embeddedworld2020
Exhibition&Conference
... it's a smarter world

Bridging the Gap Between Requirements, Architecture, and Implementation

A Systems Engineering Solution for Model-Based Design

Marc Segelken
Application Engineering
MathWorks
Ismaning, Germany
msegelke@mathworks.com

***Abstract*—How can we perform systems engineering tasks associated with standards like ARP 4754 and ISO 26262 part 4, while ensuring that the key derived requirements filter down to the implementation? Performing large-scale system design and upgrades is a task of ever-increasing complexity. Traceability and synchronization across all design levels is key to streamlining large-scale development programs. However, there is often a missing link between systems engineering and design implementation in a top-down design process.**

In this work it will be demonstrated how to bridge the gap between systems engineering and implementation in a top-down legacy systems upgrade project. The example case study shall follow the update of key system-level requirements and architectural modifications down to identification of the required design implementation. Finally, a system-level tradeoff analysis shall be performed to assess the high-level impact of the upgrade on the complete system.

Keywords—systems engineering, requirements decomposition, architecture modeling, stereotypes, architectural analysis, trade studies

I. INTRODUCTION

With the ever-increasing size and complexity of systems; requirements that must be engineered, maintained, derived, allocated, and adhered to; and constraints on performance, costs, time-to-market, power consumption, weight, and other areas, systems engineering is a challenge that needs to address these factors throughout the design of system architectures. The outcome of this process is typically a set of starting points for the design of the sub-components, with interface descriptions, sub-constraints, and derived requirements.

The following presents a top-down architecture design approach focusing on some key activities and aspects that complements Model-Based Design with architecture modeling based on stereotyped components with properties for system analysis. To enable a focus on each component without losing crucial system context information, requirements traceability for the system and (derived) component level and the use of filtered views for handling system complexity are key. An easy transition to development of the system and guaranteed consistency are other key issues for success.

II. DECOMPOSITION OF REQUIREMENTS AND ALLOCATION TO ARCHITECTURE MODEL

A systems engineering project typically begins with high-level requirements and optionally a legacy system that could partially or structurally be reused to some extent. The main task is to create an architecture with sub-components, each allocated to derived requirements to fulfill their share of the overall functionality, with as many hierarchy levels involved as appropriate. Thus, this structural decomposition is accompanied by a similar decomposition of the requirements such that the constraints of each sub-component are sufficiently defined.

Due to the creative nature of this design space exploration, many iterations and refinement steps are typically needed before a satisfactory solution is produced. Subsequent feasibility studies require considerable additional information, especially on non-functional constraints to be met by the components and the overall system. Therefore, this type of information should also be carefully decomposed throughout architecture design. Typically, several architectures, not just one, are produced and need to be evaluated and compared with respect to performance, cost, time-to-market, and other factors, to choose the final, most suitable architecture solution.

The types of requirements to be considered are outlined in the following sections.

A. Non-functional requirements

Many requirements are referring to life-cycle issues or other non-functional constraints. Possible solutions have properties such as weight, cost, reliability, development effort, and other domain-specific design data that need to fit these non-functional requirements—as well as their compositions—on each hierarchy level.

Accordingly, a hierarchy of stereotypes has to be defined, representing each type of sub-component and capturing

properties as needed, including the non-functional requirements mentioned above. This way the corresponding characteristics of the chosen components can be maintained, whether they are commercial off the shelf (COTS) or still to be developed. To perform trade studies with different component solutions and different architectures, each solution needs to be analyzed with respect to the non-functional requirements. A simple example would be the determination of the mass and therefore the weight of a certain architectural solution. The analysis here is simply to add all components' mass properties to compute the overall mass. Another simple example would be the production costs or development costs of a system, which would get computed the same way. For more complex systems, tool support is needed to get such numbers quickly while exploring different architectural solutions. With such tools, optimizing architectures based on trade study results requires much less effort.

B. Functional requirements

Temporal performance constraints aside, functional requirements are typically not addressed specifically on the architectural level, other than getting decomposed into derived requirements in parallel with the system decomposition. Performing a complete analysis at this early stage is possible with formalized requirements, but due to the difficulty of getting a complete set of requirements and assumptions, this assume-guarantee reasoning is applied very rarely in practice and is not covered by this methodology approach. Instead, simulation is proposed on the component and architecture levels to validate requirements consistency locally as well as overall system behavior.

Therefore, the ability to simulate the very same overall architecture model that was used to define components with their interfaces and interconnections is needed to avoid any mistakes caused by a rupture of systems engineering and design flow.

III. COMPLEXITY HANDLING

By definition, systems are more complex than just the software or just the hardware, or any other segmentation of the system. Focusing just on parts of the system during any design activity is mandatory to not get lost or tangled in complexity issues. However, if important context information about the role of a component or its intra-system environment is missing, specification or design flaws become inevitable.

So, a suitable subset (*view*) of the system needs to be set up to understand a specific design or analysis concern, with only the minimum required context information—everything not relevant for the task at hand needs to be hidden.

While finding an appropriate view meeting the criteria mentioned above is demanding, it is typically not sufficient to have just one view for a sub-part of the system. One-view-fits-all does not work here, since different perspectives of looking at the system require different views that are all overlapping: functional dependencies, organizational dependencies, bottleneck views, power consumption considerations, supplier dependencies, maturity levels, failure probability views, safety integrity level sections, and so on. A complete understanding of a specific design or analysis concern requires quick switching among the huge number of different groupings and filters needed on the (sub-)systems.

Since all such different views on a system always need to be consistent, tool support is crucial to define and use such views.

IV. TOOL SUPPORT

Due to the size and complexity of systems, classical approaches with drawing tools and table spreadsheets to account for custom properties and corresponding analysis are no longer appropriate. The probability of consistency issues and problems caused by out-of-date data is just too high if there is no dedicated tool support to keep data together and consistent. This is even more true for any manual approach to create something like a view on the system, focusing only on specific aspects and leaving out all the rest. Thus, systems engineering tools or development environments for software and for hardware that provide solutions for the challenges and tasks outlined above are highly recommended.

For using the architectural structure, interface definitions, allocated requirements, etc., for subsequent behavioral specification design, it is also highly recommended to have this systems engineering functionality integrated with a development environment that allows seamless continuation of work on component level as well as automatic integration in the architectural model including system simulation capabilities for validation.

For Model-Based Design—designing mixed software and hardware systems with automatic code generation and including physical parts and the environment for simulation purposes—such systems engineering functionality is available with the products System Composer and Simulink Requirements, which integrate with and extend Simulink with all the capabilities discussed above.

V. CONCLUSION

This paper outlined some key aspects of systems engineering: a) architectural decomposition with parallel requirements decomposition and allocation, b) using stereotypes for components assigning property values for all kinds of non-functional requirements or engineering information, including corresponding analysis of this information on the system level, c) handling systems' inherent complexity with a concept of different views that show only the minimum required context information for the task at hand, and d) without any risk of information loss or inconsistency, seamlessly designing component specifications based on the interface definitions from the architectural model. This includes the capability of simulating the architectural model for validation purposes based on the behavior specified for the components.

Other aspects were not discussed such as the role of an architecture model to enable communication between multiple stakeholders.

Among other things, these are solutions that are recommended to be supported by systems engineering tools to remove error-prone manual data keeping and analysis. System Composer and Simulink Requirements are extensions to Simulink and Model-Based Design that allow users to sketch

hierarchical system and software decompositions of components offering exactly these capabilities.