

UML 2.0-Based Systems Engineering Using a Model-Driven Development Approach

by Hans-Peter Hoffmann, Ph.D.

Abstract

More and more, system engineers are turning to the Unified Modeling Language (UML)[™] to specify and structure their systems. This has many advantages, including verifiability and ease of passing off information to other engineering disciplines, particularly software.

This paper describes a UML 2.0-based process that systems engineers can use to capture requirements and specify architecture. The process uses the UML exclusively for the representation and specification of system characteristics. The essential UML artifacts include use case diagrams, sequence diagrams, activity diagrams, statechart diagrams, and structure diagrams. The process is function-driven and is based heavily on the identification and elaboration of operational contracts, a message-based interface communication concept. The process is part of the I-Logix integrated systems and software engineering process HARMONY and has been applied successfully at various customer sites.

Introduction

For many years, software engineers have successfully applied the UML to model-based software engineering. There have been several attempts to introduce the UML and the underlying object-oriented methods to systems engineering in order to unify the overall development process. However, many systems engineers continue to use classical structured analysis techniques and artifacts to define system requirements and designs. One reason for this could be that systems engineering is mostly driven by functional requirements. Speaking in terms of system functionality is still considered the most natural way of expressing a design by most of the domain experts involved in the early phases of system development (electrical engineers, mechanical engineers, test engineers, marketing, and, of course, customers). Given this, the only way to unify the overall development process is to extend the UML with regard to function-driven systems engineering and to define a process that enables a seamless transfer of respective artifacts to UML-based software engineering. The release of UML 2.0 eventually provided the missing artifacts for function-driven systems engineering.

The following sections describe a UML 2.0-based process that systems engineers can use to capture requirements and specify architecture. The UML is used exclusively for the representation and specification of system characteristics. This allows a seamless transition to UML-based software development. The approach uses model execution as

a means for requirements verification and validation. The outlined process is part of the I-Logix integrated systems and software engineering process, HARMONY.

Process Overview

Figure 1 shows HARMONY, the I-Logix integrated systems and software engineering process. It is a hybrid spiral development process. Systems engineering is characterized by a sequential top-down workflow from requirements analysis to system analysis and system architectural design, with optional feedback loops to previous phases. The spiral part of the process describes the software engineering workflow. It is characterized by the iterative and incremental cycles through the software analysis and design phase, the implementation phase, and the different levels of integration and testing.

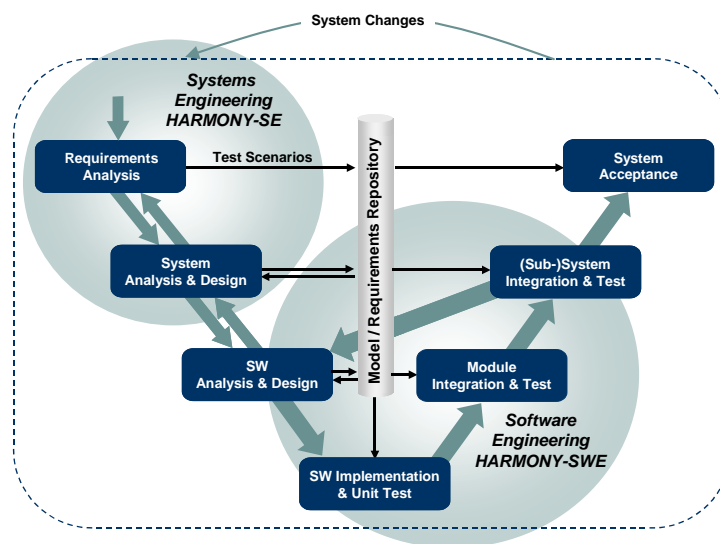


Figure 1. I-Logix Integrated Systems and Software Engineering Process (HARMONY)

HARMONY is a model-based development process using UML 2.0 as the modeling language. It provides a seamless integration between functional (systems engineering) and object (software engineering) modeling.

It is important to note the creation and the reuse of requirements-related test scenarios all along the top-down design path. These scenarios are also used to assist the bottom-up integration and test phases and, in the case of system modifications, regression test cycles.

Within HARMONY, the key objectives of the model-driven systems engineering process are:

- > Identification and derivation of required system functionality
- > Identification of associated system states and modes
- > Allocation of system functionality and modes to a physical architecture

With regard to modeling, these key objectives imply a top-down approach on a high level of abstraction. The main emphasis is on the identification and allocation of a needed functionality (e.g., a target tracker), rather than on the details of its behavior (e.g., the tracking algorithm).

Figure 2 depicts an overview of the UML-based systems engineering process. For each of the systems engineering phases, it shows the essential tasks, associated input, and work products.

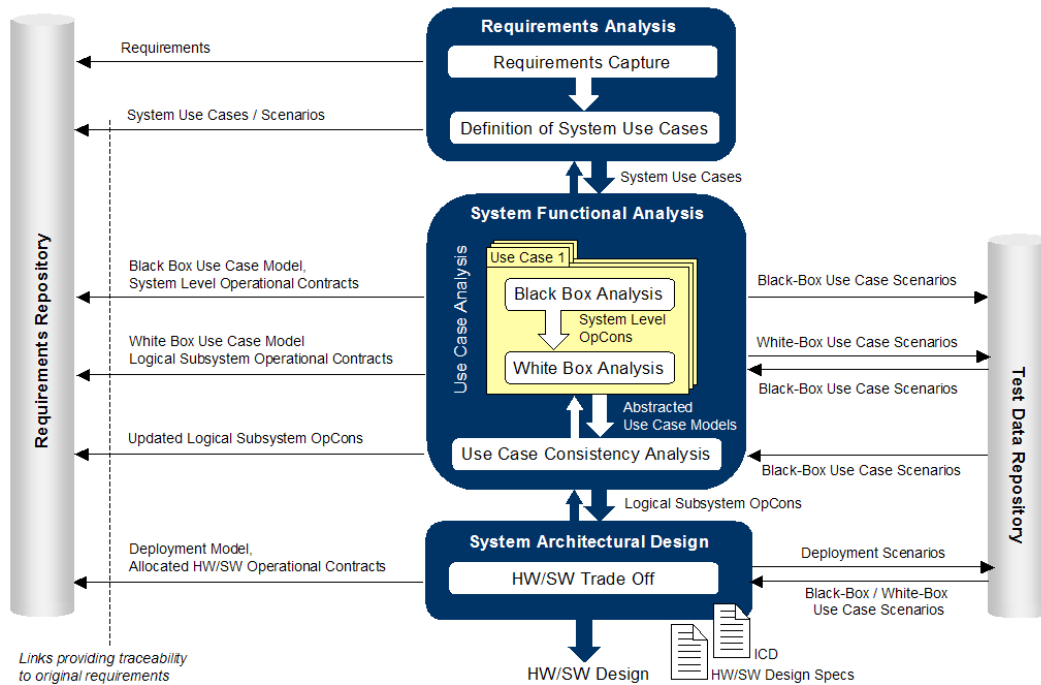


Figure 2. UML-Based Systems Engineering Process

The process is operational contract (OpCon)-driven. An operational contract specifies system behavior by adding pre- and post-conditions to the description of respective operations. Operational contracts are the primary source of traceability to the initial requirements.

The essential tasks in the requirements analysis phase are requirements capture, the creation of a requirement taxonomy, and the grouping of requirements in use cases.

The main emphasis of the system functional analysis phase is on the transformation of the identified functional requirements into a coherent description of system functions (operational contracts). For each use case, the analysis is performed in two steps: black-box analysis and white-box analysis. In black-box analysis, the system-level operational contracts are identified and verified/validated through the execution of the corresponding black-box use case model.

In white-box analysis, the system-level operational contracts are allocated to logical (functional) subsystems and further decomposed. The logical subsystem operational contracts are verified and validated through the execution of the corresponding white-box use case model.

The final step in the functional analysis phase is use case consistency analysis, in which the verified and validated use case models are integrated into a common framework and executed as concurrent processes. The use case collaboration is verified through regression testing.

The focus of the subsequent system architectural design phase is the allocation of the verified and validated operational contracts to a physical architecture. The allocation is an iterative process. It can be performed for different physical system architectures and elaborated on different allocation strategies in collaboration with domain experts (for example, mechanical engineers, electrical engineers, software engineers, and test engineers). Additionally, decisions are made on which operational contracts within a physical subsystem are handled in hardware and which are handled in software (HW/SW trade-off analysis). At this stage, quality of service requirements and safety requirements that were captured in the requirements analysis phase must be considered. The different design concepts are captured in the deployment model. The deployment model is verified through regression testing.

The following documents are generated from the deployment model and handed off to the subsequent hardware and software design:

- > HW/SW design specifications
- > Logical interface control document (ICD)

Especially with regard to the software partitions within the system architecture, the documentation may include the definition of essential subsystem use cases and associated scenarios. The use case scenarios are captured in sequence diagrams that are automatically generated during deployment model execution.

The systems engineering process is model-based, using the UML 2.0 modeling language. The essential UML artifacts are:

- > Use case diagram
- > Activity diagram
- > Statechart diagram
- > Sequence diagram
- > Structure diagram

In addition, SysML requirements diagrams are used to create a taxonomy of the captured requirements. Figure3 shows the relationship between these diagrams.

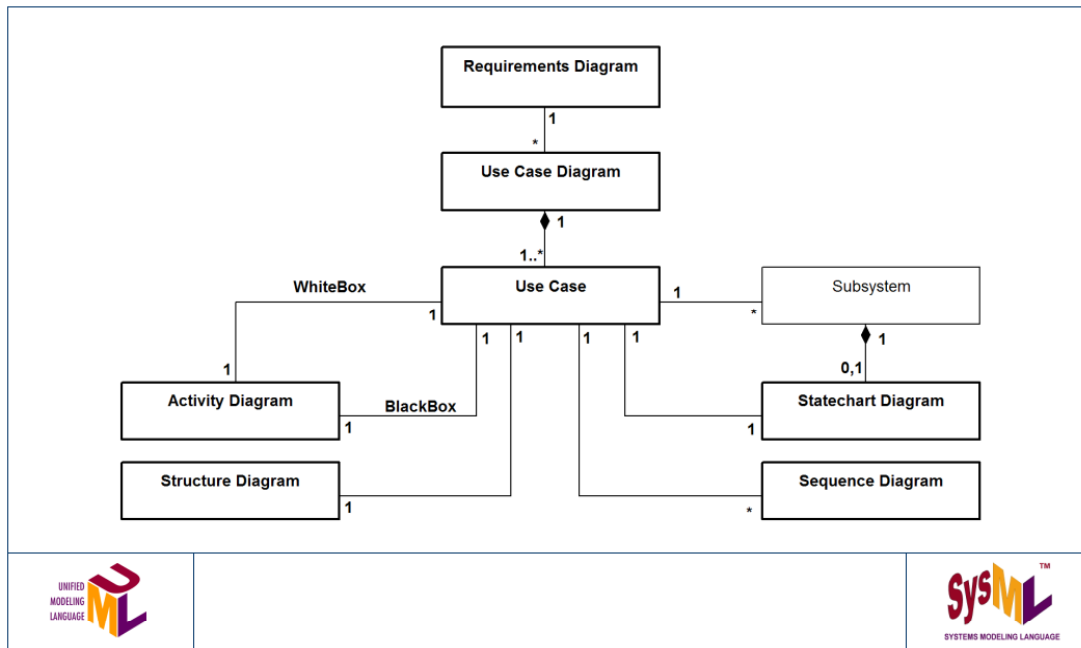


Figure 3. Essential UML/SysML Artifacts for Systems Engineering

UML Based Systems Engineering Workflow

Requirements Analysis

Requirements Capture

The requirements analysis phase starts with the analysis of the process inputs. Customer requirements are translated into a set of requirements that define what the system must do (functional requirements) and how well it must perform (quality of service requirements). SysML requirements diagrams are used to create a taxonomy of the captured requirements (Figure 4).

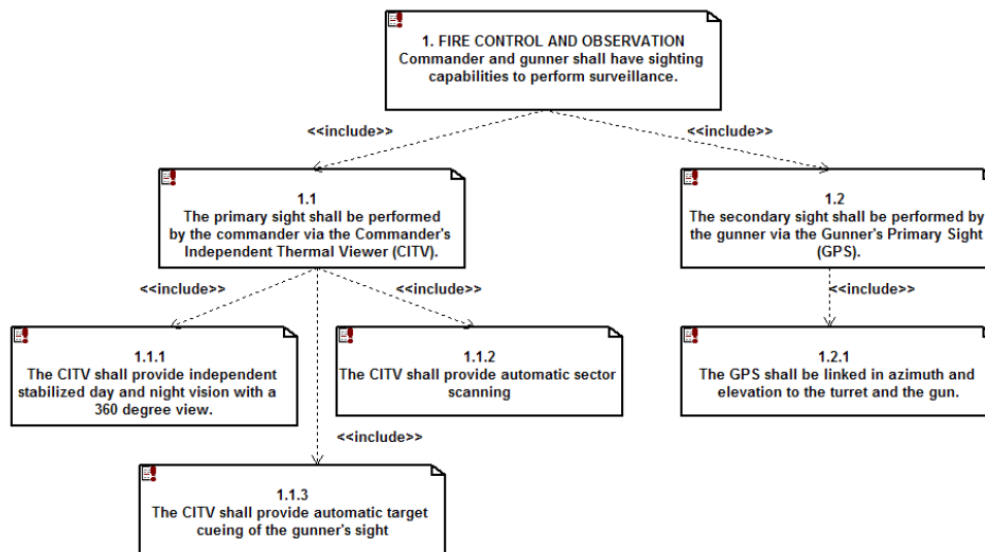


Figure 4. SysML Requirements Diagram

Sequence diagrams can be used to detail requirements. Typically, they are associated with respective requirement elements in the requirements diagram via hyperlinks. The captured requirements can be exported to a requirements management and traceability tool.

Definition of Use Cases

Once the requirements are sufficiently understood, they are clustered in use cases. A use case describes a specific operational aspect of the system (operational thread). It specifies the behavior as perceived by the users and the message flow between the users and the use case. It does not reveal or imply the system's internal structure (black-box view). A set of use cases is grouped in a use case diagram. Use cases can be structured hierarchically. There is no "golden rule" with regard to the number of use cases needed to describe a system. Experience shows that for large systems, typically 6 to 24 use cases are defined at the top level. At the lowest level a use case should be described by at least 5, with a maximum of 25 essential use case scenarios.

At this stage, emphasis is put on the identification of "sunny day" use cases, assuming an error/fail free system behavior. Exception scenarios will be identified at a later stage (=> system functional analysis) through model execution. If more than 5 error/fail scenarios are found for a use case, a separate exception use case will be defined and added to the "sunny day" use case via the include or extend relationship.

System Functional Analysis

The purpose of the system functional analysis phase is to transform the functional requirements that were identified through requirements analysis into a coherent

description of system functions (operational contracts) that will be used in the subsequent architectural design phase. The product is a description of the system in terms of functions and performance parameters, rather than a physical description.

The system functional analysis is model-based. Each use case is translated into a model and the underlying requirements subsequently verified and validated through model execution. The system is modeled using a message-driven approach (Figure 5). Characteristics of this approach are:

- > The system structure is described by means of UML 2.0 structure diagrams using blocks as basic structure elements and ports and the notation of provided / required interfaces for the definition of block interfaces.
- > Communication between blocks is based on messages/service requests.
- > System functionality is captured in activity operational contracts.
- > Functional decomposition is performed through decomposition of activity operational contracts.

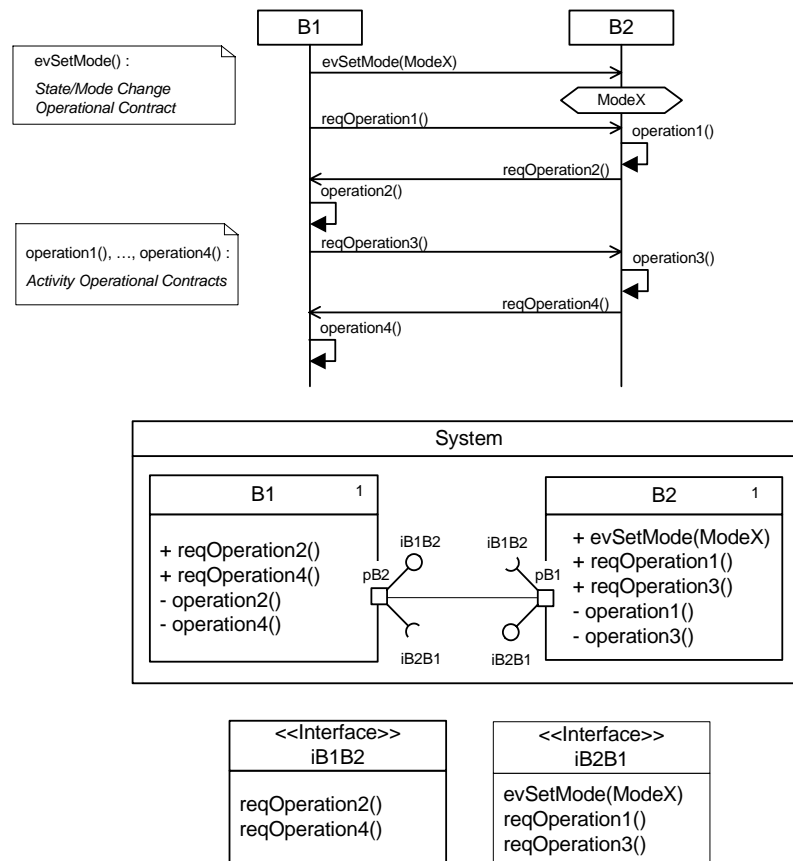


Figure 5. Message-Driven System Modeling Approach

Each use case is analyzed in two steps: black-box use case analysis and white-box use case analysis. Black-box analysis focuses on the identification of system interactions with external actors and the definition of system-level operations. In white-box analysis,

these operations are allocated to logical (i.e., functional) subsystems and further detailed.

Black-Box Analysis

Black-box use case analysis starts with the definition of the use case model context diagram (Figure 6). The UML 2.0 artifact used for this is the structure diagram. Elements of this diagram are the actors and the use case. The blocks are linked via ports. At this stage, the blocks as well as the ports are empty.

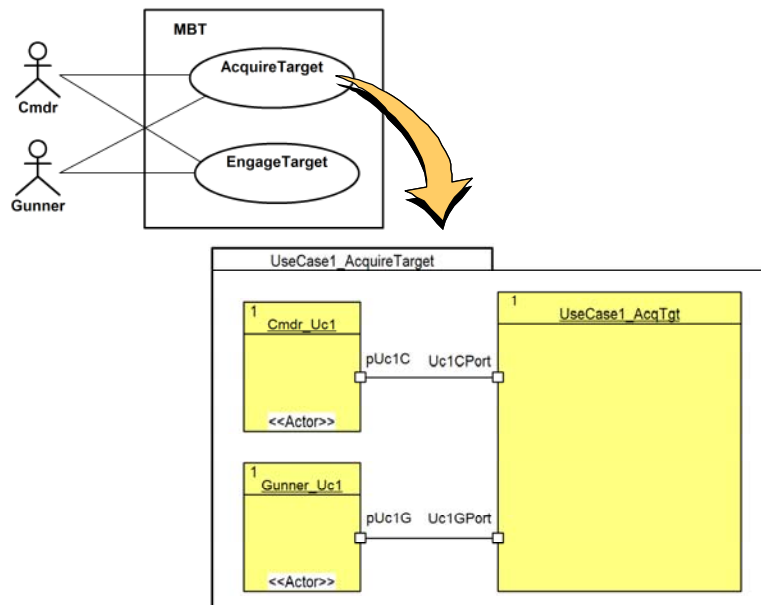


Figure 6. Definition of the Use Case Model Context Diagram

The next step in the black-box use case analysis is the identification of black-box use case scenarios. A use case scenario describes one specific path (functional flow) through a use case. It details the message flow between the actors and the use case and the resulting behavior (activity operational contracts) at the recipient. In the UML, a scenario is graphically represented in a sequence diagram. Lifelines in the black box sequence diagram are the actors and the use case (Figure 7).

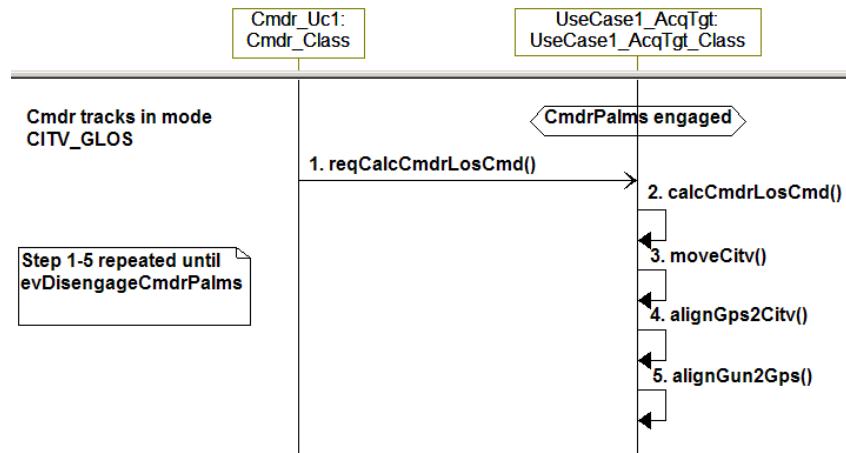


Figure 7. Black-Box Use Case Scenario UC1BB21

Once a set of essential scenarios is captured, the respective functional flow information is merged into a common use case description. The UML artifact used for this is the activity diagram. Each action block in this diagram corresponds to an activity operational contract in a sequence diagram. The use case black-box activity diagram (Figure 8) plays an essential role in the subsequent white-box analysis.

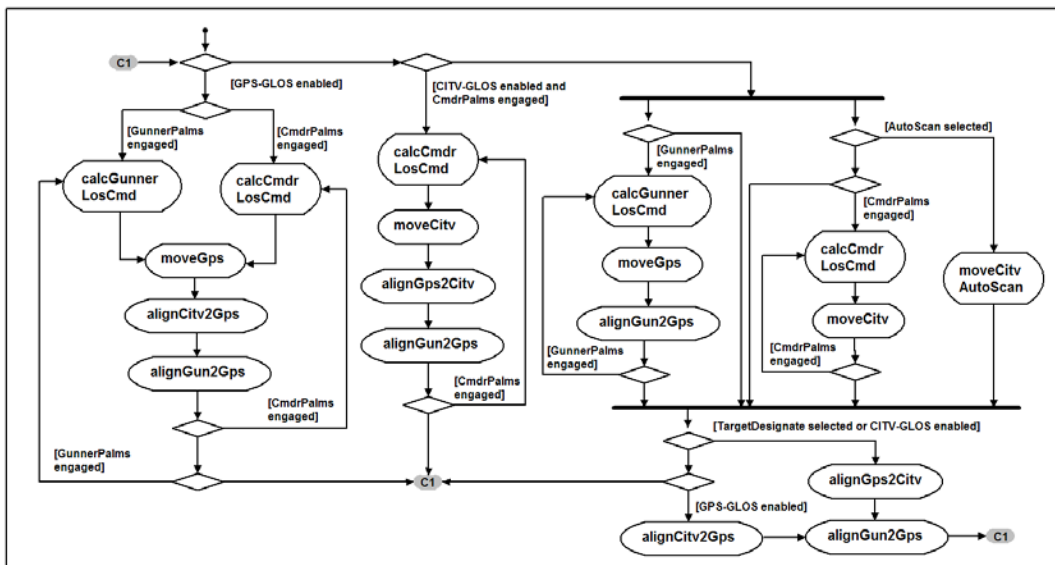


Figure 8. Use Case1 Black-Box Activity Diagram

Based on the information captured in the black-box sequence diagrams and black-box activity diagram, the blocks of the black-box use case model can now be populated. Figure 9 shows a populated black-box use case model. Received messages are marked as public (+) and allocated to the respective interfaces; operational contracts are marked as private (-).

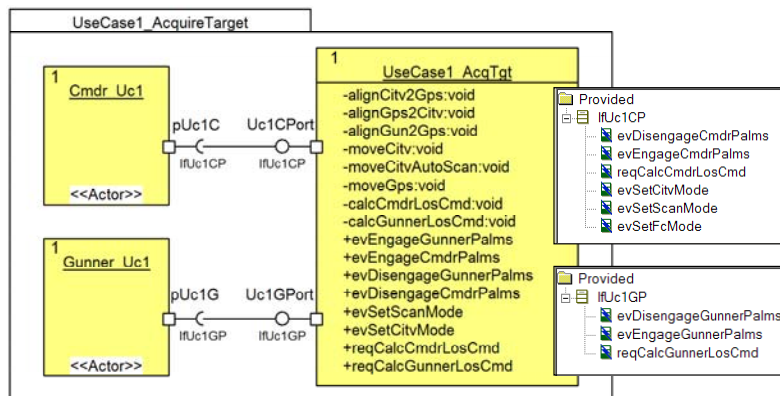


Figure 9. Populated Black-Box Use Case Model

The next step in black-box use case analysis is the description of system-level, state-based behavior. The UML artifact used for this is the statechart diagram (Figure 10). Statecharts are hierarchical state machines that visualize system states and modes and their changes as the response to external stimuli. The use case-related state-based behavior is derived from the captured use case scenarios.

At this stage, the verification and validation of the black-box use case model and the underlying requirements can start. Verification and validation are performed through model execution using the captured black-box use case scenarios as the basis for respective stimuli.

At the end of the black-box use case analysis, the black-box structure diagram and black-box statechart diagram, as well as the system-level operational contracts, are imported into the requirements repository. The operational contracts are linked to the original requirements. The black-box use case scenarios are imported into the test data repository for reuse in the subsequent phases.

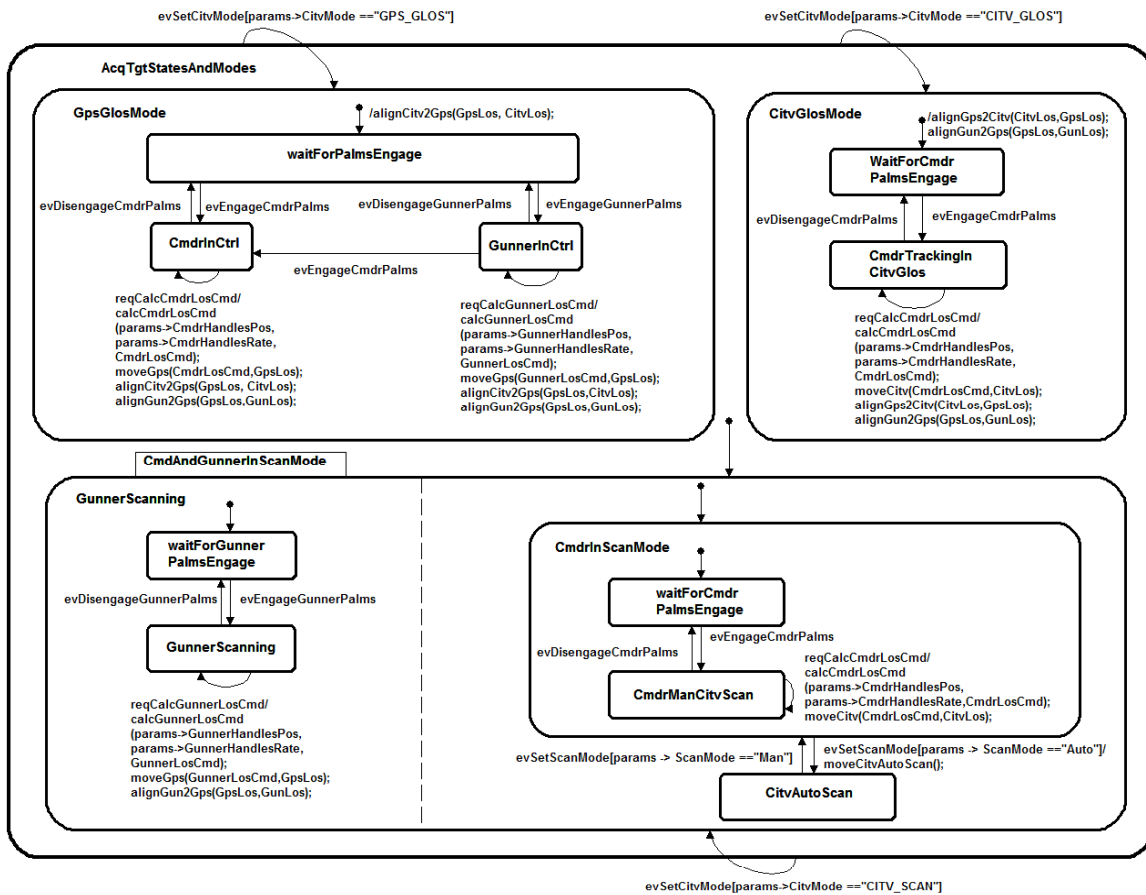


Figure 10. Black-Box State-Based Behavior Description

White-Box Analysis

White-box use case analysis starts with the partitioning of the black-box use case block into functional (logical) subsystem parts (white-box use case structure diagram, Figure 11).

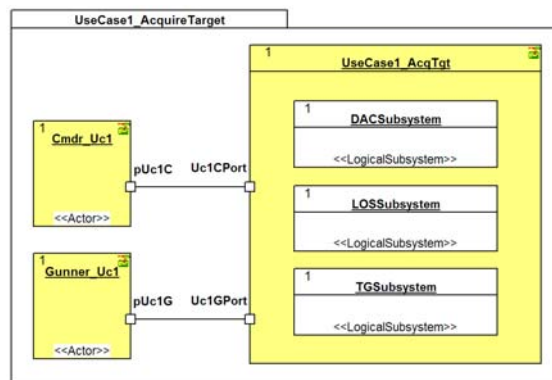


Figure 11. Definition of White-Box Use Case Model Structure

The next step is to allocate the previously defined system-level (black-box) operational contracts to respective subsystems. This is done by means of an activity diagram

(white-box activity diagram). This activity diagram is divided into columns (swim lanes), with each swim lane corresponding to a subsystem. Next, the black-box activity diagram is mapped to the white-box activity diagram structure. An essential precondition for this mapping is that the initial links (functional flow) between the operational contracts are maintained (Figure 12).

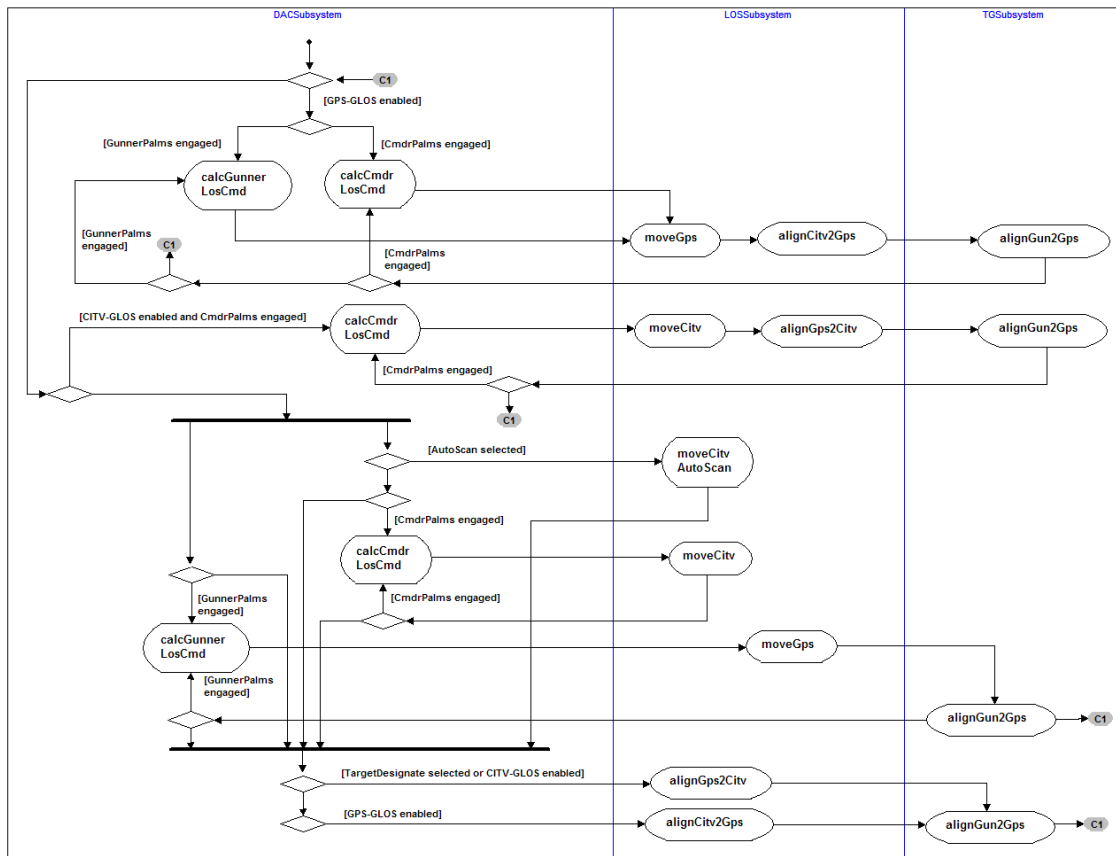


Figure 12. Use Case1 White-Box Activity Diagram

Once all system-level operational contracts are allocated to logical subsystems, they can be further decomposed. Experience shows that with regard to its “resolution,” a system-level operational contract should be decomposed no deeper than two hierarchy levels. Sometimes a sibling of the decomposed operational contracts can be allocated to a different logical subsystem.

White-box activity diagrams are complemented by the definition of white-box sequence diagrams. White-box sequence diagrams are extensions of the previously captured black-box sequence diagrams and are utilized to identify the interfaces of the functional subsystems. In white-box sequence diagrams, the use case lifeline is decomposed into a set of subsystem lifelines. The identified subsystem operational contracts are placed on the respective subsystem lifelines as operations. Messages between the lifelines that initiate the required behavior represent the interfaces between the subsystems (Figure 12).

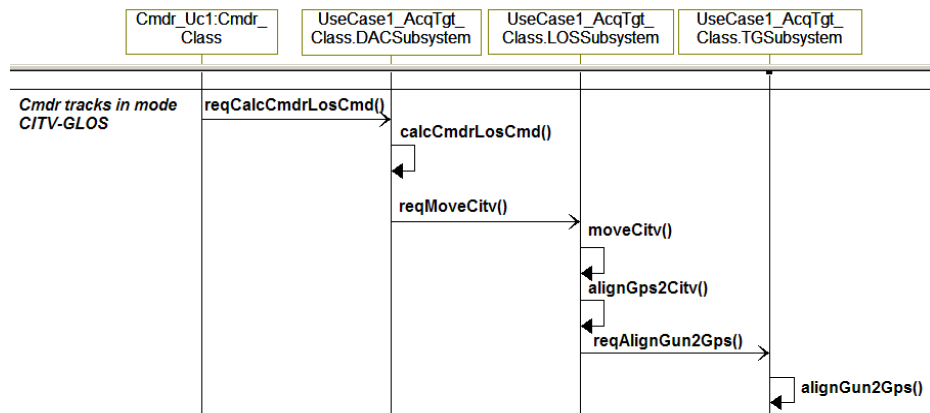


Figure 12. White-Box Use Case Scenario UC1WB21 (Decomposition of UC1BB21 of Figure 7)

Once subsystem operational contracts and the message flow between the subsystems are defined, the white-box structure diagram can be populated. Next, ports are defined and messages are allocated to interfaces (Figure 13).

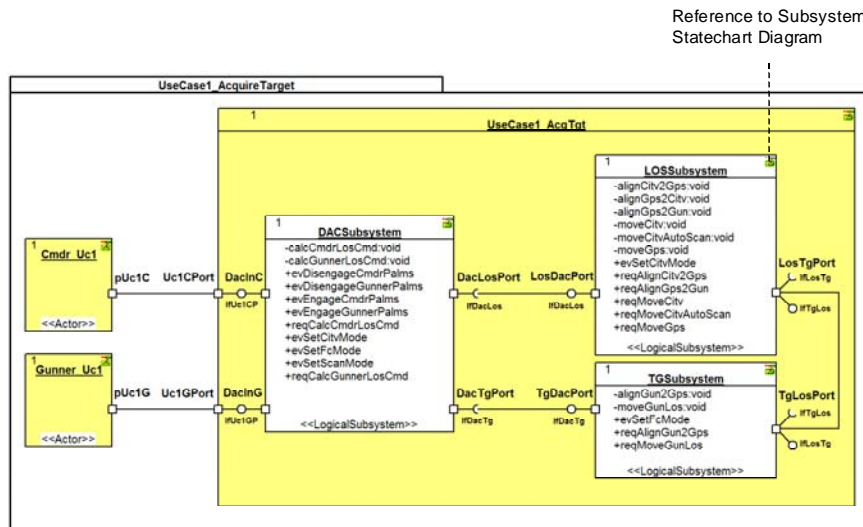


Figure 13. Populated White-box Use Case Model

As with statecharts at the system (black-box) level, statecharts are used to describe the state-based behavior (subsystem modeling) at the subsystem level.

At this stage, the verification and validation of the white-box use case model and the underlying requirements can start. Verification and validation are performed through model execution using the black-box use case scenarios as the basis for respective stimuli.

At the end of the white-box use case analysis, the white-box structure diagram and white-box statechart diagrams, as well as the subsystem-level operational contracts, are

imported into the requirements repository. The subsystem operational contracts are linked to the original requirements. The white-box use case scenarios are imported into the test data repository for reuse in the subsequent phases.

Once both the black-box and white-box use case models are verified and validated, they will be exposed (abstracted) to classes for reuse in the *use case consistency analysis*.

Use Case Consistency Analysis

The final step in the system functional analysis phase is use case consistency analysis. Even when each use case model has been individually verified and validated, there is still the risk that use case models might conflict with each other, especially when they describe common system functionality. Experience shows that use case consistency analysis is particularly needed if different authors built the use case models. In this analysis phase, instances of the verified and validated use case models are imported into a common framework (use case collaboration model, Figure 14) and are executed as concurrent processes. The use case collaboration is verified through regression testing.

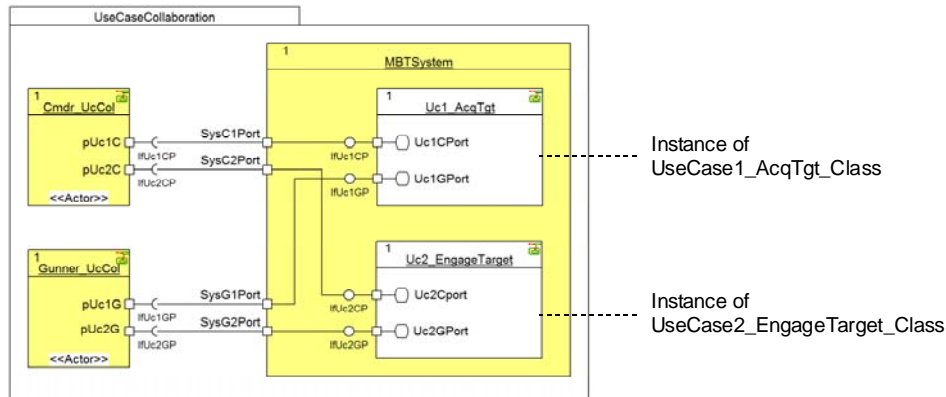


Figure 14. Use Case Collaboration Model

System Architectural Design

The focus of the system architectural design phase is the allocation of the verified and validated operational contracts to a physical architecture. The allocation is an iterative process and is performed in collaboration with domain experts (for example, mechanical engineers, electrical engineers, software engineers, and test engineers). Different architectural concepts and allocation strategies can be analyzed, taking into consideration performance and safety requirements that were captured during the requirements analysis phase.

The system architectural design phase starts with the definition of the deployment model structure. The UML 2.0 artifact used for this is the structure diagram. Constituents of this model are the actor blocks and the system block. Parts of the

system block are the physical subsystems of the chosen architecture. Links between physical subsystems are shown only if required (for example, in the case of a legacy system architecture). Depending on the system design, physical subsystems can be substructured into hardware (i.e., mechanical and/or electrical) parts and a software part.

Based on system design aspects, the logical subsystem activity operational contracts are partitioned into subsystem domains (Figure15). For those operational contracts that span more than one domain, further analysis is needed. Subsystem domain experts may participate in the analysis.

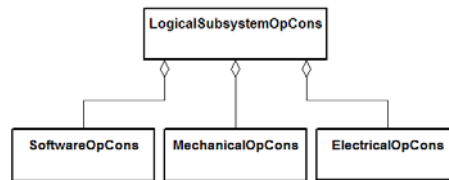


Figure 15. Partitioning of Subsystem Operational Contracts to Subsystem Domains

For a chosen implementation concept, operational contracts are then allocated to the physical subsystems by mapping white-box use case scenarios to the physical architecture accordingly (Figure16).

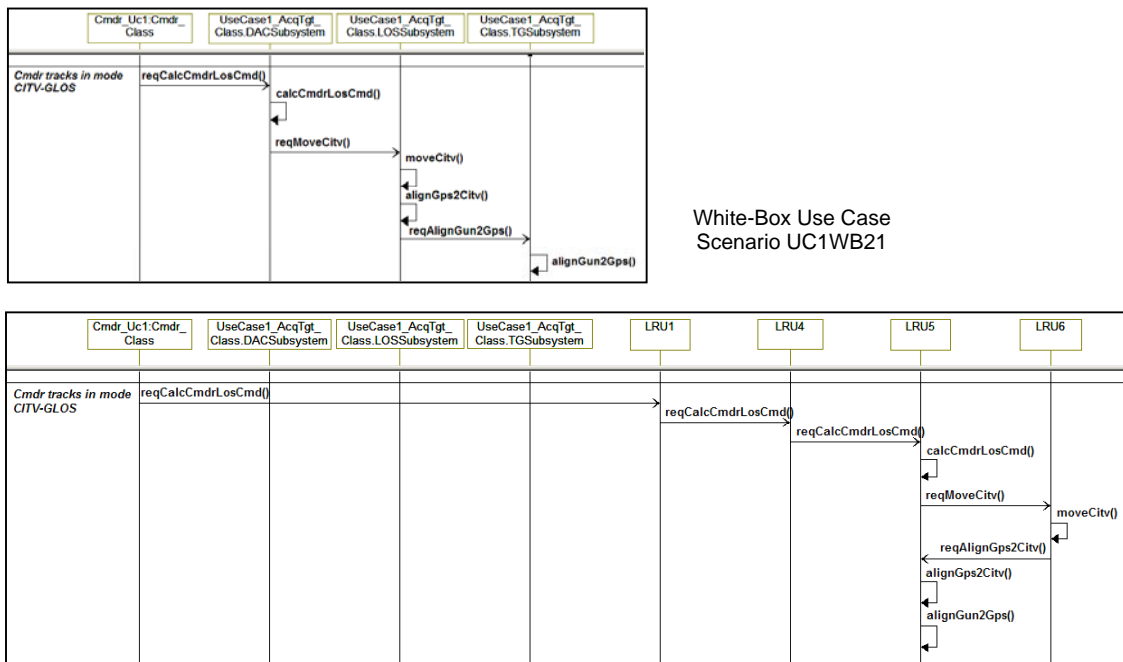


Figure 16. Mapping the White-Box Use Case Scenario UC1WB21 to a Physical Architecture

First, lifelines representing the physical subsystems are added to the white-box sequence diagram. Then, based on the implementation concept, the white-box operational contracts are “moved” to respective lifelines. In order to maintain the initial

functional flow, service requests from one physical subsystem to the other may need to be generated. In the example shown in Figure 16, the implementation concept was that LRU1 was considered the commander's I/O device. Most of the identified functionality had to be implemented in LRU5. The CITV control had to be in LRU6. In this scenario, LRU4 served as a gateway between LRU1 and LRU5. Figure 17 shows for each associated physical subsystem the allocated messages and operations.

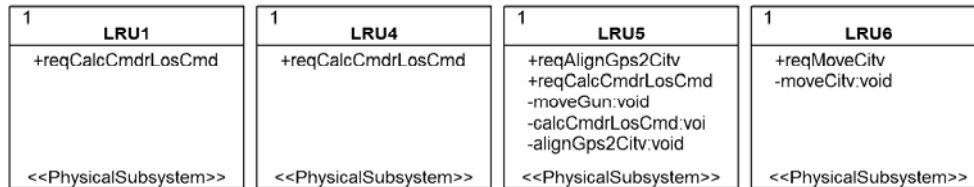


Figure 17. Messages and Operations of Scenario UC1WB21 Allocated to Physical Subsystems

By mapping the use case scenario to the deployment architecture, the links and the associated ports and interfaces are defined for each involved physical subsystem. For each physical subsystem, the resulting state-based behavior is captured in a statechart diagram. This statechart diagram will be extended incrementally for each mapped use case scenario.

The outlined process must be performed iteratively for all use cases and associated white-box use case scenarios. Figure 18 shows the final result.

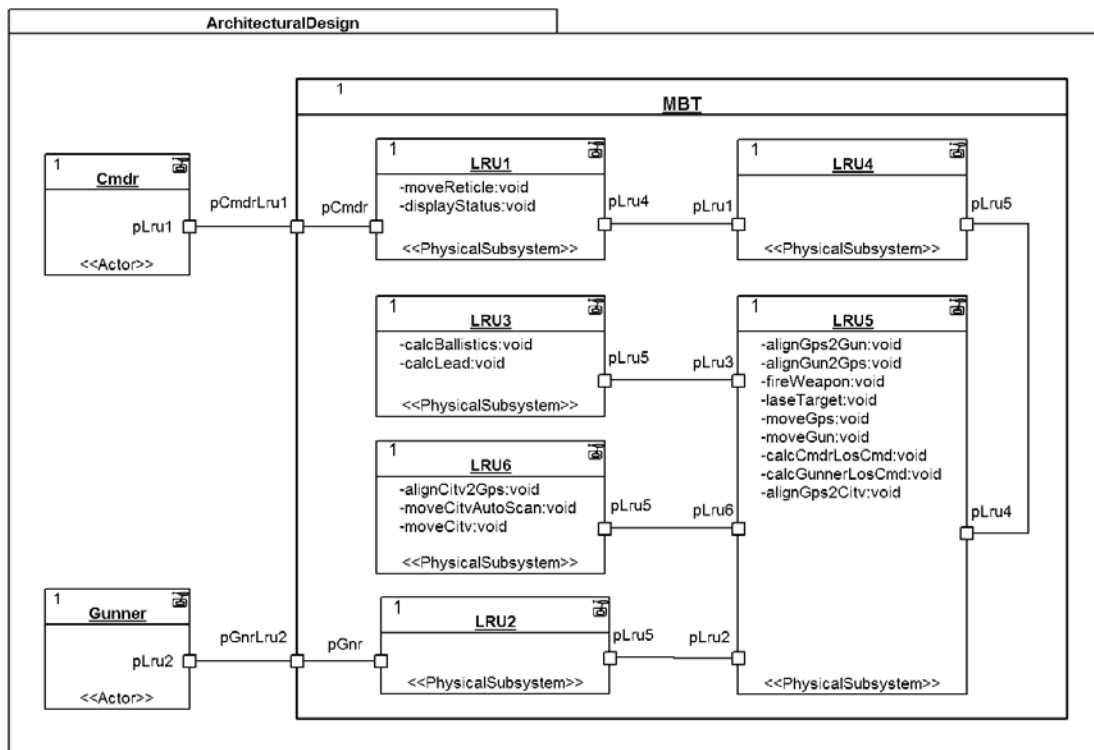


Figure 18. Activity Operational Contracts Allocated to an LRU Network Architecture (Deployment Model)

Figure 19 depicts for the chosen architectural design the resulting physical subsystem interfaces by means of an N-squared (N²) chart. An N² chart is structured by locating the nodes of communication on the diagonal, resulting in an NxN matrix for a set of N nodes. For a given node all outputs (UML 2.0 *required interfaces*) are located in a row of that node and inputs (UML 2.0 *provided interfaces*) are in the column of that node.

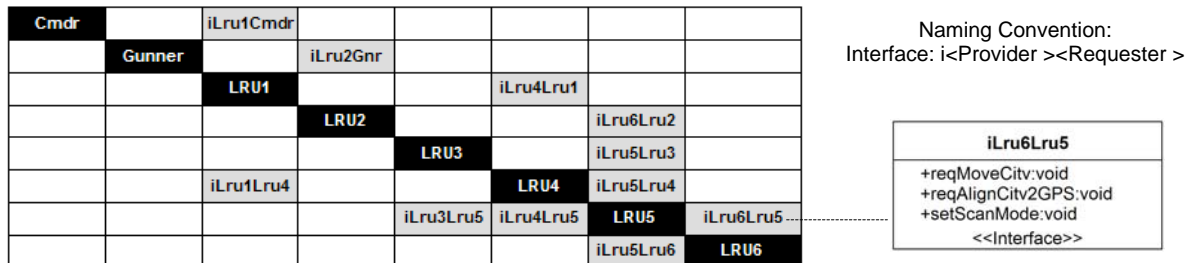


Figure 19. Documentation of Physical Subsystem Interfaces

The correctness and completeness of the deployment model is checked through model execution. Once the model functionality is verified, the architectural design can be analyzed with regard to the performance and safety requirements. Safety-related analysis typically includes FMEA and mission criticality analysis.

At the end of the system architectural design phase, the deployment model artifacts and the HW/SW assigned operational contracts are imported into the requirements repository. The operational contracts are linked to the original requirements. In addition, the following documents are generated from the deployment model as handoffs to the subsequent hardware and software design:

- > HW/SW design specifications
- > Logical interface control document (ICD)

Experience shows that, especially with regard to the software partitions within the system architecture, the documentation should include the definition of essential subsystem use cases and associated scenarios. The use case scenarios are captured in sequence diagrams that are automatically generated during deployment model execution. They are imported into the test data repository for reuse in the subsequent software design and implementation phases.

Conclusion

The UML has had a dramatic effect on software development. Although the many features within the UML can be linked to its success, one main benefit is that it enables a company to standardize on a single development language, thus facilitating communications and staff alignment between and within development groups. With UML 2.0, many of the same features and benefits of UML are now available to systems engineers. In this case, the benefit of a single language goes even further, because it does for systems engineers what it did for the software engineers. Normally, a major “minefield” exists between systems design and software development. The fact that

both entities now can communicate in a “common” language has an enormous effect on building a pathway through that minefield. That ability, plus the availability of new generation modeling tools such as Rhapsody® by I-Logix (which uses all the facilities in UML 2.0), provide users with a common and seamless environment that spans from systems to software.

BUT ... just as a automobile full of gas has a lot of potential benefits (and certainly many features), it is not until there is a “roadmap” that it all comes together and is suddenly valuable ... as with that roadmap ... people can go somewhere they haven’t been before. The integrated systems/software development process HARMONY of I-Logix is that roadmap. It provides value to both the modeling tool and the UML language. Each component (language, tool, and process) works in HARMONY to enable designers and developers to effectively and efficiently go where they have not gone before - despite their often diverse roles and responsibilities. As products continue to get more complicated and design and development teams get more distributed, and as market pressures continue to challenge development cycles and quality, the only way companies can achieve their goals will be, if they can somehow get all their resources to work in HARMONY! Now they can.

About the Author

(Hans-Peter Hoffmann is Director and Chief Methodologist for Systems Design at I-Logix, a leading Real-Time Object-Oriented and Structured Systems Design Automation Tool vendor. His focus is methodology consulting (“From Concept to Code”). He works internationally as a consultant for model-based system development in the Aerospace, Defense, and Automotive industries. You may contact him at peterh@ilogix.com.)