28th annual INCOSE international symposium
Washington, DC, USA
July 7 - 12, 2018

Reverse engineering a legacy software in a complex system: A systems engineering approach

Maximiliano Moraga
University College of Southeast Norway
Kongsberg, Norway
+47 94195982
moraga.max@gmail.com

Yang-Yang Zhao
University College of Southeast Norway
Kongsberg, Norway
+47 31009699
yangyang.zhao@usn.no

Abstract. In a complex system, a legacy software as a component is determined by various factors beyond its own capability. Lack of knowledge that shaped software, which is often the case of a legacy software, can prohibit appropriate maintenance and development to comply with the system needs. To reverse engineering legacy software for a fit with the overall system of interest is a daunting task. Existing techniques of reverse engineering are mostly from a purely technical point of view and for the single discipline of software engineering. Thus, this paper aims for an approach to properly reverse engineer the reasoning behind the legacy software developments in a complex system. By jointly apply the CAFCR model and the reverse engineering, a roadmap is created to guide incremental developments of legacy software in a complex system, which benefits both the maintenance of existing implementation and realization of new functionalities for improved system performance.

Introduction

Software development has the growing importance for many business successes. One critical issue for an existing business is the maintenance and continuous development of its software. With increasing competition, existing businesses have a tremendous pressure on the fast pace upgrading which left no time for the software to be re-created and re-implemented. In many cases, software components have been maintained and incremental improved over the years. Software engineers are usually missioned to upgrade those software capabilities to keep up with the product or process improvement. However, changes in technology and market as well as people turn-over are continuously challenging the efficiency and efficacy for each software component upgrading. For example, people turn-over could challenge the rationale behind existing design and architecture, the continuation of documentation and the traceability from software components to the overall system drivers. These risks would affect the development of new features, maintainability of existing software and estimation of the related work.

A legacy software, is a software where is not possible to understand all the fundamental concepts that shaped it as they could be neither available nor existent for understanding (Pressman, 2005). It is also a common characteristic among legacy software to have poor quality. The software itself is affected by the proper technical problems, which may be addressed with different reverse engineering techniques (Feathers, 2004) (Chikofsky & Cross II, 1990) (Ducasse & Pollet, 2009) (Belle, Boussaidi, & Kpodjedo, 2016) (Müller & Orgun, 1993). It is also affected by the problems beyond the software architecture and its realization, which resides in reverse engineering why software has been developed and architected in the specific manner to satisfy the system needs. In a complex system, the legacy software is a component among thousand others and its upgrading is determined by various change factors beyond its own capability. Thus, in a complex system, the capability to continue working on a legacy software developed many years ago and with many past environmental changes is extremely difficult but a daunting task.

The system complexity, lack of history, poor quality of the software and its subparts challenge engineers while identifying which parts should be addressed for maintenance and future development of the software. This is a typical problem for a legacy software development.

To understand why a legacy software component was shaped in any specific form in the past, it is critical that engineers have the holistic perspective of the overall system goal. In most cases where both system and software keep evolving, software engineers can deal with the technical problems of legacy software in different ways, but has difficulties to reverse engineering to the software component for the fit with the rest of the system. To address the fit of the reverse engineering of legacy software, we need to answer the following questions: “What are the important parts of legacy software for the fit with the system context?” and “How to align legacy software maintenance and development for the fit with the entire system development?”.

Literature and Knowledge Applied

The contentious development of legacy software requires a throughout understanding of the software and the big picture of system. To resolve the missing links between software components and system drivers is key to develop the software for the fulfilment of system goals. Existing techniques of software reverse engineering, covers the issue mostly for the single discipline of software engineering (Feathers, 2004) (Chikofsky & Cross II, 1990) (Ducasse & Pollet, 2009) (Belle, Boussaidi, & Kpodjedo, 2016). In a complex system, however, the software is a component of something bigger, whose usage is determined by factors beyond its own capability. In a complex system, different disciplines, sub-systems and their components interact to fulfil a system goal. Furthermore, software as a component posits new challenges in software reverse engineering. Existing methods that treat the software as a whole system cannot be sufficient to reverse engineer the software for fulfilling the needs of an overall system. Thus, a proper reverse engineering of the software which is part of a complex system cannot be accomplished by the means of software engineering alone and without an approach for complex system.

Systems engineering is known as the working methods for resolving complex systems’ issues. According to INCOSE (2016) (INCOSE, 2017), Systems Engineering (SE) is defined as an engineering discipline whose responsibility is creating and executing an interdisciplinary process to ensure that the customer and stakeholder's needs are satisfied in a high quality, trustworthy, cost efficient and schedule compliant manner throughout a system's entire life cycle. Systems engineering uses Systems Thinking (ST) to articulate the reasoning for creating and applying SE processes systematically (SEBoK authors, 2017). ST is a set of founding ideas for the development of systems theories and practices and also as a pervasive way of thinking need by those developing and applying them. Therefore, to address legacy software for its system fit, it is necessary to utilize ST for reverse engineering the reasoning of the software within its parent system. In our case, to resolve the fit between a legacy software component and a complex system is to reverse engineer the reasoning how the software component was developed according to the system needs for satisfying the stakeholder’s requirements. To visualize the system reasoning, system architecture is often used for applying systems thinking to represent systems.

According to the ISO/IEC/IEEE 42010 (ISO/IEC/IEEE, 2011) standard:

System Architecture is the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution

The ISO/IEC/IEEE 42010 (ISO/IEC/IEEE, 2011) is an international standard for the description of systems and software engineering architecture methods. CAFCR (Muller, 2004) (Customer Objectives, Application, Functional, Conceptual, Realization) is a framework for defining systems architectures, which is defined as an instance of a former version of the ISO/IEC/IEEE 42010 standard. It mainly defines architecture descriptions and a set of best practices for defining architecture frameworks. However, there are few research approaching cases of linking legacy software with SE. Despite CAFCR had never been proved in a research for dealing with legacy software, it is a proven tool for representing entire systems containing software as a subcomponent. The application of CAFCR is thus employed in this research to represent the reasoning behind the interconnections of legacy software and the system in its environment. CAFCR (Muller, 2004) addresses the systems architecting as a decomposition of the system elements contextualized in its environment. An implementation of the CAFCR framework is illustrated in Figure 1.

- The first layer at the top represents the views to decompose system architecture. These views are intended to be later complemented and integrated by the definition of qualities as the key elements that affect and interconnect the entire system. The five views are defined as: Customer objectives, Application, Functional, Conceptual and Realization view. Customer objectives and application views are the justification of the system, the why and how the customer foresees the system. Functional View represents what the system is, from an outside of the system perspective, as an interface to its context. Conceptual and realization views represent how the system is, separated in two views for stability purposes, since the realization is expected to change faster than the conceptual view.
- The second layer, called sub-methods, typically includes different modeling techniques such as context diagrams or functional models to represent the views defined in the first layer. No specific sub-methods are enforced by CAFCR. These are represented in the figure with rectangles.
- The third layer of qualities is the integration of the views using the main qualities of the system. Qualities as M*aintainability* or *Performance* are evaluated through the system views for identifying those that are affected by the different qualities, it is usually represented as needles going from the top of the system down to its lower views picking up all views related to them on their way down, in here we can see them represented as the long arrows crossing all views.
- The last layer describes the reasoning between views. This reasoning is the application of a set of steps, in order to identify most interconnections between system contexts, physical and logical components, system drivers and qualities. At the end it is the representation of the reasoning behind the system architecture. It allows engineers to have a model illustrating the reason why the system was developed in any specific form.

CAFCR also provides other layers of decomposition that is not within this research's scope, such as storytelling and life-cycle view which was included in a later version of the CAFCR framework.

CAFCR is the tool that describes the entire system architecture within its context, connecting between the key customer drivers to its lower and smaller pieces. CAFCR explains a transversal reasoning of the system architecture. Traditionally CAFCR is used for architecting systems with software components but is has not been customized or verified for systems containing legacy software. Since CAFCR helps describe a system with its different views and their relation in-between, it is possible to enable an architect to relate a software component and its sub-components to the system views. Despite ISO/IEC/IEEE 42010 has mentioned the possible use of architecture descriptions for representing legacy architectures, few research has investigated how to address that. Therefore, there is a need to extend the usage of CAFCR by integrating the use of specific reverse engineering techniques to enable the maintenance and development of legacy software in alignment to the evolution of the entire system. Next section illustrates the detailed proposition for extending CAFCR capabilities.

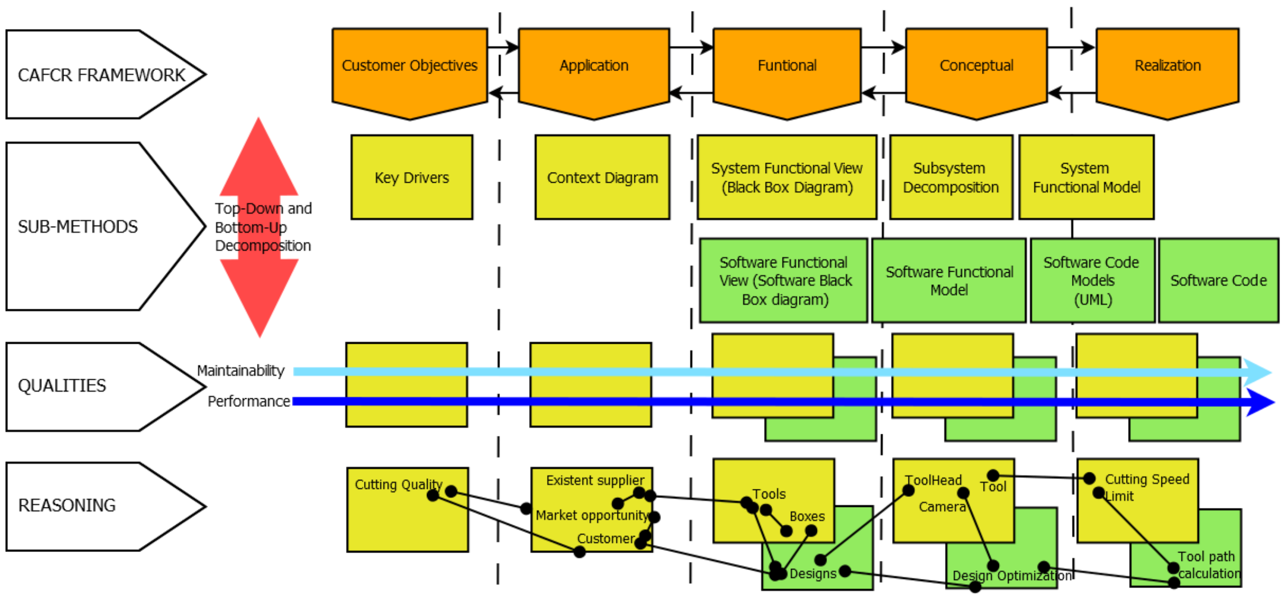
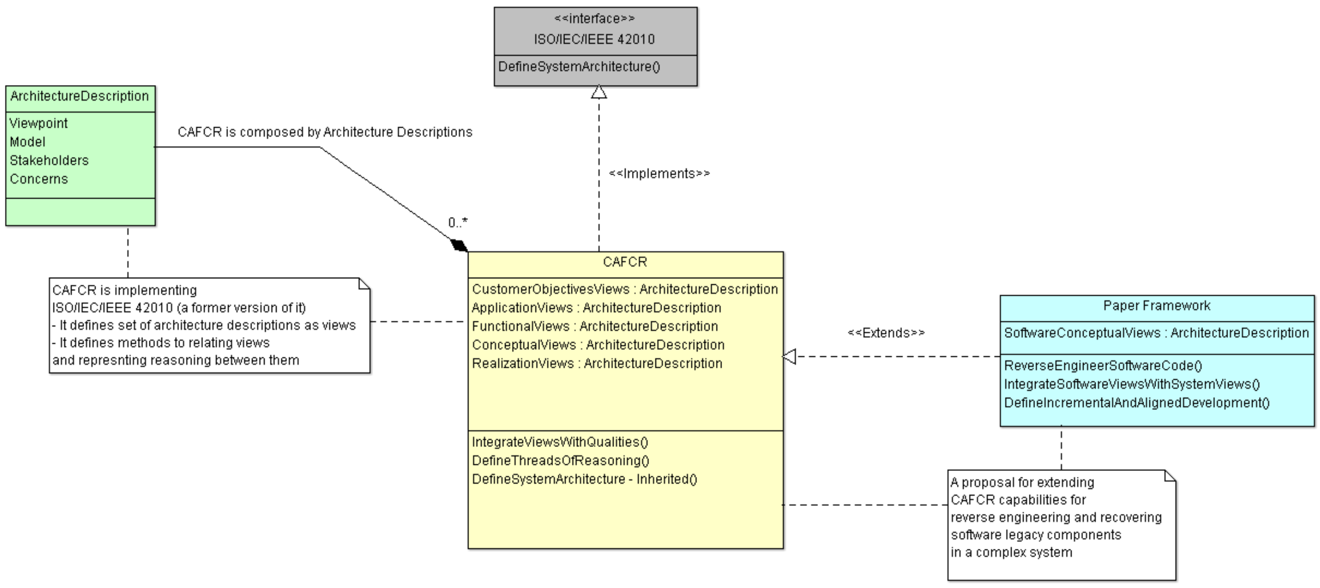


Figure 1. Example of CAFCR implementation

The Conceptual Solution

In spite of a wide application of CAFCR (Muller, 2004) as a systems architecting tool, it is inconclusive how it work for reverse engineering legacy software in a complex system. Based on both literature of reverse engineering and systems engineering, we propose the extension usage of CAFCR with this specific application. Figure 2 represents the relationship of the standard ISO/IEC/IEEE 42010 with CAFCR and a proposed extension of it in an object diagram (Purchase, Colpoys, McGill, & Britton, 2001): the standard in the grey box is the interface defining “what” should a system architecture framework represent; CAFCR in the yellow box is the implementation of this interface, defining “how” to represent the system architecture by creating architecture descriptions (defined in the green box) of the system and its components. Figure 2 inherits all capabilities of CAFCR and extends it with specific capabilities for decomposing the architecture of a legacy software component in the context of the entire system architecture. This extension of CAFCR is a conceptual solution to reverse engineer the reasoning behind legacy software to link it up with the system reasoning for resolving the fit of the software maintenance and development with the system evolution.



<<interface>>
ISO/IEC/IEEE 42010
DefineSystemArchitecture()
ArchitectureDescription
Viewpoint
Model
Stakeholders
Concerns
CAFCR is composed by Architecture Descriptions
0..*
<<Implements>>
CAFCR
CustomerObjectivesViews : ArchitectureDescription
ApplicationViews : ArchitectureDescription
FunctionalViews : ArchitectureDescription
ConceptualViews : ArchitectureDescription
RealizationViews : ArchitectureDescription
IntegrateViewsWithQualities()
DefineThreadsOfReasoning()
DefineSystemArchitecture - Inherited()
CAFCR is implementing ISO/IEC/IEEE 42010 (a former version of it)
- It defines set of architecture descriptions as views
- It defines methods to relating views and represnting reasoning between them
<<Extends>>
Paper Framework
SoftwareConceptualViews : ArchitectureDescription
ReverseEngineerSoftwareCode()
IntegrateSoftwareViewsWithSystemViews()
DefineIncrementalAndAlignedDevelopment()
A proposal for extending CAFCR capabilities for reverse engineering and recovering software legacy components in a complex system

Figure 2. Object-oriented representation of the conceptual solution

The conceptual solution should guide plans for the maintenance and development of legacy software in line with the system development and evolution. An overview of the proposed solution is illustrated in Figure 3. It is divided in two parts. The first part represents the system and software decomposition and integration powered by applying CAFCR, except for the activity 1.2 regarding software reverse engineering, which recommends a bottom-up decomposition. The second part is about the definition of plans for further development and the application of them.

a. Part 1: Decomposition, integration and reasoning reverse engineering

The activities presented in yellow boxes are with the backbone of the CAFCR framework, which are applied to decompose the system in order to identify the reasoning behind system and software architecture. The activities aren't necessarily sequential, but the given order is recommended as some enable others. The sequence represented by the orange arrow is circular for denoting the possibility of recursively applying the activities if needed. The black lines identify which activities enables others.

Activities 1.1 and 1.2. These activities aim to decompose the system and the software into the 5 views defined by CAFCR. The system can be decomposed as a combination of top-down and bottom-up for the most benefits. In the case that the legacy state of the software requires a direct decomposition and generation of views from the source code, a bottom-up approach is necessary for decomposing the legacy software. Furthermore, any modeling approach for representing the CAFCR views need to be evaluated for the specific needs or constraints of the system. The important goal of these steps is to represent the system and its software as closest as possible to reality.

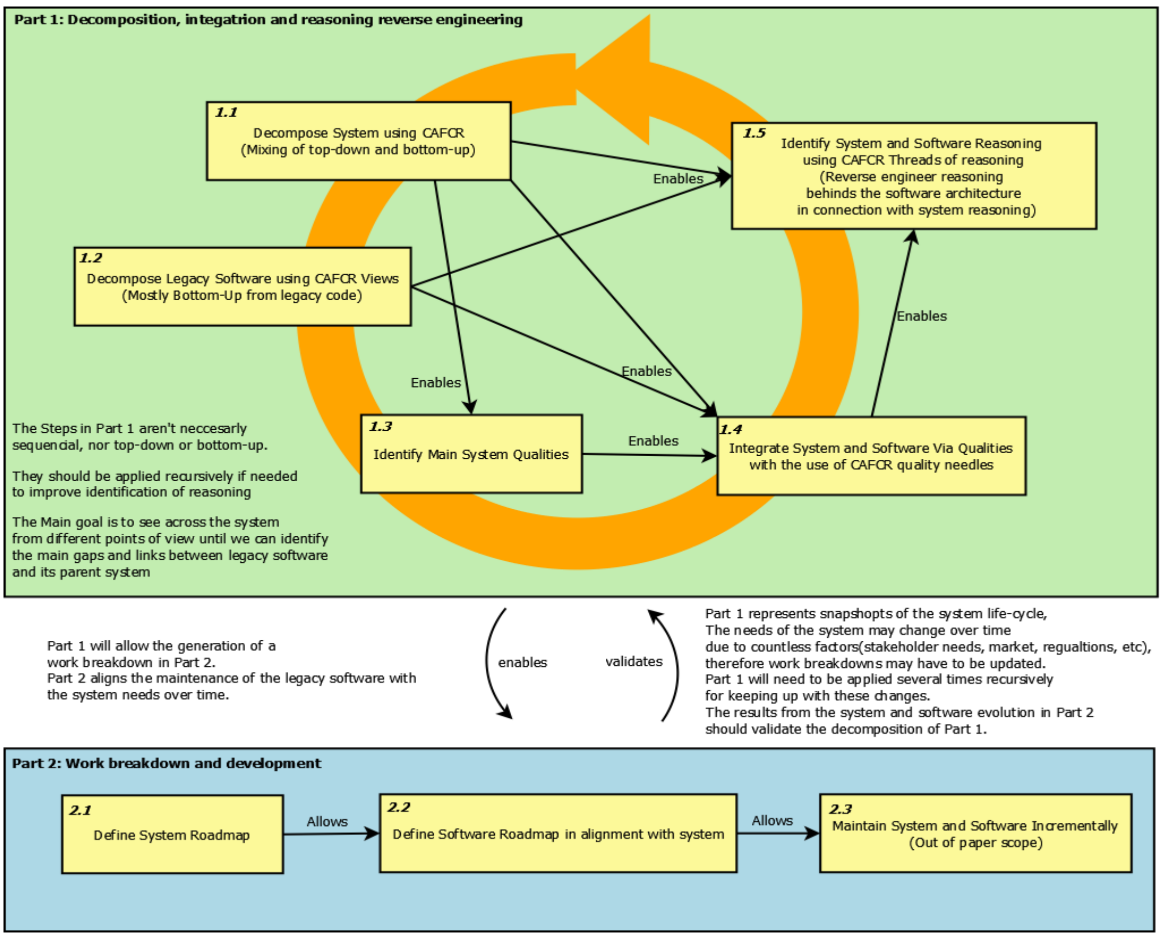
Activity 1.3 and 1.4. These activities are to identify the main system qualities and integrate the system and software across all views. Despite CAFCR provides a long list of possible qualities in a qualities checklist to help engineers identify, the qualities need to be identified from the decompositions in activity 1.1 and they are presented in any decomposition at any CAFCR view level. CAFCR integrates views by using quality needles which is about identifying views influenced by the different qualities, like pushing qualities as a needles down through all views affected by the selected quality. The identified qualities will allow the architect to integrate the different views of the system and software. The integration via qualities is the first step in reverse engineering the reasoning behind the legacy software architecture.

Activity 1.5. The CAFCR views and their integration with qualities represent incomplete pictures of the system. In order to expand the visibility of relations across the entire system, CAFCR defines the use of threads of reasoning as an overall integration of all system views and qualities. Threads of reasoning are a method that covers the relationships between views and qualities for identification of problems, gaps and possible solutions. In our case, the reasoning behind legacy software component is unknown, therefore the need for reverse engineering is to use threads of reasoning to identify the gap between software and system architectures. The identified gaps will allow engineers to identify software sub-components that are critical for the system success. It will also relate to a connection between specific input in the functional view to a function definition in the conceptual view and the actual source code of the function in the realization view, which may not necessarily be a single direction connection across views, but in the form of many to many and within themselves. These connections are representing the reasoning that shape the software code in any specific manner, therefore they help recognize what important parts of the software for fulfilling system needs.

b. Part 2: Work breakdown and development

Activity 2.1. The goal is to tell how the system will be maintained and developed in the future described as a set of tasks over a timeline. Most systems have roadmaps or other kind of tools to describe the expected evolution. It may be compressed of several layers, such as market expected evolution, system and sub-components developments plans, even sub-sub components.
Activity 2.2. The creation of a roadmap is to place the expected maintenance and development of the software in context with the system evolution. It is important that the software roadmap not only contains tasks defining possible new functionalities, but guides the improvement or reimplementation of critical legacy software subcomponents that are needed for satisfying the system needs. This roadmap will align the critical software parts identified in Part 1 with new functionalities or other expected developments for the software.
Activity 2.3. The last activity in the roadmap is about the actual implementation of tasks for further working on the tasks defined for software and system.

It is expected that a system and its architecture will evolve over time, so will the models, integration and reasoning. The above activities generate only a snapshot in the system life-cycle for the purpose of producing a development plan of the software. As long as the system evolves, Part 1 should be continuously re-applied to keep the system architecture decomposition aligned with reality. The correct application of part 1 will enable the creation of plans in part 2 by identifying the critical parts of the software for fitting in the system goals. And, the success of part 2 will validate the decompositions made in part 1.



Part 1: Decomposition, integatrion and reasoning reverse engineering
1.1 Decompose System using CAFCR (Mixing of top-down and bottom-up)
1.2 Decompose Legacy Software using CAFCR Views (Mostly Bottom-Up from legacy code)
1.3 Identify Main System Qualities
1.4 Integrate System and Software Via Qualities with the use of CAFCR quality needles
1.5 Identify System and Software Reasoning using CAFCR Threads of reasoning (Reverse engineer reasoning behinds the software architecture in connection with system reasoning)
Enables
The Steps in Part 1 aren't neccesarly sequencial, nor top-down or bottom-up.
They should be applied recursively if needed to improve identification of reasoning
The Main goal is to see across the system from different points of view until we can identify the main gaps and links between legacy software and its parent system
Part 1 will allow the generation of a work breakdown in Part 2. Part 2 aligns the maintenance of the legacy software with the system needs over time.
enables
validates
Part 1 represents snapshopts of the system life-cycle, The needs of the system may change over time due to countless factors(stakeholder needs, market, regualtions, etc), therefore work breakdowns may have to be updated. Part 1 will need to be applied several times recursively for keeping up with these changes. The results from the system and software evolution in Part 2 should validate the decomposition of Part 1.
Part 2: Work breakdown and development
2.1 Define System Roadmap
Allows
2.2 Define Software Roadmap in alignment with system
Allows
2.3 Maintain System and Software Incrementally (Out of paper scope)

Figure 3. Overview of the proposed approach activities

Data and Methods

We adopt the case study approach in this research. It is the real-life case, that COMPANY A, a multi-national engineering firm in Norway, needs to maintain and develop legacy software components for system performance improvements. We collected data through six 1-to-1 and two 1-to-many interviews and more than twenty informal interviews, that varied from quick questions in the hall to asking for specific information. Firstly, we investigate the problem within COMPANY A's context and validate the problem in the specific case —SYSTEM X. Three 1-1 interviews were performed with experienced staff within SYSTEM X across different departments, like software engineers, customer support and managers at different levels. The collected data were intended to identify the actual needs of the software, and why it was necessary to keep working with this legacy component. Then the proposed conceptual solution is applied in the COMPANY A's SYSTEM X case to illustrate and verify the usage which enables the evolution of the legacy software for the fit with its system context. The interviews were performed at different stages while applying the solution as different information was needed to complete the activities in it. Three 1-to-1 interviews were performed during the application of the solution regarding R&D and production management. Data collected from these interviews are especially relevant for the application of the first three CAFCR views, customer objectives, application and functional views. They also provide good insights in system qualities for integration. The 1-to-many interviews were performed with experienced engineers working with the system. The first 1-to-many interview was about the decomposed system and software views and their missing spots for system improvement. The second one was actually a round of meetings to apply the CAFCR steps for discovering threads of reasoning. In addition, the informal interviews where mostly for consulting engineers on different aspects of the system were to develop the different views of the solution and their integrations. Finally, we conducted expert assessment to evaluate the results of the applied solution, with a short survey. There were 9 carefully selected experts from COMPANY A participating in the survey, which six are experienced Software Engineers, two R&D managers and the product manager of the SYSTEM X. The survey was used as an indication in how well experts assess the solution for both successes of the software and system development in this industrial case.

Case Study

Based on the proposed conceptual solution and real-life data, this section presents the case study while applying the solution to the SYSTEM X in COMPANY A. This case study aims to answer what are the important parts of the COMPANY A's legacy software for the fit with the expected system development and how to align legacy software maintenance and development for the fit with the entire SYSTEM X's development.

The proposed solution defines the utilization of CAFCR for decomposing the system and software in several views. CAFCR describes the use of sub-methods for representing the decompositions. These sub-methods may vary depending of the needs of the decomposition to be performed. E.g. a sub-method of the *Application view* from CAFCR could be a context diagram, or in the case of *Conceptual view* to use a functional model. Neither the proposed solution nor CAFCR enforces the use of any specific sub-method. Instead, the selection of sub-methods is given by COMPANY A's practices to better represent the system and its legacy software. However, the sub-methods applied in this case study are only a sub-set of all available sub-methods where applied, but those presented in the following subsections are intended for illustrating and verifying the applicability of the proposed solution on a real case.

Activity 1.1: System decomposition

Decomposing a system consists of breaking up a system in smaller parts for better understandings. The SYSTEM X can been decomposed into the C-A-F-C-R views (Muller, 2004) as illustrated in Figure 1. In Activity1.1, the SYSTEM X is decomposed into the four first views, *Customer objectives*, *Application*, *Functional* and *Conceptual* views. *Realization* view will be decomposed bottom-up from the legacy software component in Activity 1.2.

Customer Objectives. To understand why legacy software is developed in any specific way, first we need to know why the system tends to be developed in a certain way. To identify the key drivers of the system we have selected a sub-method that represents the extraction of them from actual system requirements by root cause analysis. The identification of system key drivers from the requirements is represented in Figure 4. The subset of key drives in the figure is to demonstrate how to apply this decomposition in a real system. By identifying the Customer key drivers, we can also recognize important qualities of the system, as for example *Versatility* and *Productivity*.

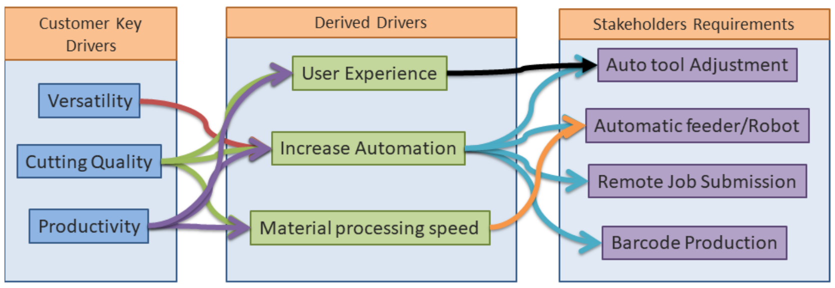


Figure 4. Key Drivers of System X

Application View. CAFCR defines *Application view* as the representation of how the customer foresees the system of interest. One of the methods in CAFCR is the use of a context diagram as a tool for understanding function allocation in the customer's context. Figure 5 is the representation of a context diagram for SYSTEM X. In Figure 5, main markets are at the top; the three blue ellipses in the middle-left represent the main competitors; the arrows identify in what market

COMPANY A competes with them. COMPANY A's suppliers are the ellipses at the bottom under COMPANY A, with some remarks on the software as the sub component of interest. In this view, the main markets are visible, *Sample/Single Making* and *Continuous Production* are defined. The identification of main markets is associated with important qualities to satisfy them.

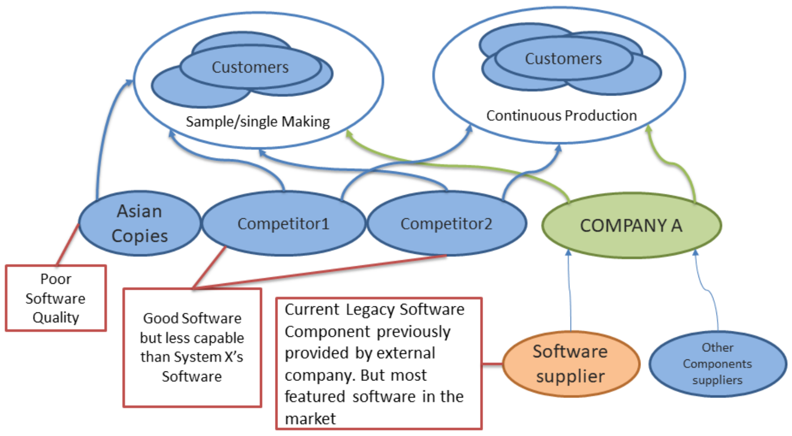

Customers
Sample/single Making
Customers
Continuous Production
Asian Copies
Competitor1
Competitor2
COMPANY A
Poor Software Quality
Good Software but less capable than System X's Software
Current Legacy Software Component previously provided by external company. But most featured software in the market
Software supplier
Other Components suppliers

Figure 5. Context Diagram for System X

Functional View. It is illustrated in Figure 6 as a black box vision of the system. It is an external point of view for identifying consumables or inputs, interfaces, restrictions, outputs. It represents the interface between the system and its environment as an illustration of all external entities affecting the shape of system implementation.

Conceptual View. The Conceptual View represents the internal decompositions of the system of interest. Figure 7 modularizes the entire SYSTEM X from a physical point of view. All or the most important components are included in this diagram. This diagram puts the components in the context within the system itself. The rectangles identify different components of the system, but our main interest is in the software highlighted with orange (Control Unit and HMI). On the other hand, Figure 8 illustrates one of the main operations of the system in a functional model. A functional model is to represent all functions involved in system functionality. It represents processing inputs, controls or output commands and the main logical internal functions or algorithms involved in system functionality. The use of both representations from Figure 7 and Figure 8 allows the traceability between physical and logical components, thus facilitates their linkages to the software.

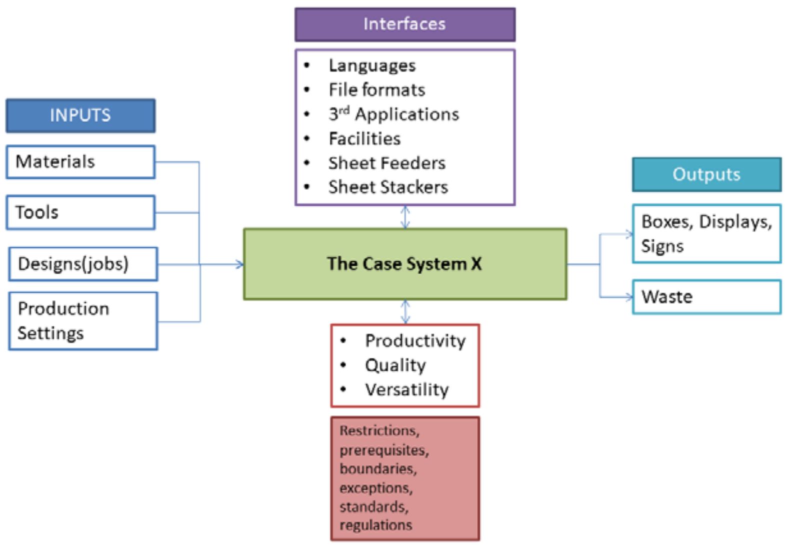


Figure 6. Black box representation of the System X

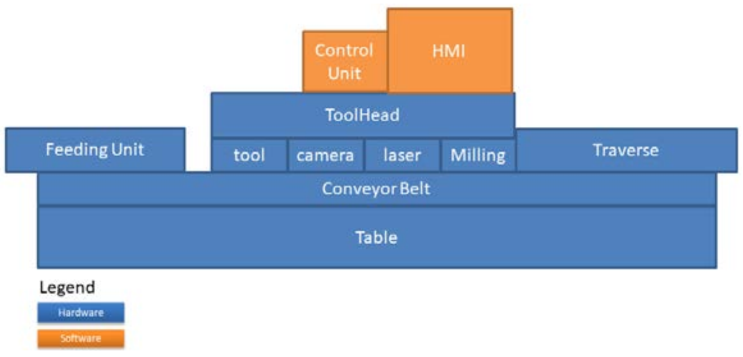


Figure 7. Sub-systems decomposition of System X

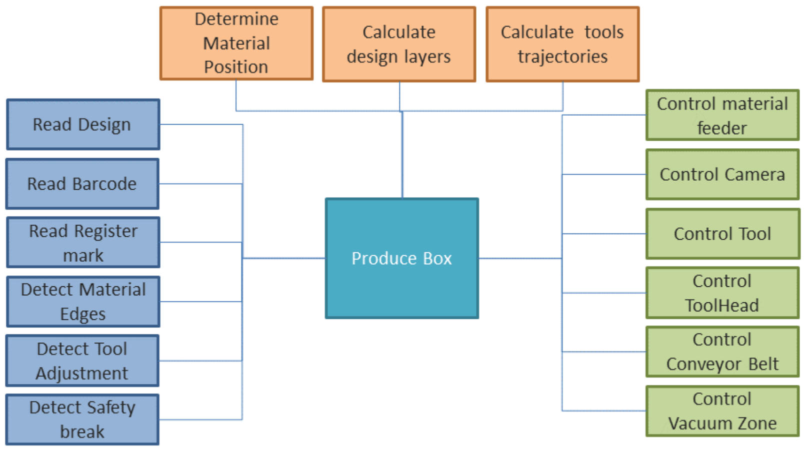


Figure 8. Functional models of System X

Activity 1.2: Software Decomposition

The proposed solution actively recommends starting decomposing legacy software with a bottom-up approach because most of the knowledge behind its architecture resides implicit in the code. Due to this decomposition from the bottom, the *Realization view* is described before the *Conceptual view*. Software engineers play a central role in reverse engineering from the software code into models that could be linked up to other system views.

Realization View. Most of the software decompositions are performed by tracing the code for identifying main components in it. These components are to be modelled by the use of UML (Purchase, Colpoys, McGill, & Britton, 2001). Reverse engineering is crucial in defining the main parts of the software for linking to system views, so they may be later maintained or reworked into better quality software to keep up with the system evolution. The challenge is to how the legacy software can be decomposed from the code so it can be later put in context with the rest of the system. In the case, engineers start with identifying objects in the code that performs most of the work during the process and interactions between them. Two main layers are found in the software, shown in Figure 9. Between these 2 layers, there exist many components. Furthermore, engineers are able to identify those components that have a bigger impact or participation in the main functionalities of the system. The light red box (*Production Manager*) is the software component, identified by the engineers, with most of the logic for the production of a box and the driving of the entire production. Figure 9 illustrates main logical objects identified in the reverse engineering of the software. The goal is to be able to know which parts of the software are important for the rest of the system.

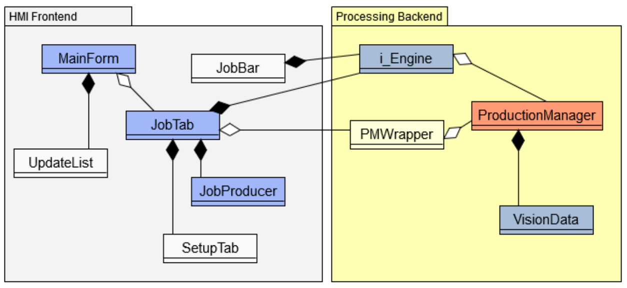


Figure 9. Simplified SYSTEM X Software UML Class Diagram

Conceptual View. By tracing the software code, it is possible to model the software code and entire functions within the software. Figure 10 represents one of the main functions of the SYSTEM X's software. It identifies main software functions (left blocks), operations (top blocks) and commands (right blocks) as the software was the main system. This model is mainly a combined analysis of software reverse engineering and the existing functionality of producing a box. It is performed very similar to the system decomposition for the conceptual view, but with a focus only in the software as it was the main system. Generating the legacy conceptual view is the most critical decomposition for generating the link between software and system. By combining software and system conceptual views, we can trace software elements to physical components, For example, Production Manager from Figure 9 is critical in the Produce box software function from Figure 10 since it contains the logic for Request a material Feed. This clearly connects to the system functional model, and to the physical component of a Sheet Feeder from Figure 10, which is important for the customer requirements related to automation.

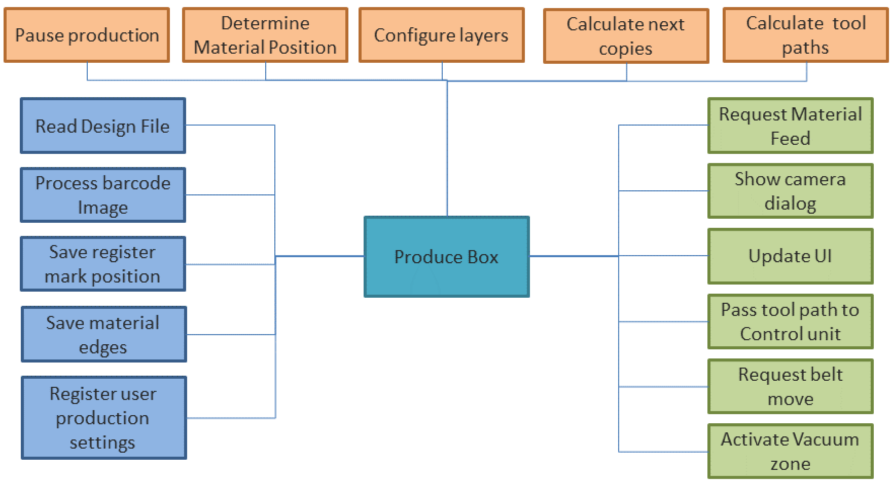


Figure 10. Software functional Model of System X

Activity 1.4: Integrate via Qualities

The Integration via qualities is defined by CAFCR as integrating needles. It focuses on making relationships between the CAFCR views, where qualities are used as needles to pick up all views that affect every quality. It is the first approach to give a context meaning to each system view. The integration with qualities can be applied for one or many qualities, depending on the purpose of the integration. In the case study, the integration is accomplished for all the main qualities identified in the previous activity, shown in Figure 11. In Figure 11, the bigger blocks represent the most important qualities for System X. Figure 12 displays the integration using *Productivity* across the five CAFCR views. The model intends to show in the white boxes what elements from the previous decompositions are critical for improving *Productivity*. In this case, *Productivity* links directly with some customer requirements and a specific market sector. We can identify some of the main interfaces and the main components affecting *Productivity*. While in realization view, we can start linking with specific software elements addressing this quality. When doing this process for other qualities, we are able to identify elements that might be important among several qualities. This model is a first step for detecting parts of the software that are critical for the system to satisfy its guiding qualities. It represents a discussion across all views for identifying those affecting a specific quality.

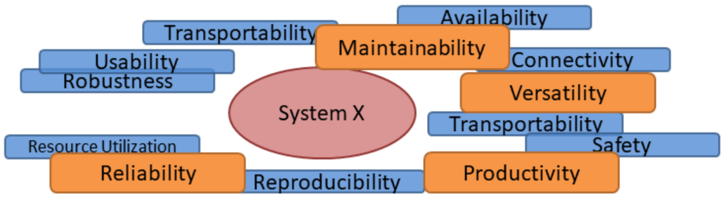


Figure 11. System X Qualities

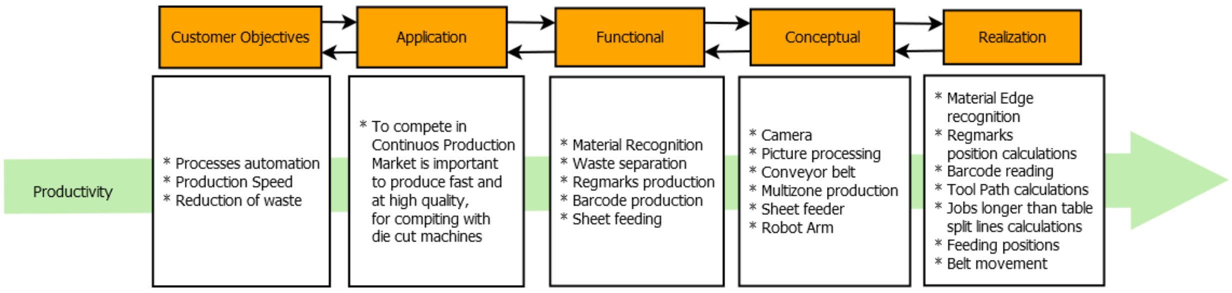
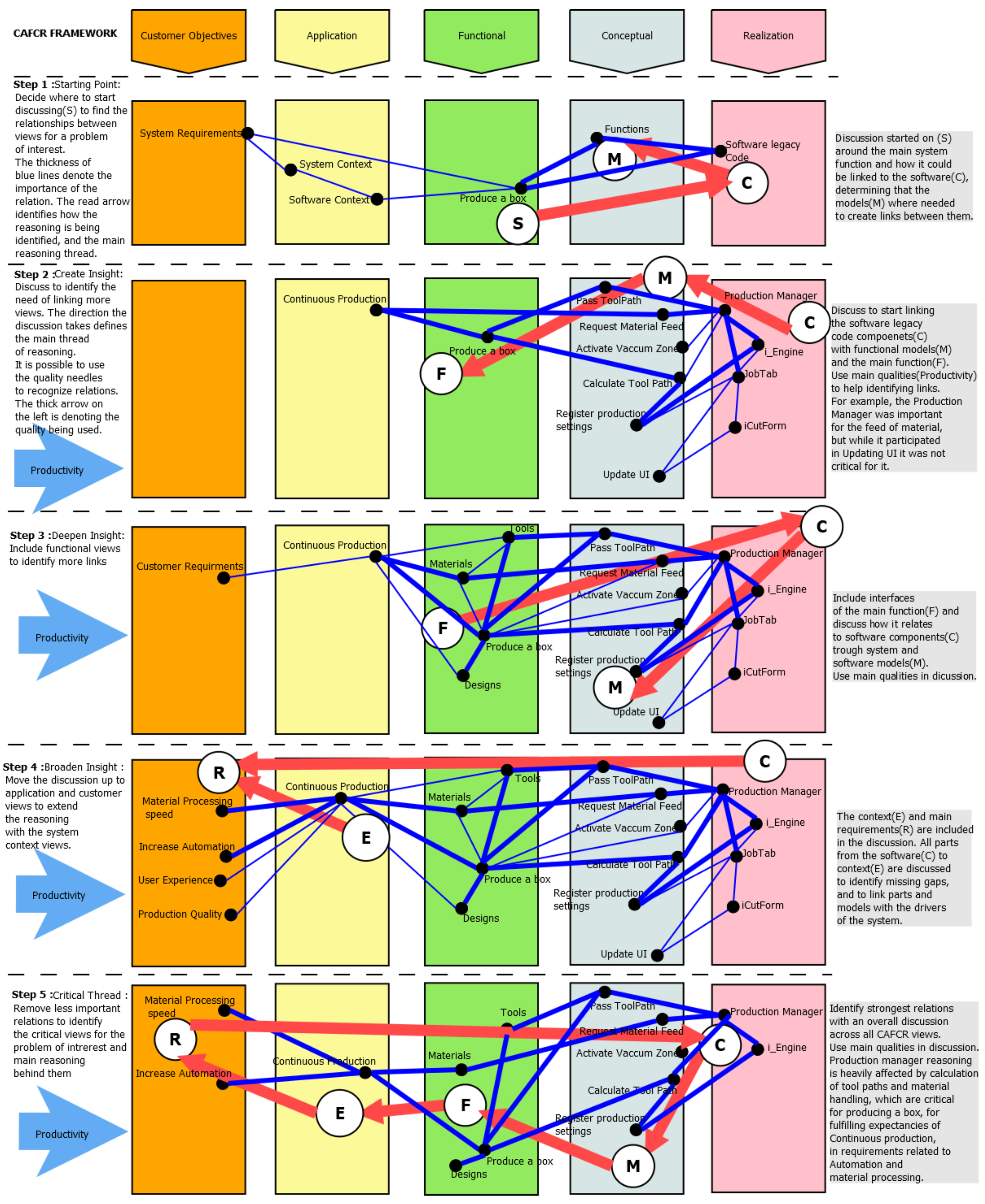


Figure 12. Integration of views via Productivity quality

Activity 1.5: Identify Threads of Reasoning

Threads of reasoning represent the integration of all sub-methods used previously during the system decomposition. The threads integrate them at all levels, as well as the relations between them and with qualities. CAFCR defines the threads identification process as a 5-step approach (Muller, 2004). To find the threads of reasoning requires discussions from a starting point about the specific need or problem. Discussions can trigger the need of adding more views and relationships in-between them. The linking between views can enable the main reasoning for the need or problem of interest. In the case study, the problem is to find the reasons behind the software implementation. Figure illustrates how the identification of threads are performed and how it connects different parts across the system, which is decomposed in the 5 CAFCR views. This figure focuses on the reasoning for the productivity quality. The back dots are the elements or issues discussed during the identification of threads. The blue lines are the relation between these elements, where the importance of the links is denoted by the thickness of the line. The main reasoning thread identified per step is represented as thick arrows. The main discussion areas and the starting and end of each step are represented as white circles.



Step 1 :Starting Point: Decide where to start discussing(S) to find the relationships between views for a problem of interest. The thickness of blue lines denote the importance of the relation. The read arrow identifies how the reasoning is being identified, and the main reasoning thread.

Discussion started on (S) around the main system function and how it could be linked to the software(C), determining that the models(M) where needed to create links between them.

Step 2 :Create Insight: Discuss to identify the need of linking more views. The direction the discussion takes defines the main thread of reasoning. It is possible to use the quality needles to recognize relations. The thick arrow on the left is denoting the quality being used.

Discuss to start linking the software legacy code compoenets(C) with functional models(M) and the main function(F). Use main qualities(Productivity) to help identifying links. For example, the Production Manager was important for the feed of material, but while it participated in Updating UI it was not critical for it.

Step 3 :Deepen Insight: Include functional views to identify more links

Include interfaces of the main function(F) and discuss how it relates to software components(C) trough system and software models(M). Use main qualities in discussion.

Step 4 :Broaden Insight : Move the discussion up to application and customer views to extend the reasoning with the system context views.

The context(E) and main requirements(R) are included in the discussion. All parts from the software(C) to context(E) are discussed to identify missing gaps, and to link parts and models with the drivers of the system.

Step 5 :Critical Thread : Remove less important relations to identify the critical views for the problem of intrerest and main reasoning behind them

Identify strongest relations with an overall discussion across all CAFCR views. Use main qualities in discussion. Production manager reasoning is heavily affected by calculation of tool paths and material handling, which are critical for producing a box, for fulfilling expectancies of Continuous production, in requirements related to Automation and material processing.

Figure 13. Threads of reasoning for the System X Legacy software

First step. It is the identification of a starting point. The case study identifies this point as the main functionality of the SYSTEM X, the production of a box. We then discuss how this could link with the software and start identifying the connection between the main system functionality, the software and the functional models.

Second step. It is the creation of insight about the problem. In our case, this step places the existing software legacy code in connection with the functional models defined in the conceptual view, including specific functions and software code pieces. This is for the intention of identifying the possible idealization paths that the existing code may have followed. It highlights the importance in the relation of specific software pieces and functions.

Third step. It is to deepen the insight of the reasoning. The discussion starts to address some of the critical interfaces of production of a box, such as materials and tools, then to link them with specific software by using functional models. At this point, it is clear what software pieces should be critical when planning the further development of the software in Activity 2.1-2.2: Roadmaps.
Fourth step. It focuses on broadening the reasoning by introducing the system context and drivers into the discussion.
Fifth step. It is for finding the threads of reasoning. It is an overall discussion for filtering out weak links. It should be clear which parts of the code are critical to the system, relations between them and to other elements.

Activity 2.1-2.2: Roadmaps

The previous activities, especially the threads of reasoning identification, enable the creation of the roadmap of work. Figure 14 is an extract of the created roadmap for developing the SYSTEM X. At the top is the expected evolution of the system for the following three years since the proposed solution is applied. The roadmap defines the expected work break down for the three years in three layers. First layer reflects the environment expectancies in the COMPANY A's business. Second and middle layer is the system in green, which contains all customer requirements for the system, which in some cases may be directly related to software and in others not. Market and system layers are from the interview data with managers. The third and last layer is the software in yellow and orange, identifying the specific work for the software development. The software layer contains the requirements and their combinations with maintaining or redesigning certain components of the legacy software for possible new functionalities. This maintenance work is highlighted in the orange boxes in the software layer. Because the CAFCR threads of reasoning visualize the connection between software parts and higher parts of the system, the roadmap can be formed for maintaining certain parts of the software even before any specific requirements.

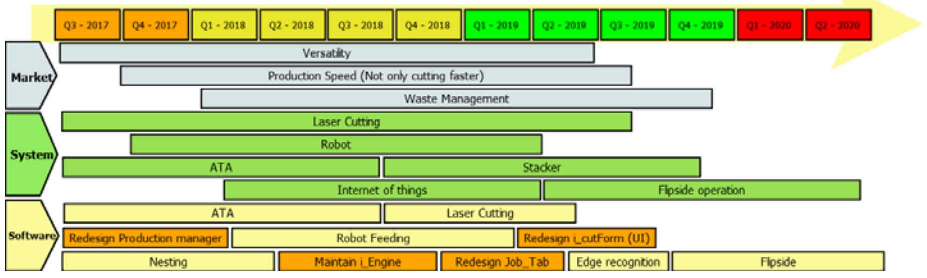


Figure 14. The Roadmap

Activity 2.3 Development

Although the actual development is not part of the paper, the development should be performed following the time schedule defined in the previous activity in practices.

Verification

The case study in COMPANY A shows a real life scenario of a system containing a software legacy component which needed to be further maintained and develop for a fit with the evolution of the SYSTEM X. The application of the solution in the case study successfully helps to develop a plan for the SYSTEM X' legacy software. This developed plan is able to address both software new functionality and the necessary work of critical parts for further developing them in alignment with the expected evolution of the complete system. Therefore, this COMPANY A's case verifies the application of the conceptual solution with an industry problem and illustrates the usage details. The solution does not enforce specific decomposition tools, but provides a framework for engineers dealing with legacy software components in a complex system. Furthermore, based on an assessment by COMPANY A's eight experts, it is found over an 80% of the participants agrees or strongly agrees the solution can help COMPANY A to solve the problem in the continuous development of the legacy software.

Conclusion

This study introduces a solution for resolving the two questions: 1) What are the important parts of legacy software for the fit with the system context? 2) How to align legacy software maintenance and development for the fit with the entire system development? Existing reverse engineering techniques mainly view the issues from a purely software engineering perspective where the software is the main system. We creatively combined the usage of systems engineering’s CAFCR model with reverse engineering to close the gap in-between systems and to enable the continuous development of a legacy software as a complex system evolves. Furthermore, the solution extends the application of CAFCR for specifically reverse engineering the reasoning behind the relationships between legacy software and its system context. The extension also provides the means for planning an align development of the software and system. The solution is verified by its application to an industrial case－SYSTEM X in COMPANY A. The case study illustrates the detailed usage of the solution in decomposing the system and software into the five CAFCR views. It is found that to decompose the software with a bottom-up approach from the source code is effective to generate the software architectural views, especially when it is a legacy component without clear architecture or documentation in a complex system. The link of software subcomponents up to system views and drivers can be identified by integrating the system and software views and reasoning about their relationships. Those links with the reasoning behind are the key to further identification what are the critical parts of the software for fulfilling the system needs. The identification of the critical parts enables us to create a maintenance and development plan by using the roadmap. In this way, the software development can be aligned with the expected evolution of the system.

References

Architecture Working Group (AWG). (2000). *IEEE Recommended Practice for Architectural.* The Institute of Electrical and Electronics Engineers, Inc.

Belle, A. B., Boussaidi, G. E., & Kpodjedo, S. (2016). Combining lexical and structural information to reconstruct software layers. *Information and Software Technology, 74*.

Chikofsky, E. J., & Cross II, J. H. (1990, January). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Softw., 7*(1), pp. 13-17.

Ducasse, S., & Pollet, D. (2009). Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering 35(4)*, 573-591.

ESKO. (2017). *ESKO Kongsberg digital finishing tables*. Retrieved 06 02, 2017, from ESKO: https://www.esko.com/en/products/kongsberg-cutting-tables

Feathers, M. (2004). *Working Effectively with Legacy Code.* Upper Saddle River, NJ, USA: Prentice Hall PTR.

INCOSE. (2017, February 23). *incose.org*. Retrieved from http://www.incose.org/AboutSE/WhatIsSE

ISO/IEC/IEEE. (2011, January). Systems and software engineering -- Architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000),* pp. 1-46.

Muller, G. (2004). *Cafcr: A multi-view method for embedded systems architecting.* Delft University of Technology; .

Müller, H., & Orgun, M. (1993). A Reverse Engineering Approach To Subsystem Structure Identification. *Journal of Software Maintenance*, 181-204.

Pressman, R. S. (2005). *Software engineering: a practitioner's approach.* Palgrave Macmillan.

Purchase, H., Colpoys, L., McGill, M. J., & Britton, C. (2001). *UML Class Diagram Syntax: An Empirical Study of Comprehension.* Sydney: Australian Computer Society, Inc.

SEBoK authors. (2017, May 11). *Guide to the Systems Engineering Body of Knowledge (SEBoK), version 1.8.* Retrieved from What is Systems Thinking?: http://sebokwiki.org/w/index.php?title=What_is_Systems_Thinking%3F&oldid=50344

Biography

Maximiliano Moraga has been a Senior Software Engineer at ESKO since 2015, a company dedicated to the research and development of computer-aided manufacturing systems. Before joining ESKO, he worked in the industry in research and development of systems combining different engineering disciplines for about 7 years. In 2008, he joined Corena as a Software Engineer to develop software for the aviation industry. In 2013, he entered FMC Technologies, an international subsea company in the Oil & Gas industries, as a Software Engineer for the development of subsea production systems. He received a bachelor in computer sciences from the University of Computer Sciences, Santiago, Chile, in 2007, and a Master in Systems Engineering from the University College of Southeast Norway, Kongsberg, Norway, in 2017.

Yang-Yang Zhao has been an Associate Professor at the Department of Science and Industry Systems in the University College of Southeast Norway since Aug. 2013. In Feb. 2018, she joined an Associate Professor position at the Department of informatics in the University of Oslo. Before that she worked as the Visiting Associate Professor in the Center for Design Research at Stanford University for the year of 2015. She also served as the In-house Professor in the Sino-Finnish Center at Tongji University for a short period. Besides, she has industrial experience in process management and business development and entrepreneurial experience in IT solutions in Asia Pacific. Her Ph.D. is in technology management and innovation strategy from the National University of Singapore. Her research interests include human-centered design, systems engineering, technological catching-up and innovation strategy in emerging markets. Her work has appeared in the Journal of System Engineering, Journal of Technology & Engineering Management, Journal of Chinese Economics & Business Studies, Total Quality Management & Business Excellence Journal and many refereed international conferences.