander/ several Compilers
Run all module tests
Nunit/Google Test
Deploy SW on HiL target PC
ElectricCommander
Run all regression tests on HiL target
PosTool + HiL simulators
Publish build and test results + SW binaries
ElectricCommander/ QlikView
Automatic installation of binaries on full system + smoke test
ElectricCommander
Fix problems found in CI and/or smoke test
~~Geo nightjob~~
~~Run all module tests~~
~~Nunit/Google Test~~
~~Run all regression tests on simulated Subsystem~~
~~PosTool~~
~~Publish test results (on project drive)~~

Figure D-4: Implement and Test cycle (proposed).

D.2.2.2 System Verification

Below the proposed System Verification process is depicted, showing the improvements (indicated in blue) identified in section D.2.1.2.1. The picture includes the currently used tooling and the proposed new tooling.

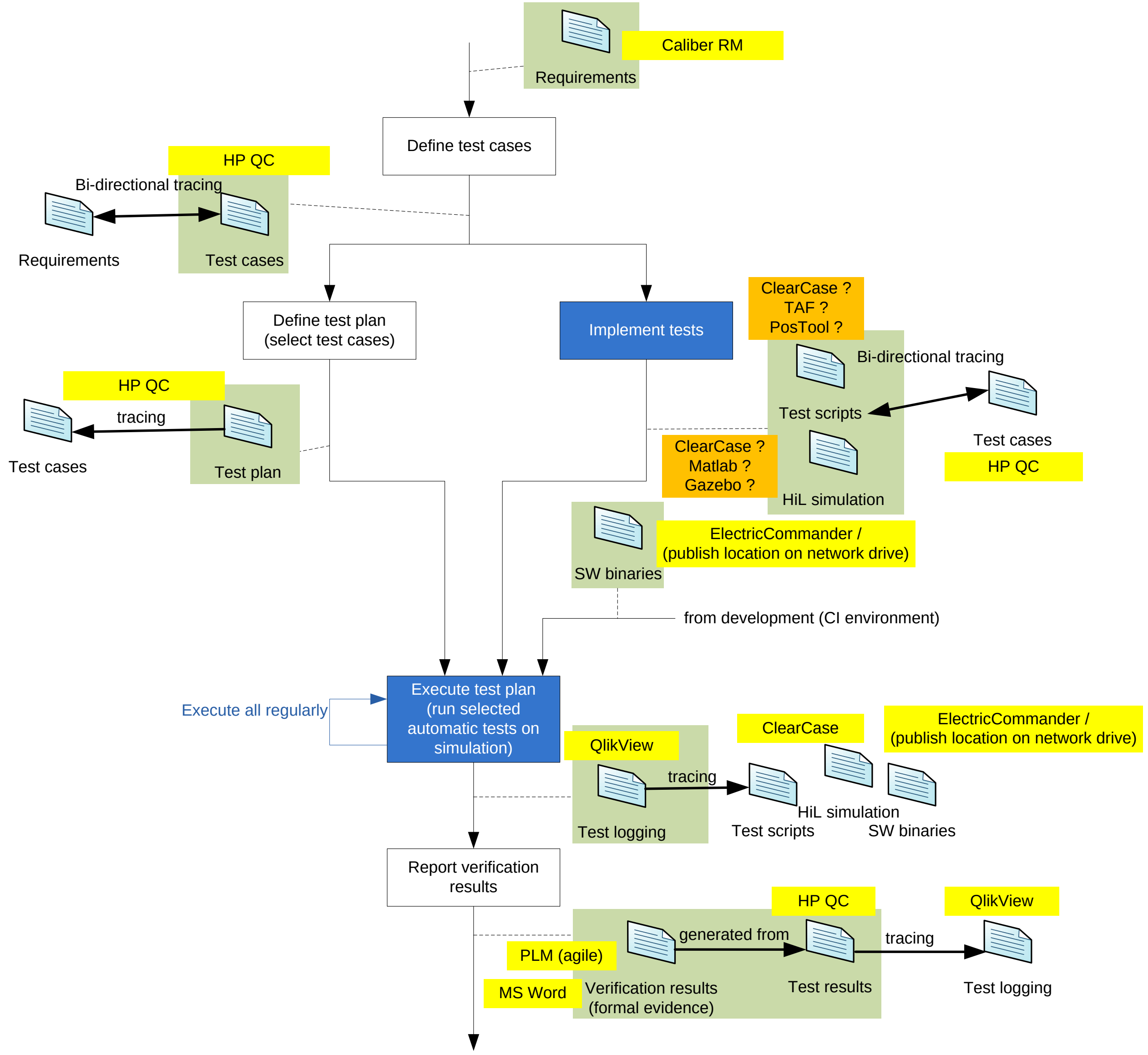
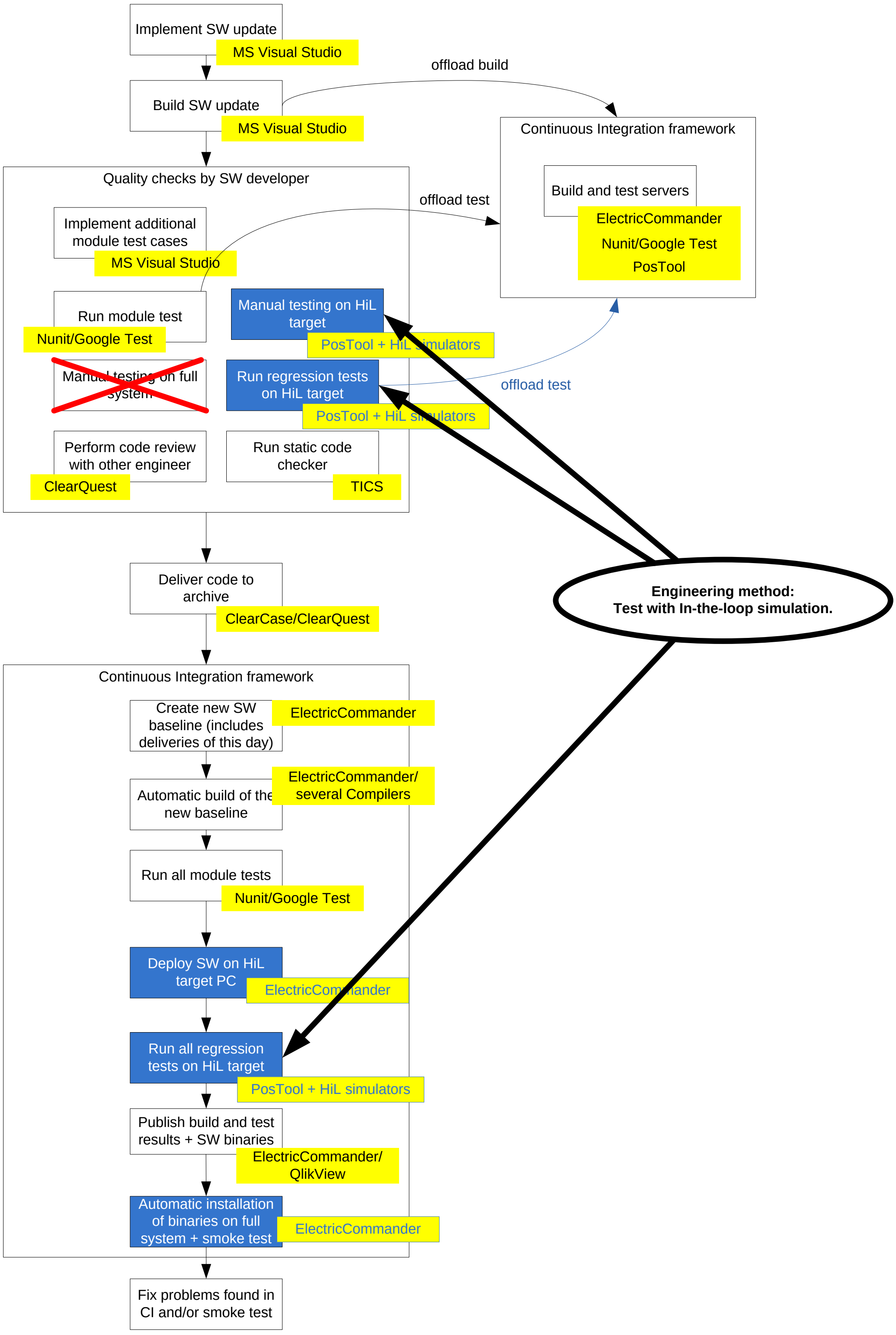


Figure D-5: System Verification process (proposed).

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).

D.3 Identification of Engineering Methods



Implement SW update
MS Visual Studio
Build SW update
MS Visual Studio
offload build
Continuous Integration framework
Build and test servers
ElectricCommander
Nunit/Google Test
PosTool
Quality checks by SW developer
offload test
Implement additional module test cases
MS Visual Studio
Run module test
Nunit/Google Test
Manual testing on HiL target
PosTool + HiL simulators
Manual testing on full system
Run regression tests on HiL target
PosTool + HiL simulators
offload test
Perform code review with other engineer
ClearQuest
Run static code checker
TICS
Deliver code to archive
ClearCase/ClearQuest
Engineering method:
Test with In-the-loop simulation.
Continuous Integration framework
Create new SW baseline (includes deliveries of this day)
ElectricCommander
Automatic build of the new baseline
ElectricCommander/ several Compilers
Run all module tests
Nunit/Google Test
Deploy SW on HiL target PC
ElectricCommander
Run all regression tests on HiL target
PosTool + HiL simulators
Publish build and test results + SW binaries
ElectricCommander/ QlikView
Automatic installation of binaries on full system + smoke test
ElectricCommander
Fix problems found in CI and/or smoke test

Figure D-4: 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,...).

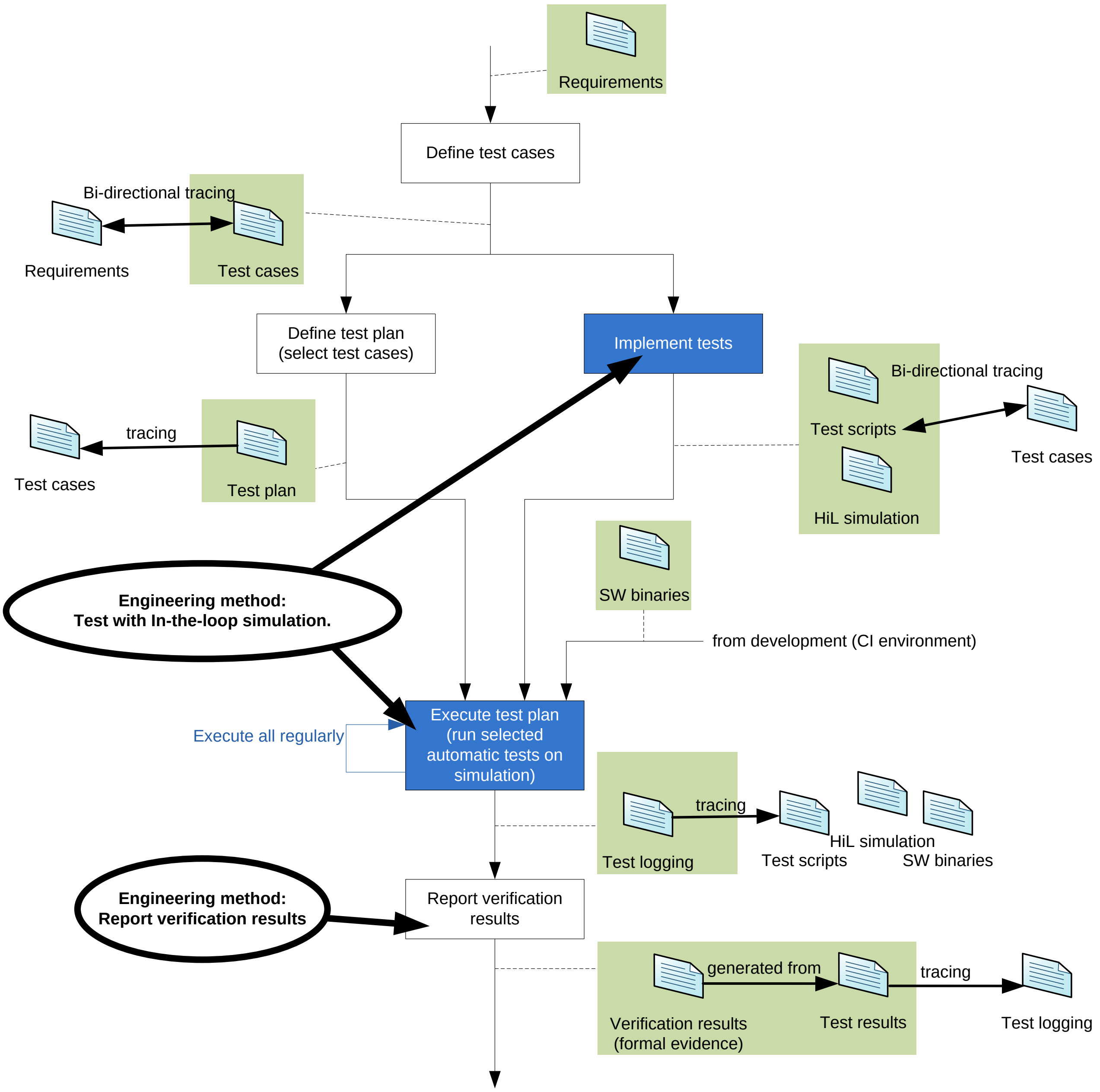


Figure D-6: identification of engineering methods (system verification).

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.

D.4 Technical case studies

In the context of Crystal Philips Healthcare wants to perform activities in the following areas to improve software testing via HiL-simulation:

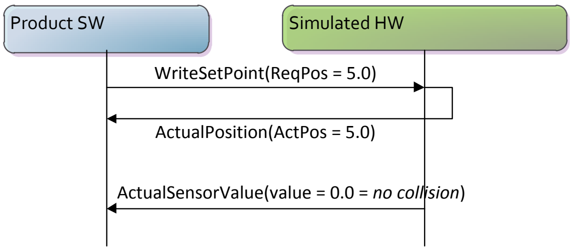
(A) Improve the simulation models of the hardware.

(B) Enable real-time execution of product software and simulated hardware.

D.4.1 Improving simulation models

A.0 The current status of simulation models

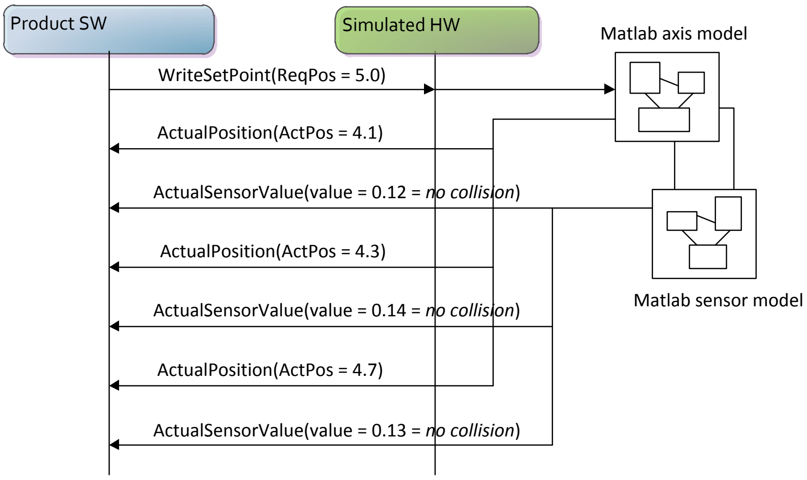
Currently an extremely simple simulation of the hardware is implemented (depicted below). Setpoints sent to position an axis (an axis is a drive/motor/load combination) are immediately returned as actual position reached by the axis. The sensors typically return some constant *good weather* value, indicating that there is no collision.


Product SW
Simulated HW
WriteSetPoint(ReqPos = 5.0)
ActualPosition(ActPos = 5.0)
ActualSensorValue(value = 0.0 = *no collision*)

A.1 Create more realistic models of hardware behavior with Matlab/Simulink

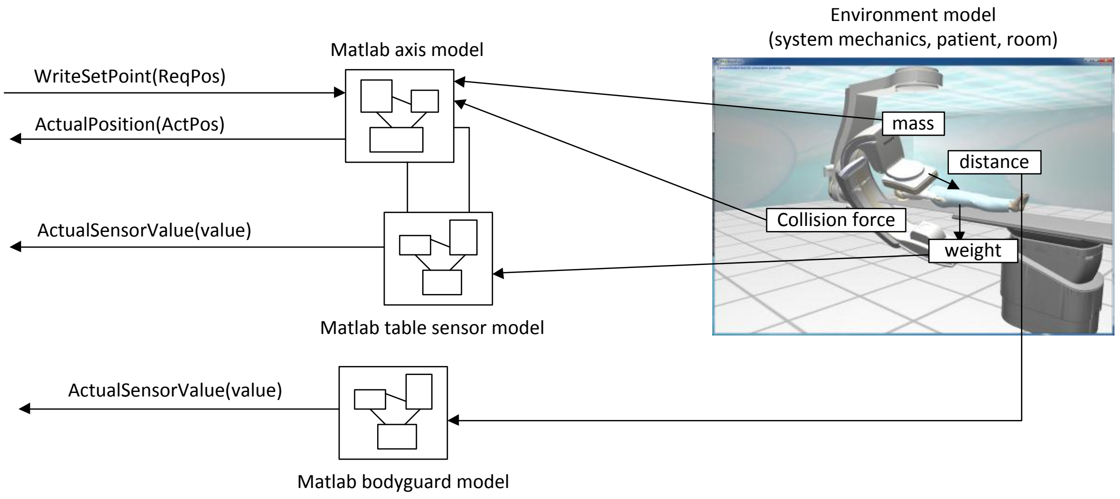
A first step towards a better simulation is to create a model of a PID-controller with the correct axis parameters, and sensor models with realistic signal characteristics, based on the hardware specs. The models interact with the software and can also interact with each other if there is some direct relation between them. See Section 0D.5 for an example of an axis and a sensor use case: the table force sensor.

Matlab/Simulink is a tool already used by system designers within iXR to do standalone simulations in the analysis phase of a project. These models typically have a very short lifetime; they are discarded once the analysis is done. We want to aim for reuse of models. Axis models used to prove the feasibility of a design should be useable as hardware simulation when testing the realization of that design.


Product SW
Simulated HW
Matlab axis model
Matlab sensor model
WriteSetPoint(ReqPos = 5.0)
ActualPosition(ActPos = 4.1)
ActualSensorValue(value = 0.12 = *no collision*)
ActualPosition(ActPos = 4.3)
ActualSensorValue(value = 0.14 = *no collision*)
ActualPosition(ActPos = 4.7)
ActualSensorValue(value = 0.13 = *no collision*)

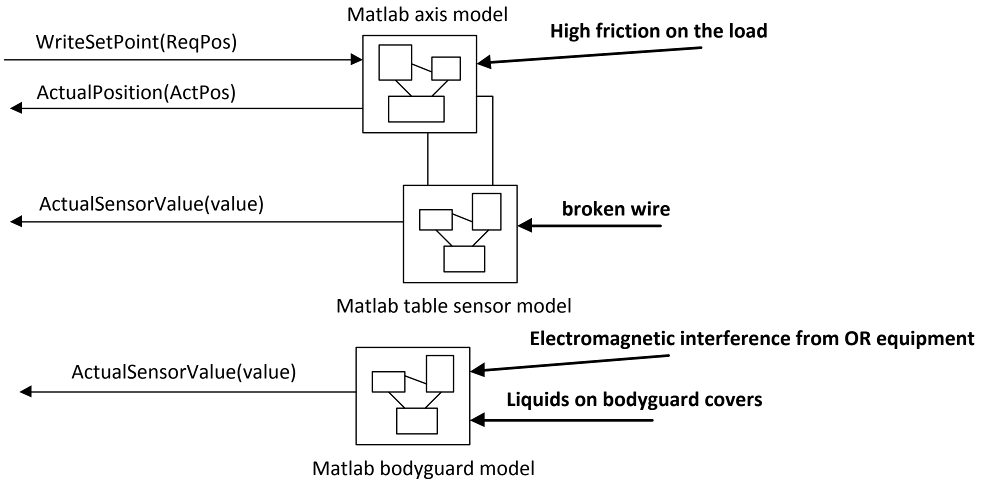
A.2 Couple Matlab/Simulink models to an environment model

A next step is to couple these Matlab models to an environment model as shown below. Instead of artificially triggering a collision on a sensor, the collision now occurs because some condition is fulfilled in the environment model that mirrors the collision condition in the real world. This can make the simulation more realistic and increase the ease of use of the simulation environment.



A.3 Fault injection/non happy-flow modeling

Another next step to enhance the models is to do fault injection. This way the software can be verified against non-ideal motors, safety requirements related to defect hardware can be verified, etc.



D.4.2 Enable real-time execution

We start by giving an overview of the different hardware motion platforms currently used in the interventional X-ray department of Philips Healthcare. Each of these platforms is a candidate for HiL simulation.

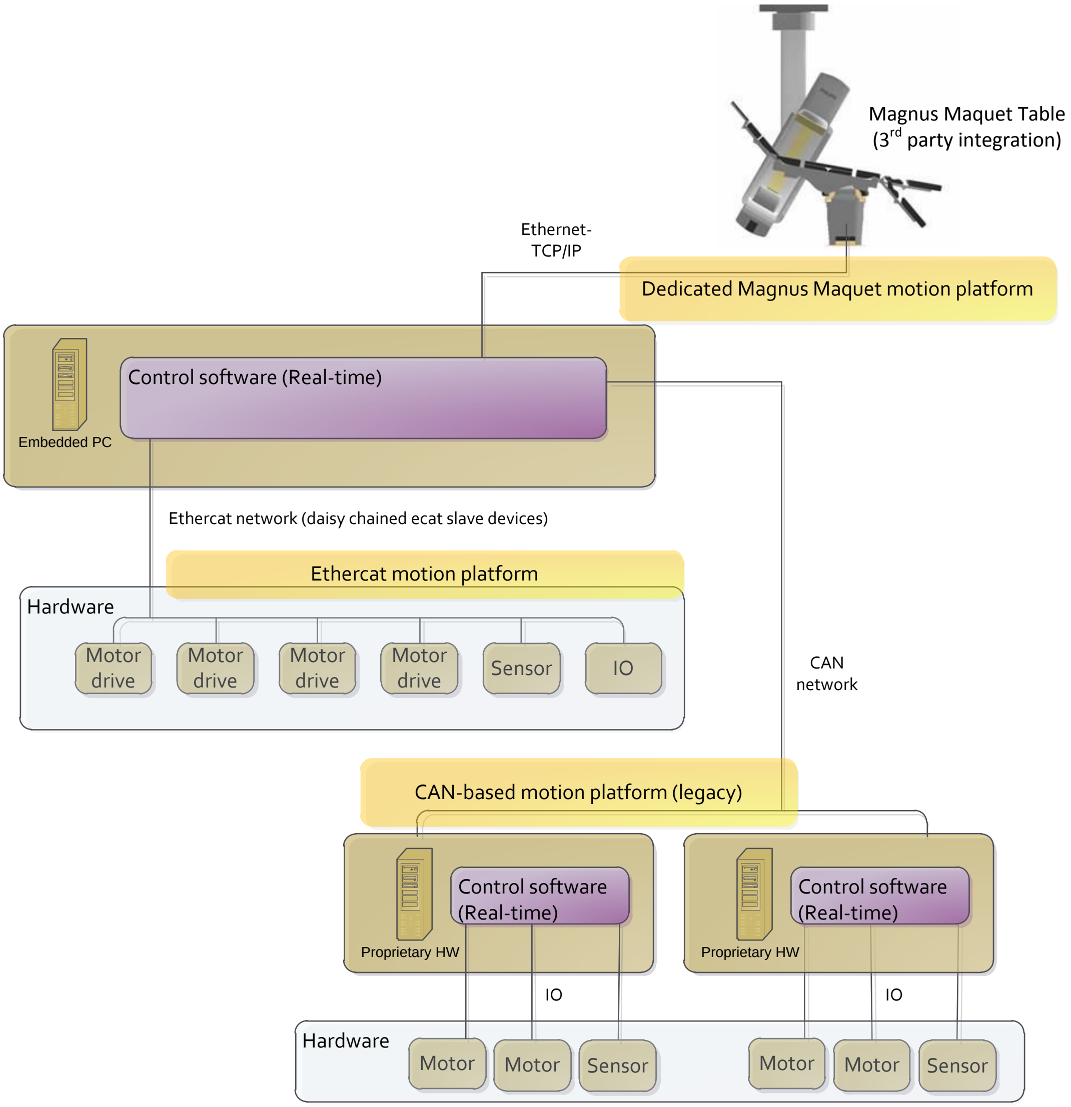


Figure 5-7: overview of hardware motion platforms.

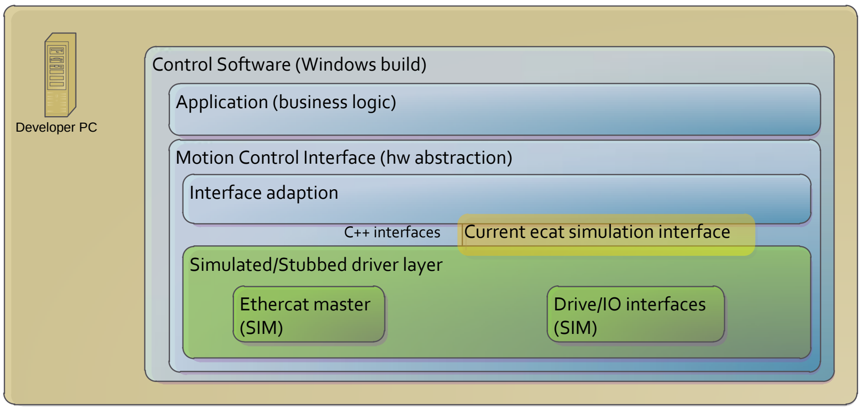
Currently the following three motion platforms are supported:

1. **Ethercat platform**: the main platform used for new developments. Real-time ethernet connects the centralized control software to motor drives and io devices (outsourced). Supports highly synchronized motion enabling the new multi-axis movements to be developed. Since most new developments target this platform this is the main candidate for HiL simulation.
2. **CAN-based platform**: legacy platform. Still in production, new developments are limited. This platform consists of a distributed control architecture, using a proprietary hardware motion infrastructure. The amount of test systems supporting this platform will decline the coming years, which is a driver for doing simulation here.
3. **Magnus Maquet platform**: dedicated platform to let the interventional X-Ray system interface with a 3rd party OR-table. There is a very limited amount of test systems with this type of table, which is the main driver for doing simulation here (a simple simulator developed by Philips Healthcare is already available).

The ethercat platform will be our main target for simulation. First steps to improve the simulation of the hardware will be done on that platform.

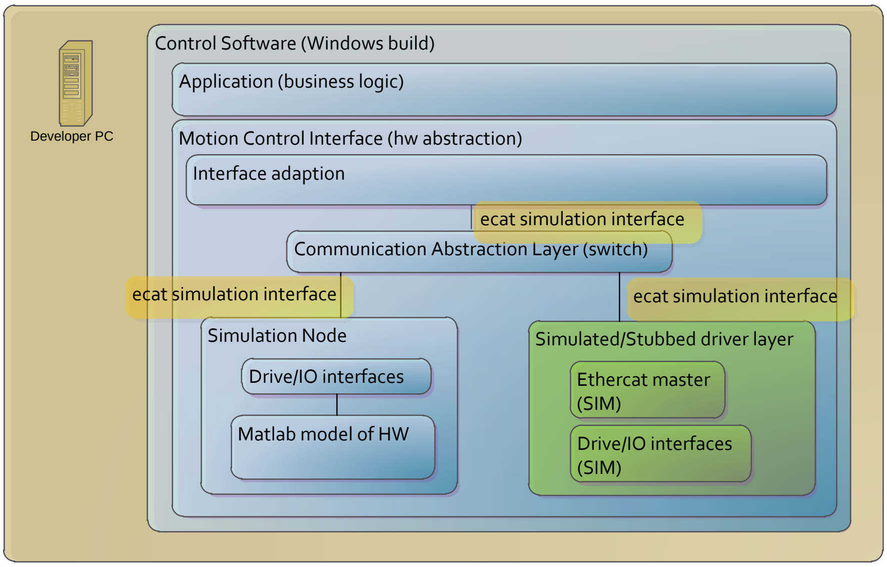
B.0 Current status of (ecat) simulation environment

The current ethercat platform simulation consists of a Windows (instead of RTOS) build of the software running on a developer PC (instead of target embedded PC), where the driver layer has been replaced by a very simple simulation (see A.0).


Developer PC
Control Software (Windows build)
Application (business logic)
Motion Control Interface (hw abstraction)
Interface adaption
C++ interfaces
Current ecat simulation interface
Simulated/Stubbed driver layer
Ethercat master (SIM)
Drive/IO interfaces (SIM)

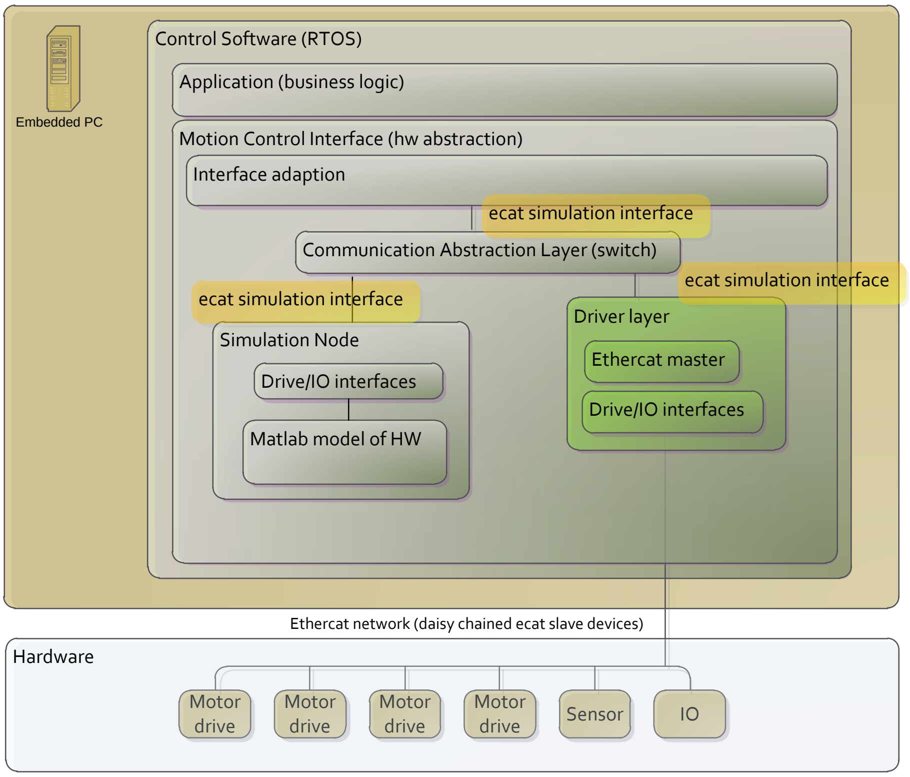
B.1 Addition of Communication Abstraction Layer

A first step in improving the simulation environment is the addition of a Communication Abstraction Layer. The main function of this component is to provide an implementation of the driver layer interfaces that handles the communication with the environment where the simulation models run (can be another PC). In this first step the simulation environment is built in the product code (as is the case for the current simulated/stubbed driver layer). Next to this the Communication Abstraction Layer will provide *switch* functionality to enable mixed simulations.


Developer PC
Control Software (Windows build)
Application (business logic)
Motion Control Interface (hw abstraction)
Interface adaption
ecat simulation interface
Communication Abstraction Layer (switch)
ecat simulation interface
ecat simulation interface
Simulation Node
Drive/IO interfaces
Matlab model of HW
Simulated/Stubbed driver layer
Ethercat master (SIM)
Drive/IO interfaces (SIM)

B.2 Real-time execution in product code RTOS environment

The next step is to build the simulator for the RTOS. This enables real-time execution of models, real-time execution of the product code on the actual target PC in a simulation mode, and this setup supports mixed simulations where only some motors and sensors are simulated.

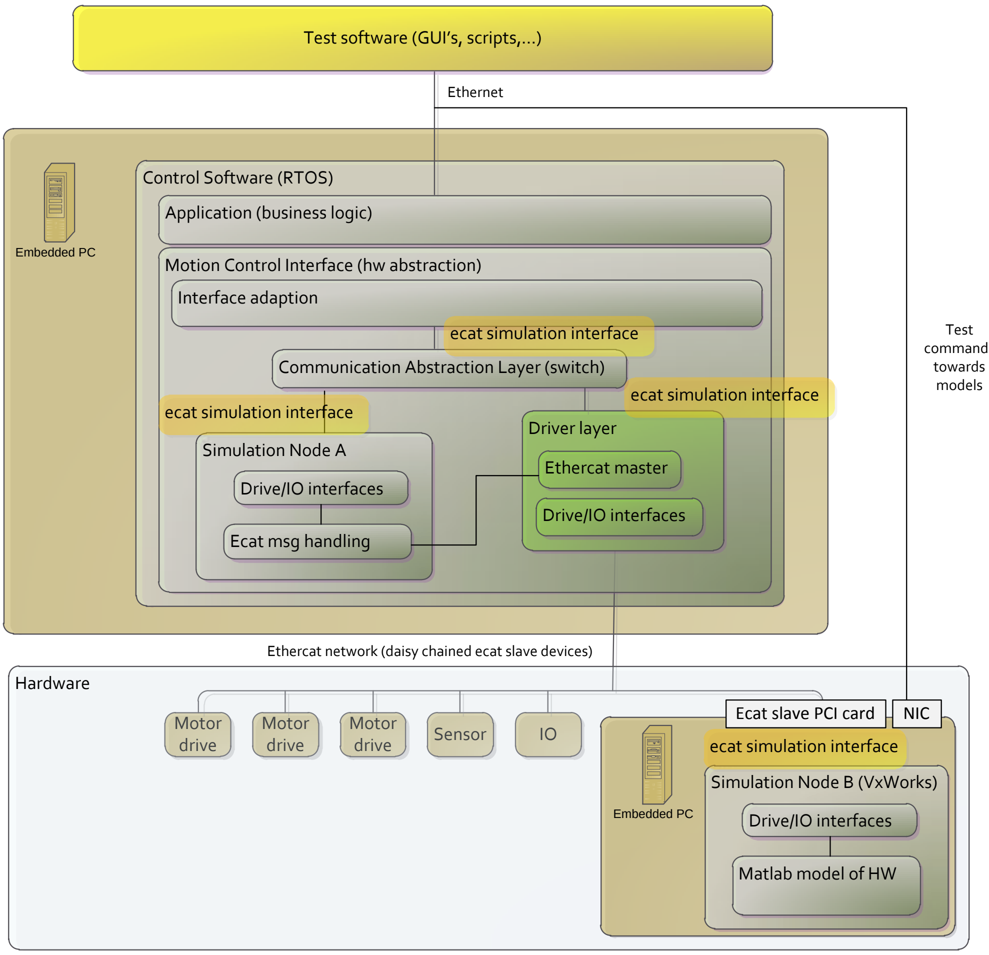

Embedded PC
Control Software (RTOS)
Application (business logic)
Motion Control Interface (hw abstraction)
Interface adaption
ecat simulation interface
Communication Abstraction Layer (switch)
ecat simulation interface
ecat simulation interface
Driver layer
Ethercat master
Drive/IO interfaces
Simulation Node
Drive/IO interfaces
Matlab model of HW
Ethercat network (daisy chained ecat slave devices)
Hardware
Motor drive
Motor drive
Motor drive
Motor drive
Sensor
IO

B.3 Offloading simulation to ethercat slave PC

In the setup described above the resources available to the RTOS must be shared by product code and simulation. When the simulation gets more complex, and hence cpu intensive, we may run out of resources. Before we get to this point we may have already have the situation that the timing of the product code is not representative anymore, because of scheduling together with the simulation.

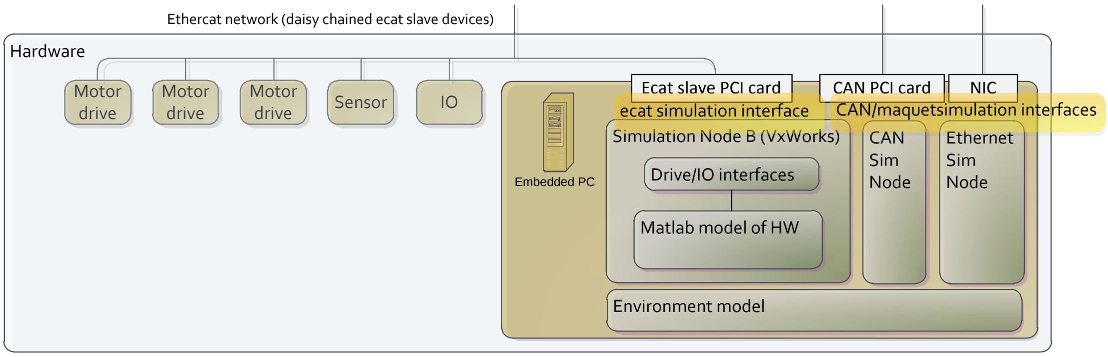
A solution, depicted below, is to offload the simulation to another PC. We need a real-time communication mechanism between the product code and the simulation: ethercat. There are PCI slave cards available turning a PC into an ethercat slave.

The test environment can control the simulation via an ethernet network interface.


Test software (GUI's, scripts,...)
Ethernet
Embedded PC
Control Software (RTOS)
Application (business logic)
Motion Control Interface (hw abstraction)
Interface adaption
ecat simulation interface
Communication Abstraction Layer (switch)
ecat simulation interface
ecat simulation interface
Simulation Node A
Drive/IO interfaces
Ecat msg handling
Driver layer
Ethercat master
Drive/IO interfaces
Test commands towards models
Ethercat network (daisy chained ecat slave devices)
Hardware
Motor drive
Motor drive
Motor drive
Sensor
IO
Ecat slave PCI card
NIC
ecat simulation interface
Embedded PC
Simulation Node B (VxWorks)
Drive/IO interfaces
Matlab model of HW

B.4 Add simulation of Ethernet and CAN motion platforms

By adding a CAN card to the simulation PC, the CAN and Ethernet based motion platforms can also be simulated. Note that the different motion platform models executed will share a real word model (c-arc is for instance controlled via ethercat, while the table is controlled via Ethernet).



.

D.5 Example use case: 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.

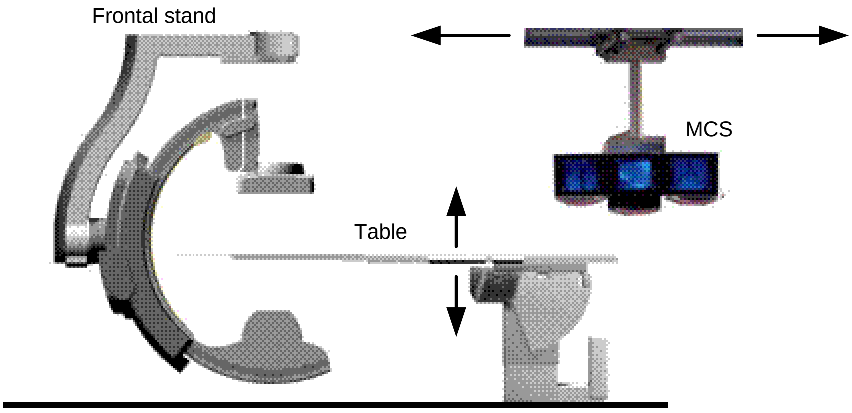


Figure D-8: Monitor Ceiling Suspension.

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

D.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. 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.
- 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.