Seminar / Workshop

Best Practices for Model-Based Systems Engineering

Hans-Peter Hoffmann, Ph.D.
Chief Systems Methodologist, IBM Rational Software
hoffmape@us.ibm.com

IBM Software
Overview

Successfully delivering complex systems requires the development of optimal designs on time, within budget and meeting quality standards. In this hands on workshop, learn how to apply IBM Rational's *Best Practices for Model-based Systems Engineering* to help improve system architecture from the start. It will be shown how modeling and model execution supports the different development phases. The role of testing, as well as requirements traceability throughout the development process, will be addressed. After an introduction to the fundamentals of the model-based systems engineering approach (essential UML/SysML artifacts for systems engineering, service request-driven system modeling approach), the model-based systems engineering workflow and the generation of associated work products will be demonstrated by means of a simple example (Security System).

1 Introduction

Fig.1-1 shows the Rational Integrated Systems / Embedded Software Development Process *Harmony* by means of the classic “V” diagram. The left leg of the “V” describes the top-down design flow, while the right hand side shows the bottom-up integration phases from unit test to the final system acceptance. Using the notation of statecharts, the impact of a change request on the workflow is visualized by the “high-level interrupt”. Whenever a change request occurs, the process will restart at the requirements analysis phase.
The Harmony process consists of two closely coupled sub-processes

- **Harmony for Systems Engineering** and
- **Harmony for Embedded Real Time Development**

The systems engineering workflow is iterative with incremental cycles through the phases requirements analysis, system functional analysis and design synthesis. The increments are use case based.

The software engineering workflow is characterized by the iterative incremental cycles through the software analysis and design phase, the implementation phase, and the different levels of integration and testing.

Regarding the systems engineering and implementation iterations, it should be noted, that the analysis iterations continue through implementation and testing, thus providing something demonstrable with each iteration.

It is important to note the creation and 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 changes, regression test cycles.

The Harmony process supports Model-Driven Development (MDD). In a model-driven development, the model is the central work product of the development processes, encompassing both analysis and design. Each development phase is supported by a specific type of model.

Models that support the requirements analysis phase are
- the Requirement Models and
- the System Use Cases Model.

A requirement model visualizes the taxonomy of requirements. The system use cases model groups requirements into system use cases. Neither of these models is executable.

In the system functional analysis phase the focus is on the translation of the functional requirements into a coherent description of system functions (operations). Each use case is translated into an executable model and the underlying system requirements verified through model execution.

There are two types of executable models supporting the design synthesis phase:
- Architectural Analysis Model(s) and
- System Architecture Model

The objective of the architectural analysis model(s) - also referred to as Trade Study Model(s) - is to elaborate an architectural concept for the implementation of the identified operations e.g. through a parametric analysis.

The system architecture model captures the allocation of the system operations to the system architecture that was elaborated in the previous architectural analysis phase. The correctness and completeness of the system architecture model is verified through model execution. Once the model is verified, the architectural design may be analyzed with regard to performance and safety requirements. The analysis may include Failure Modes Effects Analysis (FMEA), and Mission Criticality Analysis.

The baselined system architecture model defines the hand-off to the subsequent HW/SW development.
Model-driven software development is supported by the *Software Implementation Model*. This model is the basis for - either manual or automatic - code generation.

An essential element of the model-driven development process is the *Model/Requirements Repository*. It contains the configuration controlled knowledge of the system under development, i.e.
- Requirements documentation
- Requirements traceability
- Design documentation and
- Test definitions

## 2 Fundamentals of model-based systems engineering

### 2.1 Essential SysML artifacts

Fig.2.1-1 shows the SysML diagrams that are used to capture the *structural system view*.

**Fig.2.1-1**  *SysML Diagrams Capturing the Structural System View*

**Block Definition Diagram**: Defines the structural elements (Blocks) of the system and their relationship.

**Internal Block Diagram**: Defines the realization of the system structure.

**Parametric Diagram**: Defines the parametric relationship between system properties.
Fig. 2.1-2 shows the SysML diagrams that are used to capture the system behavior. **Note:** It is recommended to follow the outlined sequence of the system behavior elaboration.

**Use Case Diagram:** Defines the system scope and groups requirements into Use Cases (“Table of Contents of SuD User Manual”).

**Activity Diagram:** Shows at the functional analysis level the functional flows through the use case (“Storyboard”). At the architectural design level, it shows the allocation of decisions and operations across the system/subsystem architecture.

**Sequence Diagram:** Shows at the functional analysis level the allocation of operations and the message interactions between use cases and associated actors. At the architectural design level, it shows the allocation of operations and the message interactions between systems, subsystems, subsystem components, and actors.

**Statechart Diagram:** Aggregates the information from the activity diagram (functional flow) and the sequence diagrams (actor interactions). It puts this information into the context of system states and adds to it the system behavior due to external stimuli of different priority.
2.2 Service request-driven modeling approach

In the *Service Request-Driven Approach*, the communication between blocks is based on asynchronous messages ("service requests") via UML/SysML *Standard Ports*. A service request always is followed by an associated provided service at the receiving part – either state/mode change or operation.

First, the service requests and associated operations have no arguments. At a later stage arguments may be added to the service requests and associated operations as arguments or tags of the relevant service request and associated operation.

The approach is performed in four steps:

1. **Describe Inter-Nodal Communication**
   - e.g. derived from Activity Diagram

2. **Define Communication Nodes**
   - *e.g.*
   - Asynchronous service requests via UML/SysML Standard Ports followed by
   - Provided services at the receiving part (state/mode changes or operations)

3. **Allocate Service Requests and Operations**

4. **Create Ports and Interfaces**

In the *Service Request-Driven Approach*, the communication between blocks is based on asynchronous messages ("service requests") via UML/SysML *Standard Ports*. A service request always is followed by an associated provided service at the receiving part – either state/mode change or operation.

First, the service requests and associated operations have no arguments. At a later stage arguments may be added to the service requests and associated operations as arguments or tags of the relevant service request and associated operation.

The approach is performed in four steps:

1. **Describe Inter-Nodal Communication**
   - e.g. derived from Activity Diagram

2. **Define Communication Nodes**
   - Asynchronous service requests via UML/SysML *Standard Ports* followed by
   - Provided services at the receiving part (state/mode changes or operations)

3. **Allocate Service Requests and Operations**

4. **Create Ports and Interfaces**
3 Task Flow and Work Products in the Model-based SE Approach

3.1 Requirements Analysis

**Hands-on Exercise**

In this exercise it will be demonstrated how requirements – documented in a Word document - are imported into Rhapsody by means of the Rational **Gateway** tool.

Open the Rhapsody project **SecuritySystem_RA**.

The project structure of this model is the generic Harmony structure automatically generated by means of the **SE-Toolkit** (“Create Harmony Project”). An essential pre-condition for this is that the HarmonySE profile was added to the SysML profile.

**NOTE:** The toolkit automatically creates an empty Use Case Diagram(UCD_< project name>).

IBM Software
Task 1: Import Requirements

1. Launch Gateway: Right-click SecuritySystem_RA > Rational Rhapsody Gateway > Open

2. In Gateway, select File > Edit Project

3. Browse to the folder **Security System Nominal Requirements**

4. Drag the folder onto the Gateway project canvas

5. Drag a coverage link from the UML Model to the requirements document.

6. Click OK.
Task 2: Add Requirements to Rhapsody

1. In Gateway, select the UML Model document in the Selection pane.

2. On the Gateway Tools menu, select Add high level requirements.

3. In the dialog box, expand the Rhapsody browser structure and select the RequirementsPkg

4. Click OK.

5. Close the Gateway and return to Rhapsody
Task 3: Create Use Case Diagram

1. Double-click UCD_SecuritySystem_RA (automatically generated when creating a Harmony project).
2. Use the drawing tool to create the following Use Case Diagram:

![Use Case Diagram Image]

Task 4: Link Use Cases to System Requirements

Use cases are linked to the associated system requirements through a <<trace>> dependency. The creation of this dependency is supported by the SE-Toolkit.

1. In the tools menu select Tools > SE-Toolkit > Modeling Toolbox. In the dialog box select Dependencies, Profile: PredefinedTypes, Stereotype: trace.
2. In the UseCaseDiagramPkg select the use case Uc1ControlEntry. In the dialog box click Set Source.
3. In the Security_System_Nominal_Requirements package select requirements SRN01–SRN05. In the dialog box click Set Destination.
4. In the dialog box click Create Dependency with Stereotype.
**Task 4 Setup System Functional Analysis**

The transition from model-based Requirements Analysis to model-based System Functional Analysis is supported through the SE-Toolkit.

In the use case diagram UCD_SecuritySystem_RA right-click the use case and select **SE-Toolkit > Create System Model From Use Case**.

The toolkit feature moves the use case associated actors into the ActorPkg.

Automatically, the FunctionalAnalysisPkg is populated:

- A system block Uc_Uc1ControlEntry is created.
- An IBD_Uc1ControlEntry is created containing the instances of the actors and the use case block (no links between the parts).
- The use case – incl. its associated requirements links – is moved into the Uc1ControlEntryPkg. In addition, the toolkit created an empty Activity Diagram (Uc1ControlEntryBlackBoxView).
3.2 System Functional Analysis

Documents created at the end of the System Functional Analysis:

The System-Level Interface Control Document (ICD) defines the logical (= functional) interfaces between the (black-box) system and its actors and is the aggregate of all use case blocks interfaces of an iteration increment. This ICD is the basis for the later system-level (black-box) test definition.

**Hands-on Exercise**

In this exercise it will be demonstrated how the “use case story” is translated into an executable use case model. First, the functional flow will be captured in an Activity Diagram. Then the interactions with the environment will be derived via Sequence Diagram(s). Finally, the complete use case behavior will be captured in a Statechart Diagram based on the Activity Diagram and Sequence Diagram(s) information. The underlying requirements then will be verified/validated through model execution.

Open the Rhapsody project **SecuritySystem**

This project contains the complete – executable – model of the use case *Uc1ControlEntry* based on the nominal security system requirements imported into Rhapsody in the previous Requirements Analysis phase. **NOTE:** It is highly recommended to extend the behavior w.r.t. the extended security system requirements listed and realized in the Appendix. For the import of the respective requirements into Rhapsody via the Gateway tool please use the Word document **Security System Extended Requirements**.
Task 1: Capture Use Case Functional Flow

The location of the use case Activity Diagram is FunctionalAnalysisPkg > Use Cases > Uc1ControlEntry > ActivityViews > Uc1ControlEntryBlackBoxView

Double-click NominalUseCase

- Actions stereotyped <<MessageAction>> describe the reception or transmission of a message.
- Harmony for Systems Engineering uses a SysML activity pin stereotyped ActorPin to visualize the interaction of an action with the environment. The name of the pin is the name of the associated actor. The arrow in the pin shows the direction of the link.

In order to add an ActorPin to an action, right-click the action and select SE-Toolkit > Add Actor Pins.
Optionally extend the nominal flow w.r.t. the extended system requirements listed in the Appendix.

NOTE: In this case, the Admin needs to be added as actor to the Use Case Diagram in the RequirementsAnalysisPkg. Once the diagram is updated, right-click the use case and select `SE-Toolkit > Create System Model From Use Case`. Through this the Admin is added to the ActorPkg.

Task 2: Derive Use Case Scenario(s)

Use case scenarios are created from the Activity Diagram by means of the SE-Toolkit.

1. In the Activity Diagram window right-click and select `SE-Toolkit > Generate Sequence Diagrams`.

2. Hold down Ctrl and select in the Activity Diagram a sequence of actions. Alternatively select a single action as the source. In this case the tool will auto-create the sequence until it reaches a condition connector or a sync bar. The user is then given the choice of which path to take.

3. In the dialog box click `Set Source`

4. In the dialog box select `Create Events` and then `Create New Scenario From Activity Diagram`. 
NOTE: The Interaction Operators and the Operand Separator are added manually.

NOTE: Using the SE-Toolkit, automatically
- Actions are translated into operations (incl. associated descriptions and tags) and
- MessageActions with ActorPins are translated to respective Actor receptions.

The generated Sequence Diagram(s) are automatically stored in the Uc1ControlEntryBBScenariosPkg.
Task 3: Create Use Case Model Ports and Interfaces

The definition of ports and associated interfaces is automated in Rhapsody by means of the SE-Toolkit feature Create Ports And Interfaces. Pre-condition: All messages and operations in the sequence diagrams are realized.

Naming convention for ports: \( p<\text{TargetName}> \)

Interface names are referenced to the sender port.
Naming convention: \( i<\text{Sender}>._<\text{Receiver}> \)

1. Right-click the package Uc1ControlEntryBBScenariosPkg and select SE-Toolkit > Create Ports And Interfaces.

2. In the browser select FunctionalAnalysisPkg > Uc_Uc1ControlEntry > Internal Block Diagram. Double-click IBD_Uc1_ControlEntry.

3. Manually connect ports.
Task 4: Capture Use Case / Actor(s) Behavior in a Statechart Diagram

A Statechart Diagram describes the state-based behavior of a block. It aggregates the information from both the activity diagram (functional flow) and the sequence diagrams (interactions with the environment), and adds to it the event-driven block behavior. As the “language” of statecharts is formally defined, the correctness and completeness of the resulting behavior can be verified through model execution.

Basically, a Statechart Diagram is composed of a set of states joined by transitions and various connectors. An event may trigger a transition from one state to another. Actions can be performed on transitions and on state entry/exit.

State-based behavior of the use case block Uc_Uc1ControlEntry
In order to execute the use case model closed-loop, also the behavior of the actors has to be captured. The Rhapsody SE-Toolkit provides a feature to automatically generate the actor behavior based on the actor’s provided/required interface information:

In the use case Internal Block Diagram right-click the User block and select Se-Toolkit > Create Test Bench

The captures the User behavior in one state using MOORE syntax (= action is state). It includes already the capability to run model execution via Webify.

Alternatively, the actor behavior may be captured in a more detailed Statechart Diagram:

1. In the use case Internal Block Diagram right-click the User block and select Class > New Statechart
2. Capture the actor behavior in a statemachine.
Task 5: Verify/Validate Use Case Behavior through Model Execution

1. In the toolbar select **Window > Close All**
2. In the toolbar select **Code > Generate/Make/Run**
3. Start model execution by clicking the GO button in the animation toolbar

4. In the toolbar select **Tools > Animated Sequence Diagram**

In the FunctionalAnalysisPkg/UciControlEntryBBScenarioPkg select SC1

5. In the toolbar select **Tools > Animated Statechart**

IBM Software
6. Select Uc_Uc1ControlEntry[0]

7. In the Uc1ControlEntry Statechart open the Sub-Statechart UnlockingAndLockingAccessPoint by clicking the decomposition icon.

8. Arrange the diagrams. You may optimize the layout of each diagram by means of the *Zoom to Fit* button in the toolbar.

9. Start Webify and run the model execution. Use the provided graphical user interface to control the model execution.
You may interrupt the model execution by clicking the *Animation Break* button in the animation toolbar or terminate the model execution by clicking the *Quit Animation* button.
Once the model execution is terminated, the recorded Sequence Diagram is saved in the Uc1ControlEntryBBScenariosPkg as **Animated <Scenario Pattern ID>**.
The focus of the architectural design phase is on the allocation of functional requirements and non-functional requirements to an architectural structure. This structure may be the result of a previous trade study or a given (legacy) architecture. The allocation is an iterative process and is typically performed in collaboration with domain experts.

Architectural design is performed incrementally for each use case of an iteration by transitioning from the black-box view to the white-box view – also referred to as use case realization. The task flow is quite similar to the one outlined for the System Functional Analysis.

It starts with the definition of the system architectural structure. Based on the chosen design concept the use case block is decomposed into its relevant system architecture parts. The resulting structure is captured in a SysML Block Definition Diagram (BDD) and Internal Block Diagram (IBD).

Next, the system-level use case operations are allocated to the relevant subsystems by means of the associated Use Case White-Box Activity Diagram. Essentially, this activity diagram is a copy of the Use Case Black-Box Activity Diagram, partitioned into swim lanes, each representing a block of the system architectural decomposition hierarchy. Based on the chosen design concept, the system-level operations (activities) then are “moved” into respective block swim lanes. An essential requirement for
this allocation is that the initial links (control flow) between the activities are maintained. If an action cannot be allocated to a single block, it must be further decomposed. In this case, the sub-operations need to be linked to the parent operation through a respective dependency. An action/operation may also be allocated to more than one block, e.g. in order to meet fault tolerance requirements (architectural redundancy). In this case, the relevant operation/action is copied into the respective block swim lane and integrated into the functional flow. This process may be repeated iteratively for each level of architectural decomposition, resulting in a nested Use Case White-Box Activity Diagram.

White-box activity diagrams provide an initial estimate of the resulting load on respective communication channels, as links that cross a swim lane correspond to interfaces.

Dependent on the hand-off to the subsequent development, the subsystem block(s) - and associated white-box activity diagram may need to be further decomposed. At the lowest level, the functional allocation may address which operation should be implemented in hardware and which should be implemented in software.

From the final Use Case White-Box Diagram, associated White-Box Sequence Diagrams are derived. As outlined previously, these sequence diagrams are the basis from which ports and interfaces of the blocks at the lowest level of the system architecture are derived.

Once system-level operations are allocated to the relevant blocks at the lowest level of the architectural decomposition and associated interfaces are defined, the individual state-based behavior is captured in a Statechart Diagram. The leaf-block behavior as well as the collaboration of the decomposed subsystems then is verified through model execution.

The last step in the use case realization task flow is the allocation of non-functional requirements to the relevant part(s) and/or operations (e.g. time budgeting). Respective trace links need to be established.

The final task in the design synthesis phase is the creation/update of the Integrated System Architecture Model. This model is the aggregate of the realized use case models. The use cases collaboration as well as the correctness and completeness of the integrated system architecture model may be verified through model execution.