February 2011

Systems Engineering Best Practices with the Rational Solution for Systems and Software Engineering

Deskbook  Release 3.1.2

Model-Based Systems Engineering with Rational Rhapsody and Rational Harmony for Systems Engineering

Hans-Peter Hoffmann, Ph.D.
Chief Systems Methodologist

hoffmape@us.ibm.com

The Deskbook is written for the practitioner. Screenshots, notes and best practice tips are added to the workflow descriptions. The brief introductions are minimal rather than narrative. The Deskbook is not intended to replace IBM Rational Rhapsody training; it is intended to supplement it. It is assumed that the reader is familiar with UML/SysML and the IBM Rational Rhapsody tool.

Permission to use, copy, and distribute, this Deskbook, is granted; provided, however, that the use, copy, and distribution of the Deskbook is made in whole and not in part.

THIS DESKBOOK IS PROVIDED "AS IS." IBM MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.

IBM WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DESKBOOK OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS OF THE DESKBOOK.

The directory "Deskbook Rel3.1.2 Requirements and Models" contains the requirements specification for the Security System example and snapshots of the models generated with Rhapsody.

Copyright IBM Corporation 2006, 2011
IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Licensed Materials - Property of IBM Corporation
U.S. Government Users Restricted Rights: Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

IBM, the IBM logo, Rational, the Rational logo, Telelogic, the Telelogic logo and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both.

Other company, product, or service names may be trademarks or service marks of others.

The Rational Software home page on the Internet can be found at ibm.com/software/rational

The IBM home page on the Internet can be found at ibm.com
Foreword to the Deskbook Release 3.0

Here it is – the long awaited next iteration. I completely restructured the Deskbook, focusing more on the methodical part of Harmony for Systems Engineering and its support in Rhapsody by means of the Rhapsody SE-Toolkit. This release also extends the previous scope. It now addresses Requirements Elaboration and Traceability as well as Architectural Analysis through Trade Studies.

I want to thank two colleagues who deserve special mentions with regard to their contributions to this release: Andy Lapping and Dr. Graham Bleakley. Andy - the “Wizard Guru” – is the author of the Rhapsody-SE-Toolkit. Whenever I came up with an idea, how the SE workflow could be automated, the next day I got the wizard-based solution. Graham is the specialist for Architectural Analysis. The chapter on how to perform a trade study is based on his input. Here again, Andy automated it by respective wizards.

Any feedback for the next iteration (release) is appreciated.

Andover, 06-01-09

Foreword to the Deskbook Release 2.0

The systems engineering process is iterative. There is no reason why this should not be applicable also to this Deskbook. Meanwhile, numerous companies started to apply the Harmony for Systems Engineering process using the Deskbook as a general guideline.

This revision takes into consideration the feedback that I got from “the field”. Especially with regard to the use case analysis, I realized that the approach needed to consider the needs of projects with a large numbers of use cases. Nothing changed with regard to the workflow. What changed, is the project structure. The new project structure supports team collaboration. Each use case now may be elaborated separately and then – iteratively - merged to a common system model. The outlined process (Harmony for Systems Engineering) is tool independent.

Andover, 10-12-06
Table of Contents

1 INTRODUCTION 1
1.1 Scope .................................................................................................................................................................................................................... 1
1.2 Document Overview ............................................................................................................................................................................................. 1

2 FUNDAMENTALS OF HARMONY FOR SYSTEMS ENGINEERING ............................................................................................................ 2
2.1 Rational Integrated Systems / Embedded Software Development Process Harmony .................................................................................................................. 2
2.2 Model-based Systems Engineering Process .................................................................................................................................................................. 4
  2.2.1 Requirements Analysis .................................................................................................................................................................................. 5
  2.2.2 System Functional Analysis .......................................................................................................................................................................... 6
  2.2.3 Design Synthesis ......................................................................................................................................................................................... 10
  2.2.3.1 Architectural Analysis (Trade Study) .................................................................................................................................................... 11
  2.2.3.2 Architectural Design ............................................................................................................................................................................. 14
  2.2.3.3 Detailed Architectural Design ............................................................................................................................................................... 16
  2.2.4 Systems Engineering Hand-Off ................................................................................................................................................................... 18
2.3 Essential SysML Artifacts of Model-based Systems Engineering .................................................................................................................................................................. 19
  2.3.1 Requirements Diagram ............................................................................................................................................................................... 20
  2.3.2 Structure Diagrams ...................................................................................................................................................................................... 20
  2.3.2.1 Block Definition Diagram ......................................................................................................................................................................... 20
  2.3.2.2 Internal Block Diagram ......................................................................................................................................................................... 20
  2.3.2.3 Parametric Diagram ............................................................................................................................................................................. 22
  2.3.3 Behavior Diagrams ...................................................................................................................................................................................... 22
  2.3.3.1 Use Case Diagram ............................................................................................................................................................................... 23
  2.3.3.2 Activity Diagram ................................................................................................................................................................................... 23
  2.3.3.3 Sequence Diagram .............................................................................................................................................................................. 24
  2.3.3.4 Statechart Diagram .............................................................................................................................................................................. 24
  2.3.4 Artifact Relationships at the Requirements Analysis / System Functional Analysis Level .................................................................................................................................................................. 25
2.4 Service Request-Driven Modeling Approach .................................................................................................................................................................. 26

3 RHAPSODY PROJECT STRUCTURE .................................................................................................................................................................. 27
3.1 Project Structure Overview ................................................................................................................................................................................. 27
3.2 Requirements Analysis Package ............................................................................................................................................................................. 28
3.3 Functional Analysis Package ................................................................................................................................................................................. 29
3.4 Design Synthesis Package ............................................................................................................................................................................... 30
# Table of Contents

3.4.1 Architectural Analysis Package ................................................................................................................................. 30
3.4.2 Architectural Design Package ............................................................................................................................................. 31

3.5 System-Level Definitions .......................................................................................................................................................... 32

4 CASE STUDY: SECURITY SYSTEM ............................................................................................................................................... 33
4.1 Getting Started ............................................................................................................................................................................. 33
4.2 Requirements Analysis .................................................................................................................................................................. 34
  4.2.1 Import of Stakeholder Requirements .................................................. 35
  4.2.2 Generation of System Requirements .................................................. 36
  4.2.3 Linking System Requirements to Stakeholder Requirements .......... 38
  4.2.4 Definition of System-Level Use Cases ............................................. 40
  4.2.5 Linking Requirements to Use Cases ................................................ 41
4.3 System Functional Analysis ........................................................................................................................................................ 43
  4.3.1 Uc1ControlEntry .............................................................................. 44
    4.3.1.1 Project Structure of the Uc1 Model .............................................. 44
    4.3.1.2 Definition of Uc1 Functional Flow .............................................. 45
    4.3.1.3 Definition of Uc1 Scenarios ....................................................... 46
    4.3.1.4 Definition of Ports and Interfaces .............................................. 50
    4.3.1.5 Definition of State-based Behavior ......................................... 51
    4.3.1.6 Uc1 Model Verification ............................................................ 53
    4.3.1.7 Linking Uc1 Block Properties to Requirements ....................... 55
  4.3.2 Uc2ControlExit ................................................................................ 58
    4.3.2.1 Project Structure of the Uc2 Model .............................................. 58
    4.3.2.2 Definition of Uc2 Functional Flow .............................................. 58
    4.3.2.3 Definition of Uc2 Scenarios ....................................................... 59
    4.3.2.4 Definition of Ports and Interfaces .............................................. 60
    4.3.2.5 Definition of State-based Behavior ......................................... 60
    4.3.2.6 Uc2 Model Verification ............................................................ 61
    4.3.2.7 Linking Uc2 Block Properties to Requirements ....................... 61
    4.3.3 Merging Use Case Blocks ............................................................ 62
4.4 Design Synthesis ........................................................................................................................................................................... 63
  4.4.1 Architectural Analysis (Trade-Off Analysis) .................................... 63
    4.4.1.1 Definition of Key System Functions ......................................... 64
    4.4.1.2 Definition of Candidate Solutions .......................................... 65
    4.4.1.3 Definition of Assessment Criteria .......................................... 66
    4.4.1.4 Assigning Weights to Assessment Criteria ................................ 67
    4.4.1.5 Definition of a Utility Curve for each Criterion ......................... 68
    4.4.1.6 Assigning Measures of Effectiveness (MoE) to each Solution .... 69
    4.4.1.7 Determination of Solution ...................................................... 70
  4.4.2 Architectural Design ........................................................................ 72
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.2.1</td>
<td>Security System Decomposition</td>
<td>72</td>
</tr>
<tr>
<td>4.4.2.2</td>
<td>Graphical Allocation of Operations</td>
<td>74</td>
</tr>
<tr>
<td>4.4.2.3</td>
<td>Formalizing the Allocation of Operations</td>
<td>77</td>
</tr>
<tr>
<td>4.4.2.4</td>
<td>Allocation of Non-functional Requirements</td>
<td>79</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Detailed Architectural Design</td>
<td>81</td>
</tr>
<tr>
<td>4.4.3.1</td>
<td>Derive White-Box Sequence Diagrams</td>
<td>82</td>
</tr>
<tr>
<td>4.4.3.2</td>
<td>Definition of Ports and Interfaces</td>
<td>87</td>
</tr>
<tr>
<td>4.4.3.3</td>
<td>Definition of State-based Behavior</td>
<td>89</td>
</tr>
<tr>
<td>4.4.3.4</td>
<td>System Architecture Model Verification</td>
<td>93</td>
</tr>
<tr>
<td>5</td>
<td>HAND-OFF TO SUBSYSTEM DEVELOPMENT</td>
<td>94</td>
</tr>
<tr>
<td>6</td>
<td>APPENDIX</td>
<td>95</td>
</tr>
<tr>
<td>A1</td>
<td>Mapping Security System Requirements to Model Artifacts</td>
<td>95</td>
</tr>
<tr>
<td>A2</td>
<td>Extended Architectural Model of the Security System</td>
<td>98</td>
</tr>
<tr>
<td>A2.1</td>
<td>Extended System Architecture</td>
<td>99</td>
</tr>
<tr>
<td>A2.2</td>
<td>Extended State-based Behavior of Subsystems</td>
<td>100</td>
</tr>
<tr>
<td>A2.3</td>
<td>Verification of the Extended System Architecture Model</td>
<td>102</td>
</tr>
<tr>
<td>A3</td>
<td>Modeling Guidelines</td>
<td>103</td>
</tr>
<tr>
<td>A3.1</td>
<td>General Guidelines and Drawing Conventions</td>
<td>103</td>
</tr>
<tr>
<td>A3.2</td>
<td>Use Case Diagram</td>
<td>104</td>
</tr>
<tr>
<td>A3.3</td>
<td>Block Definition Diagram</td>
<td>105</td>
</tr>
<tr>
<td>A3.4</td>
<td>Internal Block Diagram</td>
<td>106</td>
</tr>
<tr>
<td>A3.5</td>
<td>Activity Diagram</td>
<td>108</td>
</tr>
<tr>
<td>A3.6</td>
<td>Sequence Diagram</td>
<td>111</td>
</tr>
<tr>
<td>A3.7</td>
<td>Statechart Diagram</td>
<td>113</td>
</tr>
<tr>
<td>A3.8</td>
<td>Profiles</td>
<td>115</td>
</tr>
<tr>
<td>A4</td>
<td>Guideline: Deriving a Statechart Diagram</td>
<td>116</td>
</tr>
<tr>
<td>A5</td>
<td>Guideline: Logical Decomposition of a Use Case Black-Box Activity Diagram</td>
<td>121</td>
</tr>
<tr>
<td>A6</td>
<td>Rhapsody Action Language</td>
<td>128</td>
</tr>
<tr>
<td>A7</td>
<td>Rhapsody SE-Toolkit (Overview)</td>
<td>131</td>
</tr>
<tr>
<td>7</td>
<td>REFERENCES</td>
<td>134</td>
</tr>
</tbody>
</table>
## Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Rational Integrated Systems / Embedded Software Development Process Harmony</td>
<td>2</td>
</tr>
<tr>
<td>2-2</td>
<td>Model-based Systems Engineering</td>
<td>4</td>
</tr>
<tr>
<td>2-3</td>
<td>Workflow in the Requirements Analysis Phase</td>
<td>5</td>
</tr>
<tr>
<td>2-4</td>
<td>Alternative Approaches of Building an Executable Use Case Model</td>
<td>6</td>
</tr>
<tr>
<td>2-5</td>
<td>Derivation of a Use Case Scenario from a Use Case Black-Box Activity Diagram (Industrial Automation Use Case)</td>
<td>8</td>
</tr>
<tr>
<td>2-6</td>
<td>Workflow of the Use Case Model Rainy Day Analysis</td>
<td>9</td>
</tr>
<tr>
<td>2-7</td>
<td>Workflow in the System Functional Analysis Phase</td>
<td>9</td>
</tr>
<tr>
<td>2-8</td>
<td>Subphases and Workflow in the Design Synthesis Phase</td>
<td>10</td>
</tr>
<tr>
<td>2-9</td>
<td>Workflow and Work Product</td>
<td>11</td>
</tr>
<tr>
<td>2-10</td>
<td>Different Shapes of Utility Curves</td>
<td>12</td>
</tr>
<tr>
<td>2-11</td>
<td>Tasks and Workproducts in the Architectural Design Phase</td>
<td>14</td>
</tr>
<tr>
<td>2-12</td>
<td>Nested Use Case White-Box Activity Diagram</td>
<td>14</td>
</tr>
<tr>
<td>2-13</td>
<td>Allocation of Operations to Subsystems (Use Case Fig. 2-5)</td>
<td>15</td>
</tr>
<tr>
<td>2-14</td>
<td>Workflow and Workproducts in the Detailed Architectural Design Phase</td>
<td>16</td>
</tr>
<tr>
<td>2-15</td>
<td>Derivation of White-Box Scenarios from a Use Case White-Box Activity Diagram (ref. Fig. 2-13)</td>
<td>17</td>
</tr>
<tr>
<td>2-16</td>
<td>Overview of UML/SysML Interrelationship</td>
<td>19</td>
</tr>
<tr>
<td>2-17</td>
<td>Taxonomy of SysML Diagrams Used in Harmony for Systems Engineering</td>
<td>19</td>
</tr>
<tr>
<td>2-18</td>
<td>Requirements Diagram</td>
<td>20</td>
</tr>
<tr>
<td>2-19</td>
<td>Block Definition Diagram</td>
<td>20</td>
</tr>
<tr>
<td>2-20</td>
<td>Internal Block Diagram</td>
<td>20</td>
</tr>
<tr>
<td>2-21</td>
<td>Standard Ports</td>
<td>21</td>
</tr>
<tr>
<td>2-22</td>
<td>Flow Ports</td>
<td>21</td>
</tr>
<tr>
<td>2-23</td>
<td>Constraint Block Definition in a Block Definition Diagram</td>
<td>22</td>
</tr>
<tr>
<td>2-24</td>
<td>Parametric Diagram</td>
<td>22</td>
</tr>
<tr>
<td>2-25</td>
<td>Use Case Diagram</td>
<td>23</td>
</tr>
<tr>
<td>2-26</td>
<td>Activity Diagram</td>
<td>23</td>
</tr>
<tr>
<td>2-27</td>
<td>Sequence Diagram</td>
<td>24</td>
</tr>
<tr>
<td>2-28</td>
<td>Statechart Diagram</td>
<td>24</td>
</tr>
<tr>
<td>2-29</td>
<td>SysML Artifacts Relationship at the Requirements Analysis / System Functional Analysis Level</td>
<td>25</td>
</tr>
<tr>
<td>4-1</td>
<td>Requirements Analysis Workflow and its Support through the Rhapsody SE-Toolkit</td>
<td>34</td>
</tr>
<tr>
<td>4-2</td>
<td>Stakeholder Requirements Specification</td>
<td>35</td>
</tr>
<tr>
<td>4-3</td>
<td>System Functional Analysis Workflow and its Support through the Rhapsody SE-Toolkit</td>
<td>43</td>
</tr>
<tr>
<td>4-4</td>
<td>Workflow in the Architectural Analysis Phase and its Support through the Rhapsody SE-Toolkit</td>
<td>63</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Scope

Meanwhile, many books and articles have been published about SysML, the standardized language for model-based systems engineering [1]. But in most cases, the question of how to apply it in an integrated systems and software development process has not been addressed. This deskbook tries to close the gap. Based on the tool independent Rational® Integrated Systems/Embedded Software Development Process Harmony™ it provides systems engineers with a step-by-step guide on using the SysML in a way that allows a seamless transition to the subsequent system development. In this deskbook the chosen tool is the Rational® systems and software design tool Rhapsody® Release 7.5.

The deskbook is written for the practitioner. Screenshots, notes, and best practice tips are added to the workflow descriptions. The brief introductions are minimal rather than narrative.

The deskbook does not replace the Rhapsody training documentation. It rather is intended to supplement it. It is assumed, that the reader is familiar with the UML/SysML and the Rhapsody tool.

1.2 Document Overview

The deskbook is divided into 5 sections:

- Section 1 describes the scope and structure of this book.
- Section 2 introduces the basic concepts of Harmony for Systems Engineering. It starts with an overview of how the systems engineering part of the integrated systems/embedded software development process Harmony fits into the model-driven development lifecycle. Then, the task flow and the associated work products in the different systems engineering phases are detailed. With regard to modeling, this section also provides an overview of SysML artifacts that are considered essential for model-based systems engineering, followed by an introduction to the service request driven modeling approach.
- Section 3 describes the project structure that should be followed when the Rhapsody tool is used in a model-based systems engineering project.
- Section 4 details a case study of the Harmony for Systems Engineering workflow using the Rhapsody tool. The chosen example is a Security System. The workflow starts with the import of stakeholder requirements into Rhapsody and ends with the definition of an executable system architecture model. The workflow is application oriented and focuses on the usage of the Rhapsody SE-Toolkit.
- Section 5 addresses the handoff to the subsequent subsystem (SecSysController) development.

Also provided are several appendices, including

- the documentation of the Extended System Architecture Model of the Security System which takes into consideration the inclusion of COTS components,
- a chapter about modeling/style guidelines regarding the usage of the various SysML diagrams in model-based systems engineering
- a guideline how to derive a statechart diagram from the information captured in an activity diagram and associated sequence diagrams,
- a guideline how to logically decompose a use case black-box activity diagram,
- a quick reference guide to the Rhapsody Action Language,
- an overview of the Rhapsody SE-Toolkit features that are applied in the case study, and
- a set of references.

Included to this deskbook is a volume containing

- the Security System stakeholder requirements,
- for each of the SE phases the incrementally extended Rhapsody model database:
  - SecSys_RA (Requirements Analysis),
  - SecSys_FA (System Functional Analysis),
  - SecSys_AA (Architectural Analysis),
  - SecSys_AD (Architectural Design/Detailed Architectural Design)
- the Extended System Architecture Model (SecSys_EAD).
2  Fundamentals of *Harmony for Systems Engineering*

2.1 Rational Integrated Systems / Embedded Software Development Process *Harmony*

*Fig.2-1* 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 [5,6].

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
2.2 Model-based Systems Engineering Process

Key objectives of Harmony for Systems Engineering are:

- Identification and derivation of required system functions
- Identification of associated system modes and states
- Allocation of the identified system functions and modes/states to a subsystem structure

With regard to modeling, these 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 and state-based behavior, rather than on the details of its functional behavior.

Fig.2-2 depicts an overview of Harmony for Systems Engineering. For each of the systems engineering phases, it shows the essential input and outputs.

The following paragraphs detail the workflow and artifacts of the model-based systems engineering process and outline an associated Requirements Management and Traceability (RT) concept. For a more application oriented workflow description, please refer to the case study in Section 4.
2.2.1 Requirements Analysis

The objective of the requirements analysis phase is to analyze the process inputs. Stakeholder requirements are translated into system requirements that define what the system must do (functional requirements) and how well it must perform (quality of service requirements).

The essential steps of the requirements analysis workflow are shown in Fig.2-3. It starts with the analysis and optional refinement of the stakeholder requirements. Output of this phase is the Stakeholder Requirements Specification. Essentially, stakeholder requirements focus on required capabilities. In the next step, these are transformed into required system functions ("shall" statements) and documented in the Draft System Requirements Specification. For traceability, the identified system requirements are linked to the associated stakeholder requirements.

The next major step in the requirements analysis phase is the definition of system use cases. A use case describes a specific operational aspect of the system (operational thread). It specifies the behavior as perceived by the actors (user) and the message flow between the actors and the use case. An actor may be a person, another system or a piece of hardware external to the system under development (SuD). A use case does not reveal or imply the system's internal structure (black box view).

Use cases may be structured hierarchically – but care should be taken not to functionally decompose the use cases. Use cases are not functions, they use functions. 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 may be 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, they should be grouped in a separate exception use case, which are then linked to the “sunny day” use case via an include or extend dependency.

In order to assure that all functional and associated performance requirements are covered by the use cases, respective traceability links need to be established.

Once the system-level use cases are defined and the complete coverage of the functional and associated performance requirements is assured, they need to be ranked according to their importance for the definition of the system architecture. The order defines the increments of the iterative SE workflow. At the end of each iteration this ranking might need to be updated.
### 2.2.2 System Functional Analysis

The main emphasis of the system functional analysis phase is on the transformation of the functional system requirements into a coherent description of system functions (operations). The analysis is use case-based, i.e. each system-level use case that was identified in the previous requirements analysis phase is translated into an executable model. The model and the underlying requirements then are verified through model execution.

Fig.2-4 details the modeling tasks and the associated work products. First, the use case model context is defined in an Internal Block Diagram. Elements of this diagram are instances of SysML blocks, representing the use case and its associated actor(s). At this stage, the blocks are empty and not linked.
The next step in the modeling workflow is the definition of the behavior of the use case block. It is captured by means of three SysML diagrams:

- **Activity Diagram**,  
- **Sequence Diagrams**, and  
- **Statechart Diagram**.

Each diagram plays a specific role in the elaboration of the use case behavior. The activity diagram – referred to as *Use Case Black-Box Activity Diagram* - describes the overall functional flow (storyboard) of the use case. It groups functional requirements in actions – in *Harmony for Systems Engineering* the equivalent of operations - and shows how these actions/operations are linked to each other. The sequence diagram – referred to as *Use Case Black-Box Sequence Diagram* - describes a specific path through the use case and defines the interactions (messages) between the operations and the actors. The 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.

There is no mandate directing in which order these diagrams should be generated. The order may depend on the available information and the modeler's preference. *Fig.2-4* shows three alternative approaches:

**Alternative 1** starts with the definition of *use case scenarios*. Customers often describe sequences of required system usage (e.g. *Concept of Operations*). Once a set of essential scenarios is captured, the identified functional flow is merged into a common description in an activity diagram. Ports and interfaces are created from the sequence diagrams (ref. section 2.4 *Service Request-Driven Modeling Approach*). They define the links between the actor(s) and the use case block in the use case model internal block diagram. The final step in this approach is the definition of the state-based behavior of the use case block in a statechart diagram.

**Alternative 2** starts with the definition of the *use case functional flow*. This is a common approach, if systems engineers have to elaborate requirements. Typically, customers like to express their requirements from the “big picture” point of view. Once the overall functional flow is defined, use case scenarios are derived from the activity diagram (ref. *Fig.2-5*). Ports and interfaces of the use case block are created from the sequence diagrams. Lastly, its state-based behavior is captured in a statechart diagram.

**Alternative 3** starts with the definition of the *use case state-based behavior*. This approach is recommended if the system under design (SuD) is strongly state-based. In this case, the creation of a use case black-box activity diagram may even be skipped. Use case scenarios then are derived as paths through the statechart diagram. From the sequence diagram then ports and associated interfaces are generated.

It should be noted, that regardless of which approach is chosen, the most important diagram in the system functional analysis process is the use case block statechart diagram. It comprises the information of both the black-box sequence diagrams and the use case black-box activity diagram and can be verified through model execution. The use case black-box activity diagram and the associated black-box sequence diagrams will be reused further down in the design process.

Whenever during the use case based system functional analysis new requirements are identified or high-level requirements are detailed by derived requirements, they need to be documented. Last at the end of the system functional analysis phase, these additional requirements need to be approved by the stakeholders and exported to the requirements traceability tool.

The use case model is analyzed through model execution using the black-box use case scenarios as the basis for respective stimuli. It should be noted, that - following the previously outlined key objectives of this process - the primary focus is on the verification of the generated sequences rather than on the validation of the underlying functionality.
Fig. 2-5 Derivation of a Use Case Scenario from a Use Case Black-Box Activity Diagram (Industrial Automation Use Case)
Once the use case model and the underlying functional requirements are verified, *Rainy Day Analysis* may be performed. This analysis focuses on the identification of system error / fail behavior that was not covered by the initial set of requirements.

*Fig.2-6* details the workflow and the associated work products of the rainy day analysis. It is recommended to first add respective exception behavior to the statechart diagram as this diagram depicts best the overall system behavior. If the error / fail behavior includes new functionality, the use case black-box activity diagram and – if needed – the use case model internal block diagram needs to be updated accordingly. The extended use case model is verified through model execution. It is recommended to record the respective verification scenarios and add these to the set of use case black-box sequence diagrams.

The use case modeling workflow ends with the definition of traceability links between the use case block properties and relevant system requirements. If new requirements or derived requirements were identified during the modeling process, the draft system requirements specification needs to be updated accordingly.

Once all use cases of an iteration increment are verified, there are two ways to proceed (ref. *Fig.2-7*). If use cases overlap – i.e. if they address common system requirements - the consistency between the use case models may be checked by executing respective models concurrently in a common framework (*Use Case Collaboration Model*). Constituents of the use case collaboration model are the instances of the relevant use case blocks and their associated/shared actors. The correct behavior of the use case blocks due to the inputs from the common actors may then be checked through visual inspection of the individual state-based behavior.

If consistency between the different use case models is not an issue, the identified operations, associated service requests, and attributes of each use case model are merged in a common system block SuD. In the subsequent design phase, this information will be allocated to subsystems.

The system functional analysis phase ends with the baselined *System Requirements Specification*. Another document generated at this stage is the *System-Level Interface Control Document (ICD)*. It defines the *logical* (=functional) interfaces between the (black-box) system and its actors and is the aggregate of all use case blocks interfaces. This ICD is the basis for the later system-level (black-box) test definition.
Sometimes the question comes up whether a black-box functional system model – incl. an integrated black-box statechart diagram - should be built in order to assure, that the system has been completely described by the use cases. In principal, there is no reason why it should not be done. The more pragmatic and time saving approach is to shift this issue to the subsequent design synthesis phase. The use cases should have brought enough system information to start the architectural design. What is missing will be identified later when the system architecture model will be verified through model execution.

### 2.2.3 Design Synthesis

The focus of the *design synthesis* phase is on the development of a physical architecture (i.e. a set of product, system, and/or software elements) capable of performing the required functions within the limits of the prescribed performance constraints. Design synthesis follows a top-down approach. *Fig.2-8* depicts the essential subphases and the associated workflow.

Design synthesis typically starts at each level of the architectural decomposition with an *Architectural Analysis* – also referred to as *Trade Study*. Since there may be several hardware and/or software architectures that satisfy a given set of functional and performance requirements, the optimum design concept is elaborated based upon a set of criteria (e.g. *Measures of Effectiveness, MoEs*), that are weighted according to relative importance.

The focus of the *architectural design* phase is on the allocation of system-level operations to the elements of 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. Different allocation strategies may be analyzed, taking into consideration design constraints like performance and safety requirements that were captured during the requirements analysis phase.

Depending on the hand-off to the subsequent subsystem development, the architectural analysis and architectural design phases may be repeated for the different levels of the architectural decomposition. At the lowest level, the functional allocation may address the *realization*, i.e. which operation should be implemented in hardware (e.g. mechanical or FPGA/ASIC) and which should be implemented in software.

*Fig.2-8 Subphases and Workflow in the Design Synthesis Phase*

The focus of the *Detailed Architectural Design* phase is on the definition of the ports and interfaces as well as the state-based behavior of the system blocks at the lowest level of the architectural decomposition.

The following paragraphs detail the workflow and associated work products in the different design synthesis sub-phases.
2.2.3.1 Architectural Analysis (Trade Study)

System functional analysis defines *What* the system should do but not *How* it is to be done. The objective of a *Trade Study* in the architectural analysis phase is to determine the best means of achieving the capability of a particular function in a rational manner, i.e. to identify the *How*.

One of the simplest means of determining the “how” is a technique known as the *Weighted Objectives Method*, developed by N. Cross [7]. This form of analysis is commonly used within the field of Engineering System Design to evaluate potential solutions to functional problems. It can also be used to determine the best hardware platforms for software or decide the optimum mechanical/electrical hardware split based upon non-functional requirements like a set of customer constraints, performance or cost criteria.

*Fig.2-9* depicts the workflow and the associated work products in the Architectural Analysis phase.

### Identify Key System Functions

The objective of this task is to group system functions into sub-sets to support the analysis of alternatives during architectural analysis. A key system function could be a group of system functions that

- are cohesive and/or tightly coupled or
- may be realized by a single architectural component or
- will be realized by reuse of an existing component (HW/SW) or
- may be reused within the system or
- address a specific design constraint

The next 6 tasks are performed for each selected key system function.

### Define Candidate Solutions

There is always more than one way to realize a key system function. The objective of this task is to identify possible solutions for a previously identified key system function. The solutions are elaborated in a team representing all relevant areas of expertise. At this stage, associated stakeholder requirements need to be identified and taken into consideration. Candidate solutions may take into consideration previously developed hardware and software components, non-developmental items, and COTS hardware and software.
Identify Assessment Criteria
In order to identify the best solution from a set of candidate solutions for a specific key system function, assessment criteria need to be identified. Meaningful assessment criteria are established in collaboration with stakeholders and a team representing all relevant areas of expertise. Typically, the assessment criteria are based upon customer constraints, required performance characteristics, and/or costs.

Assign Weights to Assessment Criteria
Not all assessment criteria are equal. Some are more important than others. Assessment criteria are weighted according to their relative importance to the overall solution. The weighting factors are normalized to add up to 1.0. This task should be performed in collaboration with stakeholders and relevant domain experts.

Define Utility Curves for each Criterion
The purpose of this task is to define a set of normalization curves - also known as Utility Curves or Value Functions - one for each assessment criterion that will be used to produce a dimensionless Measure of Effectiveness for each solution candidate. This curve yields a normalized value typically between 0 and 10. The input value to the curve is typically based upon equipment specifications or derived from calculations based upon possible solutions. In this case it is considered as being objective.

A utility curve may also be created by knowledgeable project members. In this case the curve reflects the consensus among the group but should be considered as subjective.

Assign Measures of Effectiveness (MoE) to Candidate Solution
In order to compare the different solutions of a key system function via weighted objectives analysis each candidate solution is characterized by a set of normalized, dimensionless values - Measures of Effectiveness (MoE) - which describe how effective a solution candidate is for a particular assessment criterion. The MoE is a normalized value computed using the utility curve and the nominal value specified for the solution candidate. The nominal values are typically determined from equipment specifications or derived from calculations based upon the relevant solution.

Determine Solution
The determination of a preferred solution is performed by means of Weighted Objectives calculation. In this analysis the MoE values for each of the assessment criteria are multiplied by the appropriate weight. The weighted values for each alternative solution then are added to obtain a total score for each solution. The solution with the highest score is selected as the implementation for that particular function.

Tab.2-1 shows for the key system function “Capture Biometric Data” in the case study described later in chapter 4, that the preferred solution is the Fingerprint Scanner.

Merge Possible Solutions to form System Architecture
The solutions identified for each key system function are merged to define the equipment breakdown structure. It is assumed that the initial key system functions were independent. Thus, the final merged solution is the preferred solution based upon the assessment criteria for the complete architecture. It will be the basis of the subsequent architectural design activities. These design decisions are captured in the Trade Study Report along with any resulting design constraints.
Tab.2-1 Weighted Objectives Table of the Key System Function “Capture Biometric Data” (ref. Case Study Chapter 4)
### 2.2.3.2 Architectural Design

*Fig. 2-11* shows the workflow and the associated work products in the architectural design phase. Based on the design concept, optionally elaborated in a previous trade study, it starts with the decomposition of the chosen system block into parts. The resulting system structure is captured in a SysML Block Definition Diagram (BDD) and in a SysML Internal Block Diagram (IBD).

![Diagram of Architectural Design](image)

Generally, there are two ways to proceed with regard to the allocation of the system block operations to the parts. If an allocation concept exists, they may be copied directly into the relevant parts. Otherwise, the allocation can be elaborated graphically for each use case 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 decomposition hierarchy. Based on the chosen design concept, the operations (= actions) then are "moved" into respective block swim lanes (ref. *Fig. 2-13*). An essential requirement for this allocation is that the initial links (functional flow) between the actions are maintained.

Use case white-box activity diagrams may be nested, thus reflecting the iterative architectural decompositions of the system under design (ref. *Fig. 2-12*).

**Fig. 2-11 Tasks and Workproducts in the Architectural Design Phase**

Generally, there are two ways to proceed with regard to the allocation of the system block operations to the parts. If an allocation concept exists, they may be copied directly into the relevant parts. Otherwise, the allocation can be elaborated graphically for each use case 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 decomposition hierarchy. Based on the chosen design concept, the operations (= actions) then are "moved" into respective block swim lanes (ref. *Fig. 2-13*). An essential requirement for this allocation is that the initial links (functional flow) between the actions are maintained.

Use case white-box activity diagrams may be nested, thus reflecting the iterative architectural decompositions of the system under design (ref. *Fig. 2-12*).

**Fig. 2-12 Nested Use Case White-Box Activity Diagram**

If an operation cannot be allocated to a single block, it must be decomposed. In this case, the associated decomposed SuD operations need to be linked to the parent operation through a respective dependency. A system-level operation may also be allocated to more than one block (e.g. architectural redundancy in order to meet fault tolerance requirements). In this case, the relevant action is copied into the respective block swim lane and integrated into the functional flow.

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

Once all system block operations, that capture system functional requirements are allocated to system parts, non-functional requirements (e.g. design constraints) are allocated to the relevant parts and respective trace links are established.
2.2.3.3 Detailed Architectural Design

The focus of the detailed architectural design phase is on the definition of ports and interfaces, as well as on the definition of the state-based behavior of the system blocks at the lowest level of the architectural decomposition. Fig.2-14 depicts the workflow and the associated work products in the detailed architectural design phase.

Leaf block ports and interfaces are identified from White-Box Sequence Diagrams. White-box sequence diagrams are derived from the use case white-box activity diagrams that were created in the previous architectural design phase. The focus of black-box sequence diagrams was on the identification of the required sequences of system functions (operations). White-box activity diagrams focus on the collaboration between the different subsystems taking into consideration the allocation of the operations. The received service requests define the (provided) interfaces of a block (ref. section 2.4).

The derivation of white-box sequence diagrams is performed iteratively for each use case white-box activity diagram.

Once ports and interfaces are defined, the resulting state-based behavior of each leaf block needs to be captured in a statechart diagram.

The correctness and completeness of the system architecture model is verified through model execution. Once the model functionality 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.
Fig. 2-15 Derivation of White-Box Scenarios from a Use Case White-Box Activity Diagram (ref. Fig.2-13)
2.2.4 Systems Engineering Hand-Off

In a Model-Driven Development the key artifact of the hand-off from systems engineering to the subsequent system development is the baselined executable model. This model is the repository from which specification documents (e.g. HW/SW Requirements Specifications, ICDs …) are generated. Scope and content of the hand-off is dependent on the characteristics of the project and the organizational structure systems engineering is embedded.

If the SuD is one specific software configuration item (CI), systems engineering may stop at the system functional analysis level. In this case, the hand-off will be executable use case models.

From the organizational point of view, if there is a separation between systems engineering and subsystems engineering, systems engineering may stop at the first level of system architecture decomposition. In this case the hand-off will be composed of relevant executable subsystem models.

If systems engineers hand-off their specifications directly to HW/SW development, the hand-off will be respective executable HW and/or SW configuration item (CI) models.

In any of these cases the hand-off packages are composed of:

- Baselined executable CI model(s)
- Definition of CI-allocated operations including links to the associated system functional and performance requirements
- Definition of CI ports and logical interfaces
- Definition of CI behavior, captured in a statechart diagram
- Test scenarios, derived from system-level use case scenarios
- CI-allocated non-functional requirements

It should be noted, that the baselined System Architecture model becomes the reference model for further development of system requirements.
2.3 Essential SysML Artifacts of Model-based Systems Engineering

SysML defines the standardized “vocabulary” of the language for model-based systems engineering. As a standard, this vocabulary needs to cover all possible applications. But SysML does not specify how to apply these words. Systems engineering is strongly communication driven. Systems engineers have to communicate with stakeholders from different domains, like electrical engineers, mechanical engineers, software engineers, test engineers, and - not to forget - the customer who is not necessarily an engineer. In such an environment it is paramount to keep the language domain independent and as simple as possible. The goal should be to minimize the amount of language elements. The fewer elements are used, the better. The compliance to a standard does not mean that all elements of this standard have to be applied. It is good practice to standardize the usage of SysML within the organization, if a company wants to deploy SysML-based systems engineering. This paragraph provides an overview of the SysML artifacts that are considered essential in the model-based systems engineering process Harmony for Systems Engineering.

SysML reuses a subset of the UML 2.1 and extended it by systems engineering specific constructs. Fig.2-16 visualizes the relationship between the UML and SysML by means of a Venn diagram, where the set of language constructs that comprise the UML and SysML languages are shown as circles marked UML 2.1 and SysML 1.0, respectively. The intersection of the two circles indicates the UML modeling constructs that SysML reuses (UML4SysML). In order to provide a seamless transition from systems engineering to software development, a respective process should focus on UML4SysML.

Fig.2-17 shows the taxonomy of SysML diagrams used in Harmony for Systems Engineering. Essentially, there are three categories of diagrams:

- Structure Diagram,
- Behavioral Diagram, and
- Requirements Diagram.

The color code of the Venn diagram is also applicable to this diagram. Some of the diagrams have two colors. This indicates that SysML extended the initial UML artifact.

The following paragraphs outline the usage of these diagrams in Harmony for Systems Engineering, and list the elements that are considered essential.
2.3.1 Requirements Diagram

A Requirements Diagram graphically shows

- the relationship among textual requirement elements
  (<<derive>>, containment)
- the relationship between requirements and model elements
  (<<trace>>, <<satisfy>>, and)
- the dependency between a requirement and a test case that
  verifies that the requirement is met (<<verify>>).

Fig.2-18 Requirements Diagram

2.3.2 Structure Diagrams

2.3.2.1 Block Definition Diagram

The SysML Block Definition Diagram is the equivalent to a class
diagram in the UML. It shows the basic structural elements (blocks)
of the system and their relationships / dependencies. Internal connectors
are not shown.

Fig.2-19 Block Definition Diagram

2.3.2.2 Internal Block Diagram

The SysML Internal Block Diagram shows the realization of the system
structure defined in the Block Definition Diagram. It is composed of a
set of nested parts (i.e. instances of the system blocks) that are inter-
connected via ports and connectors.

Fig.2-20 Internal Block Diagram
Ports

A port is a named interaction point between a block or a part and its environment. It is connected with other ports via Connectors. The SysML defines two types of ports: Standard Ports and Flow Ports. The main motivation for specifying such ports on system elements is to allow the design of modular reusable blocks, with clearly defined interfaces.

Standard Ports

A UML/SysML Standard Port is a named interaction point assigned to a block, through which instances of this block can exchange messages. It specifies the services the owning block offers (provides) to its environment as well as the services that the owning block expects (requires) of its environment.

There are two different kinds of Standard Ports:

- **Delegation or Relay** ports forward requests to other ports.
- **Behavioral** ports are parts of the block that actually implements the service.

Flow Ports

A SysML Flow Port specifies the input and output items that may flow between a block and its environment. Input and output items may include data as well as physical entities, such as fluids, solids, gases, and energy. The specification of what can flow is achieved by typing the Flow Port with a specification of things that flow.

There are two different kinds of Flow Ports:

- **An Atomic Flow Port** relays a single item that flows in or out.
- **A Non-Atomic Flow Port** relays multiple items, listed in a respective “flow specification”.

Fig.2-21 Standard Ports

Fig.2-22 Flow Ports
2.3.2.3 Parametric Diagram

A Parametric Diagram is a special type of an Internal Block Diagram. It visualizes the parametric relationship between system properties. It is an integral part of technical performance measures and trade studies.

Constraints among system properties are specified in Constraint Blocks. Constraint blocks are defined in a Block Definition Diagram and “used” in the Parametric Diagram by binding their parameters to the specific properties of a block.

![Fig.2-23 Constraint Block Definition in a Block Definition Diagram](image)

![Fig.2-24 Parametric Diagram](image)

2.3.3 Behavior Diagrams

UML/SysML provides four diagrams that express the functional and dynamic behavior of a system:

- Use Case Diagram
- Activity Diagram
- Sequence Diagram and
- Statechart Diagram

Although each diagram focuses on a specific behavioral aspect, the information provided by these diagrams overlap each other. For instance, both the sequence diagrams and the activity diagrams describe interactions. There may also be an overlap between the behavior captured in activity diagram and the statechart diagram, since SysML extended the UML activity diagrams by adding the notation of dynamic behavior (control of actions).

In order to minimize the overlap between the different behavioral diagrams, decisions should be made upfront, which role the individual diagrams should play in the context of the modeling workflow. The next step should be to “standardize” the usage of diagram elements by filtering-out in each diagram those elements that are considered essential.
2.3.3.1 Use Case Diagram

A Use Case Diagram captures the functional requirements of a system by describing interactions between users of the system and the system itself. Note that as a system is decomposed, users of a given system could be external people or other systems. A use case diagram comprises a system boundary that contains a set of use cases. Actors lie outside of the system boundary and are bound to use cases via associations.

A use case describes a specific usage (“operational thread”) of a system:
- the behavior as perceived by the users (actors) and
- the message flow between the users and the use case.

A use case does not reveal or imply the system’s internal structure (“black-box view”).

2.3.3.2 Activity Diagram

An Activity Diagram is similar to the classic flow chart. It describes a workflow, business process, or algorithm by decomposing the flow of execution into a set of actions and sub activities joined by transitions and various connectors. An activity diagram can be a simple linear sequence of actions or it can be a complex series of parallel actions with conditional branching and concurrency.

**NOTE:** In Harmony for Systems Engineering the terms activity, action and operation are synonymous.

Actions may be grouped and assigned to objects – e.g. subsystems. In this case, the activity diagram is split into swim lanes that depict the respective responsibilities.

**NOTE:** Harmony for Systems Engineering uses a SysML activity pin stereotyped ActorPin to visualize the interaction of an action/operation 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.
### 2.3.3.3 Sequence Diagram

**Sequence Diagrams** elaborate on requirements specified in use cases and activity diagrams by showing how actors and blocks collaborate in some scenarios. A sequence diagram represents one or more scenarios through a use case.

A sequence diagram is composed of vertical lifelines for the actors and blocks along with an ordered set of messages passed between these entities over a period of time.

- **Messages** are shown as horizontal lines with open arrows between the vertical object lines (lifelines).
  - **NOTE:** UML/SysML differentiates between *synchronous* and *asynchronous* messages. In Harmony for Systems Engineering, the message-based communication is described via asynchronous messages (two-line arrowhead).
- **Operations** are depicted as reflexive (synchronous) messages (full arrowhead) at associated lifelines.
- **Quality of Service (QoS)** requirements may be added as comments and/or constraints.

### 2.3.3.4 Statechart Diagram

A **Statechart Diagram** describes the state-based behavior of a block. In the Harmony for Systems Engineering workflow, it is considered the most important behavior diagram, as 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.

Statechart diagrams are finite statemachines that are extended by the notation of
- **Hierarchy**
- **Concurrency**

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.
2.3.4 Artifact Relationships at the Requirements Analysis / System Functional Analysis Level

Fig. 2-29 shows, how the different SysML artifacts are related to each other at the requirements analysis and system functional analysis level.

- A **Requirements Diagram** visualizes the dependencies of at least 3 requirements.
- A **Use Case Diagram** contains minimum one **Use Case**.
- Use cases are traced to at least one requirement.
- A use case should always have one **Activity Diagram** that captures the functional flow.
- A use case should be described by at least 5 **Sequence Diagrams**.
- When it comes to building an executable use case model, the model is described by an **Internal Block Diagram**
- The **Internal Block Diagram** should contain instances of at least two **Blocks** (use case block and actor block(s)).
- The state-based behavior of each block instance is described by a **Statechart Diagram**.

![SysML Artifacts Relationship Diagram](image-url)
2.4 Service Request-Driven Modeling Approach

In the Service Request-Driven Approach, the communication between blocks is based on asynchronous messages (“service requests”) via 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 or listed in the associated description field of the relevant service request and associated operation.

The approach is performed in four steps:

1. It starts with the definition of the network nodes by means of SysML structure diagrams, using blocks as the basic structure elements. First, these blocks are empty and not linked.

2. In the next step, the communication between the blocks is described in a UML/SysML Sequence Diagram. **NOTE**: In the Rhapsody tool the Sequence Diagram may be automatically generated from an underlying Activity Diagram by means of the SE Toolkit (ref. section 4.3.1.3)

3. The next step is the allocation of the service requests and operations to respective blocks. **NOTE**: In the Rhapsody tool this step is automated through the Auto Realize feature.

4. Based on the allocated service requests, the associated SysML Standard Ports and interfaces now can be defined. **NOTE**: In the Rhapsody tool this step is semi-automated by means of the SE-Toolkit (ref. section 4.3.1.4).
3 Rhapsody Project Structure

This section describes the project structure that should be followed when the Rhapsody tool is used in a model-based systems engineering project. The details are shown considering as an example the Security System Model of the deskbook case study.

3.1 Project Structure Overview

On the top-level, the project structure shows two types of packages:

- Packages that contain the artifacts generated in the different SE-phases, i.e.
  - RequirementsAnalysisPkg
  - FunctionalAnalysisPkg
  - DesignSynthesisPkg
- Packages that contain system-level model definitions, i.e.
  - ActorPkg
  - InterfacesPkg and
  - TypesPkg
3.2 Requirements Analysis Package

Constituents of the RequirementsAnalysisPkg are

- RequirementsPkg and
- UseCaseDiagramsPkg

Constituents of the RequirementsPkg are

- StakeholderRequirementsPkg,
- SystemRequirementsPkg, and
- DerivedRequirementsPkg

The StakeholderRequirementsPkg contains the imported stakeholder ("capability") requirements specification.

The SystemRequirementsPkg contains the actual system requirements ("shall" statements), generated from the stakeholder requirements. During requirements analysis and system functional analysis, there is no sub-structure. At the end of the architectural design phase, system requirements are grouped into allocation-related packages. Each of the packages may contain a folder with associated requirements diagrams.

During the system functional analysis and design synthesis phase additional requirements may be identified. First, they will be located in the DerivedRequirementsPkg. Once they are approved, they will be exported to DOORS, linked to the associated model artifacts and moved into allocation-related packages.

The UseCaseDiagramsPkg contains the system-level use case diagram(s).

**NOTE**: Initially, use cases and actors are located in the UseCasesPkg. In the system functional analysis phase use cases are moved into respective use case packages in the FunctionalAnalysisPkg and the associated actors are moved into the ActorsPkg.
3.3 Functional Analysis Package

System functional analysis in *Harmony for Systems Engineering* is use case based. Each use case of the system-level use case diagram(s) is translated into an executable model. The *FunctionalAnalysisPkg* contains the artifacts generated in the system functional analysis phase.

For each use case of the use case diagram, there is a package `<UseCaseName>Pkg` that contains the associated model artifacts:

- A category *blocks* containing the definition of the use case block `Uc_<UseCaseName>`. This block includes the associated statechart diagram.

- A folder *Internal Block Diagrams* with the internal block diagram `IBD_<UseCaseName>`

- A folder *Packages* that contains
  - A package `<UseCaseName>_ExecutionScopePkg` that defines the context of the use case model execution, i.e. the instances of the actor(s) and the use case block as well as the definition of their links.
  - A package `<UseCaseName>_BBScenariosPkg` which holds the use case scenarios.

- A category *Use Cases* with the use case descriptions, i.e.
  - The category *Activity Views* which contains the black-box activity diagram `<UseCaseName>_BlackBoxView` and a folder *Sequences* that contains the references to use case scenarios, which were derived from the black-box activity diagram (ref. section 4.3.1.3).
  - The category *Association Ends* which contains the definitions of the associations between the actor(s) and the use case.
  - The category *Dependencies* which contains the trace dependencies between the use case and the associated system requirements.
3.4 Design Synthesis Package

The $DesignSynthesisPkg$ is partitioned into two packages:

- $ArchitecturalAnalysisPkg$
- $ArchitecturalDesignPkg$

3.4.1 Architectural Analysis Package

The $ArchitecturalAnalysisPkg$ contains the artifacts that are created when a trade-off analysis is performed prior to the architectural design. For details please refer to section 4.4.1.
3.4.2 Architectural Design Package

Constituents of the ArchitecturalDesignPkg are:

- A folder Block Definition Diagrams with the SuD level 1 block definition diagram \texttt{BDD_{SuDName}}

- A category blocks containing the definitions of the SuD block, including instances of its parts and the definition of associated Delegation Ports.

- A folder Internal Block Diagrams with the internal block diagram of the SuD system architecture \texttt{IBD_{SuDName}}

- A folder Packages that contains
  - An ExecutionScopePkg which defines the context of the architectural model execution, i.e. the instances of the actor(s) and the SuD block as well as the definition of their links.
  - A package <Block>DecompositionPkg the constituents of which are:
    - A package <Block>WB_{AD} which contains the decomposed white-box activity diagrams of the system use cases,
    - A package <Block>WB_UcSD which holds the decomposed system use case scenarios,
    - Packages <Part>Pkg, each of which holds the definitions of the relevant part. If a part is further decomposed, it will contain a package <Part>DecompositionPkg with packages of its associated sub-blocs each of which will be decomposed according to the outlined structure.
3.5 System-Level Definitions

On the top-level of the project structure there are three packages for system-level definitions:

- **ActorPkg**
- **InterfacesPkg**
- **TypesPkg**

The **ActorPkg** contains the definitions of all the actors identified in the system-level use case diagram(s). Each actor block may contain a statechart diagram.

The **InterfacesPkg** contains the definition of interfaces and associated events. The interfaces are grouped in packages corresponding to the associated use case model(s) and the system architecture model.

The **TypesPkg** contains the system-level data definitions.
4 Case Study: Security System

Harmony for Systems Engineering is a tool independent process. In this section a case study exemplifies, how the workflow that was outlined in the previous sections is applied using the Rhapsody tool. The chosen example is a Security System.

The case study covers all phases of the model-based systems engineering process, i.e. requirements analysis, system functional analysis, and design synthesis, including associated requirements management and traceability. The workflow starts with the import of stakeholder requirements into Rhapsody and ends with the definition of an executable architecture model. The architectural design concept is elaborated through a trade study. The case study also addresses the hand-off to the subsystem development.

The Rhapsody tool supports model-based systems engineering through a special add-on – the SE-Toolkit. This toolkit contains features that automate many of the tasks in a systems engineering workflow. It should be noted, that most of these features are process-independent. The focus of this case study is on the usage of these features in the different phases of Harmony for Systems Engineering.

It is assumed, that the reader is familiar with the Rhapsody tool.

4.1 Getting Started

A Harmony for Systems Engineering compliant project structure (ref. section 2) may be created by means of the SE-Toolkit feature Create Harmony Project.

1. Start Rhapsody
2. In the main menu select File > New
3. Enter project name (SecuritySystem) and select/define the associated project directory.
4. For the Type select the SysML profile.
5. Add the Harmony profile:
   - In the main menu select File > Add Profile to Model
   - Double-click Harmony
   - Double-click Harmony.sbs
5. Right-click the project name in the browser and select SE-Toolkit\Create Harmony Project.
4.2 Requirements Analysis

The workflow followed in the case study is shown in Fig. 4-1. An important part of the workflow is the definition of trace links between the generated system requirements and the associated stakeholder requirement as well as the trace links between the system requirements and the use cases. The creation of respective dependencies is supported by the **Rhapsody SE-Toolkit**.

Once the system-level use cases are defined and the complete coverage of the functional and non-functional system requirements is assured, the requirements analysis phase ends.

**NOTE**: Requirements traceability by means of the **Rhapsody Gateway** and **Telelogic Doors** will not be addressed in this case study.

![Requirements Analysis Workflow and its Support through the Rhapsody SE-Toolkit](image_url)
4.2.1 Import of Stakeholder Requirements

Requirements analysis starts with the import of the stakeholder requirements specification into Rhapsody. The requirements may be imported manually or via the Rhapsody Gateway. In the case study the stakeholder requirements are imported from a Word document through the Rhapsody Gateway. Rhapsody Gateway puts the stakeholder requirements into the RequirementsPkg. The original Word document can be found in the Controlled Files folder. Stakeholder requirements are located in the StakeholderRequirementsPkg and stereotyped <<fromWord>>.

Security System – Stakeholder Requirements Specification

System Overview

SS1: System Summary
A security system is to be developed that controls entry and exit to a building through a single point of entry. Identification of personnel will be made by two independent checks. Each person will be photographed upon entry and their time in the building monitored.

Nominal System Specification

SS11: Security Checks
Secure areas are to be protected by two independent security checks, one based upon an employee ID and one based upon biometric data. Access to secure areas will be unavailable until the users ID is confirmed. The time between the two independent security checks will not exceed a configurable period. The user is allowed three attempts at biometric and/or card identification before access is completely disabled. Any denied access attempt is to be recorded and sent to the administrator.

Identification Requirements

SS111: Security Card
Access will be denied to any user unless he has a valid Security Card. Security cards only contain the employee name and ID and will be renewed yearly. Out of date security cards cause a denial of access.

SS112: Biometric Scan
Access will be denied to any user unless their biometric data is recognized. The biometric data is to be stored in the system database and not on the security card.

SS12: Access Priority and Time
The system will only process one user at a time, giving them sufficient time to enter and exit the area before automatically securing itself.

SS13: Exit requirements
The user is not allowed to exit until the security card has been successfully authorized.

Personnel Monitoring

MA1: Image Capture
An image is to be taken of any person, at the first attempt, when trying to access a secure area for logging time and employee ID.

MA2: Time monitoring
The time a user spends in a secure area is to be recorded. An alarm will notify administration/security if a person stays longer than 10 hours in the secure area.

Emergency Requirements

E1: Emergency Exit
In the event of an emergency the administrator can invoke a “Free Exit Mode”. All security checks for exiting the area are to be disabled until the administrator returns the system to normal working.

E2: Security Lockdown
The administrator can invoke a security lockdown mode - in this event the system will lock all access points until the administrator returns the system to normal working.

Fig.4-2 Stakeholder Requirements Specification
4.2.2 Generation of System Requirements

Stakeholder requirements typically are problem-oriented statements. They focus on required capabilities. Once they are imported into Rhapsody, they need to be converted into solution-oriented (“shall”) statements. The output of this transformation process is the – preliminary – System Requirements Specification.

System requirements are derived by decomposing the stakeholder requirements into sets of basic required functionality. Tab. 1 lists the identified system requirements. They form the basis for the subsequent system functional analysis and design synthesis phases.

System requirements are stored in the SystemRequirementsPkg of the RequirementsAnalysisPkg.
<table>
<thead>
<tr>
<th>ID</th>
<th>System Requirement</th>
<th>Req. Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS11-1</td>
<td>Two Independent Security Checks</td>
<td>Constraint</td>
</tr>
<tr>
<td></td>
<td>Secure areas shall be protected by two independent security checks.</td>
<td></td>
</tr>
<tr>
<td>SS11-2</td>
<td>Employee ID Card Identification - Entry</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>Entry shall be protected by a security check based upon employee ID.</td>
<td></td>
</tr>
<tr>
<td>SS11-3</td>
<td>Employee ID Card Identification - Exit</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>Exit shall be protected by a security check based upon employee ID.</td>
<td></td>
</tr>
<tr>
<td>SS11-4</td>
<td>Biometric Scan</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>Entry to the secure areas shall be protected by a second independent security check</td>
<td></td>
</tr>
<tr>
<td></td>
<td>based upon biometric data</td>
<td></td>
</tr>
<tr>
<td>SS11-5</td>
<td>Time Between Two Independent Checks</td>
<td>Constraint</td>
</tr>
<tr>
<td></td>
<td>The time between the two independent security checks shall not exceed a configurable</td>
<td></td>
</tr>
<tr>
<td>SS11-6</td>
<td>Configure Time Between Two Independent Checks</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>The time between two independent security checks shall be configurable.</td>
<td></td>
</tr>
<tr>
<td>SS11-7</td>
<td>Three Attempts On Employee ID Entry</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>Upon entry the user shall be allowed three attempts on card identification.</td>
<td></td>
</tr>
<tr>
<td>SS11-8</td>
<td>Three Attempts On Biometric Data Entry</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>Upon entry the user shall be allowed three biometric data entries.</td>
<td></td>
</tr>
<tr>
<td>SS11-9</td>
<td>Three Attempts On Employee ID Exit</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>Upon exit the user shall be allowed three attempts on card identification.</td>
<td></td>
</tr>
<tr>
<td>SS11-10</td>
<td>Denied Entry Notification</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>Any denied access attempt shall be logged and account details sent to the administrator</td>
<td></td>
</tr>
<tr>
<td>SS11-11</td>
<td>Denied Exit Notification</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>Any denied access attempt shall be logged and account details sent to the administrator</td>
<td></td>
</tr>
<tr>
<td>SS11-12</td>
<td>Alarm - Entry</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>On a denied entry an alarm signal shall be raised.</td>
<td></td>
</tr>
<tr>
<td>SS11-13</td>
<td>Alarm - Exit</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>On a denied exit an alarm signal shall be raised.</td>
<td></td>
</tr>
<tr>
<td>SS11-14</td>
<td>Disabling User Account</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>After three failed attempts at card identification or biometric data entry the user</td>
<td></td>
</tr>
<tr>
<td></td>
<td>account shall be disabled</td>
<td></td>
</tr>
<tr>
<td>SS11-15</td>
<td>Visualization of Security Card Check Status - Entry</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>The user shall be visually informed about the status of his/her ID card check.</td>
<td></td>
</tr>
<tr>
<td>SS11-16</td>
<td>Visualization of Security Card Check Status - Exit</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>The user shall be visually informed about the status of his/her ID card check.</td>
<td></td>
</tr>
<tr>
<td>SS11-17</td>
<td>Visualization of Biometric Data Check Status</td>
<td>Functional</td>
</tr>
<tr>
<td></td>
<td>The user shall be visually informed about the status of his/her biometric data check</td>
<td></td>
</tr>
</tbody>
</table>

Tab.4-1 System Requirements for the Security System
4.2.3 Linking System Requirements to Stakeholder Requirements

System requirements are linked to the associated stakeholder requirements through a <<satisfy>> dependency. The creation of this dependency is supported by the SE-Toolkit feature Create Dependency.

1. In the Tools Menu select Tools > SE-Toolkit>Modeling Toolbox
2. In the dialog box select Dependencies.
3. Select dependency Satisfy.
4. In the SystemRequirementsPkg select a system requirement.
5. In the ModelingToolbox dialog box click Set Source.
6. In the StakeholderRequirementsPkg select the associated stakeholder requirement.
7. In the SE-Toolkit dialog box click Set Destination.
8. In the ModelingToolbox dialog box Click Create Dependency.
Case Study: Requirements Analysis

Visualization of the <<satisfy>> Dependencies in a Requirements Diagram

1. In the RequirementsPkg create a Requirements Diagram RD_SysReqsToSHReqsLinks.
   - Time Recording (ID = MA2-1)
   - Approval of Biometric Data (ID = SS112-1)
   - Biometric Data Storage (ID = SS112-2)
   - SS1 - Entry and Exit Control System
   - SS11 - Security Checks
   - SS111 - Security Card
   - SS112 - Biometric Scan
   - SS113 - Exit requirements
   - SS12 - Access Priority and Time
   - Processing User Request (ID = SS12-1)
   - Exit Time (ID = SS12-2)
   - Access Precondition (ID = SS111-2)
   - Security Card Information (ID = SS111-1)
   - Security Lockdown (ID = E2-1)
   - Security Lockdown
   - E2 - Security Lockdown
   - E1 - Emergency Exit
   - Image Capture (ID = MA1-1)
   - Image Captue
   - MA1 - Image Capture
   - MA2 - Time monitoring
   - MA2 - Time monitoring
   - E1 - Emergency Exit
   - E1 - Emergency Exit
   - SS11 - Security Card
   - SS111 - Security Card
   - SS112 - Biometric Scan
   - Disabling User Account (ID = SS11-14)
   - Three Attempts On Employee ID Entry (ID = SS11-7)
   - Three Attempts On Employee ID Exit (ID = SS11-9)
   - Time Between Two Independent Checks (ID = SS11-5)
   - Denied Entry Notification (ID = SS11-10)
   - Denied Exit Notification (ID = SS11-11)
   - Three Attempts On Biometric Data Entry (ID = SS11-8)
   - Two Independent Security Checks (ID = SS11-1)
   - Visualization of Security Card Check Status - Exit (ID = SS11-16)
   - Visualization of Security Card Check Status - Entry (ID = SS11-15)
   - Employee ID Card Identification - Exit (ID = SS11-12)
   - Alarm - Exit (ID = SS11-13)
   - Employee ID Card Identification - Exit (ID = SS11-13)

2. Move the stakeholder requirements and system requirements into the diagram.
   - SS111 - Out of Date Cards - Exit (ID = SS111-5)
   - SS111 - Out of Date Cards - Entry (ID = SS111-4)
   - SS11 - Out of Date Cards - Exit (ID = SS11-5)
   - SS11 - Out of Date Cards - Entry (ID = SS11-4)

3. In the Tools Menu select Layout > Complete Relations > All
4.2.4 Definition of System-Level Use Cases

The system requirements of the Security System are grouped into three use cases:
- Uc1ControlEntry
- Uc2ControlExit
- Uc3ConfigureSecuritySystem

The associated actors are:
- User
- Administrator
- Camera and
- Access Point

Open UCD_SecuritySystem and draw the use case diagram with the three use cases and the associated actors.

NOTE: At this stage, the actor blocks and use cases are allocated in the UseCaseDiagramsPkg.
4.2.5 Linking Requirements to Use Cases

System functional and performance requirements are linked to the use case with a <<trace>> dependency. There are two ways to implement this dependency:

- in the browser, using the SE-Toolkit feature Create Dependency
- graphically, in a Requirements Diagram.

It is recommended to start with the toolkit feature Create Dependency. If necessary – e.g. for discussions or documentation purposes - the dependencies may then be visualized in a Requirements Diagram.

Exemplary, the linking process will be shown for the use case Uc1ControlEntry. The respective links of the other use cases are listed in the Appendix.

1. In the Tools Menu select Tools > SE-Toolkit/Modeling Toolbox
2. In the dialog box select Dependencies. Tick dependency Trace.
3. In the UseCaseDiagramsPkg select use case Uc1ControlEntry.
4. In the SystemRequirementsPkg select the requirement the use case is linked to.
5. In the ModelingToolbox dialog box click Set Source.
6. In the ModelingToolbox dialog box click Set Destination.
7. In the ModelingToolbox dialog box click Create Dependency.
Visualization of the <<trace>> Dependencies in a Requirements Diagram

1. In the SystemRequirementsPkg create a package Uc1_RequirementsPkg.
2. In the Uc1_RequirementsPkg create a Requirements Diagram RD_Uc1LinksToSysReqs.
3. Move the use case and the allocated requirements into the diagram.
4. In the Tools Menu select Layout > Complete Relations > All.
4.3 System Functional Analysis

System functional analysis is use case based. Each use case is translated into an executable model. The model and the underlying requirements then are verified through model execution. Exemplarily, the two use cases Uc1ControlEntry and Uc2ControlExit will be translated into executable models.

The system functional analysis workflow is supported by a number of features of the Rhapsody SE-Toolkit. Fig.4-3 details the workflow and lists its support through the SE-Toolkit in the respective phases.

NOTE: In the case study, the chosen approach essentially follows the “Alternative 2’’ approach described in section 2.2.3.
4.3.1 Uc1ControlEntry

4.3.1.1 Project Structure of the Uc1 Model

A project structure that complies with the recommended one outlined in section 3, may be created automatically by means of the SE-Toolkit feature **Create System Model From Use Case**.

1. Right-click use case Uc1 Control Entry and select **SE-Toolkit/Create System Model From Use Case**.

2. Uc1ControlEntry associated actor blocks are moved into the ActorPkg.

3. IBD_Uc1ControlEntry contains the instances of the actors and the use case block created through the SE-Toolkit feature (no links between the parts).

4. System Block **Uc_Uc1ControlEntry** created through the SE-Toolkit feature.

5. The use case - incl. its requirements links – is moved into the Uc1ControlEntryPkg in the FunctionalAnalysisPkg. Additionally, the Toolkit feature created an empty Activity Diagram (**Uc1ControlEntryBlackBoxView**).
4.3.1.2 Definition of Uc1 Functional Flow

There is always a discussion whether actor swim lanes should be shown in an activity diagram. In many cases this may lead to "messy", hard to read diagrams. Focus of the activity diagram should be on the system's internal functional flow.

A recommended alternative is to capture the interactions of an action with the environment by means of a SysML Action Pin, stereotyped ActorPin (e.g. readSecurityCard). In this case the name of the ActorPin must be the name of the associated actor. The arrow in the pins shows the direction of the respective link (i.e. In, Out or In/Out). The creation of actor pins is supported by the SE-Toolkit (right-click on the relevant action and select Add Actor Pins).

The SE-Toolkit feature Create New Scenario From Activity Diagram uses the pin information when deriving sequence diagrams from the activity diagram (ref. Section 4.3.1.3).

**NOTE:** The action node resetAlarm – initiated by the Administrator – was added although there is no respective system requirement. It is considered a derived requirement. Derived requirements are stereotyped <<DerivedRequirement>> and stored in the DerivedRequirementsPkg.

**NOTE:** If an activity diagram contains too many details, some actions may be placed in a Reference Activity Diagram. Do not use SubActivity Diagrams because these can contain actions only for a single swim lane. In the later white-box activity diagrams the actions may span a number of swim lanes. For more details please refer to Appendix A4.
4.3.1.3 Definition of Uc1 Scenarios

Use case scenarios are created from the black-box activity diagram by means of the SE-Toolkit feature **Create New Scenario From Activity Diagram**.

1. In the Tools Menu select **Tools > SE-Toolkit>Modeling Toolbox**.
2. In the ModelingToolbox dialog box select **Generate SDs**.
3. Hold down Ctrl and select in the black-box activity diagram a sequence of actions.
4. In the ModelingToolbox dialog box click **Set Source**.
5. In the browser select system block **Uc_Uc1ControlEntry**.
6. In the ModelingToolbox dialog box click **Set Destination**.
7. In the ModelingToolbox dialog box click **Create New Scenario From Activity Diagram**.

Alternatively select a single action as the source. The tool will auto-create the sequence until it reaches a condition connector. The user is then given the choice of which path to take.
Extending the Automatically Generated Sequence Diagram

In the automatically created sequence diagram the lifeline of the use case block shows the sequence of realized operations, which corresponds to the chosen sequence of actions in the activity diagram. The sequence diagram further shows the lifelines of those actors that interact with the use case as well as the associated message exchanges. The direction of the message exchange corresponds to the one defined in the pin of the relevant action in the activity diagram. The message names are generic and need to be manually updated and realized.
Activity View Consistency Check

The consistency between actions of the black-box activity diagram and the operations in the derived use case scenarios may be checked by means of the SE-Toolkit feature *Perform Activity View Consistency Check*.

For the selected Activity View the feature confirms that

- Each action on the activity diagram appears on at least one of the sequence diagrams referenced by the Activity View
- Each operation on the referenced sequence diagrams appears at least once on the activity diagram

Right-click Uc1_ControlEntryBlackBoxView and select *SE-Toolkit\Perform Activity View Consistency Check*.

The screenshot above shows the result of the consistency check after the first use case scenario was generated. It lists those operations that have not yet been addressed. They will be captured in the following exception scenarios.
Manually completed Use Case Scenario
_Uc1_Sc2 (Exception Scenario 1)_

_Preconditions:
Card validation failed three times_

- reqReadSecurityCard()
- validSecurityCard(CardStatus)
- displayCardStatus(CardStatus)
- Security Card Not Valid
- reqSecurityCardFailure(ScFailCount)
- ScFailCount > 3
- disableUserAccount()
- logAccountData()
- reqProcessAlert(AlertType="User Account Disabled")
- alarm()
- reqResetAlarm()

**User**
- readSecurityCard
- validateSecurityCard(CardStatus)
- scanBiometricData
- authenticateBiometricData(AuthenticationStatus)
- flagBiometricScanFailure(BsFailCount)
- flagSecurityCardFailure
- Biometric Data Not Authenticated
- BsFailCount == 3
- disableUserAccount(Admin)
- logAccountData
- alarm
- unlockAccessPoint «MessageAction» AccessPoint
- lockAccessPoint «MessageAction» AccessPoint
- resetAlarm(Admin)

**Admin**
- reqScanBiometricData()
- scanBiometricData()
- authenticateBiometricData(AuthenticationStatus)
- displayAuthenticationStatus
- displayCardStatus(CardStatus)
- displayAuthenticationStatus(AuthenticationStatus)
- logAccountData
- alarm
- unlockAccessPoint «MessageAction» AccessPoint
- lockAccessPoint «MessageAction» AccessPoint
- resetAlarm

**First Request**
- takePicture «MessageAction» Camera

Manually completed Use Case Scenario
_Uc1_Sc3 (Exception Scenario 2)_

_Preconditions:
Valid security card, Biometric Scan failed three times_

- reqScanBiometricData()
- scanBiometricData()
- authenticateBiometricData(AuthenticationStatus)
- displayAuthenticationStatus
- BsFailCount == 3
- disableUserAccount(Admin)
- logAccountData
- alarm
- unlockAccessPoint «MessageAction» AccessPoint
- lockAccessPoint «MessageAction» AccessPoint
- resetAlarm

**Admin**
- reqScanBiometricData()
- scanBiometricData()
- authenticateBiometricData(Admin)
- displayAuthenticationStatus
- displayCardStatus(CardStatus)
- displayAuthenticationStatus(AuthenticationStatus)
- logAccountData
- alarm
- unlockAccessPoint «MessageAction» AccessPoint
- lockAccessPoint «MessageAction» AccessPoint
- resetAlarm

**First Request**
- takePicture «MessageAction» Camera
**4.3.1.4 Definition of 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: \( \text{p<TargetName>} \)

Interface names are referenced to the sender port.

Naming convention: \( \text{i<Sender>_<Receiver>} \)

**NOTE:** The interface definitions and associated event definitions are allocated in the **InterfacesPkg**.

For readability reasons it is recommended not to show the interface names in the diagram. Deselect in each block the Display Option “Show Port Interfaces”.

**Internal Block Diagram IBD_Uc1_ControlEntry with Ports and Interfaces**

Right-click the package **Uc1_ControlEntry_BBScenarios** and select **SE-Toolkit/Create Ports And Interfaces**.

Manually connect ports.
4.3.1.5 Definition of State-based Behavior

State-based behavior is added to the use case block and the actors using Statechart Diagrams. The use case statechart diagram represents the aggregate of all flows in the black-box activity diagram and the associated sequence diagrams. Guidelines how to derive a statechart diagram from the information captured in the activity diagram and sequence diagrams are documented in the Appendix A3.

Actor statechart diagrams are typically very simple, just providing the required events to trigger operations / state changes in the use case block.
Case Study: System Functional Analysis

State-based Behavior of Use Case Uc1ControlEntry

A statechart should be hierarchically structured. This allows the reuse of behavior-patterns in later phases (e.g. Detailed Architectural Design).
4.3.1.6 Uc1 Model Verification

The Uc1ControlEntry model is verified through model execution on the basis of the captured use case scenarios. The correctness and completeness analysis is based on the visual inspection of the model behavior. The *Rhapsody* tool provides two ways to visualize model behavior:

- Visualization of the state-based behavior through animation of respective statecharts
- Visualization of message sequences by means of automatically generated sequence diagrams

Animated Sequence Diagram (Uc1Sc1)

Animated Statechart Diagram (Uc1ControlEntryCtr)
Case Study: System Functional Analysis

The analysis via sequence diagrams is supported by the *Rhapsody Sequence Diagram Compare* feature. This feature enables to perform comparisons between two sequence diagrams, e.g., one capturing the sequence of a required scenario and the other showing the recorded scenario. The differences between the diagrams are shown color-coded. This feature may also be used to compare two runs for regression testing.

<table>
<thead>
<tr>
<th>Arrow</th>
<th>Name</th>
<th>Color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>Blue</td>
<td>Msg matches in both SD</td>
<td></td>
</tr>
<tr>
<td>Pink</td>
<td>Pink</td>
<td>Msg is missing in the other SD</td>
<td></td>
</tr>
<tr>
<td>Green</td>
<td>Pink</td>
<td>Msg has different arguments in the other SD</td>
<td></td>
</tr>
<tr>
<td>Orange</td>
<td>Orange</td>
<td>Msg arrives at a different time in the other SD</td>
<td></td>
</tr>
<tr>
<td>Gray</td>
<td>Gray</td>
<td>Msg was excluded from comparison</td>
<td></td>
</tr>
</tbody>
</table>

Sequence Diagram Compare: Scenario Uc1_Sc1
4.3.1.7 Linking Uc1 Block Properties to Requirements

In order to assure that all Uc1 allocated functional and performance requirements are considered, traceability links from the Uc1 block properties to the system requirements are established using a satisfy dependency.

<table>
<thead>
<tr>
<th>ID</th>
<th>System Requirement</th>
<th>&lt;&lt;satisfy&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS11-3</td>
<td>Employee ID Card Identification - Exit</td>
<td>readSecurityCard(), validateSecurityCard()</td>
</tr>
<tr>
<td>SS11-4</td>
<td>Biometric Scan</td>
<td>scanBiometricData()</td>
</tr>
<tr>
<td>SS11-5</td>
<td>Time Between Two Independent Checks</td>
<td>t_Bs</td>
</tr>
<tr>
<td>SS11-7</td>
<td>Three Attempts On Employee ID Entry</td>
<td>ScFailCount, flagSecurityCardFailure()</td>
</tr>
<tr>
<td>SS11-8</td>
<td>Three Attempts On Biometric Data Entry</td>
<td>BsFailCount, flagBiometricScanFailure()</td>
</tr>
<tr>
<td>SS11-9</td>
<td>Three Attempts On Employee ID Exit</td>
<td>ScFailCount, flagSecurityCardFailure()</td>
</tr>
<tr>
<td>SS11-10</td>
<td>Denied Entry Notification</td>
<td>logEntryData(), logAccountData(), reqProcessAlert()</td>
</tr>
<tr>
<td>SS11-11</td>
<td>Denied Exit Notification</td>
<td>logExitData(), logAccountData(), reqProcessAlert()</td>
</tr>
<tr>
<td>SS11-12</td>
<td>Alarm - Entry</td>
<td>alarm()</td>
</tr>
<tr>
<td>SS11-13</td>
<td>Alarm - Exit</td>
<td>alarm()</td>
</tr>
<tr>
<td>SS11-18</td>
<td>Alarm Reset</td>
<td>resetAlarm()</td>
</tr>
<tr>
<td>SS11-14</td>
<td>Disabling User Account</td>
<td>disableUserAccount()</td>
</tr>
<tr>
<td>SS11-15</td>
<td>Visualization of Security Card Check Status - Entry</td>
<td>displayCardStatus()</td>
</tr>
</tbody>
</table>

Tab.4-2 Links of Uc_Uc1ControlEntry Block Properties to System Requirements

<table>
<thead>
<tr>
<th>ID</th>
<th>System Requirement</th>
<th>&lt;&lt;satisfy&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS11-16</td>
<td>Visualization of Security Card Check Status - Exit</td>
<td>displayCardStatus()</td>
</tr>
<tr>
<td>SS11-17</td>
<td>Visualization of Biometric Data Check Status</td>
<td>displayAuthenticationStatus()</td>
</tr>
<tr>
<td>SS11-2</td>
<td>Access Precondition</td>
<td>validateSecurityCard()</td>
</tr>
<tr>
<td>SS11-4</td>
<td>Out of Date Cards - Entry</td>
<td>validateSecurityCard()</td>
</tr>
<tr>
<td>SS11-5</td>
<td>Out of Date Cards - Exit</td>
<td>validateSecurityCard()</td>
</tr>
<tr>
<td>SS112-1</td>
<td>Approval of Biometric Data</td>
<td>authenticateBiometricData()</td>
</tr>
<tr>
<td>SS12-2</td>
<td>Entry Time</td>
<td>t_Unlocked</td>
</tr>
<tr>
<td>SS12-3</td>
<td>Exit Time</td>
<td>t_Unlocked</td>
</tr>
<tr>
<td>SS12-5</td>
<td>Automatic Securing the Secure Area - Entry</td>
<td>evAccessPointLocked()</td>
</tr>
<tr>
<td>SS12-6</td>
<td>Automatic Securing the Secure Area - Exit</td>
<td>evAccessPointLocked()</td>
</tr>
<tr>
<td>MA1-1</td>
<td>Image Capture</td>
<td>reqTakeSnapshot()</td>
</tr>
<tr>
<td>MA2-1</td>
<td>Time Recording</td>
<td>logExitData()</td>
</tr>
<tr>
<td>MA2-2</td>
<td>Alarm Conditions</td>
<td>checkForTimeLimitViolations()</td>
</tr>
</tbody>
</table>
There are two ways to implement the <<satisfy>> dependency listed in Tab.4-2.

- directly in the browser using the SE-Toolkit feature Create Dependency, or
- graphically, in a Requirement Diagram.

It is recommended to start with the SE-Toolkit feature Create Dependency. If considered necessary – e.g. for discussions or documentation purposes - the dependencies may then be visualized in a Requirements Diagram.

1. In the Tools Menu select Tools > SE-Toolkit>Modeling Toolbox
2. In the dialog box select Dependencies. Tick dependency Satisfy.
3. In the system block SecuritySystem select a property you want to link to a system requirement.
4. In the Modeling Toolbox dialog box click Set Source.
5. In the SystemRequirementsPkg select the relevant system requirement.
6. In the Modeling Toolbox dialog box click Set Destination.
7. In the Modeling Toolbox dialog box click Create Dependency.
Visualization of the Dependencies in a Requirements Diagram

1. In the RequirementsPkg create a Requirements Diagram RD_Uc1BlockLinksToSysReqs.
2. Move the operations and attributes from the Uc_Uc1ControlEntry block into the diagram.
3. Move the associated system requirements from the SystemRequirementsPkg into the diagram.
4. In the Tools Menu select Layout > Complete Relations > All

SecSysControl Block Properties Mapped to System Requirements (Excerpt)
4.3.2 Uc2ControlExit

4.3.2.1 Project Structure of the Uc2 Model

The generic project structure of the Uc2 model (=> Uc2ControlExitPkg in the FunctionalAnalysisPkg) is created by means of the SE-Toolkit feature Create System Model From Use Case.

4.3.2.2 Definition of Uc2 Functional Flow

**NOTE**: The interaction of an action node with the environment is captured by means of a SysML Action Pin, stereotyped ActoPin (e.g. readSecurityCard).

---

**Functional Flow in Uc2ControlExit (Black-Box View)**
4.3.2.3 Definition of Uc2 Scenarios

Use case scenarios are created from the black-box activity diagram by means of the SE-Toolkit feature **Create New Scenario From Activity Diagram**.

**Manually completed**
**Use Case Scenario Uc2_Sc1 (Nominal Scenario)**

**Manually completed**
**Use Case Scenario Uc2_Sc2 (Exception Scenario)**
4.3.2.4 Definition of Ports and Interfaces

**NOTE:** All use case models – and later also the architectural design model - share common actors. The structure (i.e. ports and interfaces) as well as the behavior of the actor blocks are incrementally extended according to the needs of the relevant model. In the internal structure diagram below. The use case 1 related ports (pUc1ControlEntry) of the actor blocks User, Admin, and AccessPoint intentionally are not shown (deleted from view - **not** deleted from **model**).

4.3.2.5 Definition of State-based Behavior

The statechart diagrams of the actors User and AccessPoint need to be extended, taking into consideration the generation of the User’s exit request (RequestExit) and the communication between the blocks Uc2ControlExit and AccessPoint.

Note the reuse of behavior patterns in the statechart diagram of the use case block. The system states *ProcessingSecurityCard Data* and *UnlockingAndLockingAccessPoint* are identical to the ones used in the use case block Uc_Uc1ControlEntry.
4.3.2.6 Uc2 Model Verification

The Uc2ControlExit model is verified through model execution on the basis of the captured use case scenarios. The correctness and completeness analysis is based on the visual inspection of the model behavior.

Verification of the Use Case Model Uc2ControlExit through Model Execution (Uc2Sc1)

4.3.2.7 Linking Uc2 Block Properties to Requirements

With the exception of the operation checkForTimeLimitViolations and the associated timeout attribute \( t_{Update} \) all properties of the use case ControExit are identical to the properties of the use case ControlEntry and do not need to linked.
4.3.3 Merging Use Case Blocks

Once all use case models are verified, the operations, receptions, and attributes of each use case block are merged into the system block `SecuritySystem` in the `ArchitecturalDesignPkg`

1. In the `ArchitecturalDesignPkg` create a system block `SecuritySystem`
2. Right click the block `SecuritySystem` and select `SE-Toolkit\Merge Functional Analysis`

This SE-Toolkit feature copies all events, operations and attributes from all use case blocks into the `SecuritySystem` block.

3. Duplicates are noted.

4. Copies are traced back to the origins.

`SecuritySystem Block with Merged Operations/Attributes`
4.4 Design Synthesis

4.4.1 Architectural Analysis (Trade-Off Analysis)

The focus of the Architectural Analysis is on the determination of a solution that fulfills best the required functionality identified in the system functional analysis phase. Fig. 4-4 details the architectural analysis workflow and lists its support through the Rhapsody SE-Toolkit in the respective phases.

**NOTE:** In the case study the solution of only one key system function (Capture Biometric Data) will be elaborated. Therefore the phase Merge Solutions to Form System Architecture will be skipped.

![Diagram of Architectural Analysis Workflow and Support through Rhapsody SE-Toolkit](image-url)
Case Study: Design Synthesis

4.4.1.1 Definition of Key System Functions

The objective of this stage is to group the system functions together in such a way that each group can be realized by a physical component.

Step 1: Group related system functions into key system functions

The following 3 key system functions were identified through analysis of the use case black-box activity diagrams:

**ReadCardInformation:**
- readSecurityCard
- displayCardStatus
- alarm
- resetAlarm

**CaptureBiometricData:**
- scanBiometricData
- authenticateBiometricData
- displayAuthenticationStatus

**ControlSecSys:**
- validateSecurityCard
- flagSecurityCardFailure
- flagBiometricCheckFailure
- disableUserAccount
- logAccountData
- logEntryData
- logExitData
- checkForTimeLimitViolations
- unlockAccessPoint
- lockAccessPoint

Step 2: Define and apply first cut design criteria

Typically, the first cut design criterion is to decide which of the key functions would be realized as a COTS component or developed internally. In this case study it was decided that the functions ReadCardInformation and CaptureBiometricData would be bought and the function ControlSecSys developed internally.

Due to the number of ways in which the key function CaptureBiometricData can be realized, it was decided to carry out a Trade Study. It was not considered necessary for the key function ReadCardInformation.
4.4.1.2 Definition of Candidate Solutions

The objective of this phase is to identify possible solutions for a chosen key system function.

**Step1: Identify solutions to the chosen key system function**

In this case study the chosen key function is CaptureBiometricData. Possible solutions are:

- Facial Recognition
- Fingerprint Scanner
- Optical Scanner (examining iris or retina)

**Step2: Select candidate solutions for further analysis**

Facial recognition systems are at present not very reliable technology, also they are very expensive to install and maintain. Two practical candidate solutions remain that will be carried forward for further analysis i.e.

- Fingerprint Scanner
- Optical Scanner (Cornea or Iris Scanner)

This information can now be entered into the model.

1. In the DesignSynthesisPkg create a package **ArchitecturalAnalysisPkg**
2. In the ArchitecturalAnalysisPkg create a package **TradeStudyAnalysisPkg**
3. In the TradeStudyAnalysisPkg create a package **BiometricScanTradeStudy**
4. In the BiometricScanTradeStudy package create a Block Definition Diagram called **BDD_CaptureBiometricDataOptions**
5. In the BiometricScanTradeStudy create the following blocks
   - Capture Biometric Data
   - Optical Scanner
   - FingerprintScanner
4.4.1.3 Definition of Assessment Criteria

Assessment criteria typically are based upon customer constraints, required performance characteristics, and/or cost.

Assessment criteria are normally subjective but can also be very specific. A subjective target could be low cost. A specific target could be a precise measure of accuracy i.e. +/- 0.1 mm. In this case study the assessment criteria are a mixture of both.

The assessment criteria and the associated classification in this case study are:

- Accuracy
- Purchase
- Installation and
- Maintenance Cost

The assessment criteria are captured in the model by adding to the block CaptureBiometricData for each assessment criterion a respective attribute, stereotyped <<moe>>.
4.4.1.4 Assigning Weights to Assessment Criteria

Not all assessment criteria are equal. Some are more important than others. Assessment criteria are weighted according to their relative importance to the overall solution. The weighting factors are normalized to add up to 1.0.

Step 1: Rank the assessment criteria

The ranking for the assessment criteria in this case study is

1. Accuracy
2. Security
3. Purchase Cost
4. Installation Cost
5. Maintenance

Step 2: Assign weightings to assessment criteria

In the case study the weightings of the chosen assessment criteria are

- Accuracy: 0.30
- Security: 0.25
- Purchase Cost: 0.20
- Installation Cost: 0.15
- Maintenance Cost: 0.10

These values are represented in the model by a tag called weight attached to each of the <<moe>> attributes.

1. In each <<moe>> attribute select the tab Tags and add the appropriate value.

The CaptureBiometricData block attributes are copied into the solutions blocks by means of the SE-Toolkit feature Copy MOEs to Children.

2. Right-click the CaptureBiometricData block and select SE-Toolkit\Copy MOEs to Children.
4.4.1.5 Definition of a Utility Curve for each Criterion

The utility curve is a function that compares the outcome of an objective analysis to a target and outputs a normalized value typically between 0 and 10 to indicate how well the target is met.

To determine the MoE for accuracy create a linear utility curve that examined the relationship between errors/thousand readings (0-10 errors per thousand) and a scale of 0-10.

![Accuracy Utility Curve](image)

**Accuracy Utility Curve**

**NOTE:** For a simple linear function the utility curve can be calculated from the following formula

\[ \text{MoE} = -(\text{MoE range/target range}) + \text{MOE range} \]

This simple chart yields the formula

**Accuracy MoE = -Errors Per Thousand + 10**

With regards to the **purchase cost** it is assumed that ideally the target figure that the company would wish to pay for the hardware is $0 and the maximum is $400 dollars a unit. This gives a utility curve based upon the linear graph formula described earlier - of

**Purchase cost MoE = -0.025\times\text{Purchase Cost} + 10**

![Purchase Cost Utility Curve](image)

**Purchase Cost Utility Curve**

For the **installation cost** of the hardware, a maximum budget of $1500 was estimated for 10 units. This gives a utility curve described by the function

**Installation Cost MoE = -0.0067\times\text{installation cost} + 10**

![Installation Cost Utility Curve](image)

**Installation Cost Utility Curve**
4.4.1.6 Assigning Measures of Effectiveness (MoE) to each Solution

Accuracy: Fingerprint scanners are approximately in the order of 2-3 failures per 1000. For an error per thousands value of 2.5 this yields an MoE of 7.5 for the fingerprint scanner. Optical scanning systems have failure rates of 0.001 per 1000. this yields an MoE of 9.999 or effectively 10 for the optical scanner.

Purchase Cost: For the hardware to capture biometric data it has been estimated at $110 dollars for the fingerprint scanner and $ 250 for the optical scanner. From the purchase cost utility function, a purchase cost MoE of 7.25 is calculated for the fingerprint scanner and a purchase cost MoE of 3.75 for the optical scanner.

Installation Cost: For 10 units it was estimated to be $ 600 for the fingerprint scanner and $ 1175 for the optical scanner. From the installation cost utility function, an installation cost MoE of 6.0 is calculated for the fingerprint scanner and an installation cost MoE of 2.12 for the optical scanner.

Security: It has been found that optical scanners (iris or retina) are impossible to fool, whereas fingerprint scanners have been fooled with relatively simple methods. With this mind it was decided to give fingerprint scanners a security MoE of 8.0 and optical scanners a security MoE of 10.0.

Maintenance: Both systems under consideration need little maintenance. However, optical scanners need slightly more maintenance than fingerprint scanners due to their sensitivity to light and the degree of cleanliness required. With this mind it was decided to give fingerprint scanners a maintenance MoE of 8.0 and optical scanners a maintenance MoE of 6.0.

In the browser select a block representing one of the solutions and open its features.

1. Select the attribute tab.
2. Select the attribute to be edited and in the Initial Value field enter the expected value.
3. Select and edit each attribute in turn.

Repeat steps 1-4 for each block representing a solution.

BDD_CaptureBiometricDataOption

- FingerprintScanner
- OpticalScanner
4.4.1.7 Determination of Solution

Once each of the key functions has a number of possible solutions with MoEs assigned to them, it is possible to combine the various solutions in order to determine the optimum solution for the architecture.

The means of building the possible architectures is through the Solution Architecture Diagram. It shows the component options required to build the final variant architectures for the complete architecture or key function. The two possible variant architectures in this case, consist of either the FingerprintScanner or the OpticalScanner. There are no additional components required.

**Step 1: Build Solution Architecture Diagram**

This diagram is created in the TradeStudyAnalysisPackage. It shows the composition of the final product as made up from possible solutions. Using this diagram it is possible to mix several different solutions to key functions to realize complete system architecture. In this instance there is only one component to be analyzed for each architecture.

1. In the BiometricScanTradeStudy package create a Block Definition Diagram **BDD_SolutionArchitecture**.
2. Create a block called **FingerprintScannerArchitecture**.
3. Drag on the FingerprintScanner block and using the decomposition relationship make it part of the OpticalScannerArchitecture.
4. Create a block called **OpticalScannerArchitecture**.
5. Drag on the OpticalScanner block and using the decomposition relationship make it part of the OpticalScannerArchitecture.
Step 2: Perform Weighted Objectives calculation

Once the possible solution architectures are in place, the analysis to determine the best solution from the presented options can be carried out. The means of doing this analysis is the Weighted Objectives Calculation. It is used to determine the solution for a particular function. It consists of multiplying the value for each MoE by its respective importance weighting, and then adding the resultant values together. This is carried out for each solution for each function. The sum of the combined solutions with the highest score is selected as the implementation for that particular architecture or function. The actual calculation is carried out and displayed within an Excel spreadsheet.

To support this calculation within Rhapsody and Excel, one further diagram is required: the Option Analysis Diagram. The option analysis diagram shows all the variant architecture solutions for the key function under consideration.

1. In the BiometricScanTradeStudy package create a Block Definition Diagram BDD_OptionAnalysis.
3. In the browser right-click BDD_OptionAnalysis and select SE-Toolkit\Perform Trade Analysis.

Excel will then open up with the results of the analysis. From this analysis it can be seen that the Fingerprint Scanner scores slightly higher (despite the higher scores for the optical scanner in the areas of accuracy and security) and so will be selected as the implementation of the function ScanBiometricData.
4.4.2 Architectural Design

Fig. 4-5 shows the architectural design workflow in the case study and lists its support by means of the SE-Toolkit.

Based on the design concept that was elaborated in the previous architectural analysis phase it starts with the decomposition of the system block into parts.

The allocation of the system block operations to the parts is elaborated for each use case by graphically allocating operations in the use case white-box activity diagrams (ref. section 2.2.3.2). The allocation is formalized by means of the SE-Toolkit feature Allocate Operations from Swimlanes.

Once all system block operations that capture system functional requirements are allocated to parts, non-functional requirements (e.g. design constraints) are allocated to the relevant parts and respective trace links are established.

4.4.2.1 Security System Decomposition

Based on the results of the previous trade study, the constituents of the Security System will be

- Two Card Readers (one for entry and one for exit)
- a Fingerprint Scanner
- a Control Unit

As the card readers and the fingerprint scanner are considered off-the-shelf components, the focus of the case study will be on the specification of the control unit – in the following referred to as SecSysController.

The chosen system architecture is captured in the block definition diagram BDD_SecuritySystem and the internal block diagram IBD_SecuritySystem. Both diagrams are created in the ArchitecturalDesignPkg.
By defining a composition relationship between the system block SecuritySystem and the subsystem blocks in the block definition diagram, automatically instances of the subsystem blocks are created in the SecuritySystem block. As two instances of the CardReader are needed (CardReader_Entry and CardReader_Exit) the composition relationship in the block definition diagram needs to be defined twice. Name the instances accordingly.

It is recommended to standardize the structure of the Architectural DesignPkg. If a system block is decomposed into parts, each part should be allocated to a corresponding package within a package named <SystemBlockName>DecompositionPkg. The creation of this structure is automated by means of the SE-Toolkit feature Create Sub Packages:

Right-click the SecuritySystem block, select SE-Toolkit>Create Sub Packages.
4.4.2.2 Graphical Allocation of Operations

The allocation of operations to the parts of a system block is elaborated graphically (White-Box Activity Diagram). Black-box use case activity diagrams are partitioned into swim lanes, each of which corresponds to a part of the decomposed system block (case study: CardReader_Entry, CardReader_Exit, FingerprintScanner, and SecSysController). Based on design considerations, operations (= actions) then are “moved” to respective swim lanes. An essential requirement is that the initial links between the operations are maintained.

1. In the SecuritySystemDecompositionPkg create a package SecuritySystemWB_AD.
2. In the Uc1ControlExitPkg right-click Uc1ControlEntryBlackBoxView and select Duplicate Activity View.
3. Rename the copied Activity View to Uc1_SecSysWhiteBoxView and move it into the SecuritySystemWB_AD package.
4. In the Uc2ControlExitPkg right-click Uc2ControlExitBlackBoxView and select SE-Toolkit/Duplicate Activity View.
5. Rename the copied Activity View to Uc2_SecSysWhiteBoxView and move it into the SecuritySystemWB_AD package.
6. Partition Uc1ControlEntryWhiteBoxView into swimlanes: CardReader_Entry, FingerprintScanner, and SecSysController.
7. Partition Uc2ControlExitWhiteBoxView into swimlanes: CardReader_Exit and SecSysController.

Creating a White-Box Activity Diagram:
On top of the copied black-box activity diagram create an empty activity diagram with swimlanes. Move the operations “bottom-up” into the subsystem swimlanes.
White-Box Activity Diagram Uc1ControlEntry

White-Box Activity Diagram Uc2ControlExit
**NOTE**: In order to provide the required functionality for the chosen design, two actions that do not have an associated system requirement had to be added to the white-box activity diagrams:

*enableBiometricScan* and *disableBiometricScan*

A respective derived requirement needed to be formulated and stored in the DerivedRequirementsPkg.

**Summarizing the Allocation of Operations**

For each white-box activity diagram the allocation of operations may be summarized in an Excel spreadsheet by means of the SE-Toolkit feature *Create Allocation Table*.

1. Right-click Uc1ControlEntryWhiteBoxView. Select *SE-Toolkit/Create Allocation Table*

<table>
<thead>
<tr>
<th>FingerprintScanner</th>
<th>SecSysController</th>
<th>CardReader_Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>scanBiometricData</td>
<td>validateSecurityCard</td>
<td>alarm</td>
</tr>
<tr>
<td>authenticateBiometricData</td>
<td>flagBiometricScanFailure</td>
<td>displayCardStatus</td>
</tr>
<tr>
<td>enableBiometricScan</td>
<td>disableUserAccount</td>
<td>readSecurityCard</td>
</tr>
<tr>
<td>disableBiometricScan</td>
<td>flagSecurityCardFailure</td>
<td>resetAlarm</td>
</tr>
<tr>
<td>displayAuthenticationStatus</td>
<td>logEntryData</td>
<td></td>
</tr>
<tr>
<td>logAccountData</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Allocation Table of Uc1ControlEntryWhiteBoxView (Excel Spreadsheet)*

2. Right-click Uc2ControlExitWhiteBoxView. Select *SE-Toolkit/Create Allocation Table*

<table>
<thead>
<tr>
<th>SecSysController</th>
<th>CardReader.Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>validateSecurityCard</td>
<td>alarm</td>
</tr>
<tr>
<td>checkForTimeLimitViolations</td>
<td>readSecurityCard</td>
</tr>
<tr>
<td>flagSecurityCardFailure</td>
<td>displayCardStatus</td>
</tr>
<tr>
<td>logExitData</td>
<td>resetAlarm</td>
</tr>
</tbody>
</table>

*Allocation Table of Uc2ControlExitWhiteBoxView (Excel Spreadsheet)*

Generally it is recommended to update and verify the black-box use case models, i.e. to generate new black-box use case scenarios. In this case study, this step was intentionally skipped.
4.4.2.3 Formalizing the Allocation of Operations

Once an allocation concept is elaborated, the allocation is formalized by copying the system block operations and receptions – incl. documentation and requirements dependencies - to respective subsystem blocks. This process is supported by the SE-Toolkit feature Allocate Operations from Swimlanes.

1. In the Tools Menu select Tools > SE-Toolkit > Modeling Toolbox
2. In the Modeling Toolbox dialog box select Harmony.
3. In the SecuritySystemWB_AD package select Activity Diagram(s)
4. In the Modeling Toolbox dialog box click Set Source.
5. In the ArchitecturalDesignPkg select system block SecuritySystem.
6. In the Modeling Toolbox dialog box click Set Destination.
7. In the Modeling Toolbox dialog box click Allocate Operations from Swimlanes
Case Study: Design Synthesis

The reason for the error message below is, that - as mentioned in the previous paragraph - the actions/operations `enableBiometricScan` and `disableBiometricScan` were added afterwards to the white-box activity diagram Uc1ControlEntry. Therefore they are not included in the set of merged use case operations in the SecuritySystem block.

In order to add these operations to the SecuritySystem block and to allocate them to the FingerprintScanner block:

1. In the dialog box select Errors,
2. Select Action `enableBiometricScan` and Action `disableBiometricScan`
3. click Accept Selected.
4.4.2.4 Allocation of Non-functional Requirements

So far the focus was on the allocation of system-level operations and associated functional system requirements to the parts of the chosen architectural decomposition. Latest at this stage, derived functional requirements should have been approved and linked to respective operations. The next task in the architectural design phase is the allocation of non-functional requirements. In order to assure that all non-functional requirements are considered, traceability links from the relevant subsystem block to the non-functional system requirements need to be defined using a satisfy dependency.

Once all functional and non-functional requirements are mapped to the architecture, it is recommended to reflect the partitioning by grouping the requirements in the SystemRequirementsPkg accordingly:

1. In the SystemRequirementsPkg create the subsystem packages
   - SecSysControllerReqsPkg
   - CardReaderReqsPkg
   - FingerprintScannerReqsPkg

2. Move the system requirements incl. the – approved – derived requirements into respective packages

The subsystem requirements package is part of the later hand-off from systems engineering to the subsequent subsystem development.
Visualization of the Traceability Links between Subsystem SecSysController Properties and Associated Functional / Non-functional System Requirements in a Requirements Diagram
4.4.3 Detailed Architectural Design

In this case study the architectural design phase stopped at the subsystem level. The focus of the subsequent detailed architectural design phase is on the definition of the resulting subsystem ports and interfaces and the state-based behavior of each subsystem.

Fig. 4-6 details the workflow and its support by means of the SE-Toolkit. The subsystem ports and interfaces are identified from White-Box Sequence Diagrams. White-box sequence diagrams are derived from the use case white-box activity diagrams that were created in the previous architectural design phase. The focus of black-box sequence diagrams was on the identification of the required sequences of system functions (operations). White-box activity diagrams focus on the collaboration between the different subsystems taking into consideration the allocation of the operations.

The derivation of white-box sequence diagrams is performed iteratively for each use case white-box activity diagram.

Once the state-based behavior of each subsystem block is defined, the correctness and completeness of the system architecture model is checked through model execution.
4.4.3.1 *Derive White-Box Sequence Diagrams*

**Step 1: Allocate Events**

The allocation of SecuritySystem block receptions (events) to the subsystems is performed by means of the SE-Toolkit feature *Architectural Design Wizard*.

**NOTE:** This SE-Toolkit feature may also be used for allocating operations.

1. In the ArchitecturalDesignPkg right-click SecuritySystem block and select *SE-Toolkit > Architectural Design Wizard*.
2. In the SE-Toolkit dialog box un-tick *Show Operations*.
3. In the *Parts* drop-down menu select a part.
4. In the *Unlocated Operations / Events* window select an event and click *Selected* to allocate the event to the chosen part.

**NOTE:** If the event needs to be allocated to more than one subsystem, drag & drop it from the pool to the part.
Case Study: Design Synthesis

Allocation of Security System Events to Parts through the Architectural Design Wizard

System Receptions (Events) allocated to Subsystem Blocks
Case Study: Design Synthesis

Step 2: Derive White-Box Scenarios

White-box scenarios are derived from the white-box activity diagrams by means of the SE-Toolkit feature *Create New Scenario From Activity Diagram* (ref. section 4.3.1.3). *NOTE:* Select as destination the ArchitecturalDesignPkg.

![Diagram of activity diagrams showing steps for designing authentication and security processes.](image)
White-Box Use Case Scenario WB_Uc1Sc1 (Nominal Scenario)
Manually added service requests and operations marked in red
Manually completed White-Box Use Case Scenario

**WB_Uc1Sc3 (Exception Scenario 2)**

**WB_Uc1Sc2**

**Manually completed White-Box Use Case Scenario**

**WB_Uc1Sc3 (Exception Scenario 1)**
4.4.3.2 Definition of Ports and Interfaces

Once all black-box use case scenarios are decomposed into white-box scenarios, the resulting subsystem ports and interfaces can be defined by means of the SE-Toolkit feature Create Ports And Interfaces.

Right-click the package WB_UseCaseScenarios and select SE-Toolkit\SE-Toolkit\Create Ports And Interfaces.

**NOTE**: The SE-Toolkit feature only defines the behavioral ports and associated required/defines interfaces.


3. Manually connect ports.
Documentation of System Interfaces (ICD)

A commonly used artifact for the documentation of the communication in a network is the N-squared ($N^2$) chart. In an $N^2$ chart the basic nodes of communication are located on the diagonal, resulting in an NxN matrix for a set of N nodes. For a given node, all outputs (SysML required interfaces) are located in the row of that node and inputs (SysML provided interfaces) are in the column of that node. The diagram below depicts the $N^2$ chart of the Security System architecture.

The $N^2$ chart of the Security System architecture may be generated by means of the SE-Toolkit feature \textbf{Generate N2 Matrix:}\n
In the ArchitecturalDesignPkg right-click the internal block diagram IBD_SecuritySystem and select \textit{SE-Toolkit\textbackslash Generate N2 Matrix}.

\textbf{NOTE:} In the N2 chart below the SecuritySystem column/row describes the system-level interfaces (= delegation port interfaces).

<table>
<thead>
<tr>
<th>SecuritySystem</th>
<th>User</th>
<th>Admin</th>
<th>Camera</th>
<th>AccessPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecuritySystem</td>
<td>iUser_FingerprintScanner</td>
<td>iSecSysController_Admin</td>
<td>iSecSysController_Camera</td>
<td>iSecSysController_AccessPoint</td>
</tr>
<tr>
<td>User</td>
<td>iUser_FingerprintScanner</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Admin</td>
<td>iAdmin_SecSysController</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AccessPoint</td>
<td>iAccessPoint_SecSysController</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CardReader</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FingerprintScanner</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SecSysController</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CardReader</th>
<th>FingerprintScanner</th>
<th>SecSysController</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecuritySystem</td>
<td>iUser_FingerprintScanner</td>
<td>iSecSysController_Admin</td>
</tr>
<tr>
<td>User</td>
<td>iUser_FingerprintScanner</td>
<td></td>
</tr>
<tr>
<td>Admin</td>
<td>iAdmin_SecSysController</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>iAccessPoint_SecSysController</td>
<td></td>
</tr>
<tr>
<td>CardReader</td>
<td>iCardReader_SecSysController</td>
<td></td>
</tr>
<tr>
<td>FingerprintScanner</td>
<td>iFingerprintScanner_SecSysController</td>
<td></td>
</tr>
<tr>
<td>SecSysController</td>
<td>iSecSysController_CardReader</td>
<td>iSecSysController_FingerprintScanner</td>
</tr>
</tbody>
</table>

\textit{N^2 Chart of the Security System Architecture}
4.4.3.3 Definition of State-based Behavior

The statechart diagrams of the actors User and AccessPoint need to be extended, taking into consideration the communication via the additional ports in the User block (pCardReader_Entry, pCardReader_Exit, and pFingerprintScanner) and in the AccessPoint (pSecSysController).

**System Architecture Model (Context View)**

**Extended Actor Behavior for the System Architecture Model**
Case Study: Design Synthesis

System Architecture Model (Subsystems View)

State-based Behavior of the FingerprintScanner Block
State-based Behavior of the CardReader Block

The state-based behavior of the CardReader block has to take into consideration the different roles the card reader plays in the Security System. It is either used for entry control or exit control. A Constructor need to be defined to assign the relevant instance of the CardReader block to its associated role.

1. In the CardReader block, right-click Operations > Add New > Constructor: Add the argument CR of type OMString.

2. Initialize the CardReader instances:
   - CardReader_Entry -> CardReader("Entry")
   - CardReader_Exit -> CardReader("Exit")

State-based Behavior of the CardReader Block

In the CardReader block, right-click Operations > Add New > Constructor: Add the argument CR of type OMString.

Initialize the CardReader instances:
- CardReader_Entry -> CardReader("Entry")
- CardReader_Exit -> CardReader("Exit")
Case Study: Design Synthesis

State-based Behavior of the SecSysController Block

Note the reuse of behavior patterns in the statechart diagram of the SecSysController block. The statemachines ProcessingSecurityCardData and ProcessingBiometricData are extended copies of the ones used in the use case blocks, UnlockingAndLockingAccessPoint is an unchanged copy.
4.4.3.4 System Architecture Model Verification

The system architecture model is verified through model execution on the basis of the captured use case scenarios. The correctness and completeness analysis is based on the visual inspection of the model behavior (animated statecharts and sequence diagrams).

Animated Sequence Diagram (WB_Uc1Sc1)
5 Hand-Off to Subsystem Development

In the Security System case study it was decided that the card readers and the fingerprint scanner should be COTS components while the SecSysController subsystem had to be developed. The marked packages of the Security System model on the right define the hand-off from systems engineering to the subsequent SecSysController development. It should be noted that it does not address the partitioning between hardware and software. The focus exclusively is on the required functionality and associated design constraints.

The following packages define the hand-off to the SecSysController development. It is important to note that these packages are handed-off as read-only (*Rhapsody: As Reference*):

1. The **SecSysControllerReqsPkg** contains the allocated functional and non-functional subsystem requirements.

2. The **SecSysControllerPkg** contains:
   - the definition of attributes, operations, ports, *logical* interfaces, and subsystem state-based behavior.
   - Additionally, the package **SecSysController_UcSD** contains copies of the use case scenarios confined to interactions of the SecSysController with its counterparts, i.e. subsystems and/or external actors.

3. The **SecuritySystemWB_AD** package contains the use case white-box activity diagrams associated with the SecSysController.

4. The **ActorsPkg** contains the external stimuli.

5. The **InterfacesPkg** contains the verified logical interface definitions of the system under design. Changes should be done in a controlled fashion – e.g. via configuration management.
## 6 Appendix

### A1 Mapping Security System Requirements to Model Artifacts

<table>
<thead>
<tr>
<th>ID</th>
<th>System Requirement</th>
<th>Req. Type</th>
<th>&lt;trace&gt;</th>
<th>&lt;satisfy&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS11-1</td>
<td>Two Independent Security Checks Secure areas shall be protected by two independent security checks.</td>
<td>Constraint</td>
<td>X</td>
<td>SecSysController</td>
</tr>
<tr>
<td>SS11-2</td>
<td>Employee ID Card Identification - Entry Entry shall be protected by a security check based upon employee ID.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>readSecurityCard(), validateSecurityCard()</td>
</tr>
<tr>
<td>SS11-3</td>
<td>Employee ID Card Identification - Exit Exit shall be protected by a security check based upon employee ID.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>readSecurityCard(), validateSecurityCard()</td>
</tr>
<tr>
<td>SS11-4</td>
<td>Biometric Scan Entry to the secure areas shall be protected by a second independent security check, based upon biometric data</td>
<td>Functional Requirement</td>
<td>X</td>
<td>scanBiometricData()</td>
</tr>
<tr>
<td>SS11-5</td>
<td>Time Between Two Independent Checks The time between the two independent security checks shall not exceed a configurable period.</td>
<td>Constraint</td>
<td>X</td>
<td>t_Bs</td>
</tr>
<tr>
<td>SS11-6</td>
<td>Configure Time Between Two Independent Checks The time between two independent security checks shall be configurable.</td>
<td>Functional Requirement</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SS11-7</td>
<td>Three Attempts On Employee ID Entry Upon entry the user shall be allowed three attempts on card identification.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>ScFailCount, flagSecurityCardFailure()</td>
</tr>
<tr>
<td>SS11-8</td>
<td>Three Attempts On Biometric Data Entry Upon entry the user shall be allowed three biometric data entries.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>BsFailCount, flagBiometricScanFailure()</td>
</tr>
<tr>
<td>SS11-9</td>
<td>Three Attempts On Employee ID Exit Upon exit the user shall be allowed three attempts on card identification.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>ScFailCount, flagSecurityCardFailure()</td>
</tr>
<tr>
<td>SS11-10</td>
<td>Denied Entry Notification Any denied access attempt shall be logged and account details sent to the administrator</td>
<td>Functional Requirement</td>
<td>X</td>
<td>logEntryData(), logAccountData(), reqProcessAlert()</td>
</tr>
<tr>
<td>SS11-11</td>
<td>Denied Exit Notification Any denied access attempt shall be logged and account details sent to the administrator</td>
<td>Functional Requirement</td>
<td>X</td>
<td>logExitData(), logAccountData(), reqProcessAlert()</td>
</tr>
</tbody>
</table>

Tab.A1-1 Mapping Security System Requirements to Model Artifacts
<table>
<thead>
<tr>
<th>ID</th>
<th>System Requirement</th>
<th>Req. Type</th>
<th>&lt;&lt;trace&gt;&gt;</th>
<th>&lt;&lt;satisfy&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS11-12</td>
<td>Alarm - Entry On a denied entry an alarm signal shall be raised.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>alarm()</td>
</tr>
<tr>
<td>SS11-13</td>
<td>Alarm - Exit  On a denied exit an alarm signal shall be raised.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>alarm()</td>
</tr>
<tr>
<td>SS11-14</td>
<td>Disabling User Account After three failed attempts at card identification or biometric data entry the user account shall be disabled</td>
<td>Functional Requirement</td>
<td>X</td>
<td>disableUserAccount()</td>
</tr>
<tr>
<td>SS11-15</td>
<td>Visualization of Security Card Check Status - Entry The user shall be visually informed about the status of his/her ID card check.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>displayCardStatus()</td>
</tr>
<tr>
<td>SS11-16</td>
<td>Visualization of Security Card Check Status - Exit The user shall be visually informed about the status of his/her ID card check.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>displayCardStatus()</td>
</tr>
<tr>
<td>SS11-17</td>
<td>Visualization of Biometric Data Check Status The user shall be visually informed about the status of his/her biometric data check</td>
<td>Functional Requirement</td>
<td>X</td>
<td>displayAuthenticationStatus()</td>
</tr>
<tr>
<td>SS11-18</td>
<td>Alarm Reset Once the alarm is acknowledged, it shall be reset by the Administrator</td>
<td>Functional Requirement</td>
<td>X X</td>
<td>resetAlarm()</td>
</tr>
<tr>
<td>SS11-1</td>
<td>Security Card Information Security cards shall only contain employee name and ID.</td>
<td>Constraint</td>
<td>X X</td>
<td>ValidateSecurityCard()</td>
</tr>
<tr>
<td>SS11-2</td>
<td>Access Precondition Access to the secure area shall only be allowed with a valid security card.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>validateSecurityCard()</td>
</tr>
<tr>
<td>SS11-4</td>
<td>Out of Date Cards - Entry Out of date cards shall deny entry.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>validateSecurityCard()</td>
</tr>
<tr>
<td>SS11-5</td>
<td>Out of Date Cards - Exit Out of date cards shall deny exit.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>validateSecurityCard()</td>
</tr>
<tr>
<td>SS11-6</td>
<td>Approval of Biometric Data The user shall not be allowed access unless his/her biometric data are recognized.</td>
<td>Functional Requirement</td>
<td>X</td>
<td>authenticateBiometricData()</td>
</tr>
<tr>
<td>SS11-7</td>
<td>Biometric Data Storage Biometric data shall be stored in the system database and not on the security card.</td>
<td>Constraint</td>
<td>X</td>
<td>FingerprintScanner</td>
</tr>
<tr>
<td>SS11-8</td>
<td>Biometric Scan Activation The biometric scan device shall be activated only when there is a request.</td>
<td>Functional Requirement</td>
<td></td>
<td>enableBiometricScan(); disableBiometricScan()</td>
</tr>
</tbody>
</table>

Tab.A1-2 Mapping Security System Requirements to Model Artifacts (cont’d)
<table>
<thead>
<tr>
<th>ID</th>
<th>System Requirement</th>
<th>Req. Type</th>
<th>&lt;&lt;trace&gt;&gt;</th>
<th>&lt;&lt;satisfy&gt;&gt;</th>
</tr>
</thead>
</table>
| SS12-1 | Processing User Request  
The system shall only process one user at a time. | Constraint      | X         | X           | SecSysController |
| SS12-2 | Entry Time  
The user shall be given sufficient time to enter the secure area.         | Constraint      | X         |             | t_Unlocked       |
| SS12-3 | Exit Time  
The user shall be given sufficient time to exit the secure area.            | Constraint      | X         |             | t_Unlocked       |
| SS12-4 | Configure Entry and Exit Time  
The time to enter and exit the area shall be customizable. | Functional Requirement | X        |             |               |
| SS12-5 | Automatic Securing the Secure Area - Entry  
Once the user has entered the area, the system shall automatically secure itself. | Functional Requirement | X        |             | evAccesspointLocked() |
| SS12-6 | Automatic Securing the Secure Area - Exit  
Once the user has exited the area, the system shall automatically secure itself. | Functional Requirement | X        |             | evAccesspointLocked() |
| MA1   | Image Capture  
An image shall be taken of any person, at the initial attempt, when trying to access a secure area for logging time and employee ID. | Functional Requirement | X        |             | reqTakeSnapshot() |
| MA2-1 | Time Recording  
The time a user spends in a secure area shall be recorded.                   | Functional Requirement | X        |             | logExitData()    |
| MA2-2 | Alarm Conditions  
An alarm shall notify if a person stays longer than 10 hours in the secure area. | Functional Requirement | X        |             | checkForTimeLimitViolations() |
| E1    | Emergency Exit  
In the event of an emergency the administrator can invoke a "Free Exit Mode". All security checks for exiting the area shall be disabled until the administrator returns the system to normal working. | Functional Requirement | X        |             |               |
| E2    | Security Lockdown  
The administrator can invoke a security lockdown mode - in this event the system shall lock all access points until the administrator returns the system to normal working. | Functional Requirement | X        |             |               |

**Tab.A1-3  Mapping Security System Requirements to Model Artifacts (cont'd)**
A2  Extended Architectural Model of the Security System

In the case study the focus of the system architecture model was exclusively on the required functionality. Design aspects were allocated to the respective subsystems as non-functional requirements. All interfaces of the architectural components were functional also referred to as logical. This paragraph shows, how to consider realization aspects in the system architecture model of the Security System.

Based on the results of the trade study it was decided that both the card readers and the fingerprint scanner should be COTS components. The chosen devices are shown in the picture below.

The table below specifies the control logic to be implemented in the SecSysController.

<table>
<thead>
<tr>
<th></th>
<th>LED1</th>
<th>LED2</th>
<th>Alarm</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Red</td>
<td>Red</td>
<td>CR_Entry, CR_Exit</td>
<td></td>
</tr>
<tr>
<td>Valid Card</td>
<td>Green</td>
<td>Yellow</td>
<td>only CR_Entry =&gt; ready for FingerprintScan</td>
<td></td>
</tr>
<tr>
<td>Pass</td>
<td>Green</td>
<td>Green</td>
<td>CR_Entry &amp;&amp; approved FingerprintScan, CR_Exit</td>
<td></td>
</tr>
<tr>
<td>Fail &lt;=3</td>
<td>Yellow</td>
<td>Red</td>
<td>CR_Entry, CR_Exit</td>
<td></td>
</tr>
<tr>
<td>Fail &gt;3</td>
<td>Red</td>
<td>Red</td>
<td>X CR_Entry, CR_Exit</td>
<td></td>
</tr>
</tbody>
</table>

Also the fingerprint scanner device displays the authorization result and the fingerprint status via two LEDs. But these LEDs are controlled by the device itself. The communication between the device and the SecsysController is via a bus. The table below specifies the control logic to be implemented in the FingerprintScanner.

<table>
<thead>
<tr>
<th></th>
<th>LED1</th>
<th>LED2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>Red</td>
<td>Red</td>
</tr>
<tr>
<td>Enabled</td>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td>Pass</td>
<td>Green</td>
<td>Green</td>
</tr>
<tr>
<td>Fail &lt;=3</td>
<td>Green</td>
<td>Yellow</td>
</tr>
</tbody>
</table>

From the system modeling point of view it was decided to model

- The bus communication between the devices and the SecSysController using SysML/UML Standard Ports.
- The control of the card readers’ LEDs and alarm device using SysML Flow Ports.

With regard to the card reader devices the following assumptions were made:

- The security card information is provided to the SecSysController via a bus communication
- Two LEDs display the result of the security card validation as well as the Security System status. The LEDs are connected to and controlled by the SecsysController.
- Also the alarm device within the card readers is connected to and controlled by the SecSysController.
A2.1 Extended System Architecture

Copy **IBD_SecuritySystem**, name the new Internal Block Diagram **IBD_SecuritySystem_Extended**.

Add (Atomic) Flowports to **itsSecSysController**, **CardReader_Entry**, and **CardReader.Exit**.

**NOTE**: Flowports that are not associated with a **CardReader** instance are **Removed from View** in the respective part.

In the **TypesPkg** define the type of the flowport attribute **LEDx xxxx** as Enumeration with the values **Red, Yellow, Green**.
A2.2 Extended State-based Behavior of Subsystems

1. In the CardReaderPkg right-click the block CardReader. Select Add New Statechart, and name it CardReaderCtrl_Extended.

2. Right-click the Statchart and select Set As Main Behavior.

3. In the FingerprintScannerPkg right-click the block FingerprintScanner. Select Add New Statechart and name it FingerprintScannerCtrl_Extended.

4. Right-click the Statchart and select Set As Main Behavior.
Extended State-based Behavior of the SecSysController Block

- ProcessingSecurityCardData_Extended
  - ValidatingSecurityCardData
    - [UserRequest="Entry"]
      - setLED1_Entry(Green);
      - setLED2_Entry(Yellow);
    - [UserRequest="Exit"]
      - setLED1_Exit(Red);
      - setLED2_Exit(Yellow);
  - [CardStatus="Valid"]
    - setLED1_Entry(Green);
    - setLED2_Entry(Yellow);
  - Fail
    - setLED1_Entry(Red);
    - setLED2_Entry(Red);
    - setAlarm_Entry(true);
    - setLED1_Exit(Red);
    - setLED2_Exit(Red);
    - setAlarm_Exit(true);
  - ScFailCount=0
    - BsFailCount=0
    - if (UserRequest="Entry")
      - setLED1_Entry(Yellow);
      - setLED2_Entry(Red);
    - else
      - setLED1_Exit(Yellow);
      - setLED2_Exit(Red);

- ProcessingBiometricData_Extended
  - flagBiometricScanFailure(BsFailCount);
  - waitForBiometricScanInfo
    - reqEnableBiometricScan to pFingerprintScanner
    - reqDisableBiometricScan to pFingerprintScanner
    - BfFailCount=0
  - tm(t_Bs)
    - BsTimeout
      - Failed3Times
      - Authenticated
        - setLED2_Entry(Green);
      - [else]
        - reqSetAuthenticationStatus
          - setLED2_Entry(Green);
  - BiometricScanFailure
    - flagBiometricScanFailure(BsFailCount);
  - reqDisableBiometricScan to pFingerprintScanner
  - reqAlarm to pCardReader_Entry

- UnlockingAndLockingAccessPoint
  - reqTakeSnapshot to pCamera
  - reqProcessAlert("Use r Acce ss Disa bled") to pAdmin

- ExitControl
  - reqValidateSecurityCard
    - [IS_PORT(pCardReader_Entry)]/
      - UserRequest="Entry";
    - [IS_PORT(pCardReader_Exit)]/
      - UserRequest="Exit";

- CardValid
  - Fail3Times
  - CardValid
  - Fail
  - /logEntryData();

- TimeLimitMonitor
  - /tm(t_Update)
  - CheckingForTimelimitViolations
    - /checkForTimeLimitViolations(TimeLimitFlag);
    - /TimeLimitFlag=0;
    - [TimeLimitFlag=1]
      - reqProcessAlert("TimeLimitViolation") to pAdmin

A2.3 Verification of the Extended System Architecture Model

The extended system architecture model is verified through model execution on the basis of the captured use case scenarios. The correctness and completeness analysis is based on the visual inspection of the model behavior (animated statecharts and sequence diagrams).

User Interfaces for the Model Execution

- **Panel Diagram**

- **Webify User Interface**
A3  **Modeling Guidelines**

This chapter specifies the guidelines and best practices to model a system using SysML. These guidelines are a symbiosis of many years of modeling experience in different industry branches (Aerospace, Defense, Automotive, Telecom, Medical, Industrial Automation, and Consumer Electronics) and have been proven to significantly enhance the readability of model-based specifications.

It starts with general guidelines and drawing conventions. SysML diagrams that are considered essential and associated elements then are discussed in detail. Finally, an approach which extends the SysML profile for project-specific needs is described.

A3.1 **General Guidelines and Drawing Conventions**

The following guidelines and drawing conventions are recommended for all diagrams:

- Create simple, focused diagrams with a small number of elements. As a rule of thumb, avoid placing more than ten major elements (block, use case, actor, etc.) on a diagram.

- Ensure all diagrams can be printed on standard 8.5x11 or A4 paper.

- Arrange elements in diagrams to avoid crossing of lines. All lines should be straight or rectilinear.

- Create elements with a consistent size. Avoid clutter and chaos by arranging elements with equidistant spacing and alignment.

- The default Rhapsody fonts, shapes, symbols, line styles, and colors shall be used consistently in all packages in the model.

- Position related elements close together in diagrams.

- Ensure elements in diagrams have the same level of abstraction.

- Organize diagrams in a hierarchical fashion. Locate diagrams in packages corresponding to their relative position in the system hierarchy.

- Ensure accurate and complete descriptions are entered for all model elements to assist in understanding the model and to facilitate the eventual hand-off of the model. These descriptions must also support the auto-generated documentation from the model.

- Avoid excessive use of description notes in diagrams. It’s generally recommended to put these descriptions in the description field of the corresponding graphical artifact.

- Do not use *comments* in the model.
A3.2 Use Case Diagram

Use Case Diagrams capture the functional requirements of a system by describing interactions between users of the system and the system itself. Users of a given system could be external people or other systems. A use case diagram is comprised of a system boundary that contains a set of use cases. Actors lie outside of the system boundary and are bound to use cases via associations.

Elements and Artifacts

Use Case: A use case defines the system context. Name use cases using verbs that describe their ultimate goal.

Actor: A role that an external user plays with respect to the system. Note that external users could be people or other systems. Use domain-specific, role-based names for actors.

System Boundary: Distinguishes the border between the actors and the system containing the use cases.

Association: Connects an actor with a use case, indicating which actors carry out which use cases.

Dependency: Connects two use cases, indicating which use cases depend on other use cases. For simplicity, only the <<include>> stereotype should be used for use case dependencies. Other stereotypes, like <<extend>>, should be avoided.

Guidelines and Drawing Conventions

• A system typically has many use cases. To manage this complexity, group use cases into Use Case Diagrams.

• Ensure each use case has a clear goal and that its functionality falls within the bounds of the system. Keep the goal broad enough to break the use case down into several scenarios (rule of thumb: $5 < n < 25$ “sunny day scenarios”).

• Every actor in a use case diagram must be associated with one or more use cases. Every use case must be directly associated with at least one actor.

Naming Conventions

• When multiple use case diagrams are defined, use case diagrams shall be numbered: UCD$<N_r>$ <Use Case Diagram Name>

• When multiple use case diagrams are defined, the name of a use case shall include the reference to its associated use case diagram: UCD$<N_r>\_UC<N_r>$ <Use Case Name>.

• Note: Use case names may have spaces

• The use case name shall start with a verb.
A3.3 Block Definition Diagram

The SysML Block Definition Diagram shows the basic structural elements (blocks) and their relationships / dependencies. Basic structural elements may be actors and subsystems or interfaces.

**Elements and Artifacts**

- **Block**: An entity that can contain data and behavior. A system block may be decomposed into sub-blocks. A system block is a reusable design element.

- **Actor**: A role that an external user plays with respect to the system. *Note*: This element is not shown in the *Rhapsody* toolbar. The actor needs to be defined in the browser (-> *ActorsPkg*) and then dragged into the block definition diagram.

- **Interface**: A contract comprised of event receptions and/or operations. In *Harmony for Systems Engineering* an interface only contains event receptions. Any system block that realizes the interface must fulfill that contract. An interface does not contain behavior.

- **Association**: Represents a bidirectional relationship between system blocks and actors.

- **Directed Association**: Represents a uni-directional relationship between system blocks and actors.

- **Directed Composition**: Shows the hierarchical decomposition of a system block into its sub-blocks.

- **Generalization**: Shows the relationship between a more general system block and a more specific system block. The more specific system block is fully consistent with the more general system block and contains additional information or behavior.

- **Dependency**: Shows the relationship between two system blocks in which one block requires the presence of another block.

**Guidelines and Drawing Conventions**

- Use the *Label* feature on the Display Options to keep block names simple within block definition diagrams, even when they are referencing blocks across packages.

- Blocks should not show attributes, operations and ports.

- Use the composition relationship to show block decomposition – do not show blocks inside other blocks.

- The stick figure should only be used to visualize actors that are external to the system. Actors that represent subsystems should be shown as (color-coded) blocks.

**Naming Conventions**

The name of a block definition diagram should have the pre-fix “BDD_”.
A3.4 Internal Block Diagram

The SysML Internal Block Diagram shows the realization of the system structure defined in the Block Definition Diagram. It is comprised of a set of nested parts (i.e. instances of the blocks) that are interconnected via ports and connectors.

Elements and Artifacts

- **StandardPort**: A named interaction point assigned to a block, through which instances of this block can exchange messages.

- **FlowPort**: specifies the input and output items that may flow between a block and its environment. Input and output items may include data as well as physical entities, such as fluids, solids, gases, and energy.

- **Connector**: A connection between two ports through which information flows via interfaces. When two parts share the same parent part, the connection between the two blocks is modeled with a single connector. However, when two parts have different parents, the connection between the two parts requires multiple connectors routed through delegation ports.
Guidelines and Drawing Conventions

- Show part decomposition by placing sub parts inside of their owning part.

- When possible, try to arrange parts in a vertical fashion. Also, try to place ports that communicate outside of the system tier on the left side of the block and ports that communicate within the system tier on the right side of the block.

- Use the Label feature on the Display Options to keep part names simple within internal block diagrams, even when they are referencing parts or system blocks across packages.

- Depending on the level of detail you are trying to convey in the diagram, you may hide or show attributes, and operations. All communication between parts occurs through ports and well defined interfaces.

- Depending on the level of detail you are trying to convey in a specific diagram, you may hide the pictograms of port interfaces (lollipop/socket) to avoid clutter.

- Avoid creating “gigantic” internal block diagrams that show all port connections between every part in the system, as these diagrams quickly become over-cluttered and unreadable. Instead create separate internal block diagrams with a mission focused on showing a specific collaboration or part decomposition.

Naming Conventions

- The name of an internal block diagram should have the pre-fix “IBD_”.

- Parts should keep the default name (its<BlockName>) created by Rhapsody. Only in use case models, the actor instance names should refer to the use case: (Ucd<Nr>_.) Uc<Nr>A_<ActorName>.

- Naming convention for ports: p<CounterpartName>

- Port names should be placed inside the associated part.

- Interface names should be referenced to the sender port. Naming convention: i<Sender>_<Receiver>
A3.5 Activity Diagram

The Activity Diagram describes a workflow, business process, or algorithm by decomposing the flow of execution into a set of actions and sub-activities. An activity diagram can be a simple linear sequence of actions or it can be a complex series of parallel actions with conditional branching and concurrency. Swim lanes can be added to the activity diagram to indicate the entities responsible for performing each activity.

NOTE: In Harmony for Systems Engineering the terms activity, action and operation are synonymous.

If an activity diagram is used to describe a workflow there are no limitations of using elements (Rhapsody mode: “Analysis”). If an activity diagram is used to describe a macro containing operations (Rhapsody mode: “Design”) there are limitations. The reason for the limitations is that SW-code for testing purposes is generated.

Elements and Artifacts

Action: An action represents a primitive operation. In Harmony for Systems Engineering also actions stereotyped <<MessageAction>> are used. These actions contain only messages to and/or from an actor.

Subactivity: A subactivity that is further decomposed into a set of actions and sub-activities.

NOTE: A decomposed subactivity cannot partitioned into swim lanes. If a decomposed sub activity needs to be partitioned into swim lanes (ref. architectural design) the parent action should be decomposed using a Call Behavior action (ref. Appendix A5).

Call Behavior: References to another activity diagram as an activity (ref. Appendix A5).

Control Flow: Actions are linked via control flow. Execution begins in an activity when a transition flows into it. A transition from an activity fires when the activity has completed and any guard conditions on the transition have been met.

Initial Flow: A control flow that leads to the initial action in the activity diagram.

Fork Node: A compound control flow that connects a single control flow to multiple concurrent activities.

Join Node: A compound control flow that merges the control flow from multiple concurrent activities.

Swim Lane Frame: Draws a frame around the entire set of activities so that they can be partitioned into swim lanes.

Swim Lane Divider: Places a vertical partition within the swim lane frame. Each swim lane represents an entity that is responsible for performing the activities in that swim lane. Control flows can cross swim lanes.

Decision Node: A condition connector splits a single control flow into multiple branches, each containing a guard. The guards on each branch should be orthogonal conditions, though they do not need to cover all possibilities. An “else” guard should be added to provide a default branch when no other guards are met.

Activity Final: Terminates the control flow of the activity diagram.

Diagram Connector: A diagram connector helps manage diagram complexity by allowing jumping to different sections of the activity diagram to avoid line crossing.

Action Pin: In SysML the Input/Output shows the input data of an action. In Harmony for Systems Engineering action pins - stereotyped <<ActorPin>> - are used to depict the link between an action and an actor. In this case the name of the pin has the name of the associated actor. The arrow in the pin shows the direction of the respective link (i.e. In, Out or In/Out).
Activity Diagram

Guidelines and Drawing Conventions

- In Harmony for Systems Engineering, the activity diagram is used exclusively to describe the functional flow through a use case. Therefore, select the activity diagram mode "Analysis".

- Document the pre-conditions in the respective tag of the diagram.

- Actor swim lanes should not be used. The link of an activity to the actor should be described through action pins, stereotyped <<ActorPIN>>.

- When performing a recursive decomposition of a complex system, the activities at one system tier can become the use cases in the next lower system tier.

- Activity diagrams should flow vertically from top to bottom. The initial activity should be located near the top of the diagram and any termination states should be located near the bottom of the diagram.

- Use the statechart action language to express guards to provide the best transition to statechart diagrams. See the appendix A5 for more details on Rhapsody’s action language.

- All control flow lines should be rectilinear or straight. Control flows should not cross each other or cross through activities.

- Diagram connectors should only be used when the readability of an activity diagram is disturbed by a direct control flow.

- Control flows and initial flows cannot have triggers.

- To reference another activity diagram as an activity, drag that activity diagram from the browser onto the diagram. This creates a call behavior activity that links to the external activity diagram (ref Appendix A5).

- Generally, an activity should correspond to an operation to be performed in the associated block. Exception: Activities stereotyped <<Message Action>> which describe the reception or transmission of a message, e.g.
Note: In the case of a message exchange with external actors respective actor pins need to be added to the message action, e.g.

- All activities should have only one exit transition. Any scenarios where multiple transitions flow out of an activity should be explicitly drawn using a condition connector or a fork node.

**Naming Conventions**

- The diagram shall have the associated use case name in plain text at the top of the diagram.
- Activity names shall start with a verb, beginning with a lower case letter, and map directly to the names of operations on system blocks.
A3.6 Sequence Diagram

Sequence Diagrams elaborate on requirements specified in use cases and activity diagrams by showing how actors and blocks collaborate in some behavior. A sequence diagram represents one or more scenarios through a use case.

A sequence diagram is comprised of vertical lifelines for the actors and blocks along with an ordered set of messages passed between these entities over a period of time.

Elements and Artifacts

- **Instance Line**: Draws a vertical lifeline for an actor or block.
- **Message**: Creates a horizontal message line between two lifelines or looped back onto the same lifeline. All messages between blocks are considered asynchronous. Reflexive (loop back) messages are considered synchronous operations and represent simple, private activities within the block.
- **Condition Mark**: Represents a mode/state change in a block. Can also be used to specify preconditions and post conditions for each instance on the sequence diagram.
- **Time Interval**: An annotation on a lifeline that identifies a time constraint between two points in the scenario.
- **Interaction Occurrence (Reference Sequence Diagram)**: Helps manage scenario complexity by cross-referencing other sequence diagrams.
- **Interaction Operator**: Helps to group related elements in a sequence diagram. This includes the option of defining specific conditions under which each group of elements will occur.
- **Operand Separator**: Used to create subgroups of interaction operators (e.g. concurrent operations or alternatives).
- **Partition Line**: Used to divide a scenario into sections of related messages. Each partition line has its own text field used to describe that section of the scenario.
- **Constraint**: A semantic condition or restriction expressed as text.
Guidelines and Drawing Conventions

- Pre- and post-conditions should be documented in condition marks on respective lifelines or in respective tags of the diagram.
- If possible, arrange lifelines such that the message exchange occurs in a “general” left-to-right flow from the top of the sequence down to the bottom. In other words, arrange the order of lifelines to minimize message zigzagging.
- For documentation reasons the print-out of a scenario should be captured on one page.
- Divide complicated scenarios into manageable, well-documented, logically related groups of messages using partition lines.
- Interaction Operators should not be nested deeper than 3 hierarchy levels.
- Extract reused portions of scenarios into separate sequence diagrams that are included using interaction occurrences.
- All message lines should be horizontal, rather than diagonal. Asynchronous messages between blocks have an open arrowhead and synchronous, reflexive messages have a filled arrowhead.
- Stereotype messages according to their associated protocol (e.g. M1553, Ethernet, etc …).
- Use the statechart diagram action language to express constraints to provide the best transition to statechart diagrams. See the appendix A6 for more details on Rhapsody’s action language.
- If a condition mark represents a mode/state change in reaction to a respective message, the condition mark should match the name of the state in the statechart diagram.
- Do not show operations on the actor lifelines.
- Do not use timeout in a sequence diagram. Rather describe a time constraint by means of Time Intervals.

Naming Conventions

The following table summarizes the recommended naming conventions for asynchronous messages:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req&lt;Service&gt;</td>
<td>Used to request a service (operation) on a block. These messages are followed by a reflexive message on the receiving block indicating the execution of the service. Example: reqReadSecurityCard The corresponding reflexive message name excludes the “req” prefix and begins with a lower case letter: Example: readSecurityCard</td>
</tr>
<tr>
<td>ret&lt;Service&gt;Status</td>
<td>Used to provide results of a service (operation) back to the requester. Example: retAuthenticateBiometricDataStatus</td>
</tr>
<tr>
<td>ev&lt;Event&gt;</td>
<td>Used to send a notification of change Example: evAccessPointLocked</td>
</tr>
</tbody>
</table>
A3.7 Statechart Diagram

A Statechart Diagram shows the state-based behavior of a block across many scenarios. It is comprised 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. See the appendix for more details on Rhapsody’s action language.

Classically, a statechart diagram depicts the behavior of reactive blocks—that is, blocks that maintain their history over time and react to events. However, when modeling a system, the behavior of blocks is always captured in statechart diagrams backed by supporting attributes and operations, as all communication between blocks occurs through ports using asynchronous events.

Elements and Artifacts

- **State**: A state typically models a period of time during the life of a block while it is performing an activity or waiting for some event to occur. States can also be used to model a set of related values in a block. A state that contains multiple sub states is called an “or” state or composite state. A state that contains two or more concurrent regions is called an “and” state or orthogonal state. Actions can be performed on state entry and exit.

- **Transition**: A transition from a state defines the response of a block in that state to an event. Transitions may flow through one or more connectors (defined below) and ultimately route to a new state or loop back to the original state. Transitions can have actions and guards that make them conditional.

- **Default Transition**: A transition that leads to the state (or the sub state in an “or” state or “and” state) that should be entered by default.

- **And Line**: Used to create an “and” state by dividing a state into multiple orthogonal, concurrent regions.

- **Fork Synch Bar**: A compound transition that connects a single transition to multiple orthogonal destinations.

- **Join Synch Bar**: A compound transition that merges transitions from different orthogonal states.

- **Condition Connector**: A condition connector splits a single transition into multiple branches, each with a guard. The guards on each branch should be orthogonal conditions, though they do not need to cover all possibilities. An “else” guard can be added to provide a default branch when no other guards are met.

- **History Connector**: A history connector is placed in an “or” state to remember its last active sub state. When the “or” state is re-entered, it automatically returns to that sub state. The transition coming out of the history connector is the default transition taken when there is no history.

- **Termination Connector**: A termination connector destroys the block.

- **Junction Connector**: A junction connector helps manage diagram complexity by combining several incoming transitions into a single outgoing transition.

- **Diagram Connector**: A diagram connector helps manage diagram complexity by allowing jumping to different sections of the statechart diagram to avoid line crossing.

- **EnterExit Point**: A connector that links transitions across statechart diagrams.

- **Send Action State**: Graphical representation of a send signal action.
### Guidelines and Drawing Conventions

- If possible, Statechart diagrams should flow vertically from top to bottom. The initial state should be located near the top of the diagram and any termination connectors should be located near the bottom of the diagram.

- Typically, all states should have at least one entry transition and at least one exit transition. A “dead end” state should be a very rare thing!

- Avoid nesting of states beyond 3 or 4 levels. Ensure complex nesting is simplified with sub state diagrams.

- All transition lines should be rectilinear or straight. Transitions should not cross each other or cross through states.

- Labels should be positioned on the left-hand side of the arrow direction.

- For readability reasons, use **Mealy** syntax (event [condition]/action on transition) wherever possible. Always place the action on a transition on a new line from the event and guard.

- **Moore** syntax (= action on entry, reaction in state) should be avoided unless necessary. This feature allows a block to react to events within a state without actually leaving that state via a transition. Exceptions to this rule include
  - protocol state machines for actors that respond to an input with a specific output,
  - message routing state machines that forward requests from one subsystem to another subsystem, and
  - actions in action states (ref Appendix A4).

  Never use “action on exit”.

- **Diagram Connectors** should only be used when the readability of a statechart diagram is disturbed by a direct transition.

- It is essential that the **EnterExit Points** connectors have meaningful names and the two charts that are connected can be shown side by side, with the connecting transition being easily identifiable. Using similar positions of the connector on each chart may facilitate this.
Naming Conventions

- State names should be verbs and indicate the current mode or condition of the block. Typically names are in the present tense. Names must be unique among sibling states and should never be the same as the name of a block or an event.
- Avoid names like "idle" or "wait".

A3.8 Profiles

A profile extends the UML/SysML with domain-specific tags and stereotypes. It also allows certain tool-specific properties to be overridden to support modeling in a specific domain. These customizations can be applied to the entire model or to specific model elements.

Exemplarily Tab.A3-2 shows the properties of a project-specific profile that supports the modeling guidelines outlined in the previous sections. Tab.A3-1 depicts the definition of element tags that was added to the profile in order to support the documentation.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Applicable To</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreCondition</td>
<td>Use Case</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Sequence Diagram</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Primitive Operation</td>
<td></td>
</tr>
<tr>
<td>PostCondition</td>
<td>Use Case</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Sequence Diagram</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Primitive Operation</td>
<td></td>
</tr>
<tr>
<td>Constraint</td>
<td>Use Case</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Sequence Diagram</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Primitive Operation</td>
<td></td>
</tr>
</tbody>
</table>

Tab.A3-1 Project-Specific Tags Defined in a Profile

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity_diagram&gt;Transition&gt;line_style</td>
<td>rectilinear_arrows</td>
</tr>
<tr>
<td>Activity_diagram&gt;DefaultTransition&gt;line_style</td>
<td>straight_arrows</td>
</tr>
<tr>
<td>Statechart&gt;Transition&gt;line_style</td>
<td>rectilinear_arrows</td>
</tr>
<tr>
<td>Statechart&gt;DefaultTransition&gt;line_style</td>
<td>straight_arrows</td>
</tr>
<tr>
<td>Statechart&gt;CompState&gt;ShowCompName</td>
<td>false</td>
</tr>
<tr>
<td>SequenceDiagram&gt;General&gt;HorizontaMessageType</td>
<td>Event</td>
</tr>
<tr>
<td>SequenceDiagram&gt;General&gt;SelfMessageType</td>
<td>PrimitiveOperation</td>
</tr>
<tr>
<td>SequenceDiagram&gt;General&gt;ShowwAnimStateMark</td>
<td>false</td>
</tr>
<tr>
<td>ObjectModelGe&gt;Actor&gt;ShowName</td>
<td>Name_only</td>
</tr>
<tr>
<td>ObjectModelGe&gt;Class&gt;ShowName</td>
<td>Name_only</td>
</tr>
<tr>
<td>ObjectModelGe&gt;Object&gt;ShowName</td>
<td>Name_only</td>
</tr>
<tr>
<td>ObjectModelGe&gt;Inheritance&gt;line_style</td>
<td>rectilinear_arrows</td>
</tr>
<tr>
<td>ObjectModelGe&gt;Depends&gt;line_style</td>
<td>rectilinear_arrows</td>
</tr>
<tr>
<td>ObjectModelGe&gt;Class&gt;ShowPorts</td>
<td>false</td>
</tr>
<tr>
<td>ObjectModelGe&gt;Class&gt;ShowPortsInterfaces</td>
<td>false</td>
</tr>
<tr>
<td>UseCaseGe&gt;Actor&gt;ShowName</td>
<td>Name_only</td>
</tr>
<tr>
<td>UseCaseGe&gt;UseCase&gt;ShowName</td>
<td>Name_only</td>
</tr>
</tbody>
</table>
**A4  Guideline: Deriving a Statechart Diagram**

This guideline describes how to derive state-based behavior from the information captured in an activity diagram and associated sequence diagrams. The steps are detailed using as example the use case Uc2_ControlExit.

Fig.A4-1 shows the activity diagram Uc2_ControlExitBlackBoxView of the use case Uc2_Control Exit. It describes the functional flow of the use case by decomposing the flow of execution into a set of action nodes joined by transitions and condition connectors.

Fig.A4-2 and Fig.A4-3 show the black-box sequence diagrams that were generated from the black-box activity diagram by means of the Rhapsody SE Toolkit feature Create New Scenario From Activity Diagram. The information from the activity diagram and its associated sequence diagrams will be used to identify and capture the state-based system behavior in a statechart diagram.
Step 1: Identify Wait States and Action States

**Step 1.1: Identify Wait States**

In a Wait State an object waits for an event to happen. It consumes time while waiting for the event.

In the use case black-box activity diagram identify actions with IN actor pins. In the use case black-box sequence diagrams (Fig.A4-2, Fig.A4-3) identify the messages (receptions) that trigger the selected actions. For each of the identified actions create in the statechart diagram a wait state named `WaitFor<ReceptionName>`. In cases where the use case black-box sequence diagram shows a timeout event (Fig.A4-2: `t_Unlocked`), create in the statechart diagram a wait state with a name that describes the actual system status (Fig.A4-4: `AccessPointUnlocked`).

**Fig.A4-4 Wait States of Uc2_ControlExit**

**Step 1.2: Identify Action States**

An action state is a state whose purpose is to execute an entry action, after which it takes a completion transition to another state. It is a kind of dummy state that is useful for organizing state machines into logical structures.

In the use case black-box activity diagram identify actions with multiple outgoing completions with guard conditions. For each of these actions create in the statechart diagram an action state with the name of the action (naming convention: `<ActionName>ing`) and allocate the relevant action to it using MOORE syntax.

**Fig.A4-5 Action States of Uc2 ControlExit**

**NOTE:** Besides the output-relevant action, an action-state may also have additional context-related actions allocated to it (Fig.A4-5: action state `ValidatingSecurityCard`).
Step 2: Connect States

Step 2.1: Identify the initial state

Mark the initial state with a Default Connector. If attributes need to be initialized (e.g. ScFailCount in Fig.A4-6), add respective actions to the default connector.

Step 2.2: Identify transitions, triggering events, and associated actions

The transitions between the states and associated triggering events – including guarded condition(s) - are identified through analysis of the captured black-box use case sequence diagrams.

Select a use case scenario. Replicate the scenario in the statechart diagram as follows:
Start from the initial state. In the sequence diagram identify the event and – if needed – guarded condition(s) that trigger a transition and the associated action(s). In the statechart diagram identify the target state. Connect the two states. Label the transition following MEALEY syntax: Event [Condition] / Action. If the target state is an action state, add to the transition label only those actions that are not allocated to the state. Proceed in the sequence diagram and repeat the outlined connecting steps in the statechart diagram.

Repeat the replication of scenarios in the statechart for all captured use case scenarios.

Step 2.3: Execute the Statechart

Verify the correctness of the captured state-based behavior through model execution using the black-box use case scenarios as the basis for respective stimuli.
Step 3: Structure the Statechart hierarchically

Step 3.1: Identify state hierarchies

Once the flat statechart is verified, look for ways to simplify it by structuring it hierarchically. Identify states that can be aggregated. Grouping criteria could be e.g.

- System modes
- System phases or
- Reuse of state patterns

Also look for situations where the aggregation of state transitions simplifies the statechart. Inspection of the flat statechart reveals that

- ValidatingSecurityCard,
- FlagingSecurityCardFailure, and
- WaitFor_reqReadSecurityCard in the case of a card failure

can be considered sub-states of a composite state called ProcessingSecurityCard (Fig.A4-7). As ScFailCount is a local attribute, its initialization is added to the default entry of the composite state. Furthermore, the substates FlagingSecurityCardFailure and WaitFor_reqReadSecurityCard can be aggregated in the composite state ValidationFail, thus denoting the fail mode within the ProcessingSecurityCard state.

Note the different transitions out of the composite state. In the case of CardStatus=="Pass" the triggering condition and associated action is captured in the top-level statechart (Fig.A4-8) as a high-level interrupt. In the case of a third-time failure, the respective triggering condition and associated action is captured within the ProcessingSecurityCard state and linked to the top-level statechart via an EnterExit point (Fail3Times).

States in the flat statechart Fig.A4-6 that relate to the access point control can be aggregated into the composite state UnlockingAndLockingAccessPoint, as shown in Fig.A4-8. This state includes the messages sent to the access point. Furthermore, the states WaitFor_evAccessPointUnlocked and WaitFor_evAccessPointLocked can be merged to one wait state called WaitForAccessPointFeedback. The exit out of the composite state is captured in the top-level statechart.
Fig.A4-9 shows the final structure of the top-level statechart of the use case Uc2ControlExit.

**Step 3.2: Execute the Statechart**

Verify the correctness of the captured state-based behavior through model execution using the black-box use case scenarios as the basis for respective stimuli.
A5  Guideline: Logical Decomposition of a Use Case Black-Box Activity Diagram

An important artifact in the Harmony for Systems Engineering workflow is the activity diagram associated with a use case. In the Functional Analysis phase the use case black-box activity diagram describes the functional flow (storyboard) of a use case. It groups functional requirements in actions – in Harmony for Systems Engineering the equivalent of operations - and shows the associated flow of information within the system and between the system and its environment (actors). In the architectural design phase the use case white-box activity diagram supports the allocation of actions/operations to an architectural structure.

A system-level use case and the associated black-box activity diagram describe the required functionality at a high level of abstraction – in this section referred to as logical decomposition level 0. In order to provide the relevant information for the implementation, actions/operations identified at this level may need to be detailed further. Experiences show, that if a level 0 action/operation needs to be logically decomposed, the decomposition seldom exceeds level 2.

This section describes the workflow, associated work products, and provides guidelines that should be followed when activities/operations of a use case black-box activity diagram are logically decomposed. The different steps are outlined using the industrial automation example shown in Section 2.2.2.

NOTE: For didactical reasons I did not show hierarchy in Fig.2-5, the use case black-box activity diagram.

The chosen system is a complex multi-axis manufacturing machine. Key components of this machine are 10 movable axes with partly overlapping working envelopes. For the case study the use case covering the Homing And Manual Mode was chosen (Fig.A5-1).

Fig.A5-1 Use Case Diagram UCD_MultiAxesSystem

Fig.A5-2 shows the functional and state-based behavior of the use case at the top-level – also referred to as decomposition level 1. The sequence diagram in Fig.A5-2 describes the use case scenario of a manual movement of AxisB. The associated operation will be decomposed in the following paragraphs.

The logical decomposition is performed in 8 steps.

Step 1: Identify operations that need to be decomposed

A prerequisite for this step is, that the use case block behavior at the chosen system decomposition level was verified through model execution.

Step 2: Rank the identified operations

In order to achieve a common level of decomposition, the identified operations need to be ranked according to their importance. Each of the selected operations then will be decomposed by only one level before the next level of decomposition may be considered.

The next steps are iteratively repeated for each selected operation.
Fig.A5-2 Use Case HomingAndManualMode: Functional and State-based Behavior at the System Level (Level 0)
Step 3: Decompose the selected operation

The workflow is shown for the operation moveAxisB in Fig.A5-2.

**Step 3.1: Create a Reference Activity Diagram**

The decomposition of an operation is described in a new Activity Diagram – in the following referred to as Reference Activity Diagram. Do not use the Rhapsody feature Subactivity. The reason for this is, that a subactivity chart cannot be partitioned into swim lanes. The capability to allocate the decomposed operations to swimlanes is crucial in the architectural design phase.

1. Right-click Uc1HomingAndManualBlackBoxView > Add New > Diagrams > Activity Diagram

2. Name the original Activity Diagram Level0 and name the new Activity Diagram Level1_moveAxisB

**Step 3.2: Define the Functional Flow at the Decomposed Level**

In Fig.A5-3 the activity diagram Level1_moveAxisB captures the check for possible axis envelope overlaps as well as direction dependent and axis position dependent (homed/not homed) speed limitations of the manual movement.

**Step 3.3: Derive Scenarios from the Reference Activity Diagram**

Capture the essential scenarios by means of the SE-Toolkit feature Create New Scenario From Activity Diagram. Check the consistency between the actions in the Reference Activity Diagram and the operations in the derived scenarios by means of the SE-Toolkit feature Perform Activity View Consistency Check.

---

**Fig.A5-3 Reference Activity Diagram Level1_moveAxisB and Associated Derived Scenarios**
Appendix

Step 4: Link the Reference Activity Diagram to the parent activity diagram

1. Move the Reference Activity Diagram Level1_moveAxisB into the parent activity diagram.

   **NOTE:** Rhapsody creates in the parent activity diagram a Call Behavior Activity with the name of the reference activity diagram.

2. Replace the action that was decomposed by the call behavior activity.

   **NOTE:** Do not add actor pins to the activity as the interactions with the actors is captured in the lower hierarchy.

Verify the link by creating an extended black-box use case scenario by means of the SE-Toolkit feature **Create New Scenario from Activity Diagram**.

**Fig.A5-4 Linking the Reference Activity Diagram to the Parent Activity Diagram**

**Fig.A5-5 Automatically generated Extended Uc1Sc1 Sequence Diagram**

**NOTE:** There is no need to manually complete the service request messages in the automatically generated sequence diagram. For the verification of the link only the correct sequence of operations is relevant.
Step 5: Link RefAD scenarios to parent black-box use case scenarios

In the use case black-box scenarios replace the operation that was decomposed and its associated service request message by an Interaction Operator that references to the RefAD scenarios. Name it 
\(<DecomposedOperationName>_\_Scenarios\).

The interaction operator contains the links to the scenarios that were derived from the Reference Activity Diagram.

Step 6: Update port(s) and interface(s) of the use case block

Update the use case block port(s) and interface(s) by means of the SE-Toolkit feature Create Ports And Interfaces.

Step 7: Extend the state-based behavior of the use case block

There are two ways to proceed:

- Alternative 1: Modify/extend the existing use case block statechart.
- Alternative 2: Create a copy of the existing use case block statechart and modify/extend the copy.

It is recommended to create a new use case block statechart and make it Main Behavior as shown below.

Fig.A5-7 shows the extended statechart of the use case block. Note the EnterExit point (UnsafePos) that was added to the state movingAxisB in the top-level statechart. It captures the exit from the substate in case of an anticipated envelope conflict with other axes.

Step 8: Execute the use case model

Verify the correctness of the extended use case block through use case model execution (Fig.A5-8).
Appendix

Figure A5.7  Extended State-Based Behavior of the Use Case Block

Figure A5.8  Animated Sequence Diagram (Extended Uc1Sc1)
**Outlook: Allocation of the decomposed operations in the Architectural Design phase**

In the architectural design phase the level 1 use case black-box activity diagram and its associated Reference Activity Diagram(s) are partitioned into swim lanes that represent the components of the architectural structure. Based on the chosen design concept, the operations then are “moved” into respective swim lanes. Fig.A5-9 shows the allocation of the constituents of the decomposed operation moveAxisB.

![Allocation of RefAD_MoveAxisB Operations to Subsystems in the Architectural Design Phase](image-url)
A6  \textit{Rhapsody Action Language}\footnote{This section provides a brief introduction to the action language applied in the \textit{Rhapsody} tool.}

Basic Syntax

The language is case sensitive. That is, “evmove” is different from “evMove”. Each statement must end with a semi-colon. All names must start with a letter and cannot contain spaces. Special characters are not permitted in names, except for underscores (_). However, a name should never start with an underscore. The following words are reserved and should not be used for names: \texttt{asm, auto, break, case, catch, char, class, const, continue, default, delete, do, double, else, enum, extern, float, for, friend, GEN, goto, id, if, inline, int, IS\_IN, IS\_PORT, long, new, operator, OPORT, OUT\_PORT, params, private, protected, public, register, return, short, signed, sizeof, static, struct, switch, template, this, throw, try, typedef, union, unsigned, virtual, void, volatile, while.}

Assignment and Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{X=1}</td>
<td>(Sets X equal to 1)</td>
</tr>
<tr>
<td>\texttt{X=Y}</td>
<td>(Sets X equal to Y)</td>
</tr>
<tr>
<td>\texttt{X=X+5}</td>
<td>(Adds 5 to X)</td>
</tr>
<tr>
<td>\texttt{X=X-3}</td>
<td>(Subtracts 3 from X)</td>
</tr>
<tr>
<td>\texttt{X=X*4}</td>
<td>(Multiplies X by 4)</td>
</tr>
<tr>
<td>\texttt{X=X/2}</td>
<td>(Divides X by 2)</td>
</tr>
<tr>
<td>\texttt{X=X%5}</td>
<td>(Sets X to the remainder of X divided by 5)</td>
</tr>
<tr>
<td>\texttt{X++}</td>
<td>(Increments X by 1)</td>
</tr>
<tr>
<td>\texttt{X--}</td>
<td>(Decrements X by 1)</td>
</tr>
</tbody>
</table>

Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{X==5}</td>
<td>(X equal to 5)</td>
</tr>
<tr>
<td>\texttt{X!=5}</td>
<td>(X not equal to 5)</td>
</tr>
<tr>
<td>\texttt{X&lt;3}</td>
<td>(X less than 3)</td>
</tr>
<tr>
<td>\texttt{X&lt;=3}</td>
<td>(X less than or equal to 3)</td>
</tr>
<tr>
<td>\texttt{X&gt;4}</td>
<td>(X greater than 4)</td>
</tr>
<tr>
<td>\texttt{X&gt;=4}</td>
<td>(X greater than or equal to 4)</td>
</tr>
<tr>
<td>\texttt{X&gt;2 &amp;&amp; X&lt;7}</td>
<td>(X greater than 2 \texttt{and} X less than 7)</td>
</tr>
<tr>
<td>\texttt{X&lt;2 | X==7}</td>
<td>(X less than 2 \texttt{or} X equal to 7)</td>
</tr>
</tbody>
</table>

Printing

The “cout” operator prints to the screen. Elements to be printed are separated by the “\texttt{<<}” operator. Text strings are surrounded by double quotes. Attributes are referenced using their names. The “\texttt{endl}” operator prints a carriage return. So, to print out the current value of \texttt{X}, use the following command:

\begin{verbatim}
cout << “The value of X is “ << X << endl;
\end{verbatim}

If the current value of \texttt{X} is 5, this statement prints the following message on the screen:

\begin{verbatim}
The value of X is 5
\end{verbatim}
Conditional Statements

Conditional statements begin with the keyword “if” followed by a conditional expression in parenthesis, followed by the statement to execute if the condition evaluates to true. You can optionally add the “else” keyword to execute a statement if the condition evaluates to false. The “else” clause can contain another nested “if” statement as well. For example:

```cpp
if (X<=10)
    X++; else
    X=0;
```

Multiple statements can be grouped together by placing them in curly braces.

```cpp
if (X<=10)
    { X++;     cout << "The value of X is " << X << endl; }
else
    { X=0;     cout << "Finished" << endl; }
```

Incremental Looping Statements

Incremental looping is accomplished using the “for” statement. It holds three sections separated by semicolons to specify: 1) an initialization statement, 2) a conditional expression, and 3) an increment statement. For example, to iteratively set the value of X from 0 to 10 while printing out its value:

```cpp
for (X=0; X<=10; X++)
    cout << X << endl;
```

Conditional Looping Statements

The “while” statement is used for conditional looping. This statement has a single conditional expression and iterates so long as it evaluates to true. The previous example could be implemented using a “while” statement as follows:

```cpp
X=0;
while(X<=10)
    { cout << X << endl;     X++; }
```

Invoking Operations

To invoke an operation on a block, use the operation name followed by parenthesis. For example, to invoke the “go” operation:

```cpp
go();
```

If an operation takes parameters, place them in a comma-separated list. For example, to invoke the “min” operation with two parameters:

```cpp
min(X,Y);
```

Generating Events

The “OUT_PORT” and “GEN” keywords are used to generate events through ports. For example, to send an event named “evStart” out the port named “p2”, issue the following statement:

```cpp
OUT_PORT(p2)->GEN(evStart);
```

To generate an event with parameters, place them into a comma-separated list. For example, to generate an event named “evMove” with two parameters for velocity and direction:

```cpp
OUT_PORT(p2)->GEN(evMove(10,2));
```

NOTE: The “OPORT” keyword can be used in place of “OUT_PORT”.

```cpp
```
Appendix

Referring to Event Parameters in Transitions

The "params" keyword followed by the "->" operator is used to reference the parameters of the event that caused the current transition. For example, if an event named "evMove" has a parameter named "velocity", that parameter can be referenced using "params->velocity". This syntax can also be embedded in statements within the action on the transition. For example:

```c
if (params->velocity <= 5)
```

Testing the Port on which an Event Arrives

The "IS_PORT" keyword is used to test whether the event that caused the current transition arrived through a specific port. For example:

```c
if (IS_PORT(p2))...
```

Testing the State of a State Machine

The "IS_IN" keyword is used to test whether a state machine is in a specific state. For example, to test whether the state machine of a block is in a state called "Accelerating":

```c
if (IS_IN(Accelerating))
```
## A7 Rhapsody SE-Toolkit (Overview)

<table>
<thead>
<tr>
<th>SE-Toolkit Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Add Hyperlink(s)</td>
<td>Adds a hyperlink from the source(s) to the destination(s).</td>
</tr>
<tr>
<td>1.2 Add Anchor(s)</td>
<td>Adds an anchor from the source(s) to the destination(s)</td>
</tr>
<tr>
<td>1.3 Add SD Ref(s)</td>
<td>Adds selected sequence diagram(s) as Referenced Sequences to the use case.</td>
</tr>
<tr>
<td>1.4 Add Event Reception(s)</td>
<td>Adds receptions of the chosen events to the target interface.</td>
</tr>
<tr>
<td>1.5 Add Event Reception(s)</td>
<td>Adds receptions of the chosen events to the target interface.</td>
</tr>
<tr>
<td>1.6 Merge Blocks</td>
<td>Copies any operations, receptions, and attributes from the source blocks to a single destination block.</td>
</tr>
<tr>
<td>1.7 Create Dependency</td>
<td>Creates dependencies between model elements.</td>
</tr>
<tr>
<td>1.8 Populate Activity Diagram</td>
<td>For each reflexive message on the selected sequence(s) an action is created on the selected activity diagram</td>
</tr>
<tr>
<td>1.9 Allocate Operations from Swimlanes</td>
<td>Copies operations allocated to a swimlane in a White-Box Activity Diagram into the relevant sub-system block.</td>
</tr>
<tr>
<td>1.10 Create New Scenario from Activity Diagram</td>
<td>Creates a sequence diagram from selected actions in an activity diagram. If the source is a single action then the user will be asked to choose a path each time a condition connector is encountered</td>
</tr>
</tbody>
</table>

*Tab.A7-1 Rhapsody SE-Toolkit Features*
<table>
<thead>
<tr>
<th>SE-Toolkit Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2  Create Harmony Project</td>
<td>Creates a Harmony for Systems Engineering compliant project structure</td>
</tr>
<tr>
<td>3  Create System Model from Use Case</td>
<td>Creates a Harmony for Systems Engineering compliant package structure for the use case model</td>
</tr>
<tr>
<td>4  Auto-Rename Actions</td>
<td>Harmonizes the action statement and action name in an activity diagram.</td>
</tr>
<tr>
<td>5  Add Actor Pins</td>
<td>Adds SysML action pins stereotyped &lt;&lt;ActorPin&gt;&gt; to the selected action on an activity diagram. User selects the direction and the actor from a drop down list.</td>
</tr>
<tr>
<td>6  Perform Activity View Consistency Check</td>
<td>Checks the consistency between actions of the black-box activity diagram and the operations in the derived use case scenarios.</td>
</tr>
<tr>
<td>7  Create Ports and Interfaces</td>
<td>Createsbehavioral ports and associated interfaces based on scenarios captured in sequence diagrams</td>
</tr>
<tr>
<td>8  Connect Ports</td>
<td>Creates links between ports on an internal block diagram</td>
</tr>
<tr>
<td>9  Create Initial Statechart</td>
<td>Creates wait state(s) and action states based on the information captured in an Activity Diagram .</td>
</tr>
<tr>
<td>10 Merge Functional Analysis</td>
<td>Copies all operations, event receptions and attributes from all use case blocks into the selected block</td>
</tr>
<tr>
<td>11 Duplicate Activity View</td>
<td>Makes a copy of an activity view and strips away any referenced scenarios</td>
</tr>
<tr>
<td>12 Create Sub Packages</td>
<td>Creates a package per subsystem and moves subsystem blocks into those packages.</td>
</tr>
<tr>
<td>13 Architectural Design Wizard</td>
<td>Copies operations from one architectural layer to another and tracks when operations have been allocated.</td>
</tr>
<tr>
<td>14 Perform Swimlane Consistency Check</td>
<td>Checks consistency between the allocated actions in swimlanes against the allocated operations in subsystem blocks.</td>
</tr>
<tr>
<td>15 Create Allocation Table</td>
<td>Summarizes the allocation of operations of a white-box activity diagram in an Excel spreadsheet.</td>
</tr>
<tr>
<td>16 Create Allocation CSV File</td>
<td>As ‘Create Allocation Table’ – except in a CSV form. Added to the model as a controlled file.</td>
</tr>
<tr>
<td>17 Generate N2 Matrix</td>
<td>Creates an Excel spreadsheet of the provided and required interface matrix from an internal block diagram</td>
</tr>
<tr>
<td>18 Copy MoEs to Children</td>
<td>Copies the MoE attributes of the key function block into the solution blocks.</td>
</tr>
<tr>
<td>19 Copy MoEs from Base</td>
<td>Copies the MoE attributes of the key function block into a selected solution block.</td>
</tr>
<tr>
<td>20 Perform Trade Analysis</td>
<td>Calculates for a set of solutions a Weighted Objectives Table and displays the results in an Excel spreadsheet.</td>
</tr>
</tbody>
</table>

Tab.A7-2 Rhapsody SE-Toolkit Features cont’d
Harmony/SE Workflow and its Support through the Rhapsody Toolkit

**Requirements Analysis**

1. Import Stakeholder ReqSpec
2. Generate System Requirements
3. Define System-Level Use Cases
4. Define Use Case Model Context

**System Functional Analysis**

1. Link Sys Reqs to Stakeholder Reqs with satisfy Dependency
2. Link Use Case to Reqs with trace Dependency
3. Define UC Functional Flow
4. Define UC Scenarios
5. Define Ports and Interfaces
6. Derive UC Statebased Behavior
7. Verify / Validate UC Model
8. Link UC Block Properties to Reqs with trace Dependency
9. Derive White-Box Sequence Diagrams
10. Merge Propertyed of UC Blocks in SuD Block
11. Decompose System Block into Parts
12. Graphically allocate Operations in UC White-Box Activity Diagram
13. Formalize Allocation by copying Operations into Parts
14. Allocate Non-functional Reqs and define Traceability Links
15. Derive White-Box Sequence Diagrams

**Architectural Design**

1. Merge Propertyed of UC Blocks in SuD Block
2. Decompose System Block into Parts
3. Graphically allocate Operations in UC White-Box Activity Diagram
4. Formalize Allocation by copying Operations into Parts
5. Allocate Non-functional Reqs and define Traceability Links
6. Derive White-Box Sequence Diagrams
7. Define Ports and Interfaces
8. Define State-Based Behavior of Blocks
9. Verify System Architecture Model
10. Merge Propertyed of UC Blocks in SuD Block
11. Decompose System Block into Parts
12. Graphically allocate Operations in UC White-Box Activity Diagram
13. Formalize Allocation by copying Operations into Parts
14. Allocate Non-functional Reqs and define Traceability Links
15. Derive White-Box Sequence Diagrams
16. Define Ports and Interfaces
17. Define State-Based Behavior of Blocks
18. Verify System Architecture Model
7 References

[1] SysML 1.0 Specification  
http://www.omgsysml.org

Hans-Peter Hoffmann, CrossTalk, Vol. 18 No. 11, November 2005

Hans-Peter Hoffmann, Proceedings of INCOSE 2006 16th International Symposium


Bruce Douglass, I-Logix/Telelogic Whitepaper 2006

Bruce Douglass, I-Logix/Telelogic Whitepaper 2006

Nigel Cross, Wiley, 1989