Submethods in the CR Views

-

logo
TBD

Gerrit Muller
Buskerud University College
Frogs vei 41 P.O. Box 235, NO-3603 Kongsberg Norway
gaudisite@gmail.com

Abstract

This chapter describes the *Conceptual* view and the *Realization* view. Both views are supported by a set of submethods to describe multi-disciplinary design, for example several decompositions and models are provided.

version: 1.5 status: finished October 20, 2017

1 Introduction

Decomposition is widely used in the conceptual and realization view. Section 2 describes a few *decompositions*, and introduces *interfaces*. From components to system qualities is more than a simple accumulation of component data. The system behavior and characteristics are described by *qualities* in Section 3. The conceptual and realization views also provide information to support the project manager. Section 4 describes some submethods to support project management.

2 Decomposition

Decomposition and modularity are well known concepts, which are the fundamentals of software engineering methods. A nice article by Parnas [11] discusses decomposition methods.

The decomposition can be done along different axes. Subsection 2.1 shows *construction* as axis, and Subsection 2.2 shows the *functional* decomposition. The decomposition into concurrent activities and the mapping on processes, threads and processors is called the execution architecture, which is described in Subsection 2.4.

The design of complex systems always requires multiple decompositions, for instance a construction and a functional decomposition. Subsection 2.3 describes a submethod to cope with multiple decompositions. The relations between the decompositions are described by mappings, described in Subsection 2.5.

Decompositions results in components. The interfacing between components is discussed in Subsection 2.6.

2.1 Construction Decomposition

The construction decomposition views the system from the construction point of view, see Figure 1 for an example. In this example the decomposition is structured to show layers and the degree of domain know-how. The vertical layering defines the dependencies: components in the higher layers depend on components in the lower layers. Components are not dependent on components at the same or higher layer. The amount of domain know how provides an indication of the added value of the components. More generic components are more likely to be shared in a broader application area, and are more likely to be purchased instead of being developed.

The construction decomposition is mostly used for the design management. It defines units of design, as these are created and stored in repositories and later updated. The atomic units are aggregated into compound design units. In software the compound design units are often called *packages*, in hardware they are called *modules*. The blocks in Figure 1 are at the level of these packages and modules.

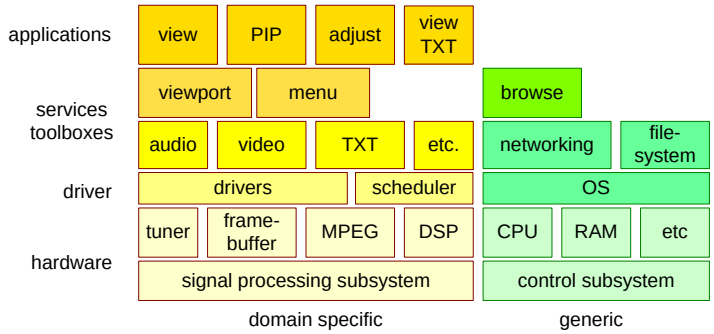

Figure 1: Example of a construction decomposition of a simple TV. The vertical axis is used for layers, where higher layers depend on lower layers, but not vice versa. In horizontal direction the left hand side shows the domain specific components, the right hand side shows the more generic components.

Packages and modules are used as unit for testing and release and they often coincide with organizational ownership and responsibility.

In hardware this is quite often a very natural decomposition, for instance into cabinets, racks, boards and finally integrated circuits, Intellectual property (IP) cores and cells. The components in the hardware are very tangible. The relationship with a number of other decompositions is reasonably one to one, for instance with the work breakdown for project management purposes.

The construction decomposition in software is more ambiguous. The structure of the code repository and the supporting build environment comes close to the hardware equivalent. Here files and packages are the aggregating construction levels. This decomposition is less tangible than the hardware decomposition and the relationship with other decompositions is sometimes more complex.

2.2 Functional Decomposition

The functional decomposition decomposes end user functions into more elementary functions. The elementary functions are internal, the decomposition in elementary functions is not easily observable from outside the system. In other words, the **what** is worked out in **how**. Be aware of the fact that the word *function* in system design is heavily overloaded. No attempt is made to define the functional decomposition more sharply, because a sharper definition does not provide more guidance to architects. Main criterium for a good functional decomposition is its useability for design. A functional decomposition provides insight how the system will accomplish its job. MASCOT [2] is an example of a method where a functional decomposition is based on data flow.

Figure 2 shows an example of (part of) a functional decomposition for a camera

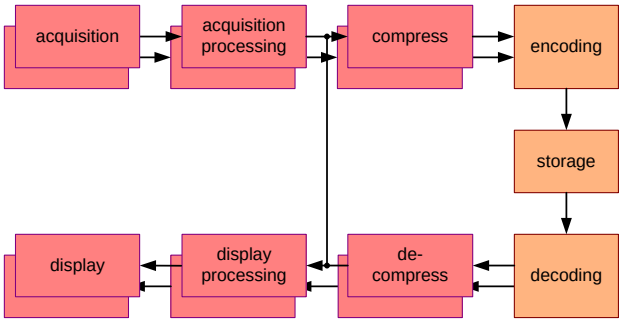

Figure 2: Example functional decomposition camera type device

type device. It shows a data flow with communication, processing, and storage functions and their relations. This functional decomposition is **not** addressing the control aspects, which might be designed by means of a second functional decomposition, this time taken from the control point of view.

2.3 Designing with Multiple Decompositions

Most designers don't anticipate cross system design issues. During the preparation of design team meetings designers often do not succeed in submitting system level design issues. This limited anticipation is caused by the locality of the viewpoint, implicitly chosen by the designers. The designers are, while they working on a component, concerned about many design characteristics. Examples of design characteristics are *Signal to noise ratio (SNR)*, *accuracy*, *memory usage*, *processor load*, and *latency*.

Figure 3 shows a method to help designers to find system design issues, based on the *Question space*. The question space is a three dimensional space. Two dimensions are the decomposition dimensions (construction and functional); the last dimension is the design characteristic dimension. The design characteristics on this axis must be specific and quantifiable. A source of inspiration to find these characteristics are the qualities, described in Chapter **??**, where the challenge is to find the specific and quantified characteristics that contribute to the quality.

For every point in this 3D space a question can be generated in the following way:
How about the <*characteristic*> of the <*component*> when performing <*function*>?
Which will result in questions like:
How about the *memory usage* of the *user interface* when *querying the database*?

The designers will not be able to answer most of these questions. Simply asking these questions helps the designer to change the viewpoint and discover many potential issues. Fortunately, most of the (not answered) questions turn out

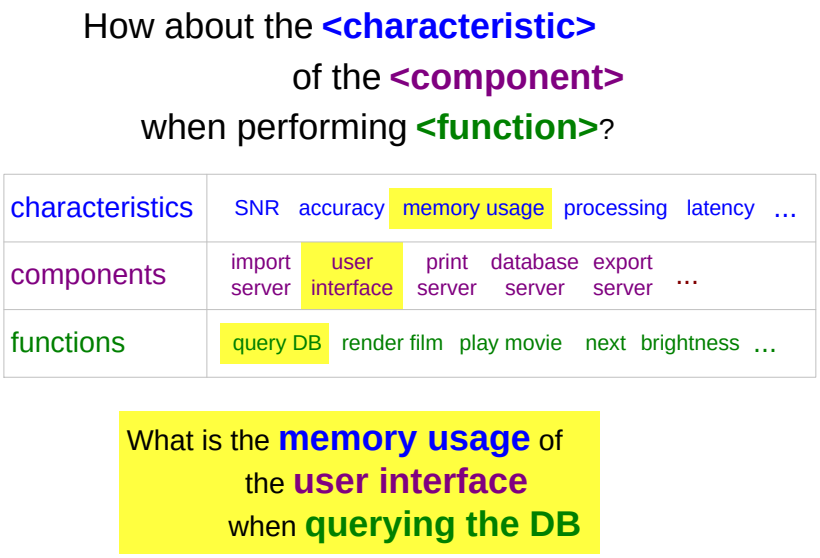

Figure 3: The question generator for multiple decompositions generates a question for every point in the Question space. The generic question is shown at the top. An example is shown below. The table shows a partial population for the three dimensions. The question at the bottom is generated by substituting one value from every row.

to be irrelevant. The answer to the memory usage question above might be *insignificant* or *small*. The more detailed memory usage questions are irrelevant as long as the total functionality fits in the available memory.

The architect can apply a priori know-how to select the most relevant questions in the 3D space, for instance:

Critical for system performance Every question that is directly related to critical aspects of the system performance is relevant. For example *What is the CPU load of the motion compensation function in the streaming subsystem?* will be relevant for resource constrained systems.

Risk planning wise Questions regarding critical planning issues are also relevant. For example *Will all concurrent streaming operations fit within the designed resources?* will greatly influence the planning if resources have to be added.

Least robust part of the design Some parts of the design are known to be rather sensitive, for instance the priority settings of threads. Satisfactory answers should be available, where a satisfactory answer might also be *we scheduled a priority tuning phase, with the following approach.*

Suspect parts of the design Other parts of the design may be suspect for a number of reasons. Experience, for instance, learns that response times and throughput do not get the required attention of software designers (experience-based suspicion). Or we may have to allocated an engineer to the job with insufficient competence (person-based suspicion).

Some questions address a line or a plane in the multi dimensional space. An example of such an improved question is a memory budget for the system, thereby addressing all memory aspects for both functions and components in one budget.

2.4 Execution Architecture

The execution architecture is the run-time architecture of a system. The process[1] decomposition plays an important role in the execution architecture. Figure 4 shows an example of a process decomposition.

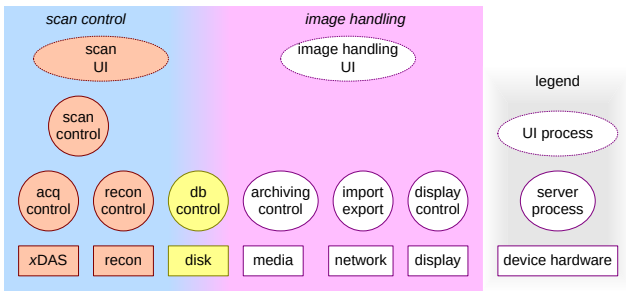

Figure 4: An example of a process decomposition of a MRI scanner.

One of the main concerns for process decomposition is concurrency: which concurrent activities are needed or running, and how do we synchronize these activities? Two techniques to support asynchronous functionality are widely used in operating systems: processes and threads. Processes are self sustained, which own their own resources, especially memory. Threads have less overhead than processes. Threads share resources, which makes them more mutually dependent. In other words processes provide better means for separation of concerns.

The execution architecture must map the functional decomposition on the process decomposition. This mapping must ensure that the timing behavior of the system is within specification. The most critical timing behavior is defined by the dead lines. Missing a dead line may result in loss of throughput or functionality. The timing behavior is also determined by the choice of the synchronization methods, by the granularity of synchronization and by the scheduling behavior. The most common technique to control the scheduling behavior is by means of priorities.

[1] Process in terms of the operating system

This requires, of course, that priorities are assigned. Subsystems with limited concurrency complexity may not even need multiple threads, but these subsystems can use a single thread that keeps repeating the same actions all the time. The mapping is further influenced by hardware software allocation choices, and by the construction decomposition. A well known method in the hard real time domain is DARTS (Design Approach for Real Time Systems) [3]. This methods provides guidelines to identify hard real time requirements, translate them in activities and to map activities on tasks. DARTS then describes how to design the scheduling priorities.

In practice many components from the construction decomposition are used in multiple functions, and are mapped on multiple processes. These shared components are aggregated in shared or dynamic-link libraries (dll's). Sharing the program code run-time is advantageous from memory consumption point of view.

2.5 Relations between Decompositions

The decompositions that are made as part of the design are related to each other. A mapping or allocation is required to relate a decomposition with another decomposition. For instance the functional decomposition can be mapped on the construction decomposition: functions are allocated to components in the construction decomposition. Another example is that functions are mapped on threads in the execution architecture.

The difficult aspect of these mappings is that in most systems $n : m$ mappings are needed. Every decompositions serves its own purposes, such as *construction* and *configuration management* in the construction decomposition, *performance* and *image quality* in the data flow functional decomposition, and *timing* and *concurrency* in the execution architecture. Each decomposition must clearly serve its intended purpose. On top of that a clear mapping strategy must be described to relate the decompositions.

2.6 Interfaces

The interfaces are the complement of the components in a decomposition. A lot of work on interface specifications has been done, for instance in KOALA [14]. KOALA adds the notion of *provides* and *requires* interfaces to formalize dependency relations. A powerful decoupling step is the use of *protocols* as described by Jonkers [7]. Protocols according to [7] describe the functional and dynamic behavior of interfaces.

In Subsection **??** the interfaces were already discussed in the context of external interfaces in the functional view. The internal interface can be specified analogous to the external interfaces. Part of the internal interface is also specified by an internal information model, for instance modeled via entity relationship diagrams.

The internal information model abstracts from the implementation, by modelling the data concepts, relationships and activities. The internal information model extends the external information model with data that are introduced as design concepts. It can, for instance, show caches, indices and other structures that are needed to achieve the required performance.

3 Quality Design Submethods

This section discusses submethods to achieve the objectives for some of the qualities that will be discussed in Chapter **??**. *Performance* is discussed in Subsection 3.1. *Budgetting*, a submethod that can be used for several qualities, is described in Subsection 3.2. Submethods for *Safety*, *Reliability* and *Security* are discussed in Subsection 3.3. *Start up* and *Shutdown* are discussed in Subsection 3.4. Subsection 3.5 describes briefly submethods with respect to *Value* and *Cost*. Subsection 3.6 discusses granularity, an important the design consideration.

3.1 Performance Modeling

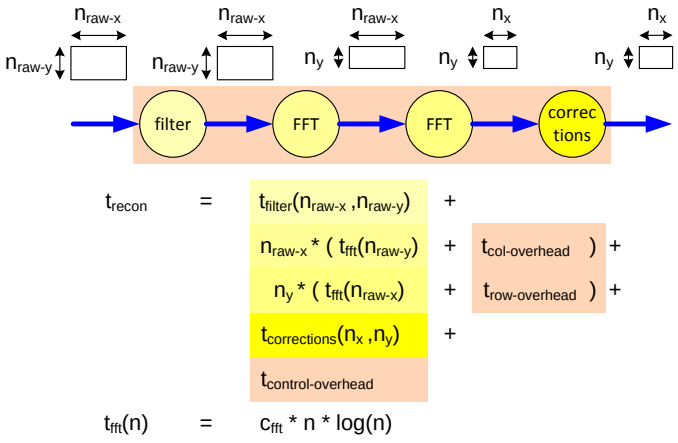

Figure 5: Flow model and analytical model of the image reconstruction in MR scanners. The analytical model is an algorithm to calculate the inherent computational costs.

System performance is being tackled by using complementary models, such as visual models and analytical models. For instance, flow can be visualized by showing the order, inputs, outputs and the type of data, and flow performance can be described by means of a formula. In figure 5 the performance is modeled by a visual model at the top and an analytical model below. The analytical model is

entirely parameterized, making it a generic model that describes the performance over the full potential range. For every function in the visual model the order of the algorithm is determined and the parameterization for the input and output data. The analytical model should be a manageable formula to provide insight in the performance behavior. In this example for MR reconstruction Fourier transforms are order $n * log(n)$, while the other computations are order n.

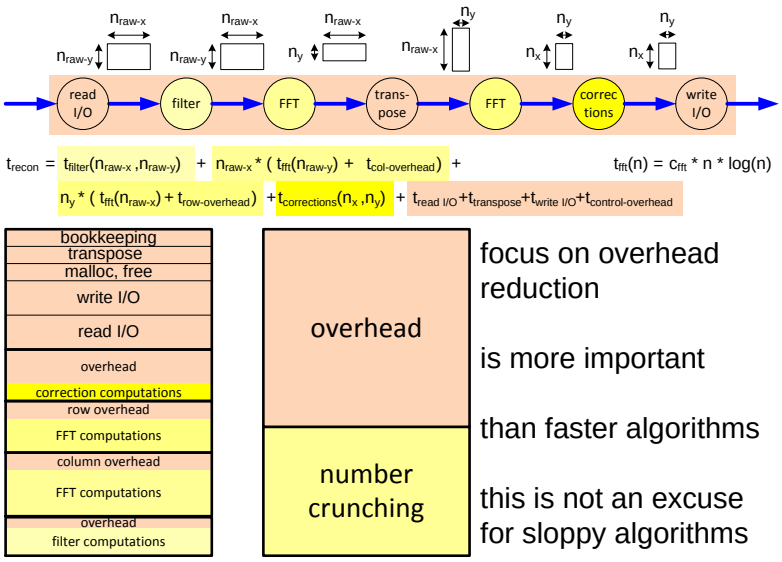

Figure 6: Example of performance analysis and evaluation. Implementation specific functions are added to the flow model and the analytical model. Below timing measurements are added, and classified as overhead and number crunching.

The implementation of the system often reveals additional contributions to the processing time, resulting in an improved model, as shown in Figure 6. The pipeline model at the top of Figure 6, is extended with data transfer functions. The measurement below the model shows that a number of significant costs are involved in data transfer and control overhead. The original model of Figure 5 focuses on processing cost, including some processing related overhead. In product creation[2] the overhead plays a dominant role in the total system performance. Significant overhead costs are often present in initialization, I/O, synchronization, transfers, allocation and garbage collection (or freeing if explicitly managed).

Analytical performance models as shown in Figure 6 are powerful means to design, analyze and discuss performance. The difficulty in developing these models

[2]observed and coped with this problem in the following product developments: 1980 Video Display unit, 1981 Oncology Support, 1984 Digital Cardio Imaging, 1984 MRI user interface, 1986 MRI data acquisition, 1992 Medical Imaging workstation, 2002 Audio/Video processing.

is in finding a manageable level of abstraction without losing too much predictive value. To develop the analytical model algorithmic analysis and empirical analysis need to be combined. It is my experience that analytical models with a manageable level of abstraction can be made for a wide variety of systems: MRI scanners, Digital Cardio Imaging, Medical Imaging Workstation, Wafersteppers, and Audio and Video processing systems.

The actual characteristics of the technology being used must be measured and understood in order to make a good (reliable, cost effective) design. The basic understanding of the technology is created by performing micro-benchmarks: measuring the elementary functions of the technology in isolation. Figure 7 lists a typical set of micro-benchmarks to be performed. The list shows infrequent and often slow operations and frequently applied operations that are often much faster. This classification implies already a design rule: slow operations should not be performed often[3].

	infrequent operations, often time-intensive	*often repeated operations*
database	start session finish session	perform transaction query
network, I/O	open connection close connection	transfer data
high level construction	component creation component destruction	method invocation same scope other context
low level construction	object creation object destruction	method invocation
basic programming	memory allocation memory free	function call loop overhead basic operations (add, mul, load, store)
OS	task, thread creation	task switch interrupt response
HW	power up, power down boot	cache flush low level data transfer

Figure 7: Typical micro-benchmarks for timing aspects

The results of micro-benchmarks should be used with great care. The measurements show the performance in totally unrealistic circumstances, in other words it is the best case performance. This best case performance is a good baseline to understand performance, but when using the numbers the real life interference (cache disturbance for instance) should be taken into account. Sometimes additional measurements are needed at a slightly higher level to calibrate the performance estimates.

[3]This really sounds as an open door. However, I have seen many violations of this entirely trivial rule, such as setting up a connection for every message, performing I/O byte by byte et cetera. Sometimes such a violation is offset by other benefits, especially when a slow operation is in fact not very slow and when the brute force approach is both affordable as well as extremely simple.

The standard work about performance issues in computer architectures is the book by Hennesey and Patterson [5]. Here modelling and measurement methods can be found that can serve as inspiration for performance analysis of embedded systems.

3.2 Budgets

The implementation can be guided by making budgets for the most important resource constraints, such as memory size, response time, or positioning accuracy. The budget serves multiple purposes:

- to make the design explicit
- to provide a baseline to take decisions
- to specify the requirements for the detailed designs
- to have guidance during integration
- to provide a baseline for verification
- to manage the design margins explicitly

The simplification of the design into budgets introduces design constraints. Simple budgets are entirely static. If such a simplification is too constraining or costly then a dynamic budget can be made. A dynamic budget uses situational determined data to describe the budget in that situation. The architect must ensure the manageability of the budgets. A good budget has tens of quantities described. The danger of having a more detailed budget is the loss of overview.

step	example
1A measure old systems	micro-benchmarks, aggregated functions, applications
1B model the performance starting with old systems	flow model and analytical model
1C determine requirements for new system	response time or throughput
2 make a design for the new system	explore design space, estimate and simulate
3 make a budget for the new system:	models provide the structure measurements and estimates provide initial numbers specification provides bottom line
4 measure prototypes and new system	micro-benchmarks, aggregated functions, applications profiles, traces
5 Iterate steps 1B to 4	

Figure 8: Budget-based design flow

Figure 8 shows a budget-based design flow. The starting point of a budget is a model of the system, from the conceptual view. An existing system is used to

get a first guidance to fill the budget. In general the budget of a new system is equal to the budget of the old system, with a number of explicit improvements. The improvements must be substantiated with design estimates and simulations of the new design. Of course the new budget must fulfill the specification of the new system, sufficient improvements must be designed to achieve the required improvement.

Early measurements in the integration are required to obtain feedback once the budget has been made. This feedback will result in design changes and could even result in specification changes.

3.3 Safety, Reliability and Security

The qualities *safety*, *reliability* and *security* share a number of concepts, for example:

- containment
- graceful degradation
- interlock, for instance dead man switch
- detection and tracing of failures,
- logging of operational data for post mortem analysis, e.g. flight recorder
- redundancy

All three qualities are covered by an extensive set of methods. Highly recommended is the work of Neumann [10]. A lot of literature is based on work in the aerospace industry. A good starting point to the literature is the home page of the International Council on Systems Engineering [6].

A common guideline in applying any of these concepts is that the more critical a function is, the higher the understandability should be, or in other words the simpler the applied concepts should be. Many elementary safety functions are implemented in hardware, avoiding large stacks of complex software.

Specialized engineering disciplines exist for Safety, Reliability and Security. These disciplines have developed their own methods. One class of methods relevant for system architects is the class of analysis methods that start with a (systematic) brainstorm, see figure 9. The Medical HACCP Alliance [13] provides extensive documentation for Hazard Analysis And Critical Control Point (HACCP) method for medical devices. A more systematic analysis provides input to improve the design.

Walk-through is another effective assessment method. A few use cases are taken and together with the engineers the implementation behavior is followed for these cases. The architect will especially assess the understandability and simplicity of the implementation. An implementation that is difficult to follow

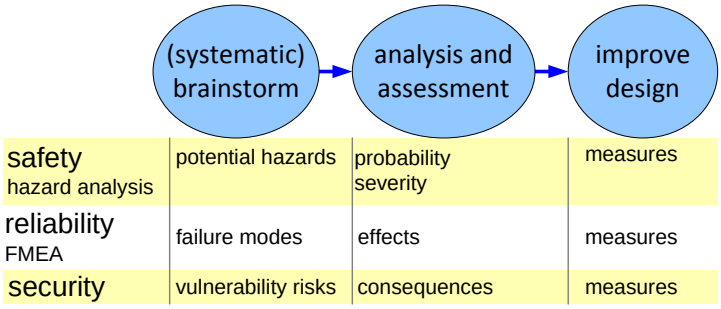

Figure 9: Analysis methods for safety, reliability and security

with respect to safety, security or reliability is suspect and at least requires more analysis.

3.4 Start up and Shutdown

The start up and the shutdown of the system are related to many components and functions of the system. One of the common patterns is the *run level* concept. The start up and shutdown are performed in phases, with increasing functionality and increasing integration, see for instance [8].

The current trend with more sophisticated power management, software downloading and roaming access networks increases the importance of clear design concepts to support these types of functionality.

3.5 Value and Cost

Many design decisions are made on an evaluation of the value of a design option versus the cost of this option. The production cost[4] of a system can be managed by making a decomposition of the cost and using the decomposition to create a cost budget.

Determination of the value of a design option is much more difficult. The value depends on the viewpoint. Some features are valuable for a particular stakeholder, for instance diagnostics for a service engineer and debugging for the developer. In general multiple viewpoints need to be somehow accumulated to create an *integral value*. QFD [12] uses multiple mappings with weight factors to "add" values together and create an integral value.

[4]Within Philips the term Material and Labor Cost (MLC) is frequently used. The MLC determines the fixed cost of a product. The investment costs are variable costs. Different accounting practices are used to cope with the investment costs in the integral cost of the product.

3.6 Granularity Determination

The granularity of operations is an important design choice. Fine granularity offers flexibility and fast response, at the cost of more overhead per operation. Coarse granularity creates less overhead, at the cost of less flexibility and longer latencies. The determination of the granularity is an optimization problem that can be solved by applying optimization techniques from the operational research, see for an introduction [1].

Examples of operations where the unit size of operation has to be chosen are: buffering, synchronization, processing and input/output. In the case of video processing examples of operation sizes are: pixel, line and frame. In real video processing systems at this moment the unit size is typical a quarter of a video frame. The latency of the video chain is critical for two reasons: the time difference between audio and video must be small (lip-synchronization) and zapping must be fast. Smaller unit sizes create too much overhead, larger unit sizes create too much latency.

4 Project Management Support

The architect supports the project leader. Typical contributions of the architect are an initial *work breakdown* and an *integration plan*. Many more project management submethods exist, see for instance [4], but most of them are less relevant for the system architect.

A work breakdown is in fact again another decomposition, with a more organizational point of view [4]. The work in the different work packages should be cohesive internally, and should have low coupling with other work packages.

Figure 10 shows an example of a work breakdown. The entire project is broken down in a hierarchical fashion: project, segment, work package. In this example color coding is applied to show the technology involved and to show development work or purchasing work. Both types of work require domain know how, but different skills to do the job.

Schedules, work breakdown and many technical decompositions are heavily influenced by the integration plan. Integration is the effort of combining the components into a (sub)system, and to get the integrated (sub) system to work in the intended way. During the integration many specification and design inconsistencies, oversights, misunderstandings and mistakes are detected, analyzed and solved. Integration typically costs a lot of time and effort. The risk in a project is that the integration takes too much time and effort. Sufficient and regular attention for the integration viewpoint makes the risk better manageable.

Figure 11 shows an example of an integration plan. The systems that are used for the actual integration, the *integration vehicles*, are the limiting resource for integration. The integration plan is centered around 3 tiers of integration vehicles:

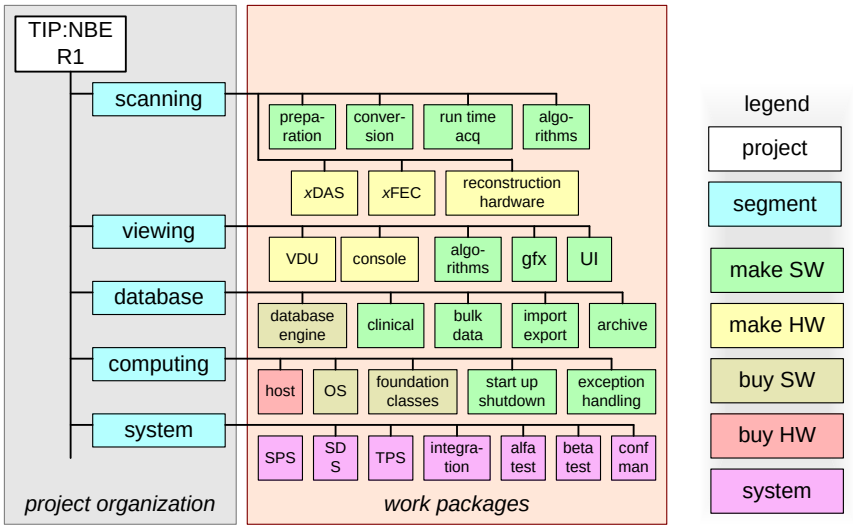

Figure 10: An example of a work breakdown from MRI scanner development. The project is organized in segments. The work in every segment is decomposed in work packages.

- partial systems to facilitate SW testing
- existing HW systems
- new HW systems

The partial systems for SW testing consist mostly of standard computer infrastructure. This computer infrastructure is very flexible and accessible from software point of view, but far from realistic from hardware viewpoint. The existing and new HW systems are much less accessible and more rigid, but close to the final product reality. The new HW system will be available late and hides many risks and uncertainties. The overall strategy is to move from good accessible systems with few uncertainties to less accessible systems with more uncertainties. A new application is first tested on a partial system for software testing. Then this application is tested on systems with existing hardware, with little hardware uncertainties. Finally this application is tested on the new base system. In general integration plans try to avoid stacking too many uncertainties by looking for ways to test new modules in a stable known environment, before confronting new modules with each other.

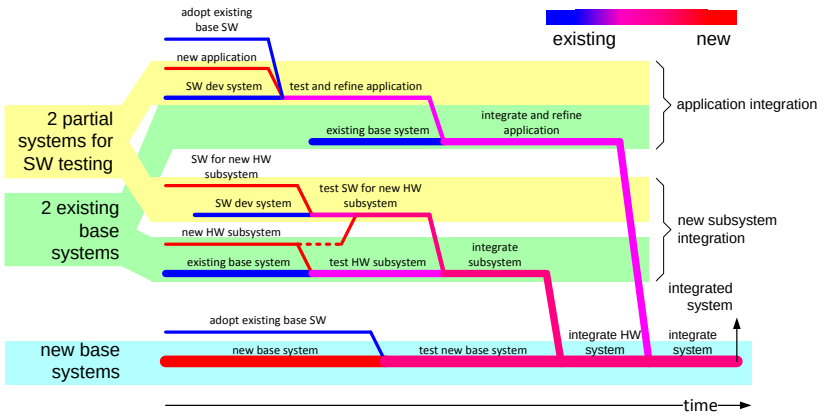

Figure 11: Example of an integration plan, with three tiers of integration vehicles. In this example two partial systems for software testing, two existing base systems and one new base system

5 Overview of the Submethods in the CR views

Figure 12 shows an overview of the submethods that are discussed in this chapter. These submethods are positioned in the *Conceptual View* and the *Realization View*. This positioning is not a black and white proposition, many submethods address aspects from multiple views. However, the positioning based on the essence of the submethod helps to select the proper submethod.

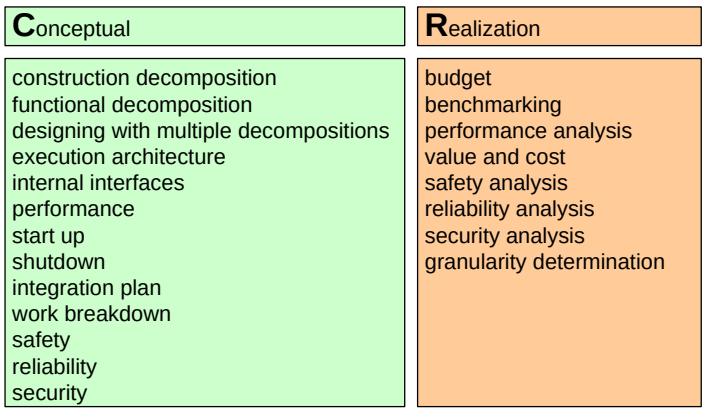

Figure 12: Overview of the submethods discussed in this chapter, positioned in the CR views

References

[1] J. E. Beasley. OR-notes. `http://mscmga.ms.ic.ac.uk/jeb/or/basicor.html`.

[2] Per Bjurï¿½us and Axel Jantsch. MASCOT: A specification and cosimulation method integrating data and control flow. `http://jamaica.ee.pitt.edu/Archives/ProceedingArchives/Date/Date2000/papers/2000/date00/pdffiles/03a_2.pdf`, 2000.

[3] H Gomaa. *Software Design Methods for Real-time Systems*. Addison-Wesley, 1993.

[4] Robert J. Graham and Randall L. Englund. *Creating an Environment for Successful Projects; The Quest to Manage Project Management*. Jossey-Bass Publishers, San Fransisco, CA, 1997.

[5] John L. Hennessy, David A. Patterson, and David Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[6] INCOSE. International council on systems engineering. http://www.incose.org/toc.html, 1999. INCOSE publishes many interesting articles about systems engineering.

[7] Hans Jonkers. Interface-centric architecture descriptions. In *WICSA 2001, Amsterdam*, 2001.

[8] James Mohr. The linux tutorial; run-levels. http://www.linux-tutorial.info/cgi-bin/display.pl?65&99980&0&3, 1997.

[9] Gerrit Muller. The system architecture homepage. http://www.gaudisite.nl/index.html, 1999.

[10] Peter G. Neumann. Homepage peter g. neumann a.o. about safety, security and reliability. http://www.csl.sri.com/users/neumann/.

[11] David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, pages 128–138, March 1979. This article can also be found in "Software Fundamentals, Collected Papers by David Parnas", Addison-Wesley.

[12] QFD Institute. QFD institute. http://www.qfdi.org/, 2000.

[13] The Medical HACCP Alliance. Hazard analysis and critical control point. http://medicalhaccp.ag.vt.edu/, 1998.

[14] Rob van Ommering. Building product populations with software components. In *ICSE 2002*, 2002.

History

Version: 1.5, date: April 8, 2003 changed by: Gerrit Muller

- repaired broken reference

Version: 1.4, date: April 7, 2003 changed by: Gerrit Muller

- added subsection "Relations between Decompositions"
- clarified text of execution architecture
- clarified the value of KOALA
- replaced flow models by *models that visualize flow*
- added legend to process decomposition figure
- small textual improvements
- changed status to finished

Version: 1.3, date: February 27, 2003 changed by: Gerrit Muller
- changed the figure with the question generator
- added short description to DARTS reference
- explained the construction of the analytical performance models
- added a discussion about the difficulties and value of analytical performance models

Version: 1.2, date: November 20, 2003 changed by: Gerrit Muller
- added description of relation *design characteristics* and *qualities*
- removed configuration management remark from execution architecture subsection
- changed status into "concept"
- added overhead terms to analytical formula of reconstruction performance analysis
- added *old* and *new* to figure describing budget methods
- renamed Section "Qualities" in "Qaulity Design Submethods"
- renamed Subsection "Granularity" in "Granularity Determination"
- added Section "Overview"

Version: 1.1, date: October 30, 2003 changed by: Gerrit Muller
- added Subsection Granularity

Version: 1.0, date: October 14, 2003 changed by: Gerrit Muller
- removed Figure execution architecture
- status changed into 'draft'
- replaced budget figure
- improved figure work breakdown

Version: 0.1, date: September 22, 2003 changed by: Gerrit Muller
- moved "Execution architecture" into Section "Decomposition"
- added "Budgets"
- added improved performance model
- added "Micro benchmarks" to the Subsection "Performance"
- added Analysis methods to Subsection "Safety, Reliability and Security"

Version: 0, date: July 28, 2003 changed by: Gerrit Muller
- Created, no changelog yet