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

Deskbook Release 4.1

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

© 2011 IBM Corporation

The file "Deskbook Rel 4.0.pdf" is the latest version of the “Systems Engineering Best Practices with the Rational Solution for Systems and Software Engineering Deskbook Release 4.0” (“Deskbook”), released July 2013.

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 Rel.4.0 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.

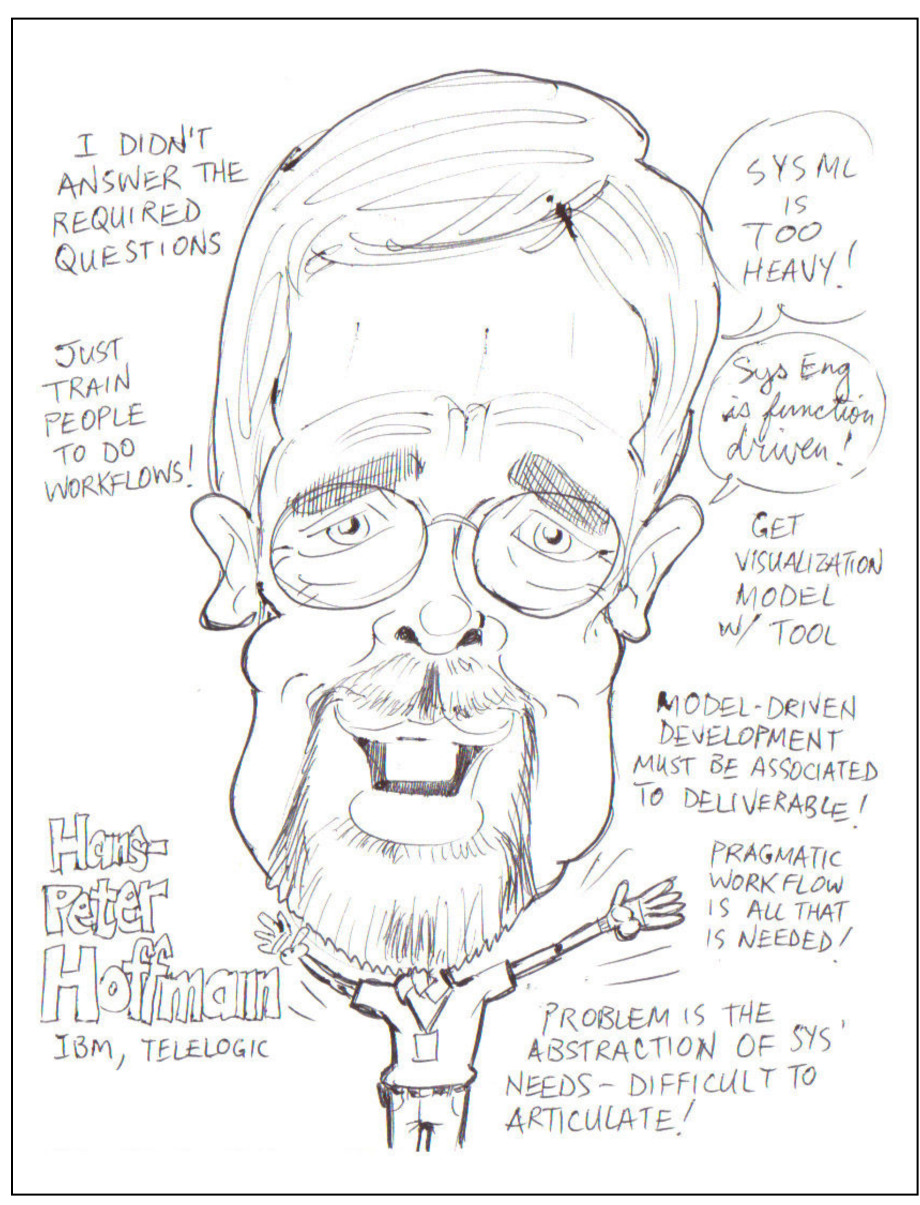
Licensed Materials - Property of IBM Corporation
U.S. Government Users Restricted Rights: Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

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**



The Author - by *J. Rick White*

Foreword to the Deskbook Release 4.1

Here it is – the next iteration. Two chapters were added to the Appendix, extending the scope of the previous release. It now addresses the model-based system design approach in case of a change request to a legacy system and a model-based testing approach for the verification of hand-off models by means of the *Rhapsody* tool add-ons *TestConductor* (TC) and *Automatic Test Generation* (ATG).
The testing chapter is a contribution of *Dr. Udo Brockmeyer* (BTC Embedded Systems AG, Germany) and his team. Thank you all.

Boston , February 2014

Foreword to the Deskbook Release 4.0

The systems engineering process is iterative. There is no reason why this should not be applicable also to the Deskbook.

This release outlines a new approach – the *Use Case Realization Approach*. Experiences in several complex applications show that this approach significantly streamlines the development of an Integrated System Architecture. Also, the collaboration between the tools *Rhapsody* and *DOORS* via the *Rhapsody Gateway* tool is addressed in more detail.

Since I first introduced the Deskbook over seven years ago, the Deskbook has been used by customers all over the world. Besides the English release there is also a Japanese and a Chinese translation available. I want to thank *Chiori Asada* and her team in Japan for their effort translating the Release 3.1 into Japanese. For China, *Lian Gu* personally translated the Release 4.0 into Chinese. This release will be available in China as an IBM booklet July 2013. I also want to express my appreciation to Lian for her translation.

I also want to thank two colleagues who deserve special mention with regard to their contributions to this release: *Andy Lapping* and *Pavel Vodov*. Andy – the “Wizard Guru” – is the author of the *Rhapsody SE-Toolkit*. Pavel detailed the collaboration between the tools *Rhapsody* and *DOORS*. Working with them has been a distinct pleasure for me.

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

Boston, June 20, 2013

© Copyright IBM Corporation 2006. 2010. All Rights Reserved.

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 10
2.2.3.2 Architectural Design 13
2.2.4 Systems Engineering Hand-Off 17
2.3 ESSENTIAL SYSML ARTIFACTS OF MODEL-BASED SYSTEMS ENGINEERING 18
2.3.1 Requirements Diagram 19
2.3.2 Structure Diagrams 19
2.3.2.1 Block Definition Diagram 19
2.3.2.2 Internal Block Diagram 19
2.3.2.3 Parametric Diagram 21
2.3.3 Behavior Diagrams 21
2.3.3.1 Use Case Diagram 22
2.3.3.2 Activity Diagram 22
2.3.3.3 Sequence Diagram 23
2.3.3.4 Statechart Diagram 23
2.3.4 Artifact Relationships at the Requirements Analysis / System Functional Analysis Level 24
2.4 SERVICE REQUEST-DRIVEN MODELING APPROACH 25

3 *RHAPSODY* PROJECT STRUCTURE 26

3.1 PROJECT STRUCTURE OVERVIEW 26
3.2 REQUIREMENTS ANALYSIS PACKAGE 27
3.3 FUNCTIONAL ANALYSIS PACKAGE 28
3.4 DESIGN SYNTHESIS PACKAGE 29
3.4.1 Architectural Analysis Package 29
3.4.2 Architectural Design Package 30
3.5 SYSTEM-LEVEL DEFINITIONS 31

4 CASE STUDY: SECURITY SYSTEM 32

4.1 CASE STUDY WORKFLOW 32
4.2 CREATION OF A HARMONY PROJECT STRUCTURE 33

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.3 REQUIREMENTS ANALYSIS ... 34
4.3.1 DOORS: Import of Stakeholder Requirements ... 35
4.3.2 DOORS: Import of System Requirements ... 36
4.3.3 Linking System Requirements to Stakeholder Requirements ... 38
4.3.4 DOORS -> Gateway -> Rhapsody: Import of System Requirements ... 41
4.3.5 Definition of System-Level Use Cases ... 42
4.3.5.1 Linking Requirements to Use Cases ... 43
4.3.6 Rhapsody -> Gateway -> DOORS: Export of Use Cases ... 46
4.4 SYSTEM FUNCTIONAL ANALYSIS ... 48
4.4.1 Uc1ControlEntry ... 49
4.4.1.1 Definition of Model Context ... 49
4.4.1.2 Definition of Functional Flow ... 52
4.4.1.3 Derivation of Black-Box Use Case Scenarios ... 53
4.4.1.4 Definition of Ports and Interfaces ... 57
4.4.1.5 Definition of Use Case Behavior ... 58
4.4.1.6 Use Case Model Verification ... 60
4.4.1.7 Linking Model Properties to Requirements ... 62
4.4.2 Uc2ControlExit ... 64
4.4.2.1 Definition of Model Context ... 64
4.4.2.2 Definition of Functional Flow ... 64
4.4.2.3 Derivation of Black-Box Use Case Scenarios ... 65
4.4.2.4 Definition of Ports and Interfaces ... 66
4.4.2.5 Definition of Use Case Behavior ... 66
4.4.2.6 Use Case Model Verification ... 67
4.4.2.7 Linking Model Properties to Requirements ... 67
4.5 DESIGN SYNTHESIS ... 68
4.5.1 Architectural Analysis (Trade-Off Analysis) ... 68
4.5.1.1 Definition of Key System Functions ... 69
4.5.1.2 Definition of Candidate Solutions ... 70
4.5.1.3 Definition of Assessment Criteria ... 71
4.5.1.4 Assigning Weights to Assessment Criteria ... 72
4.5.1.5 Definition of a Utility Curve for each Criterion ... 73
4.5.1.6 Assigning Measures of Effectiveness (MoE) to each Solution ... 74
4.5.1.7 Determination of Solution ... 75
4.5.1.8 Documentation of the Solution in the ArchitecturalDesignPkg ... 77
4.5.2 Architectural Design ... 78
4.5.2.1 Use Case Realization Uc1ControlEntry ... 79
4.5.2.1.1 Update of the ArchitecturalDesignPkg ... 79
4.5.2.1.2 Allocation of System Block Properies to Parts ... 80
4.5.2.1.2.1 Allocation of Operations to Parts ... 80
4.5.2.1.2.2 Allocation of Attributes and Events to Parts ... 84
4.5.2.1.3 Derivation of White-Box Sequence Diagrams ... 85
4.5.2.1.4 Definition of Ports and Interfaces ... 88

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.1.5 Definition of Realized Use Case Behavior 90
4.5.2.1.6 Realized Use Case Verification 93
4.5.2.1.7 Allocation of Non-functional Requirements 93
4.5.2.2 Use Case Realization Uc2ControlExit 94
4.5.2.2.1 Update of the ArchitecturalDesignPkg 94
4.5.2.2.2 Allocation of System Block Properies to Parts 94
4.5.2.2.3 Derivation of White-Box Sequence Diagrams 94
4.5.2.2.4 Definition of Ports and Interfaces 95
4.5.2.2.5 Definition of Realized Use Case Behavior 95
4.5.2.2.6 Realized Use Case Verification 96
4.5.2.2.7 Allocation of Non-functional Requirements 96
4.5.2.3 Integrated Use Case Realization 97
4.5.2.3.1 Creation of Base IA Model 98
4.5.2.3.2 Configuring Realized Use Case Model Handoff 99
4.5.2.3.3 Integration of Realized Use Case 100
4.5.2.3.4 Verification of Use Cases Collaboration 108

5 HAND-OFF TO SUBSYSTEM DEVELOPMENT 109

6 APPENDIX 115

A1 MODELING GUIDELINES 115
A1.1 General Guidelines and Drawing Conventions 115
A1.2 Use Case Diagram 116
A1.3 Block Definition Diagram 117
A1.4 Internal Block Diagram 118
A1.5 Activity Diagram 120
A1.6 Sequence Diagram 123
A1.7 Statechart Diagram 125
A1.8 Profiles 127
A2 DERIVING A STATECHART DIAGRAM 128
A3 USAGE OF ACTIVITY DIAGRAM INFORMATION IN THE SE WORKFLOW 133
A6 RHAPSODY ACTION LANGUAGE 136
A5 CHANGE REQUEST-DRIVEN SYSTEM DESIGN APPROACH 139
A6 USING MODEL-BASED TESTING FOR THE VERIFICATION OF HAND-OFF MODELS 147
A7 RHAPSODY SE-TOOLKIT (OVERVIEW) 153

7 REFERENCES 156

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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 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 tools are the Rational® systems and software design tool *Rhapsody® Release 8.01* and the requirements management and traceability tool *DOORS*® Release 9.3.

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 *DOORS* and ends with the definition of an executable integrated 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 (*SecSysControlle*r) development.

Also provided are several appendices (Section 6) including

- 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 chapter about the usage of Activity Diagram information in the SE workflow,
- a quick reference guide to the *Rhapsody* Action Language,
- an overview of the *Rhapsody SE-Toolkit* features
- a chapter outlining the model-based system design approach in the case of a change request to a legacy system
- a chapter about a model-based testing approach for the verification of hand-off models by means of the *Rhapsody* tool add-ons *TestConcuctor* (TC) and *Automatic Test Generation* (ATG).

Included to this deskbook is a volume containing

- the SecSys Stakeholder and System Requirements
- for each of the SE phases the incrementally extended *Rhapsody* model database.
- *DOORS* archive of the SecSys requirements
- *Rhapsody Gateway* custom types file

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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.

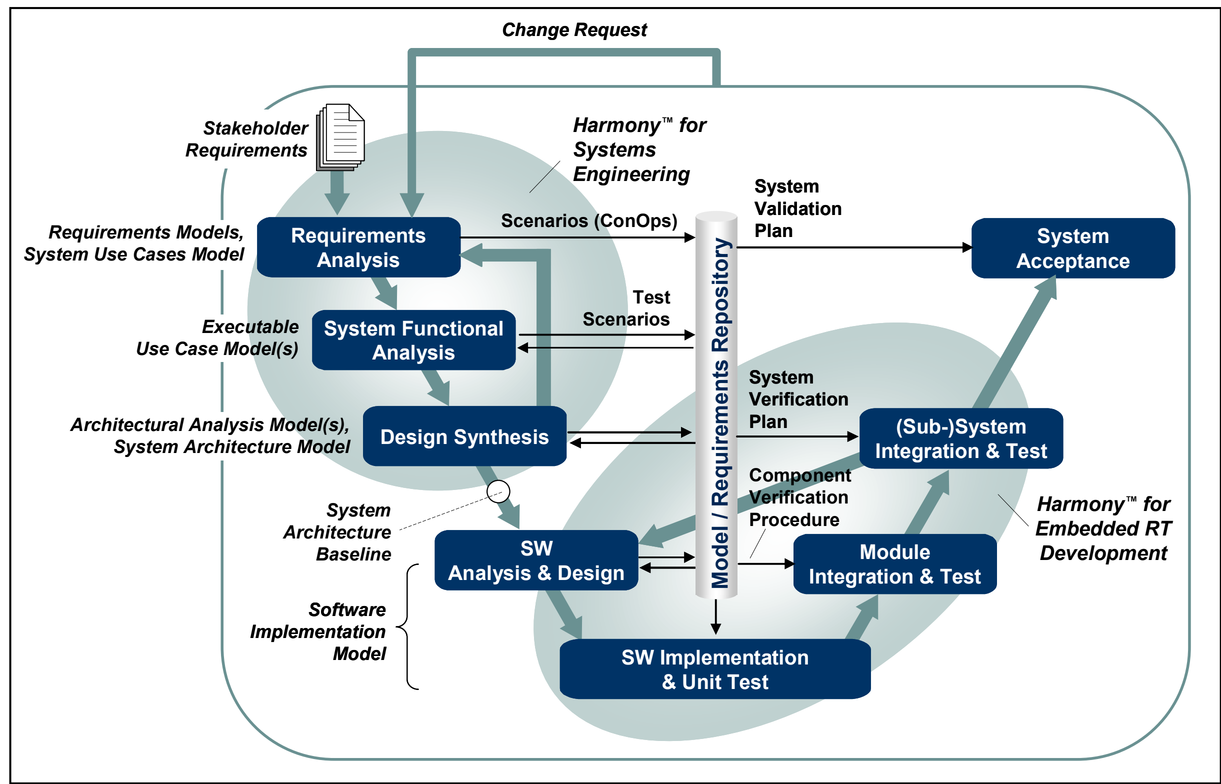


Fig. 2-1 Rational Integrated Systems / Embedded Software Development Process Harmony

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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 [3].

The analysis iterations for systems engineering and implementation continue through implementation and testing, to provide demonstrable results with each iteration to continually validate behavior.

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

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

Models that support the *requirements analysis* phase are

- the *Requirement Models* and
- the *System Use Cases Model*.

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

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

There are two types of executable models supporting the design synthesis phase:

- *Architectural Analysis Model(s)* and
- *System Architecture Model*

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

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

The baselined system architecture model defines the hand-off to the subsequent HW/SW development.

Model-driven software development is supported by the *Software Implementation Model*. This model is the basis for - either manual or automatic - code generation.

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

- Requirements documentation
- Requirements traceability
- Design documentation and
- Test definitions

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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.

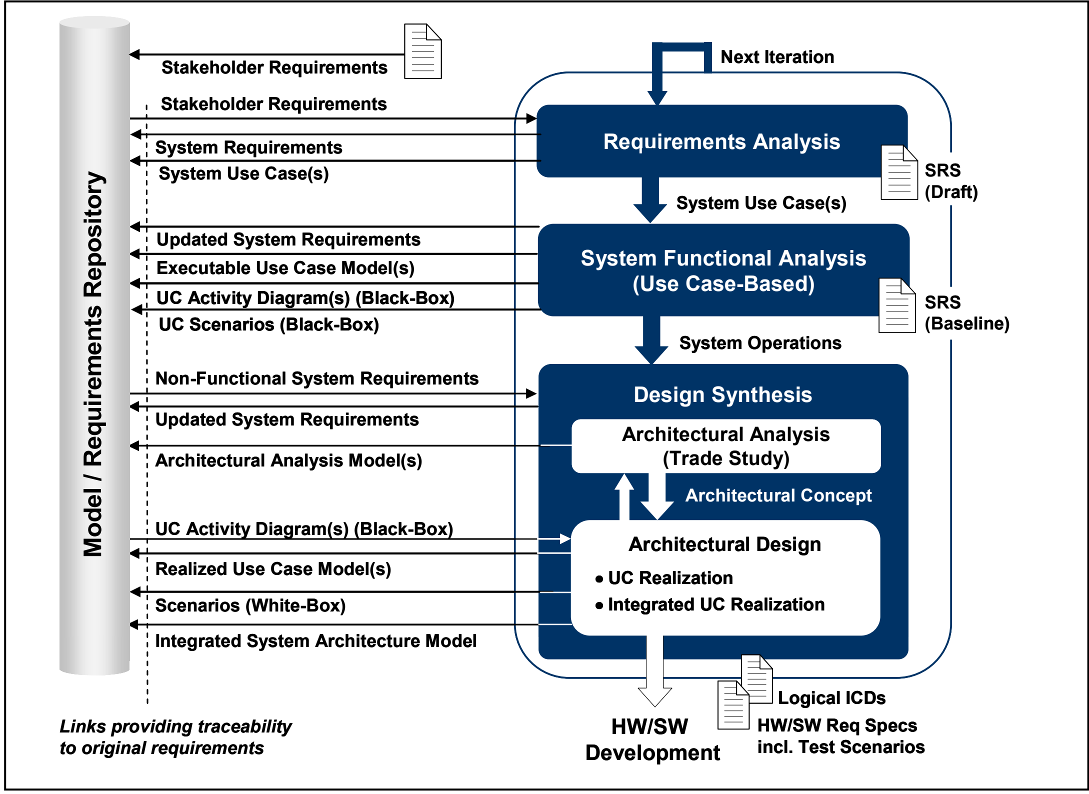


Fig. 2-2 Model-based Systems Engineering

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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.*

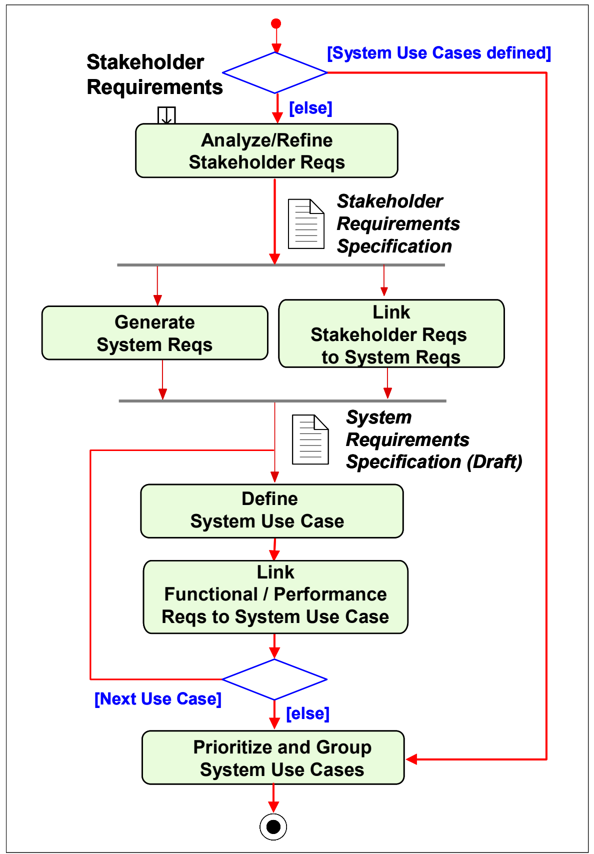


Fig. 2-3 Workflow in the Requirements Analysis Phase

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 10 to 15 system 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 selected set of use cases defines the *increments* of the iterative SE workflow. At the end of each iteration this ranking might need to be updated.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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.

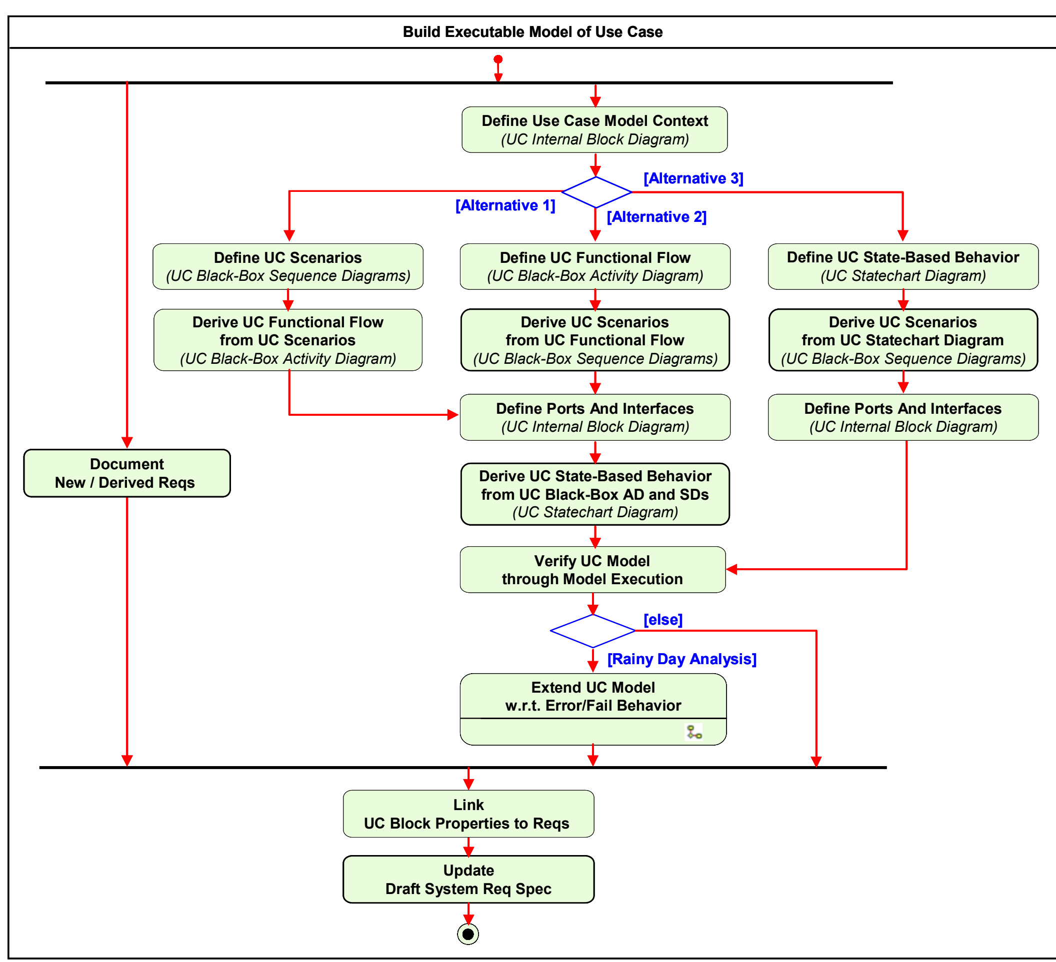


Fig. 2-4 Alternative Approaches of Building an Executable Use Case Model

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

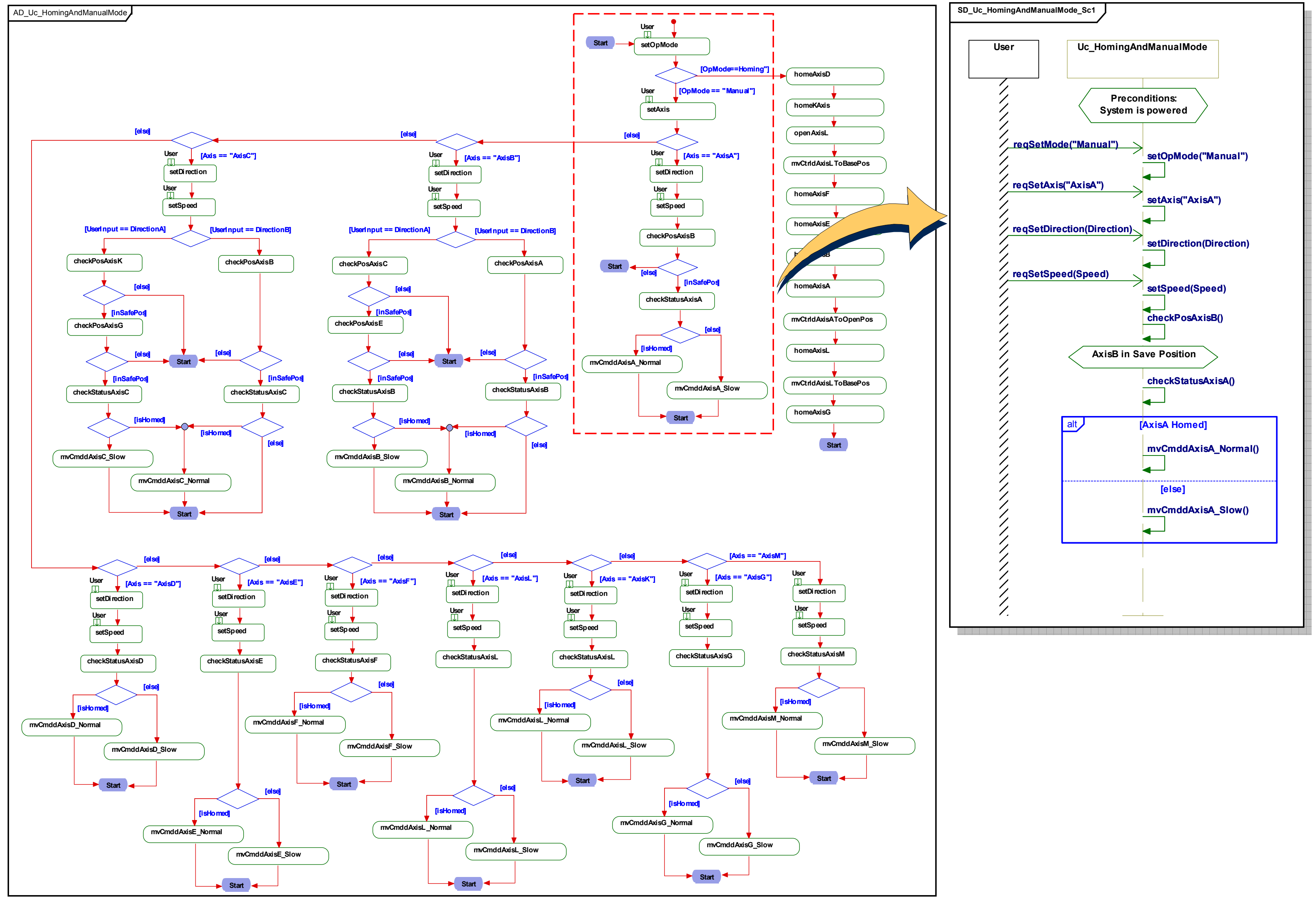


Fig. 2-5 Derivation of a Use Case Scenario from a Use Case Black-Box Activity Diagram (Industrial Automation Use Case)

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

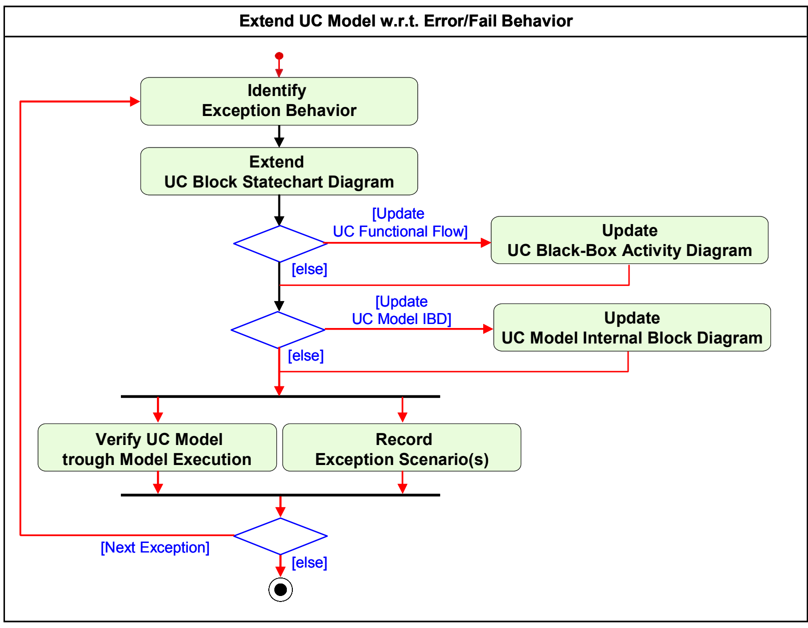


Fig. 2-6 Workflow of the Use Case Model Rainy Day Analysis

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 fail behaviour scenario as well as the internal block diagram needs to be updated accordingly. The extended use case model is verified through model execution.

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, 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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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 is split into two sub-phases

- *Architectural Analysis* and
- *Architectural Design*.

2.2.3.1 Architectural Analysis

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 [4]. 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. 2-7 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

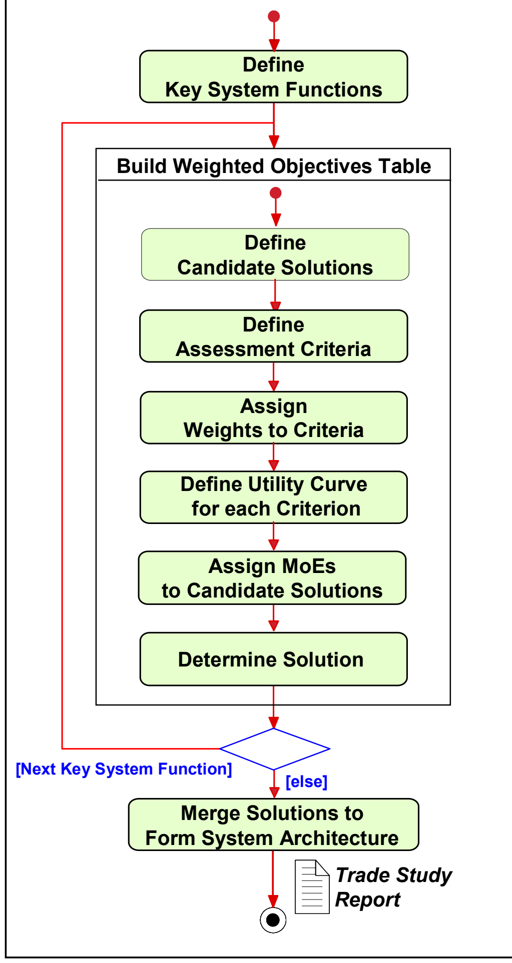


Fig. 2-7 Workflow and Work Product in the Architectural Analysis Phase

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

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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.
Fig. 2-9 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.

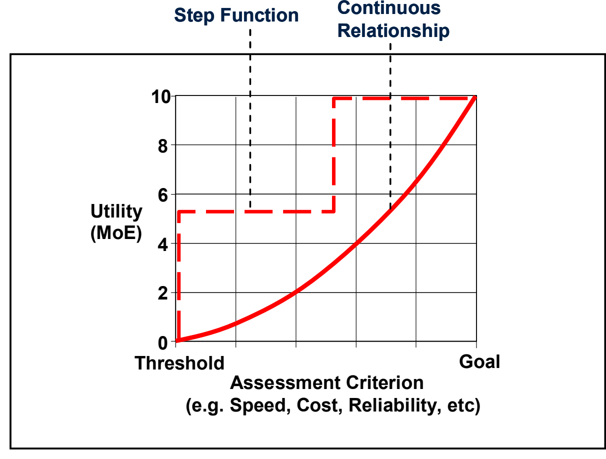
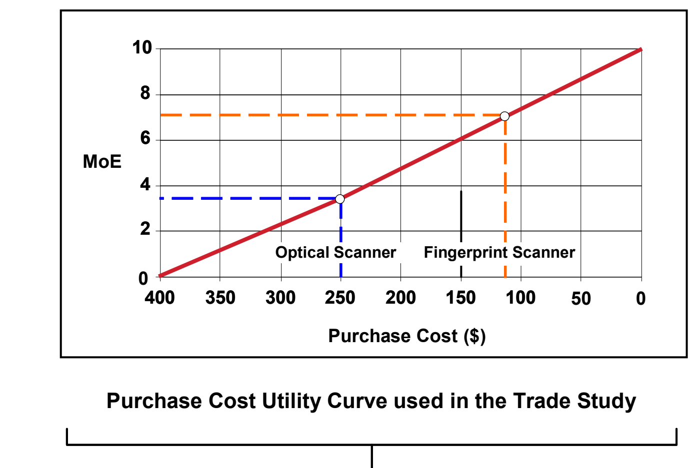


Fig. 2-8 Different Shapes of Utility Curves

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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



Purchase Cost Utility Curve used in the Trade Study

Solution Alternatives	Solution Criteria										Weighted Total
	Accuracy Wt = 0.3		Purchase Cost Wt = 0.2		Installation Cost Wt = 0.15		Maintenance Cost Wt = 0.1		Security Wt = 0.25		
	MoE	W	MoE	W	MoE	W	MoE	W	MoE	W	
Fingerprint Scanner	7.5	2.25	7.25	1.45	6.0	0.9	8.0	0.8	8.0	2.0	7.4
Optical Scanner	10.0	3.0	3.75	0.75	2.12	0.318	6.0	0.6	10.0	2.5	7.168

MoE = Measurement of Effectiveness Value W = Weighted Value = Wt * MoE Weighted Total = Σ W(Solution Alternative)

Fig. 2-9 Weighted Objectives Table of the Key System Function “Capture Biometric Data” (ref. Case Study Chapter 4)

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

2.2.3.2 Architectural Design

The focus of the architectural design phase is on the allocation of functional requirements and non-functional requirements to an architectural structure. This structure may be the result of a previous trade study or a given (legacy) architecture. The allocation is an iterative process and is typically performed in collaboration with domain experts.

Architectural design is performed incrementally for each use case of an iteration by transitioning from the black-box view to the white-box view – also referred to as ***use case realization*** (ref. *Fig. 2-10*). The taskflow is quite similar to the one outlined for the System Functional Analysis

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

Next, the system-level use case operations are allocated to the system structure. Generally, there are two ways to proceed. If an allocation concept exixts, they may be copied directly into the relevant parts. Otherwise, the allocation can be elaborate graphically by means of the *Use Case White-Box Activity Diagram*. Essentially, this activity diagram is a copy of the *Use Case Black-Box Activity Diagram*, partitioned into swim lanes, each representing a block of the system architectural decomposition hierarchy. Based on the chosen design concept, the system-level operations (= actions) then are “moved” into respective block swim lanes (ref. *Fig. 2-12*) 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. 2-11*).

If an action cannot be allocated to a single block, it must be decomposed. In this case, the sub-operations need to be linked to the parent operation through a respective dependency.

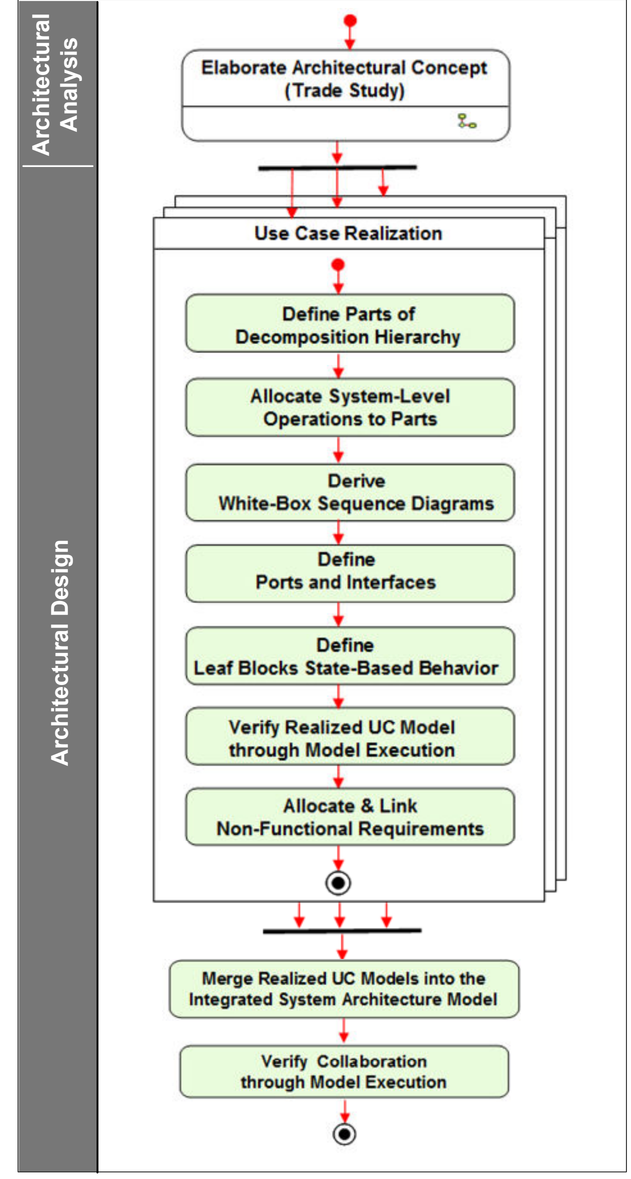


Fig. 2-10 Workflow in the Architectural Design Phase

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

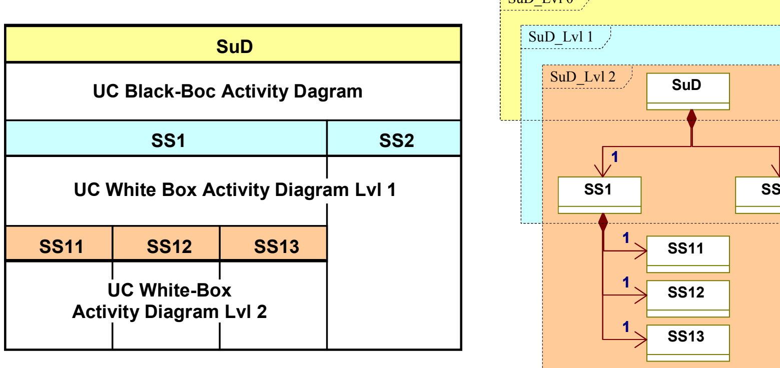


Fig. 2-11 Nested Use Case White-Box Activity Diagram

An action/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 operation/action is copied into the respective block swim lane and integrated into the functional flow.

The white-box activity diagram provides an initial estimate of the resulting load on respective communication channels, as links that cross a swim lane correspond to interfaces.

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

From the final *Use Case White-Box Diagram*, associated *White-Box Sequence Diagrams* are derived (ref. *Fig. 2-13*). As outlined previously, these sequence diagrams are the basis from which ports and interfaces of the blocks at the lowest level of the system architecture are derived.

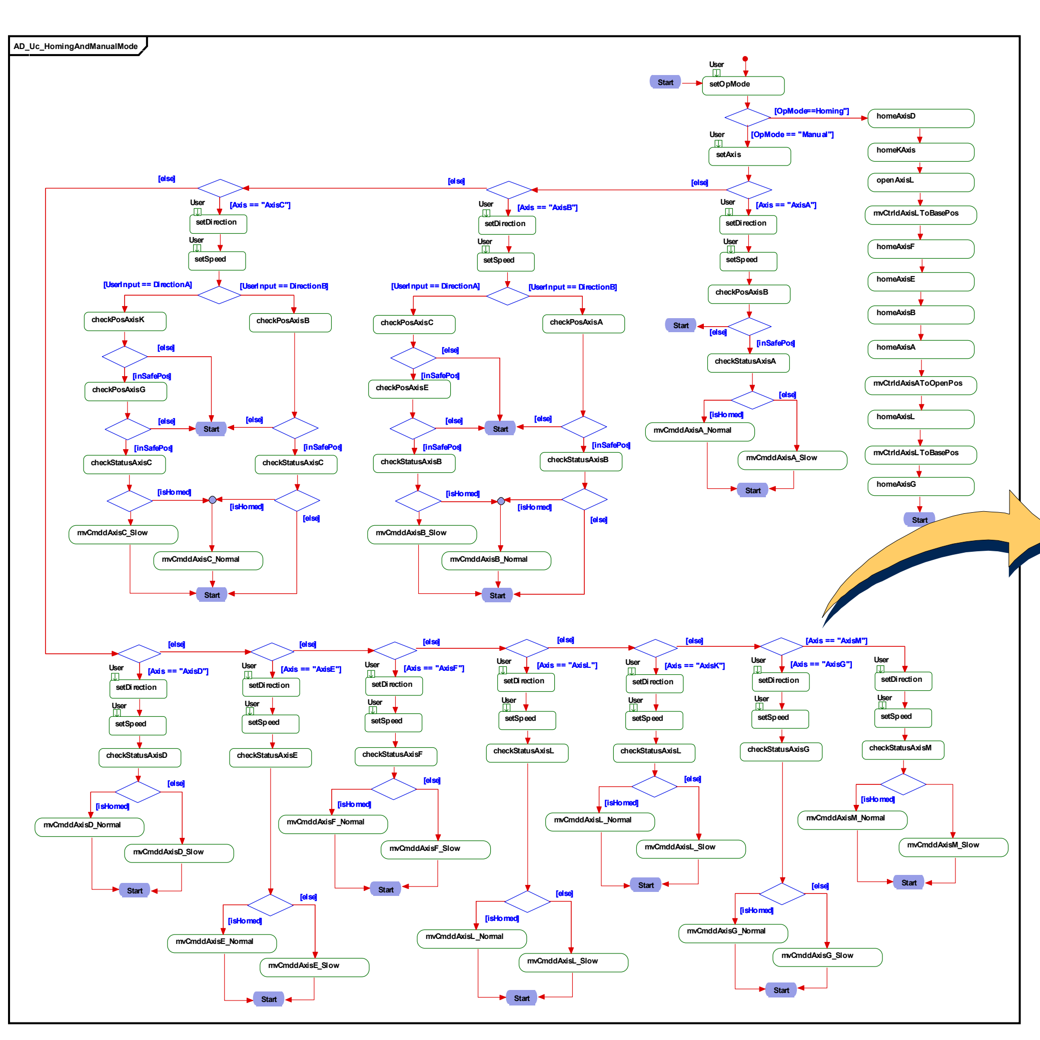
Once system-level operations are allocated to the relevant blocks at the lowest level of the architectural decomposition and associated ports and interfaces are defined, the individual state-based behavior is captured in a statechart diagram. The leaf-block behavior as well as the collaboration of the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

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

The final task in the architectural design phase is the creation/update of *the Integrated System Architecture Model.* This model is the aggregate of the realized use case models. It is the aggregate of the baselined realized use case models

The use cases collaboration as well as the correctness and completeness of the Integrated System Architecture Model may be verified through model execution.

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.



Use Case Black-Box Activity Diagram

Fig. 2-12 Allocation of Operations to Subsystems (Use Case Fig. 2-5)

Use Case White-Box Activity Diagram Decomposition Lvl 1

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

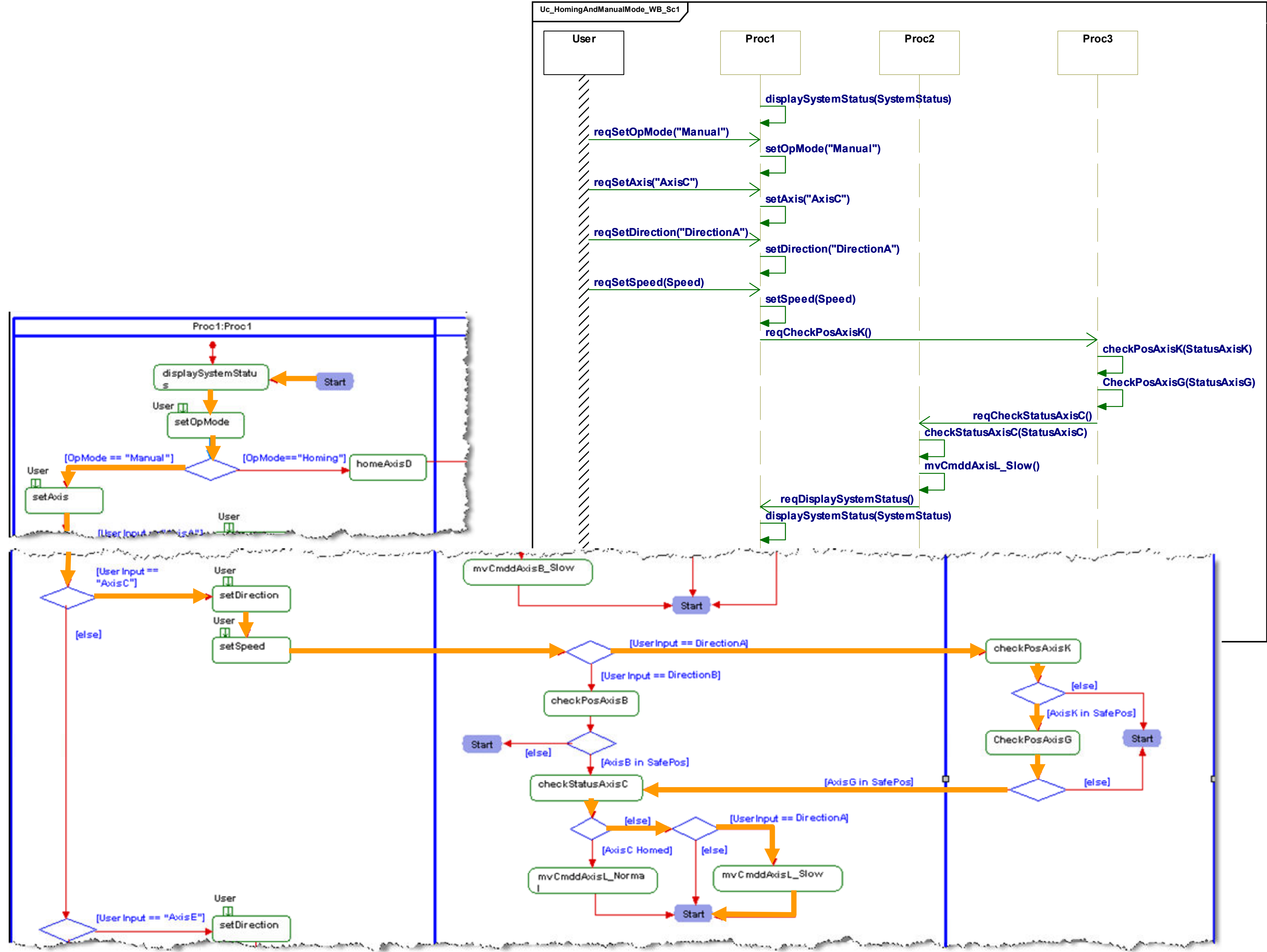


Fig. 2-13 Derivation of White-Box Scenarios from a Use Case White-Box Activity Diagram (ref. Fig. 2-5)

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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 Integrated System Architecture 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 configuation 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 and attributes including links to the associated system functional and performance requirements
- Definition of CI ports and *logical* – optionally *operational* - interfaces
- Definition of CI behavior, captured in a statechart diagram
- Test scenarios – also referred to as *Integration Test Scenarios* – derived from system-level use case scenarios
- CI-allocated non-functional requirements

It should be noted, that the baselined Integrated System Architecture Model becomes the reference model for further development of system requirements.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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 2.3 and extended it by systems engineering specific constructs. *Fig. 2-14* 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 2.3 and SysML 1.2, 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.

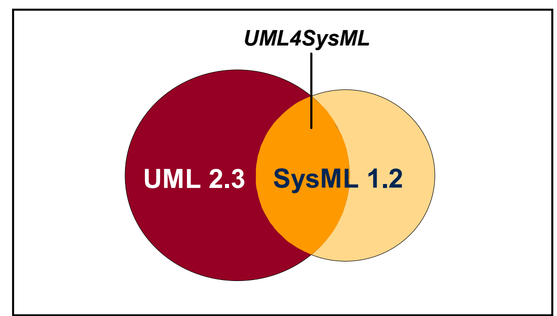


Fig. 2-14 Overview of UML/SysML Interrelationship

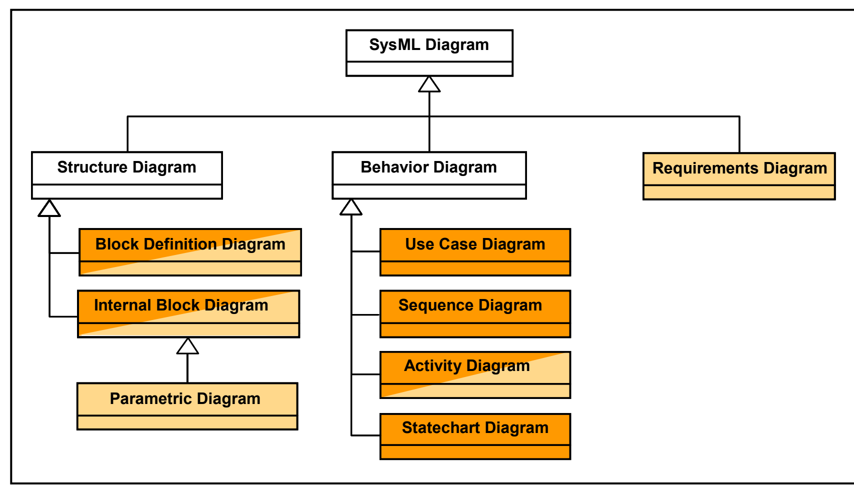


Fig. 2-15 Taxonomy of SysML Diagrams Used in Harmony for Systems Engineering

Fig. 2-15 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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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>>).

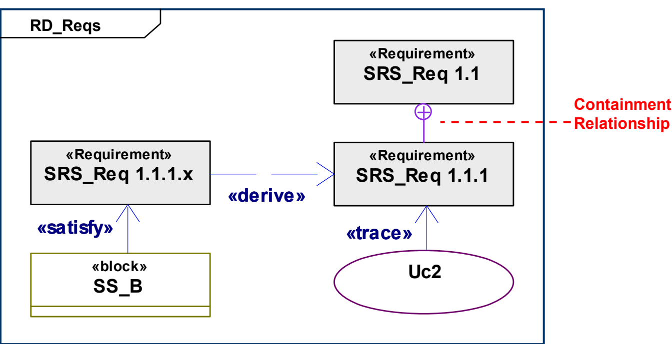


Fig. 2-16 Requirements Diagram

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.

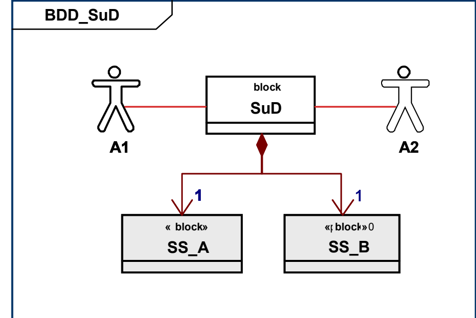


Fig. 2-17 Block Definition Diagram

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.

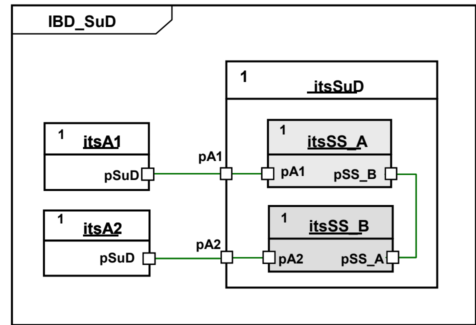


Fig. 2-18 Internal Block Diagram

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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.

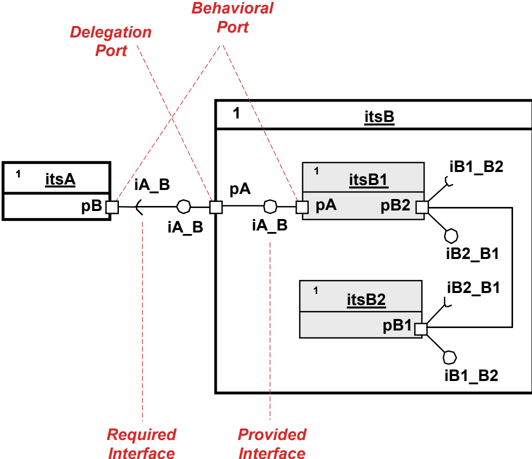


Fig. 2-19 Standard Ports

A standard port is specified via its *provided* and *required* ***interfaces***. A provided interface (denoted by a lollipop symbol) specifies a set of messages received at that port from elements outside the block. A required interface (denoted by a socket symbol) specifies a set of messages sent from that port to elements outside of the block. Thus, by characterizing an interface as required or provided, the direction of the constituent messages at the port is defined.

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”.

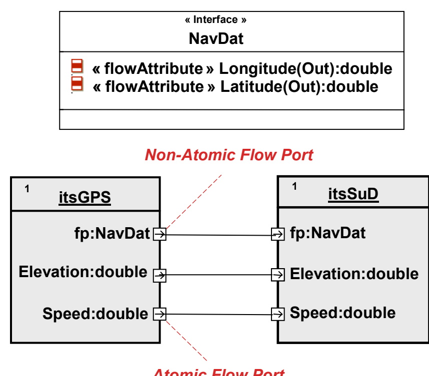


Fig. 2-20 Flow Ports

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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

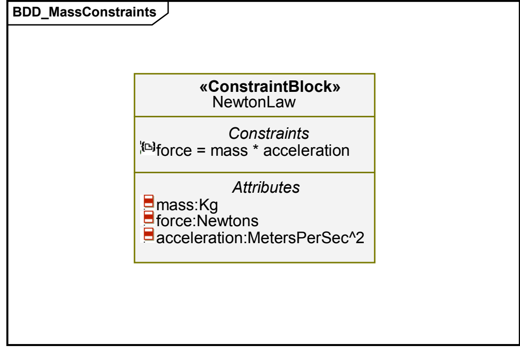


Fig. 2-21 Constraint Block Definition in a Block Definition Diagram

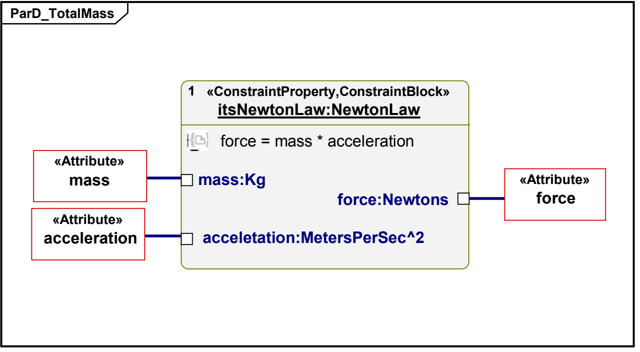


Fig. 2-22 Parametric Diagram

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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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").

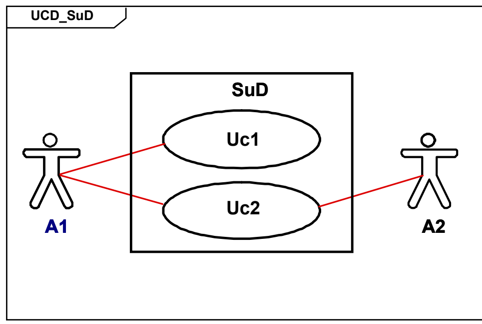


Fig. 2-23 Use Case Diagram

When use cases get too complex, *dependencies* between use cases may be defined:

- *<<include>>*
  One use case includes another
- *<<extend>>*
  One use case provides an optional extension of another
- *Generalization*
  One use case is a more specialized or refined version of another

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.

<u>NOTE</u>: 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.

<u>NOTE</u>: *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.

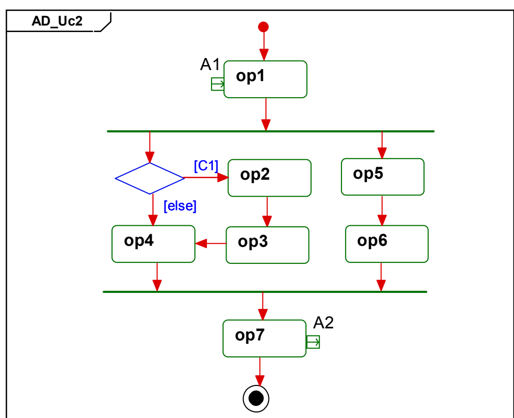


Fig. 2-24 Activity Diagram

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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 behavior. 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*).
  <u>NOTE</u>: 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.

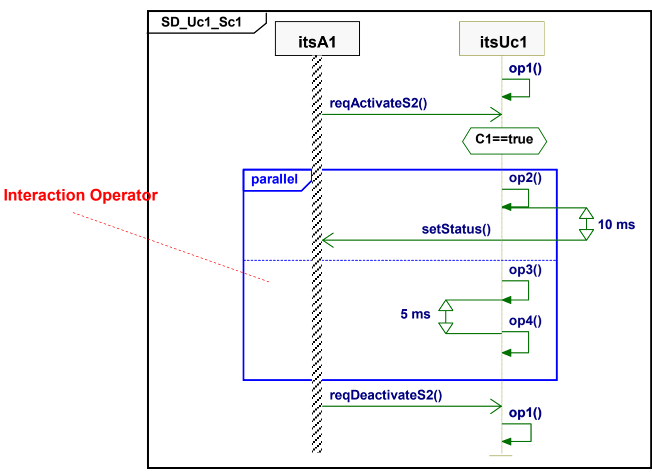


Fig. 2-25 Sequence Diagram

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

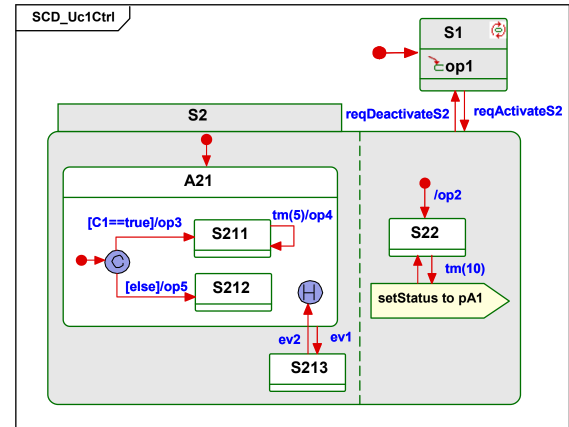


Fig. 2-26 Statechart Diagram

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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

Fig. 2-27 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 block properties are described by operations, attributes, ports and interfaces.
- The state-based behavior of each block instance is described by a *Statechart Diagram*.

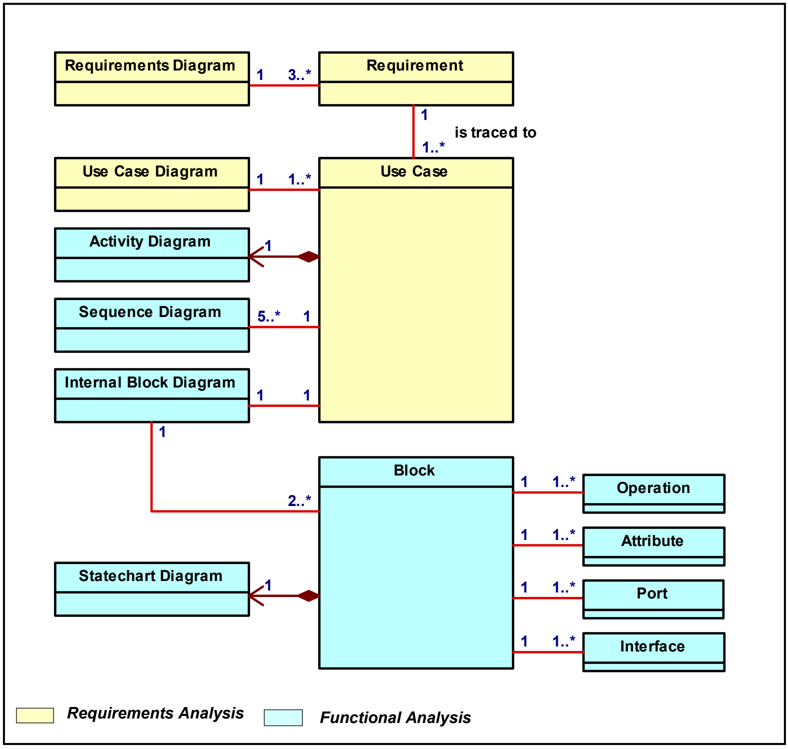
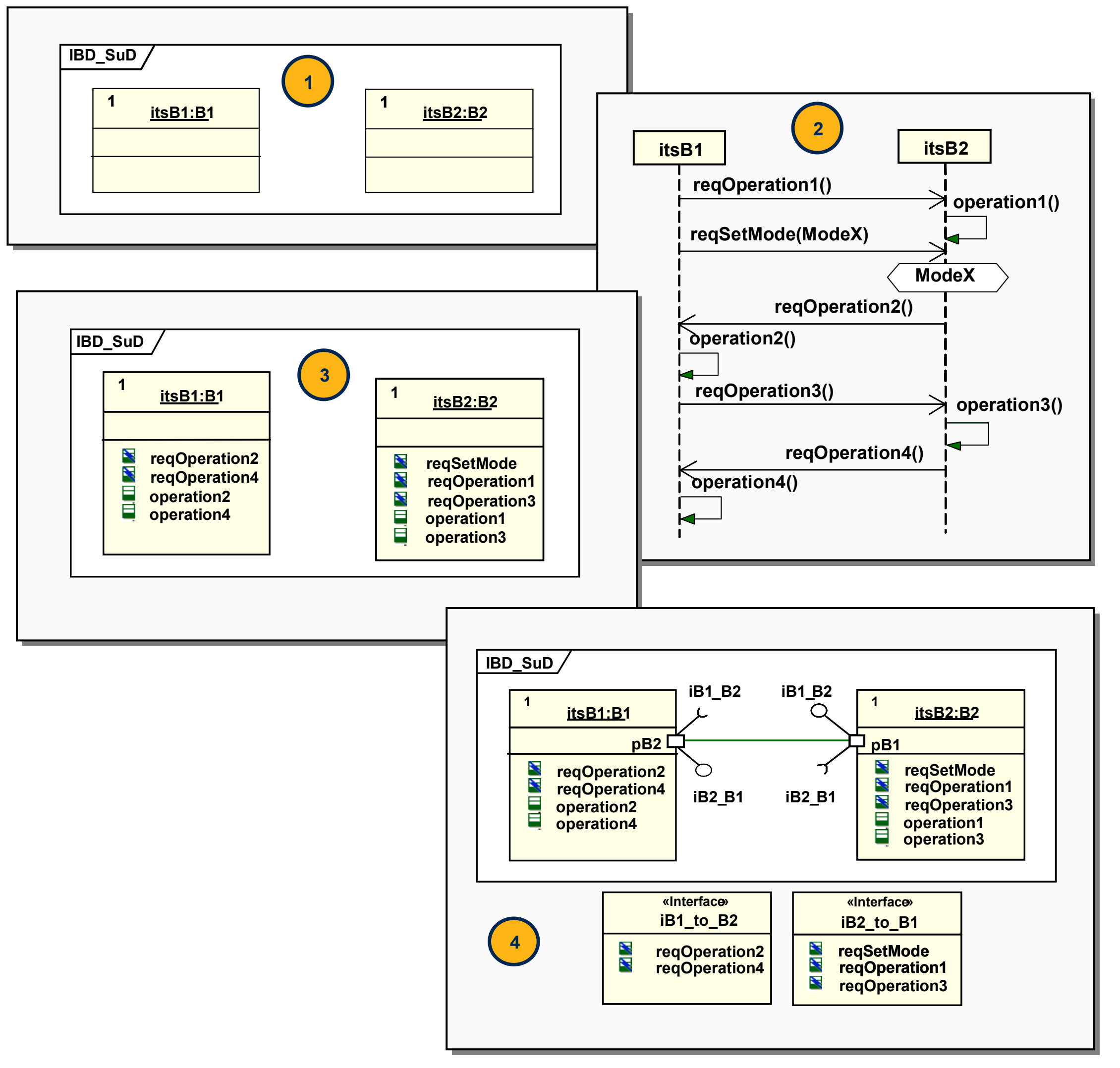


Fig. 2-27 SysML Artifacts Relationship at the Requirements Analysis / System Functional Analysis Level

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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 4.4.1.3).

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 4.4.1.4).

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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

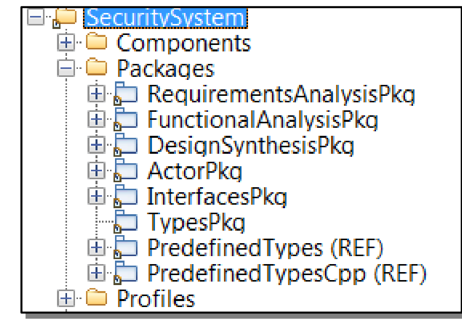
On the top-level, the project structure shows two types of packages:

- 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*



Project Structure Overview

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

3.2 Requirements Analysis Package

Constituents of the RequirementsAnalysisPkg are

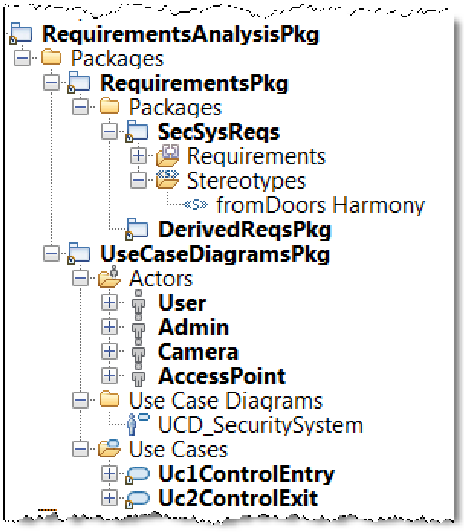
- *RequirementsPkg* and
- *UseCaseDiagramsPkg*

The RequirementsPkg contains the system requirements ("shall" statements) generated from the stakeholder requirements and imported from DOORS.

During the system functional analysis and design synthesis phase additional requirements may be identified. Temporarily, they will be located in the DerivedRequirementsPkg. Once they are approved through model execution, the system requirements database in DOORS will be updated accordingly. The updated system requirements then are exported from DOORS to Rhapsody and linked to the associated model artifacts.

The UseCaseDiagramsPkg contains the use cases incl the system requirements related dependencies, the actors as well as the use case diagram(s)

<u>*NOTE*</u>: Initially, use cases and actors are located in the UseCasesPkg. In the system functional analysis phase the use cases are moved into respective use case packages in the *FunctionalAnalysisPkg* and the associated actors are moved into the *ActorsPkg.*



Requirements Analysis Package

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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.

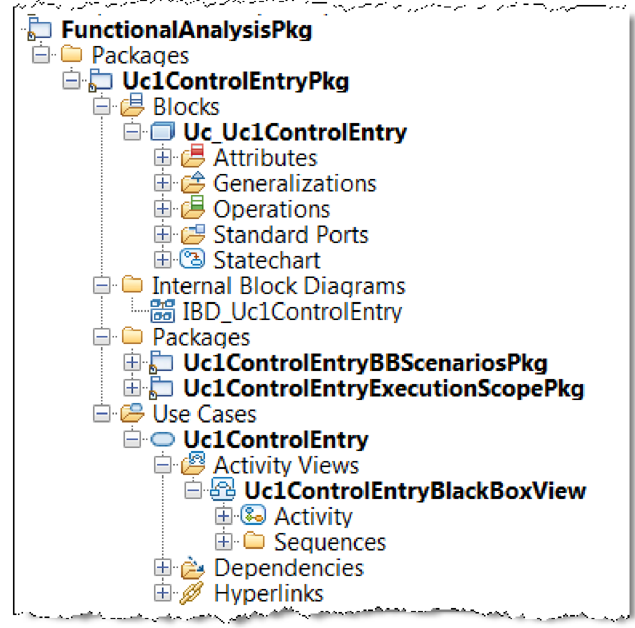
For each use case of the use case diagram, there is a package <*UseCaseName*>**Pkg** that contains the associated model artifacts:

- 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 4.4.1.3).
  - 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.



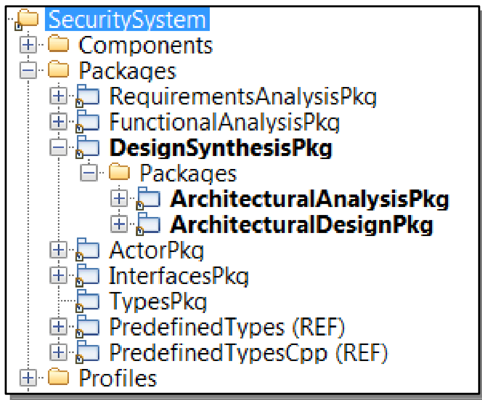
Functional Analysis Package

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

3.4 Design Synthesis Package

The *DesignSynthesisPkg* is partitioned into two packages

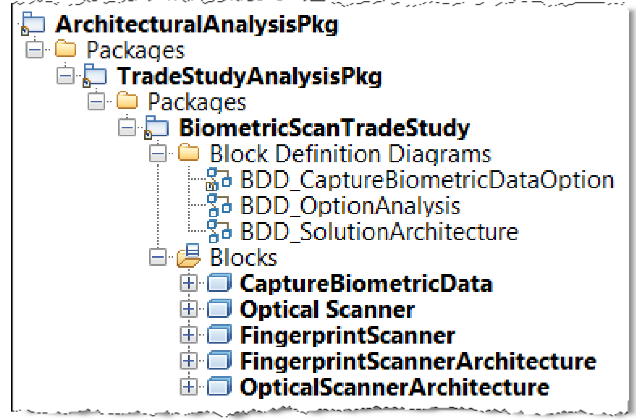
- *ArchitecturalAnalysisPkg* and
- *ArchitecturalDesignPkg*



Design Synthesis Package

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 4.5.1.



Architectural Analysis Package

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

3.4.2 Architectural Design Package

Constituents of the *ArchitecturalDesignPkg* are:

- A folder *Block Definition Diagrams* with the SuD level 1 block definition diagram **BDD_**<*SuDName*>

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

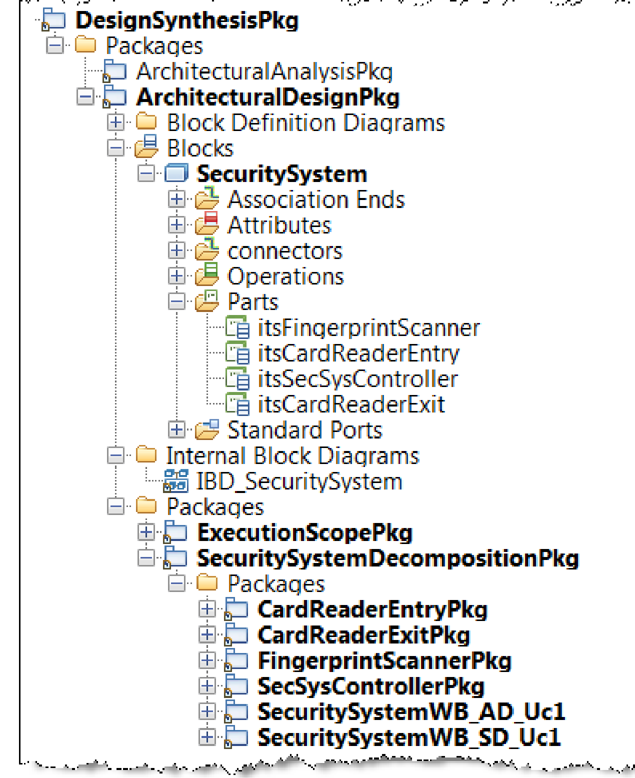
- A folder *Internal Block Diagrams* with the internal block diagram of the SuD system architecture **IBD_**<*SuDName*>

- 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:

    - 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-blocks each of which will be decomposed according to the outlined structure.
    - A package <*Block*>**WB_AD** which contains the decomposed white-box activity diagram(s) of the system use case(s),
    - A package <*Block*>**WB_UcSD** which holds the decomposed system use case scenarios,



Architectural Design Package

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

3.5 System-Level Definitions

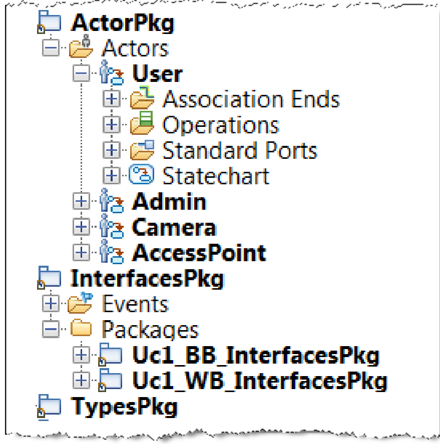
On the top-level of the project structure there are three packages for system-level definitions:

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

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

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

The *TypesPkg* contains the system-level data definitions.



System-Level Definitions

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4 Case Study: Security System

Harmony for Systems Engineering is tool independent. 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 *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*.

4.1 Case Study Workflow

Fig. 4-1 provides an overview of the MbSE workflow followed in the case study. It shows for each of the SE phases the generated key handoff artifacts together with the associated *Rhapsody* projects. The reason for splitting the workflow into different *Rhapsody* projects is, that it supports the collaboration of distributed teams.

The workflow is use case based. It starts with the import of the elaborated system requirements from DOORS to the *Rhapsody* project <*SuD Name*>**_RA** and the definition of the system-level use cases. The handoff to the subsequent functional analysis phase are the use cases and the associated system requirements.

In the functional analysis phase the chosen use cases are transformed into executable black-box (BB) use case models. The modeling is performed for each use case in a separate *Rhapsody* project (**Uc**<*Nr*><*Use Case Name*>). The verified/validated black-box use case models and associated functional requirements are the input to the subsequent design synthesis phases.

Architectural analysis is performed in a separate *Rhapsody* project <*SuD Name*>**_AA** based on the verified/validated functional system requirements. Additionaly, non-functional requirements (design constraints) are taken into consideration. The elaborated system architecture structure is the handoff to the subsequent architectural design phase.

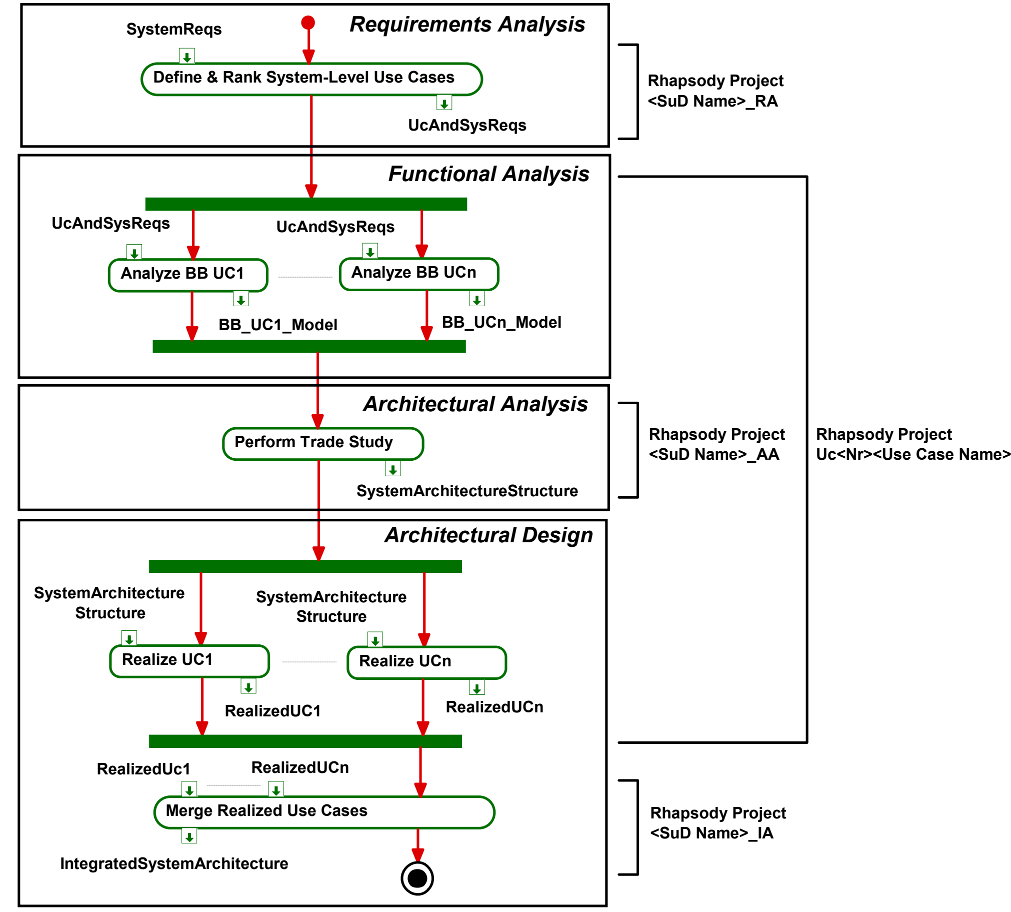


Fig. 4-1 MbSE Workflow and Associated Rhapsody Projects

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Architectural design is performed in two steps.

First, each of the black-box use case models is ***realized***, i.e. transformed into a white-box model based on the system architecture structure provided by the architectural analysis model. The realization is performed in respective *Rhapsody* projects defined in the functional analysis phase.

The correctness and completeness of each realized use case model is verified through model execution.

Once all use cases of an increment are realized, they are merged in the *Integrated System Architecture* model. The merger is performed in a separate Rhapsody project *<SuD Name>*_**IA**.

The baselined Integrated System Architecture model is the key artifact of the handoff to the subsequent system development. It is the repository from which specification documents (HW/SW Requirements Specifications, ICD's , …) are generated.

4.2 Creation of a Harmony Project Structure

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

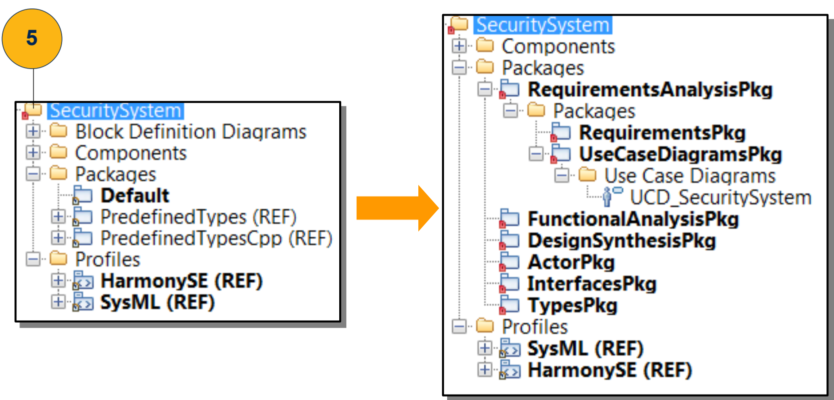
1. Start *Rhapsody*

2. In the main menu select *File > New*
   Enter project name (e.g. *SecuritySystem*) and select/define the associated project directory.

3. Select the *SysML* profile and create project..

4. 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.*



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.3 Requirements Analysis

The workflow followed in the case study is shown in *Fig. 4-2*. It starts with the import of the stakeholder requirements and derived system requirements – both captured as Word documents – into Doors. Once imported, the system requirements are linked to the stakeholder requirements via <<satisfy>> dependency and the complete coverage will be ensured
The next step in the Requirements Analysis workflow is the export of the system requirements from DOORS to Rhapsody. This is performed via *Rhapsody Gateway.*

In Rhapsody, the imported system requirements are grouped into use cases and respective <<trace>> links from the use cases to the system requirements are established.
Subsequently, the use cases incl. their links are exported from Rhapsody via Gateway to DOORS.

It should be noted that the outlined workflow will be applied whenever there will be a change or update of the requirements.

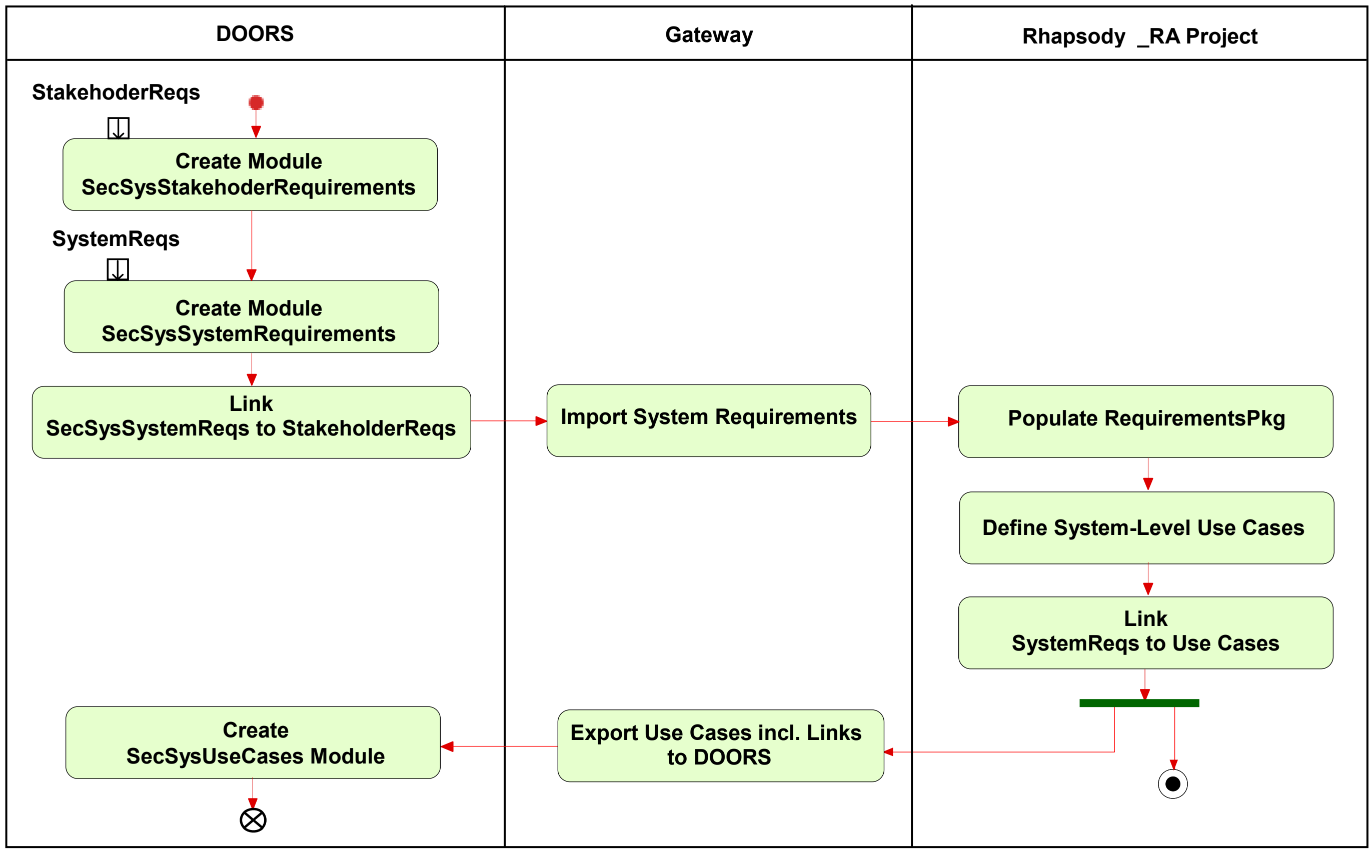
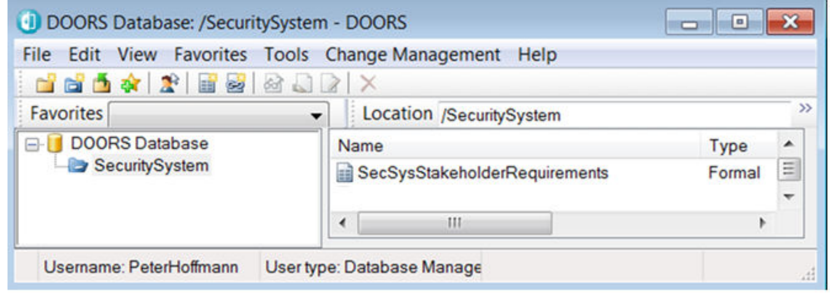


Fig. 4-2 Requirements Analysis Workflow

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.3.1 DOORS: Import of Stakeholder Requirements



1. Open DOORS. In the Database Explorer, select the project **SecuritySystem**.

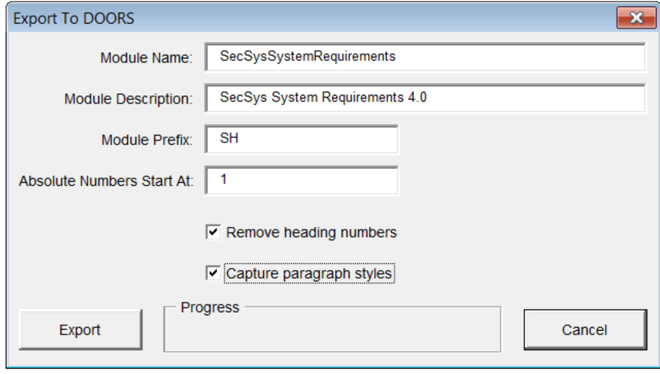
2. Open in the volume included to this Deskbook the Word document **SecSys Stakeholder Requirements 4.0.doc**

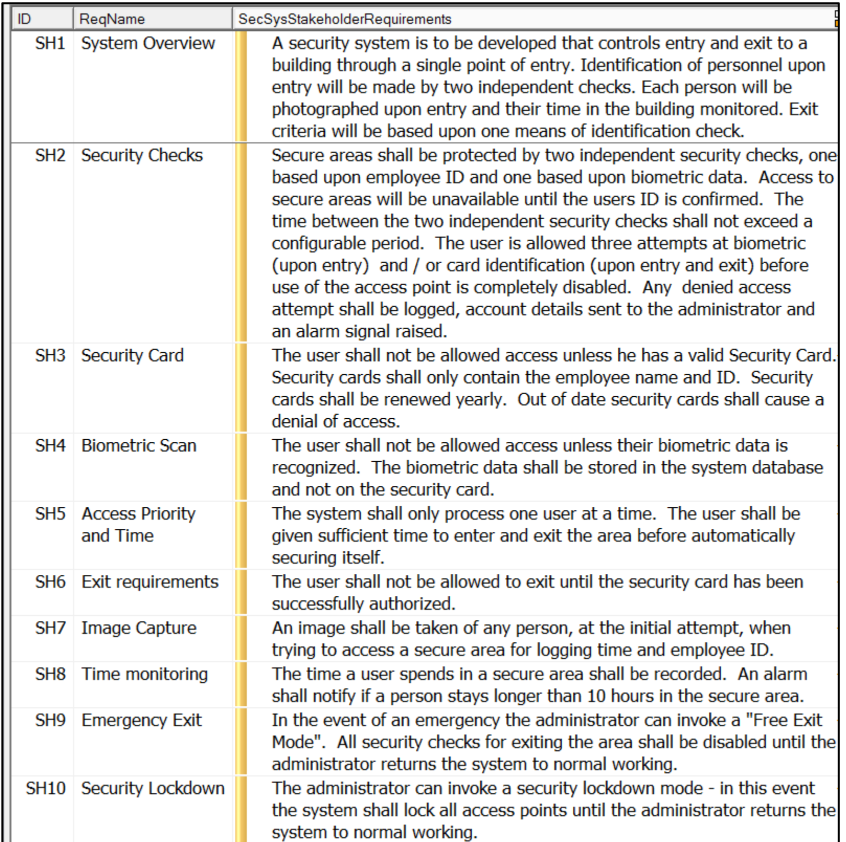
3. In the Word toolbar, click on the *Export to DOORS* icon.

4. Specify the Module Name: **SecSysStakeholderRequirements**. Specify Prefix: **SH**.

5. Click *Export*.

6. Switch to DOORS and save the new module.



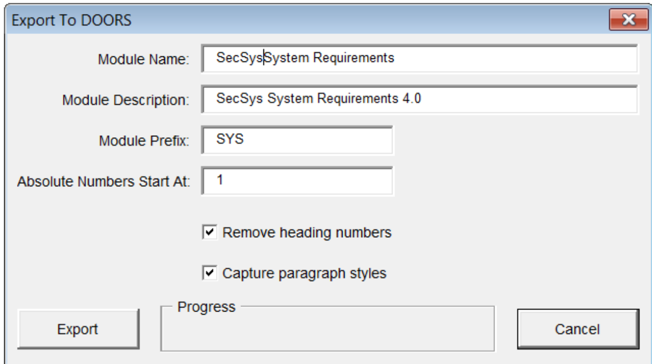


ID	ReqName	SecSysStakeholderRequirements
SH1	System Overview	A security system is to be developed that controls entry and exit to a building through a single point of entry. Identification of personnel upon entry will be made by two independent checks. Each person will be photographed upon entry and their time in the building monitored. Exit criteria will be based upon one means of identification check.
SH2	Security Checks	Secure areas shall be protected by two independent security checks, one based upon 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 shall not exceed a configurable period. The user is allowed three attempts at biometric (upon entry) and / or card identification (upon entry and exit) before use of the access point is completely disabled. Any denied access attempt shall be logged, account details sent to the administrator and an alarm signal raised.
SH3	Security Card	The user shall not be allowed access unless he has a valid Security Card. Security cards shall only contain the employee name and ID. Security cards shall be renewed yearly. Out of date security cards shall cause a denial of access.
SH4	Biometric Scan	The user shall not be allowed access unless their biometric data is recognized. The biometric data shall be stored in the system database and not on the security card.
SH5	Access Priority and Time	The system shall only process one user at a time. The user shall be given sufficient time to enter and exit the area before automatically securing itself.
SH6	Exit requirements	The user shall not be allowed to exit until the security card has been successfully authorized.
SH7	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.
SH8	Time monitoring	The time a user spends in a secure area shall be recorded. An alarm shall notify if a person stays longer than 10 hours in the secure area.
SH9	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.
SH10	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.

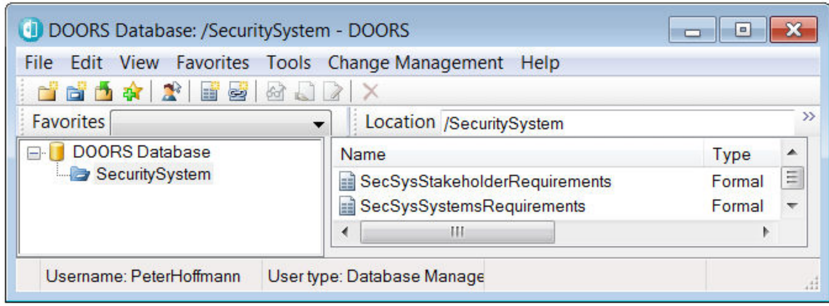
Stakehoder Requirements Inported into DOORS

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.3.2 DOORS: Import of System Requirements



1. Open DOORS. In the Database Explorer, select the project **SecuritySystem**.
2. Open in the volume included to this Deskbook the Word document **SecSys System Requirements 4.0.doc**
3. In the Word toolbar, click on the *Export to DOORS* icon.
4. Specify the Module Name: **SecSysSystemsRequirements**. Specify Prefix: **SYS**.
5. Click *Export*.
6. Switch to DOORS and save the new module.



 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

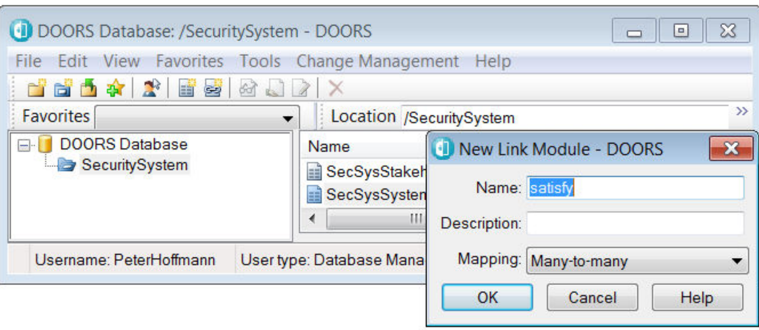
ID	ReqName	SecSys System Requirements 4.0	RequirementType
SYS1	Three Attempts On Employee ID Entry	Upon entry the user shall be allowed three attempts on card identification.	Functional
SYS2	Three Attempts On Biometric Data Entry	Upon entry the user shall be allowed three biometric data entries.	Functional
SYS3	Disabling User Account	After three failed attempts at card identification or biometric data entry the user account shall be disabled.	Functional
SYS4	Denied Entry Notification	Any denied access attempt shall be logged and account details sent to the administrator.	Functional
SYS5	Out of Date Cards	Out of date cards shall deny entry and invalidate the card.	Functional
SYS6	Authorization of Security Card – Entry	Access to the secure area shall only be allowed with a valid security card.	Functional
SYS7	Two Independent Security Checks	Secure areas shall be protected by two independent security checks.	Functional
SYS8	Alarm – Entry	On a denied entry an alarm signal shall be raised.	Functional
SYS9	Employee ID Card Identification – Entry	Entry shall be protected by a security check based upon employee ID.	Functional
SYS10	Visualization of Security Card Check Status – Entry	The user shall be visually informed about the status of his/her ID card check.	Functional
SYS11	Security Card Information	Security cards only contain the employee name and ID and will be renewed yearly.	Functional
SYS12	Visualization of Biometric Data Check Status	The user shall be visually informed about the status of his/her biometric data check.	Functional
SYS13	Approval of Biometric Data	The user shall not be allowed access unless his/her biometric data are recognized.	Functional
SYS14	Biometric Scan	Entry to the secure areas shall be protected by a second independent security check, based upon biometric data.	Functional
SYS15	Image Capture	An image shall be taken of any person, at the initial attempt, when trying to access a secure area.	Functional
SYS16	Three Attempts On Employee ID Exit	Upon exit the user shall be allowed three attempts on card identification.	Functional
SYS17	Time Limit Violation	An alarm shall notify if a person stays longer than 10 hours in the secure area.	Functional

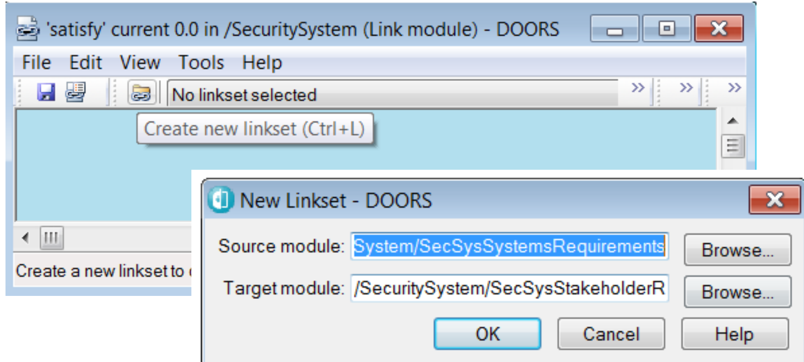
ID	ReqName	SecSys System Requirements 4.0	RequirementType
SYS18	Denied Exit Notification	The administrator shall be notified about any denied exit. The notification shall include user account details.	Functional
SYS19	Alarm – Exit	On a denied exit an alarm signal shall be raised.	Functional
SYS20	Employee ID Card Identification – Exit	Exit shall be protected by a security check based upon employee ID.	Functional
SYS21	Visualization of Security Card Check Status – Exit	The user shall be visually informed about the status of his/her ID card check.	Functional
SYS22	Security Lockdown	In the event of a security lockdown, the system shall lock all access points until the administrator returns the system to normal working.	Functional
SYS23	Emergency Exit	In the event of an emergency all security checks for exiting the area shall be disabled until the administrator returns the system to normal working.	Functional
SYS24	Authorization of Security Card – Exit	The user shall not be allowed to exit until the security card has been successfully authorized.	Functional
SYS25	Entry Time	The user shall be given sufficient time to enter the secure area.	Non-Functional
SYS26	Time Between Two Independent Checks	The time between the two independent security checks shall not exceed a configurable period.	Non-Functional
SYS27	Processing User Request	The system shall only process one user at a time.	Non-Functional
SYS28	Biometric Data Storage	Biometric data shall be stored in the system database and not on the security card.	Non-Functional
SYS29	Time Recording	The time a user spends in a secure area shall be recorded.	Non-Functional
SYS30	Exit Time	The user shall be given sufficient time to exit the secure area.	Non-Functional
SYS31	Automatic Securing the Secure Area – Entry	Once the user has entered the secure area the system shall automatically secure itself.	Functional
SYS32	Automatic Securing the Secure Area – Exit	Once the user has exited the secure area the system shall automatically secure itself.	Functional
SYS33	Configuration of Entry and Exit Time	The time to enter and exit the secure area shall be customizable.	Non-Functional

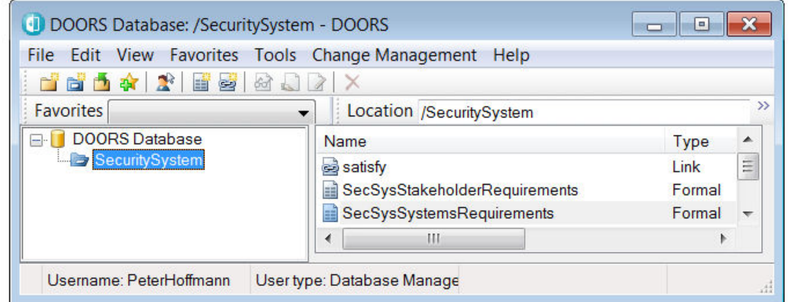
System Requirements Imported into DOORS

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.3.3 Linking System Requirements to Stakeholder Requirements







1. In the DOORS Database Explorer select the **SecuritySystem** project.
2. In the menu, select *File / New / Link Module…*
3. Name the link module **satisfy** and click *OK*
4. In the pop-up window select *Create new linkset*.
5. For the Source module, click *Browse…* and select the module SecSysSystemsRequirements. For the Target module, click *Browse…* and select the module SecSysStakeholderRequirements.
6. Click *OK*.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

7 Open the SecSysSystemsRequirements module.
Go to *File / Module Properties.*

8 In the *Linksets* tab, click *Add*.

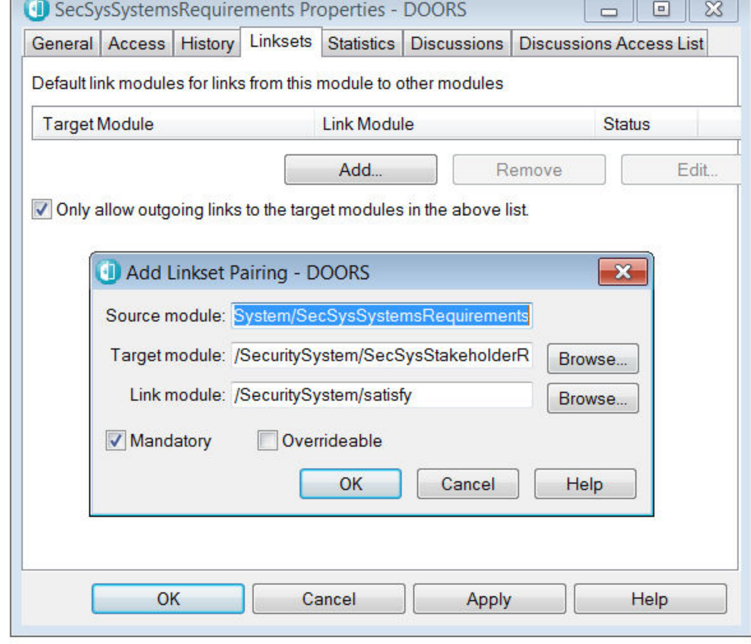
9 For the *Target module*,
select the SecSysStakeholderRequirements module.
For the *Link module*,
select the *satisfy* link module.

10 Tick the “Mandatory” box.

11 In the *Linksets* tab, tick
Only allow outgoing links to the target modules in the above list.

NOTE: This will prevent accidental links created in the wrong direction.

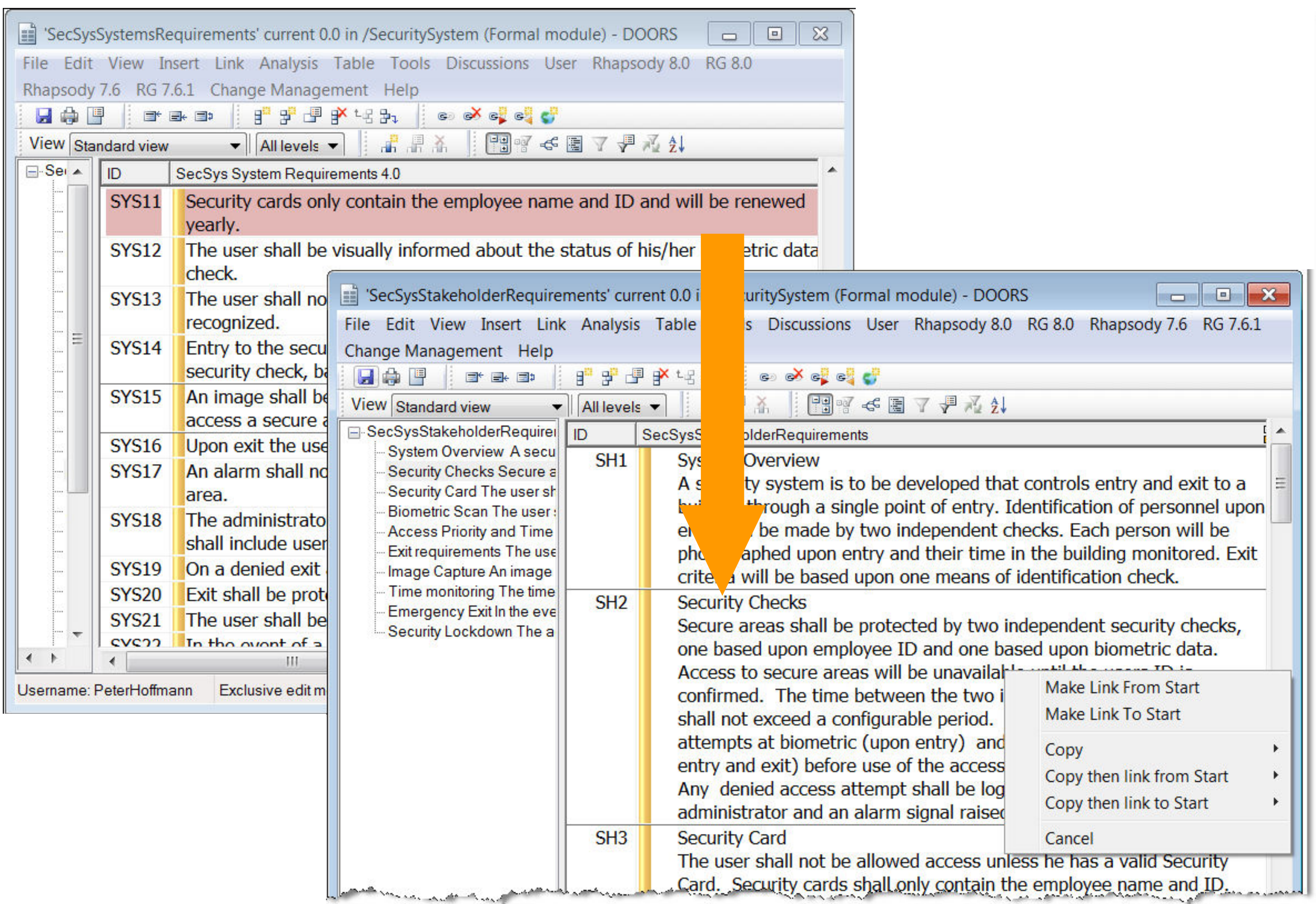
12 Click *OK*.



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

13

With the SecSysStakeholderRequirements and SecSysSystemsRequirements modules both open on the screen, drag a System Requirement onto the appropriate Stakeholder Requirement. Select *Make Link from Start* to establish the link. Repeat to create all necessary links.

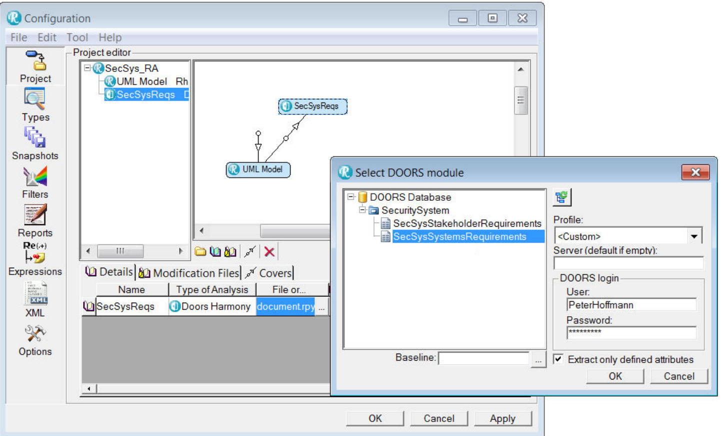


© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.3.4 DOORS -> Gateway -> Rhapsody: Import of System Requirements

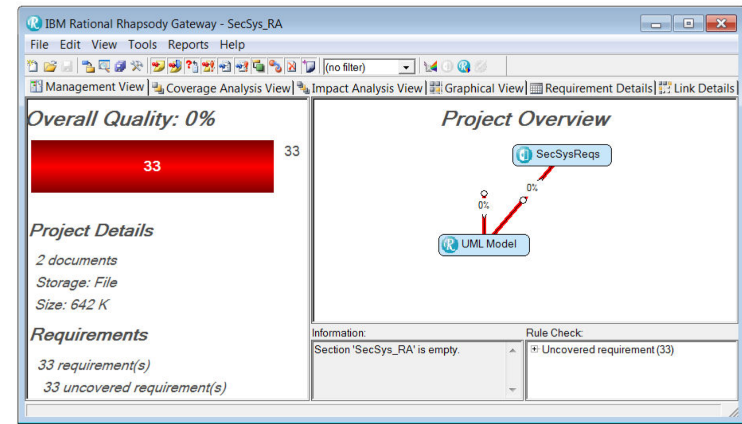
The import of the system requirements from DOORS into Rhapsody is performed in two steps. First the requirements are imported into the Rhapsody Gateway tool. Then the requirements are imported from Gateway into Rhapsody, i.e. the Rhapsody project **SecSys_RA**.

1. Create a *Harmony* compliant Rhapsody project (ref. Section 4.2) and name it **SecSys_RA.**
2. In the Rhapsody browser right-click **SecSys_RA** and select *Rational Rhapsody Gateway / Open*



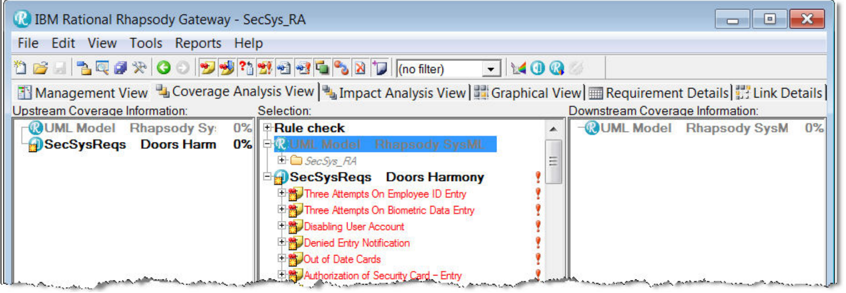
Import Requirements into Gateway

3. Select *File / Edit Project* to open the configuration window
4. *Add a document to the canvas* and name it SecSysReq
5. Add a coverage link from UML Model to SecSysReq
6. Select as *Type of Analysis* **DOORS Harmony**
7. Select in the *File or ..* pop-up window *DOORS Database / SecuritySystem / SecSysRequirements*
8. Tick *Extract only defined attributes* and confirm (OK)
9. Confirm the Configuration setup (OK)

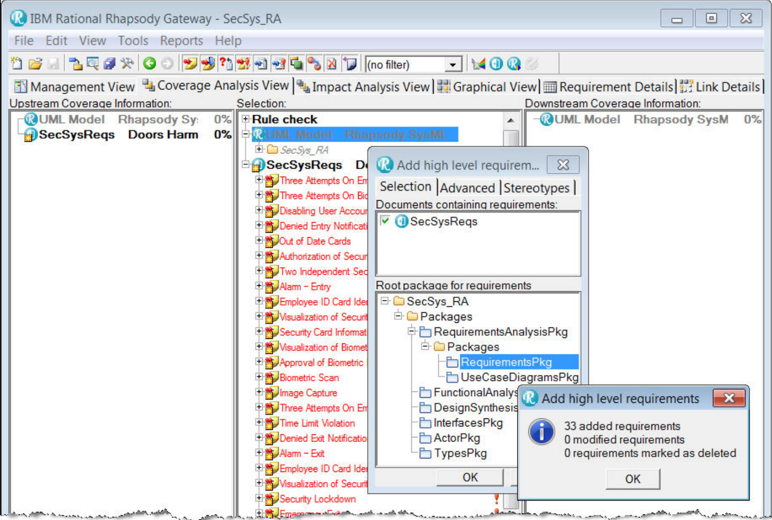


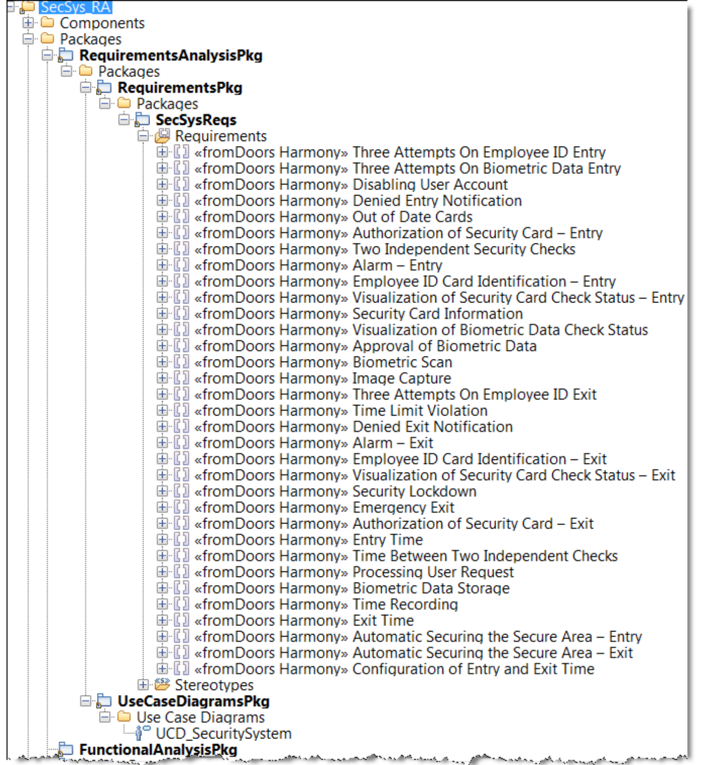
Import Requirements into Rhapsody

10. In the Coverage Analysis View select *UML Model* and *Tools / Add high level requirements*
11. In the pop-up window select as *Root package for requirements* RequirementsPkg
12. Confirm setup (OK)



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.





Imported System Requirements in SecSys_RA

4.3.5 Definition of System-Level Use Cases

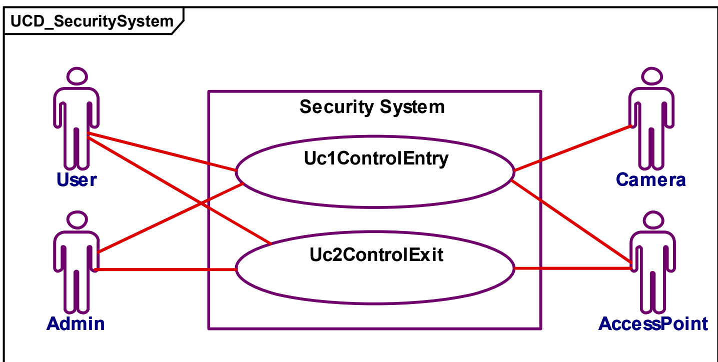
The system requirements of the Security System are grouped into two use cases:

- Uc1ControlEntry
- Uc2ControlExit

The associated actors are

- User
- Administrator
- Camera and
- Access Point

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



Use Case Diagram of the Security System

In order to support the update in case of requirements changes, define the use cases as *Units*:

Right-click the use case and select *Create Unit*.

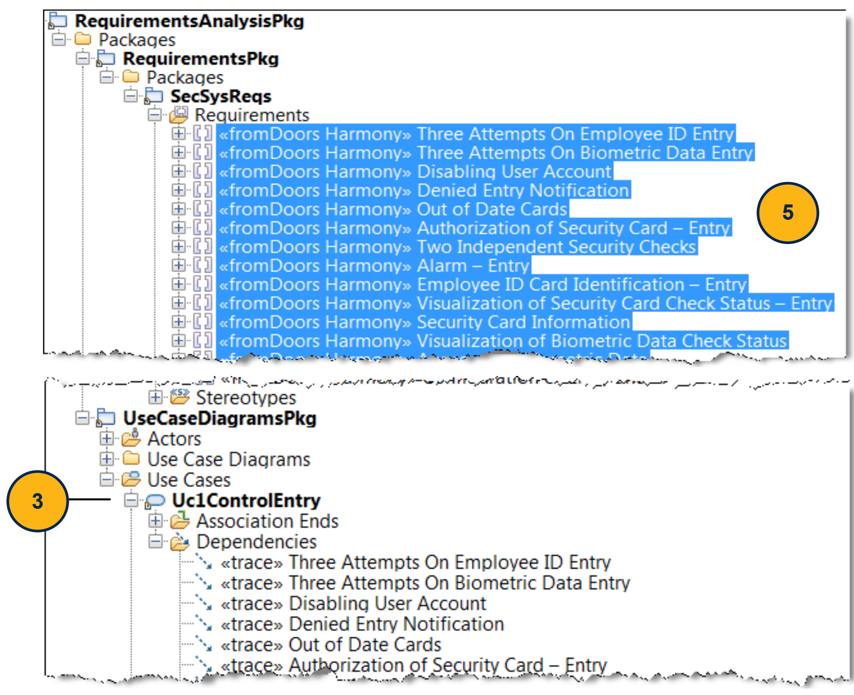
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.3.5.1 Linking Requirements to Use Cases

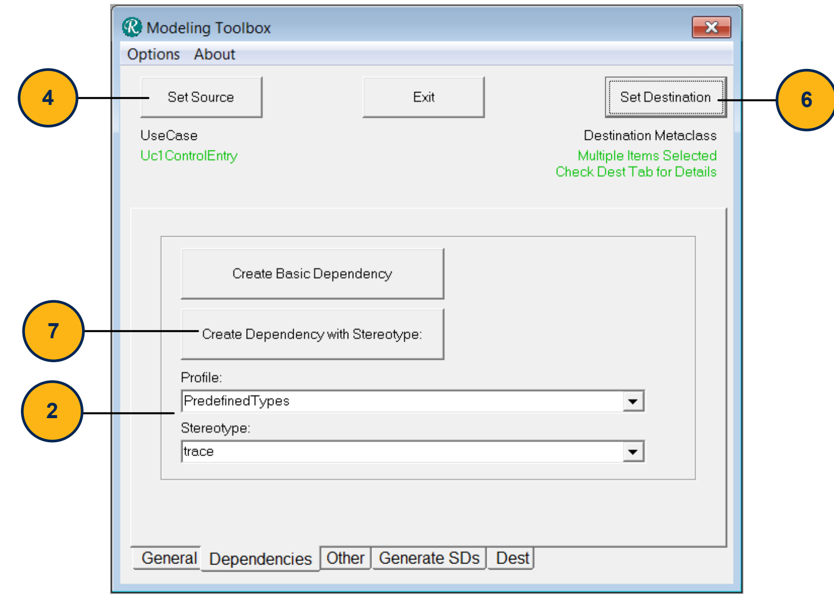
System functional and non-functional system requirements are linked to the use case with a <<trace>> dependency by means of the SE-Toolkit feature ***Create Dependency***.

<u>*NOTE:*</u> A system requirement may be associated to more than one use case.

Exemplary, the linking process is shown for the use case Uc1ControlEntry.



1. In the Tools Menu select *Tools > SE-Toolkit > Modeling Toolbox*
2. In the dialog box select *Dependencies*. Select Profile: *PredefinedTypes* Select Stereotype: *trace*.
3. In the UseCaseDiagramsPkg select use case Uc1ControlEntry.
4. In the ModelingToolbox dialog box click *Set Source.*
5. In the SecSysReq package select the requirements the use case is linked to.
6. In the ModelingToolbox dialog box click *Set Destination.*
7. In the ModelingToolbox dialog box click *Create Dependency with Stereotype.*



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Visualization of the Use Case Links to the Functional / Non-Functional System Requirements (Matrix View)

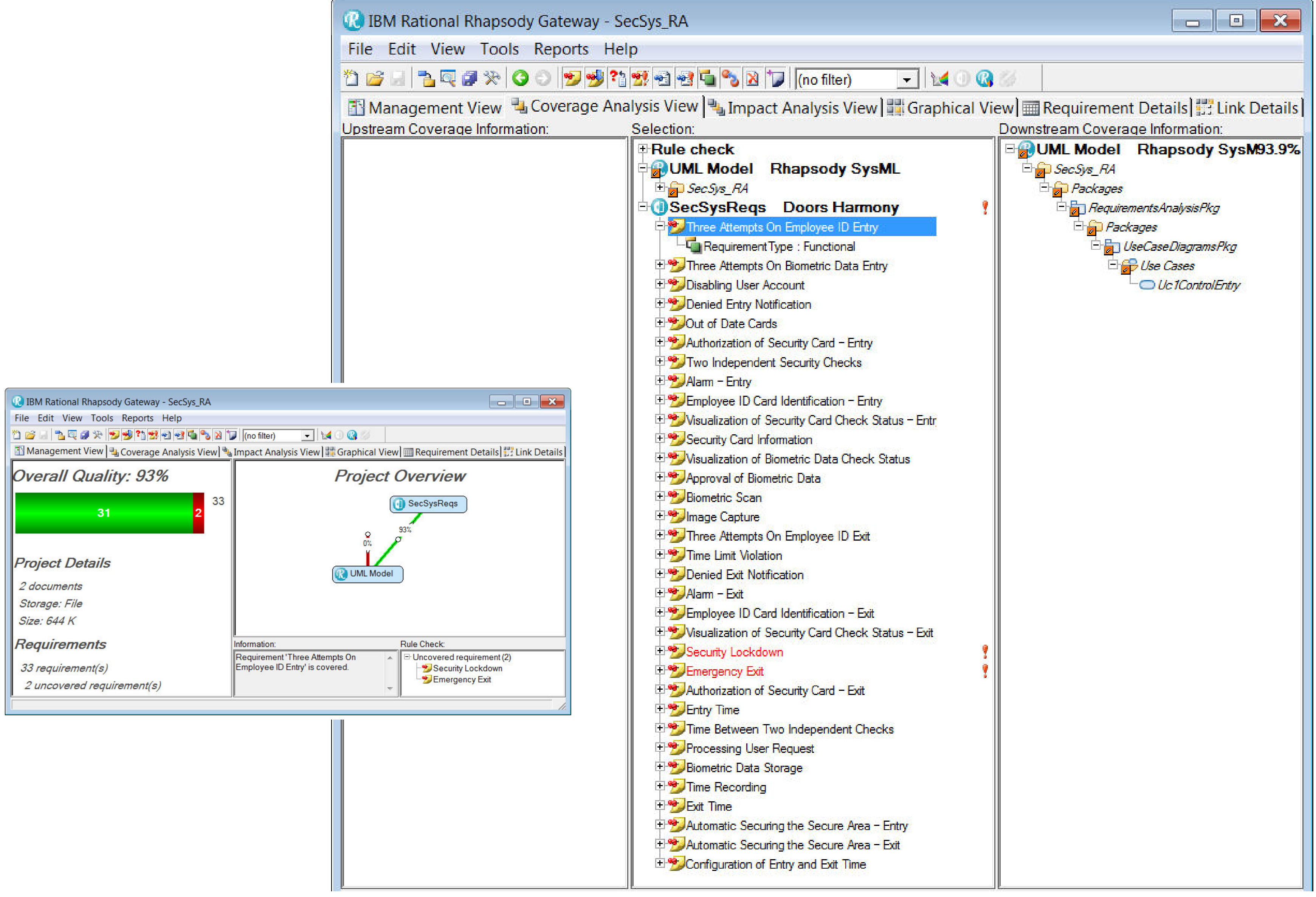
From: UseCase **Scope:** UseCaseDiagramsPkg

To: Requirement **Scope:** SecSysReq

	Uc1ControlEntry	Uc2ControlExit
Three Attempts On Employee ID Entry	Three Attempts On Employee ID Entry	
Three Attempts On Biometric Data Entry	Three Attempts On Biometric Data Entry	
Disabling User Account	Disabling User Account	
Denied Entry Notification	Denied Entry Notification	
Out of Date Cards	Out of Date Cards	
Authorization of Security Card - Entry	Authorization of Security Card - Entry	
Two Independent Security Checks	Two Independent Security Checks	
Alarm - Entry	Alarm - Entry	
Employee ID Card Identification - Entry	Employee ID Card Identification - Entry	
Visualization of Security Card Check Status - Entry	Visualization of Security Card Check Status - Entry	
Security Card Information	Security Card Information	Security Card Information
Visualization of Biometric Data Check Status	Visualization of Biometric Data Check Status	
Approval of Biometric Data	Approval of Biometric Data	
Biometric Scan	Biometric Scan	
Image Capture	Image Capture	
Three Attempts On Employee ID Exit		Three Attempts On Employee ID Exit
Time Limit Violation		Time Limit Violation
Denied Exit Notification		Denied Exit Notification
Alarm - Exit		Alarm - Exit
Employee ID Card Identification - Exit		Employee ID Card Identification - Exit
Visualization of Security Card Check Status - Exit		Visualization of Security Card Check Status - Exit
Security Lockdown		
Emergency Exit		
Authorization of Security Card - Exit		Authorization of Security Card - Exit
Entry Time	Entry Time	
Time Between Two Independent Checks	Time Between Two Independent Checks	
Processing User Request	Processing User Request	Processing User Request
Biometric Data Storage	Biometric Data Storage	
Time Recording		Time Recording
Exit Time		Exit Time
Automatic Securing the Secure Area - Entry	Automatic Securing the Secure Area - Entry	
Automatic Securing the Secure Area - Exit		Automatic Securing the Secure Area - Exit
Configuration of Entry and Exit Time	Configuration of Entry and Exit Time	Configuration of Entry and Exit Time

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Visualization of the Requirements Coverage in Gateway

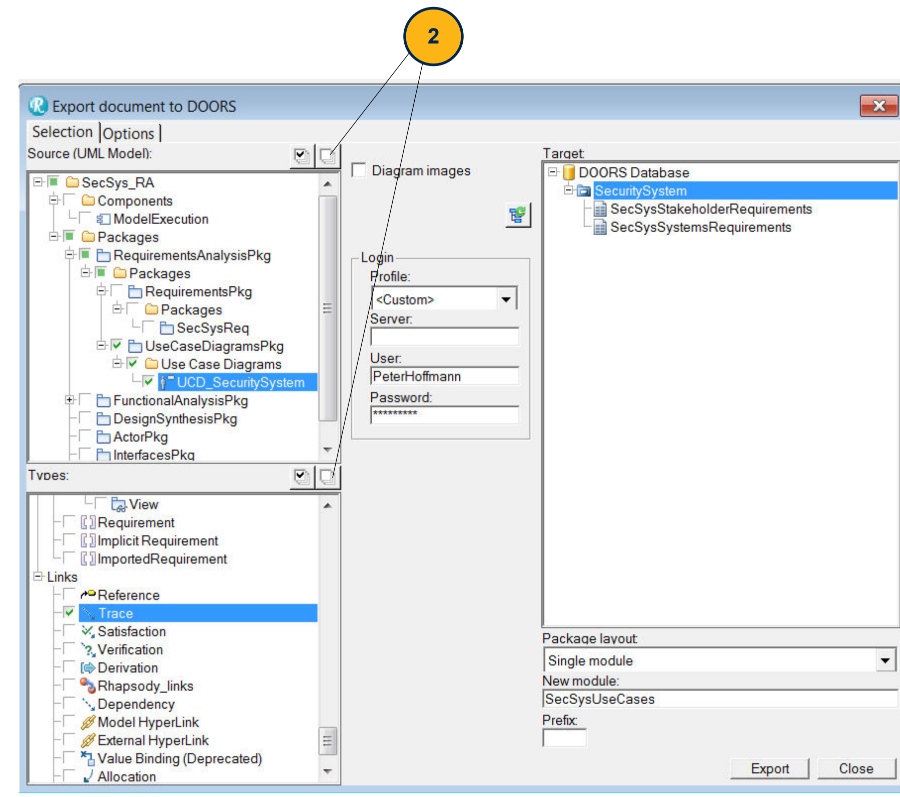


NOTE: The system requirements *Security Lockdown* and *Emergency Exit* intentionally were not linked to any use case

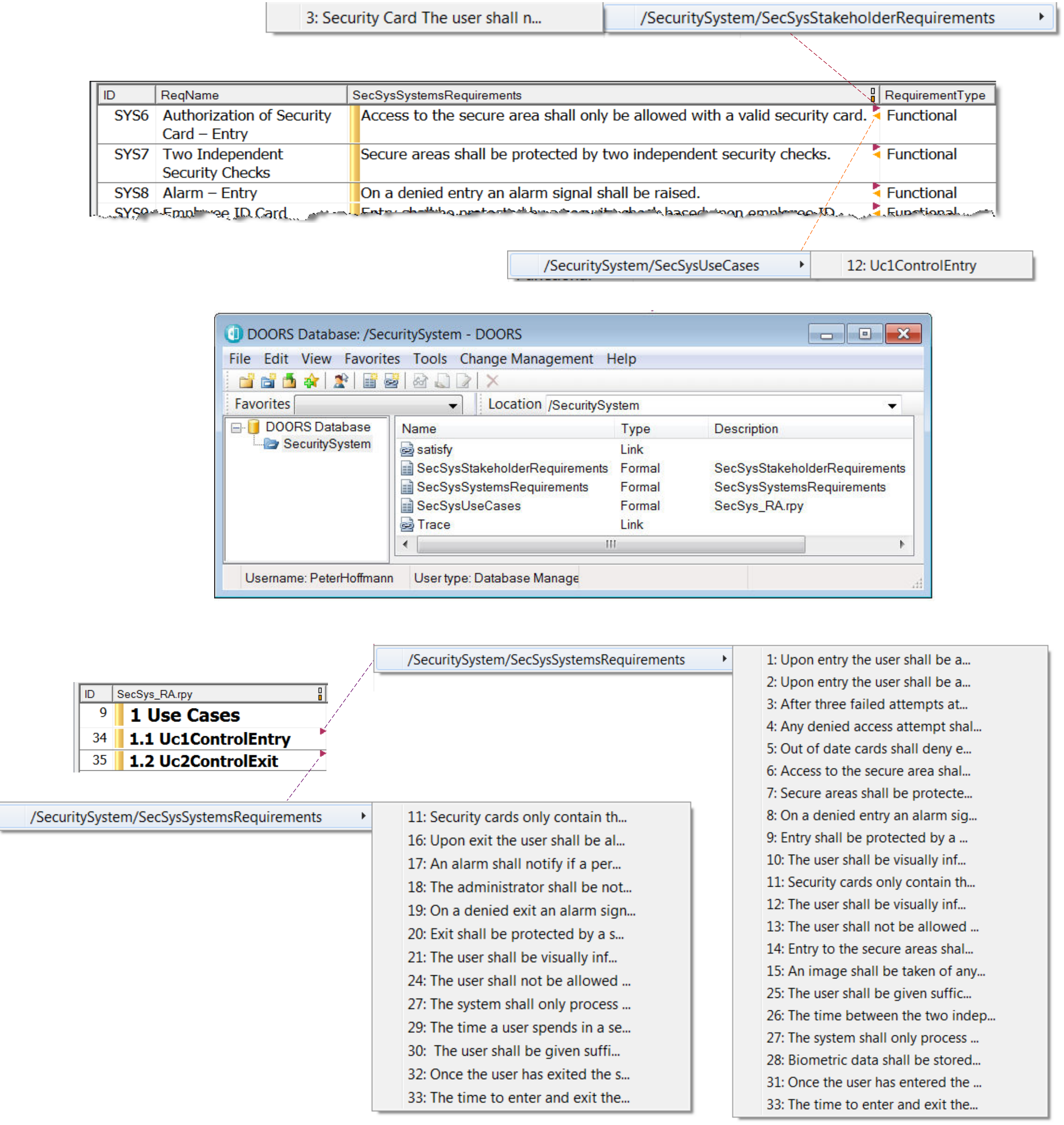
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.3.6 Rhapsody -> Gateway -> DOORS: Export of Use Cases

1. In the Coverage Analysis View select *UML Model* and *Tools / Export Documents to DOORS*
2. In the *Source (UML Model)* window and *Types* window *Deselect all* links
3. In the *Source (UML Model)* window select *UCD_SecuritySystem*
4. In the *Types* window select
   - *Elements / Use Case* and
   - *Links / Trace*
5. Click *Update Tree* button and select *SecuritySystem*
6. Define as DOORS *New module* **SecSysUseCases**
7. Click *Export*



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



DOORS Database and Links between Stakeholer Requirements, System Requirements and System Use Cases

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.4 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 validated 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 2.2.2 .

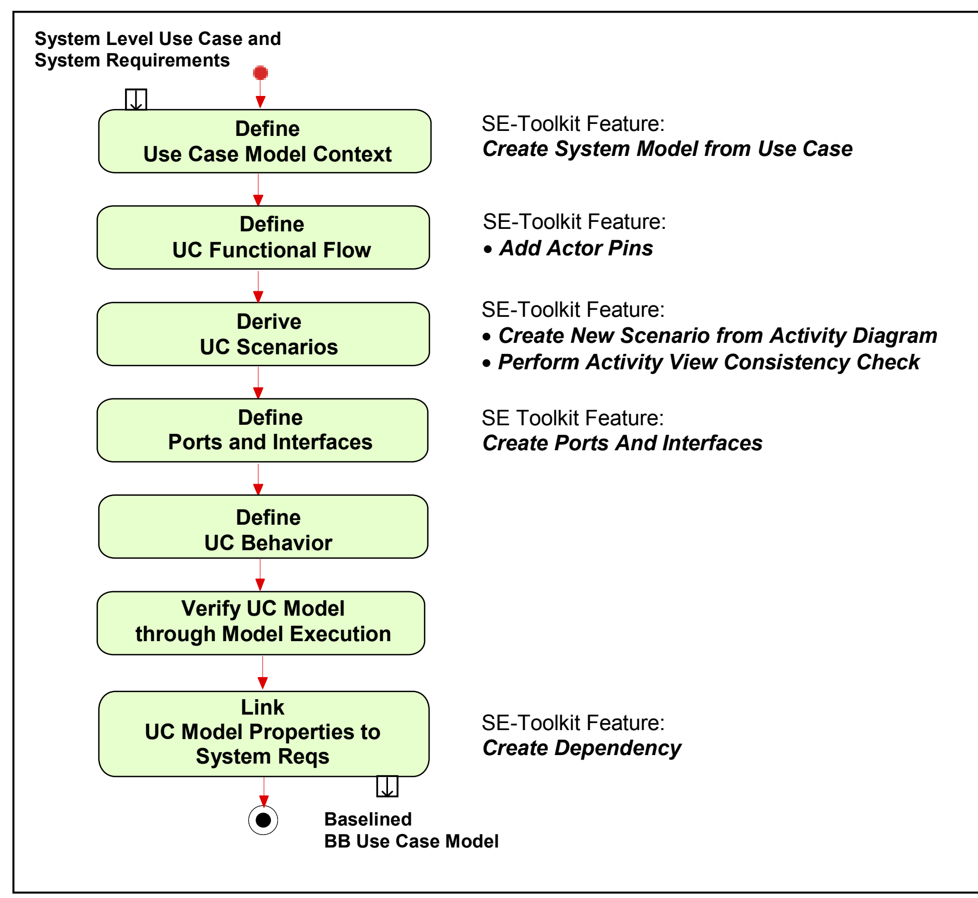


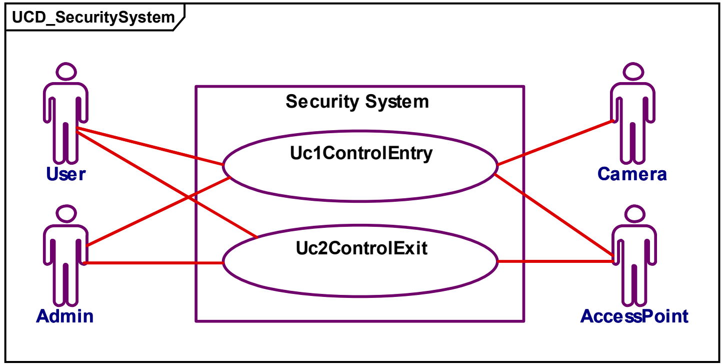
Fig. 4-3 System Functional Analysis Workflow and its Support through the Rhapsody SE-Toolkit

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.4.1 Uc1ControlEntry

4.4.1.1 Definition of Model Context

The elaboration of of the use case Uc1ControlEntry will be performed in a separate Rhapsody project.



Imported UCD_SecuritySystem

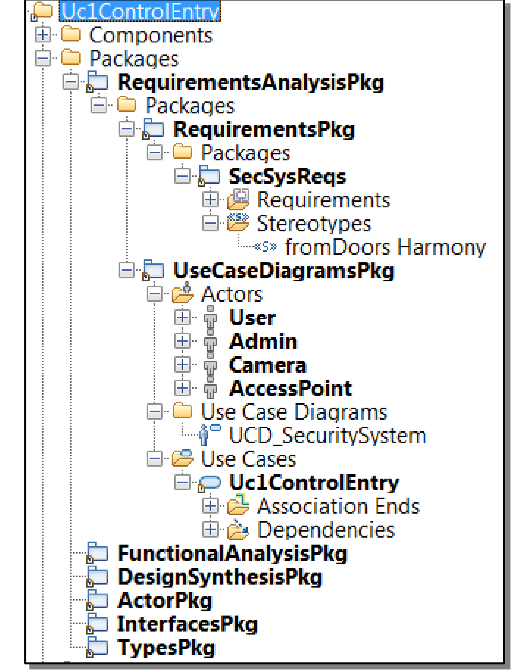
1. Create a *Harmony* compliant Rhapsody project and name it **Uc1ContolEntry**

2. In the Rhapsody main menu select *File > Add to Model As unit* navigate to the SecSys_RA project and double-klick *SecSys_RA.rpy*

3. In the dialog box select
   - UseCaseDiagramsPkg.sbs,
   - RequirementsPkg.sbs

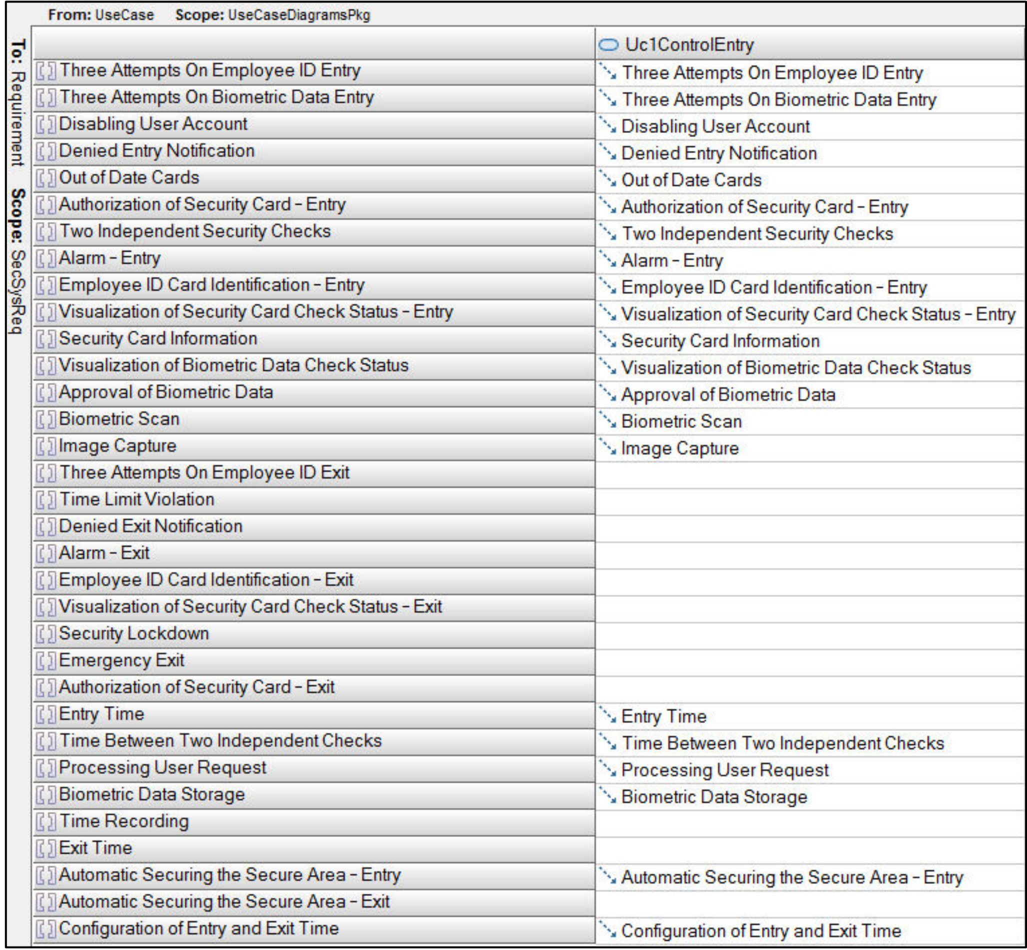
4. Click Ok,

In the imported use case diagram UCD_SecuritySystem you may *Delete from Model* the use case Uc2ControlExit.



Uc1ConrolEntry-focused Project Structure

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



From: UseCase Scope: UseCaseDiagramsPkg

To: Requirement Scope: SecSysReq

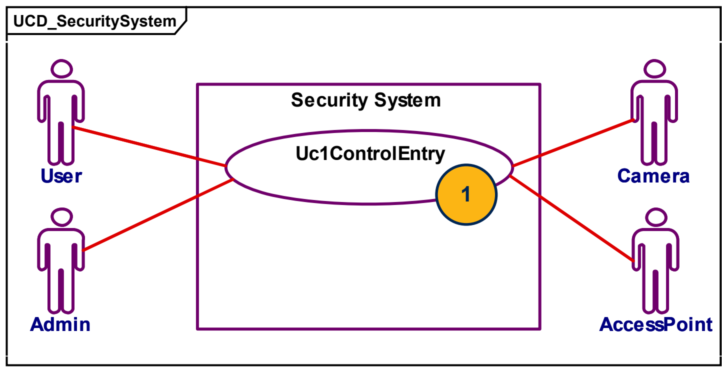
	Uc1ControlEntry
Three Attempts On Employee ID Entry	Three Attempts On Employee ID Entry
Three Attempts On Biometric Data Entry	Three Attempts On Biometric Data Entry
Disabling User Account	Disabling User Account
Denied Entry Notification	Denied Entry Notification
Out of Date Cards	Out of Date Cards
Authorization of Security Card - Entry	Authorization of Security Card - Entry
Two Independent Security Checks	Two Independent Security Checks
Alarm - Entry	Alarm - Entry
Employee ID Card Identification - Entry	Employee ID Card Identification - Entry
Visualization of Security Card Check Status - Entry	Visualization of Security Card Check Status - Entry
Security Card Information	Security Card Information
Visualization of Biometric Data Check Status	Visualization of Biometric Data Check Status
Approval of Biometric Data	Approval of Biometric Data
Biometric Scan	Biometric Scan
Image Capture	Image Capture
Three Attempts On Employee ID Exit	
Time Limit Violation	
Denied Exit Notification	
Alarm - Exit	
Employee ID Card Identification - Exit	
Visualization of Security Card Check Status - Exit	
Security Lockdown	
Emergency Exit	
Authorization of Security Card - Exit	
Entry Time	Entry Time
Time Between Two Independent Checks	Time Between Two Independent Checks
Processing User Request	Processing User Request
Biometric Data Storage	Biometric Data Storage
Time Recording	
Exit Time	
Automatic Securing the Secure Area - Entry	Automatic Securing the Secure Area - Entry
Automatic Securing the Secure Area - Exit	
Configuration of Entry and Exit Time	Configuration of Entry and Exit Time

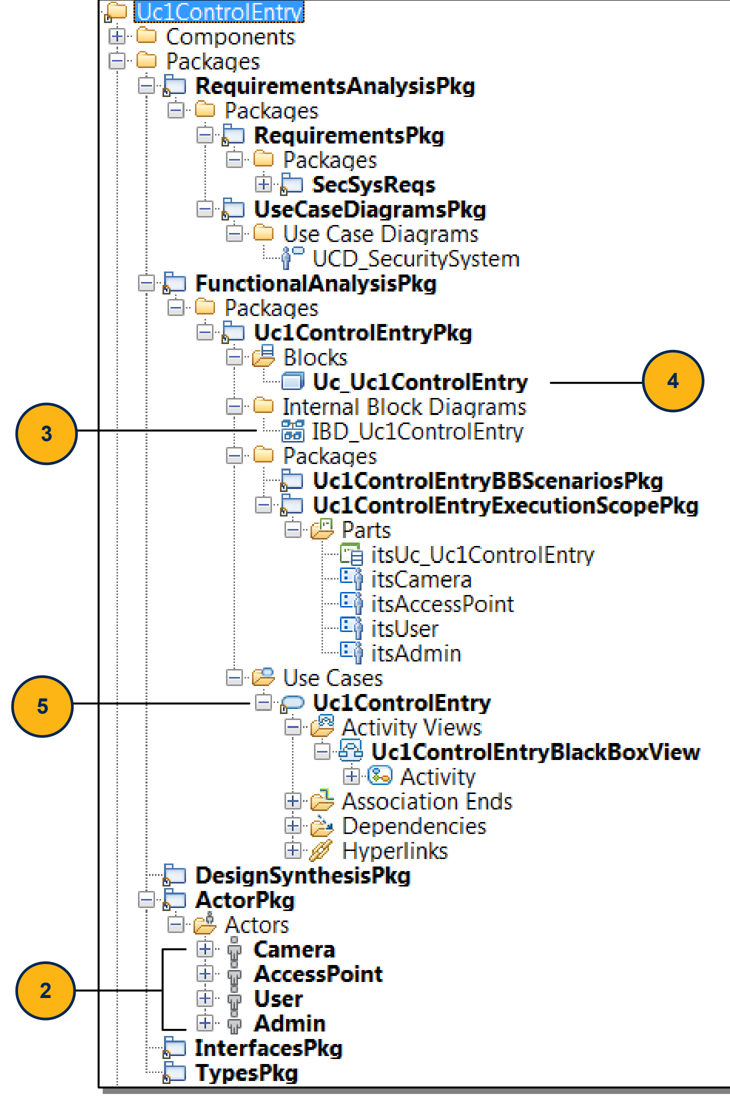
Uc1ControlEntry related System Requirements

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A Functional Analysis project structure that complies with the recommended one outlined in Section 3.3, may be created automatically by means of the SE-Toolkit feature ***Create System Model From Use Case***.

1 Right-click use case Uc1ControlEntry 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.

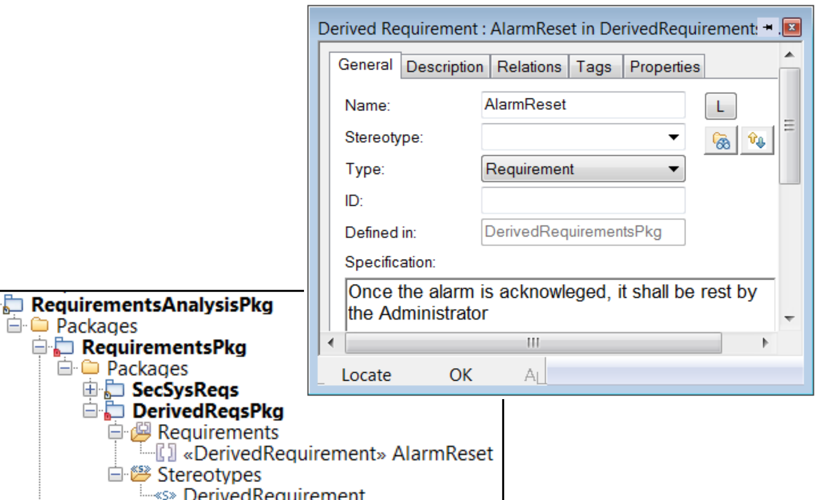
5 The use case - incl. its requirements links – is moved into the Uc1ControlEntryPkg in the FunctionalAnalysisPkg. Additionally, the Toolkit feature created an empty Activity Diagram (**Uc1ControlEntryBlackBoxView**).

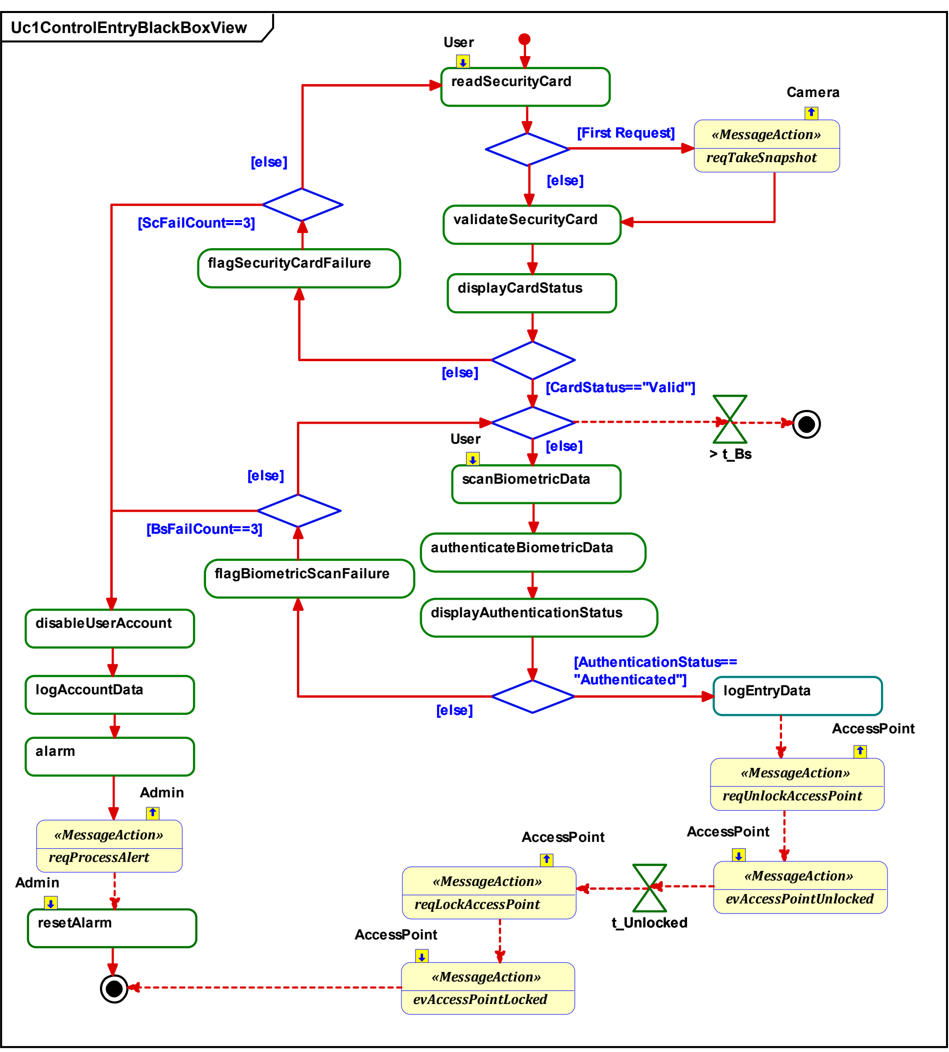
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.4.1.2 Definition of 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.

<u>*NOTE*</u>: 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 – temporarily (!) - in the ***DerivedRequirementsPkg***.





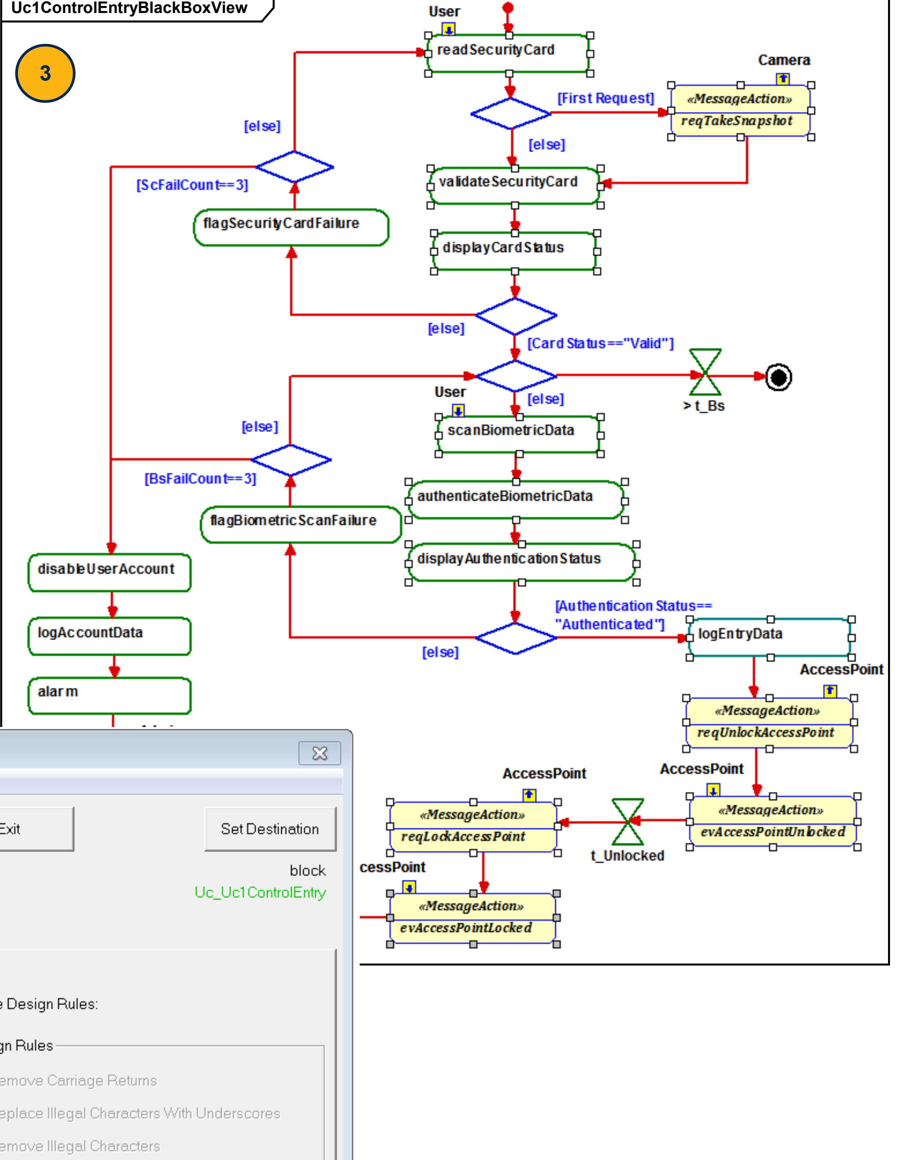
If an activity diagram contains too many details, some actions may be placed in a Reference Activity Diagram. Do not use SubActivity Diagrams because these can contain actions only for a single swim lane. In the later white-box activity diagrams the actions may span a number of swim lanes.

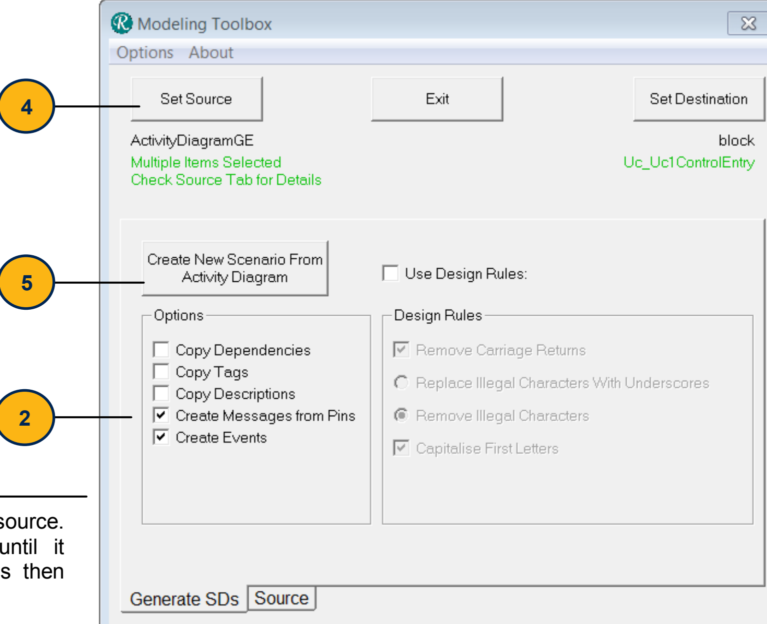
 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.4.1.3 *Derivation of Black-Box Use Case Scenarios*

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

1. In the activity diagram window right-click and select *SE-Toolkit > Generate Sequence Diagram.*
2. In the ModelingToolbox dialog box tick *Create Messages from Pins* and *Create Events.*
3. Hold down Ctrl and select in the black-box activity diagram a sequence of actions. [1])
4. In the ModelingToolbox dialog box click *Set Source.*
5. In the ModelingToolbox dialog box click *Create New Scenario From Activity Diagram.*

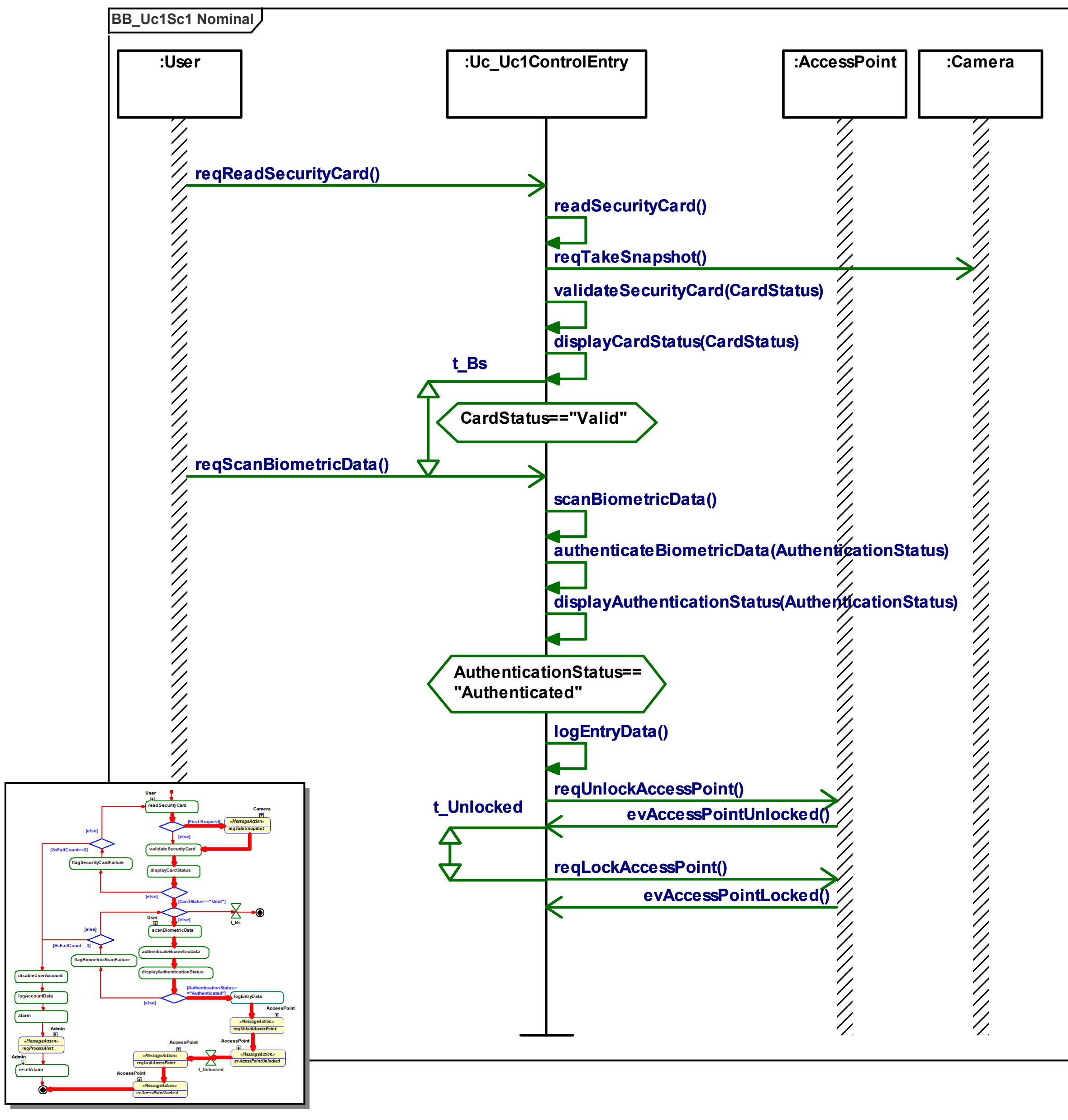




NOTE:
The created Sequence Diagram is automatically stored in the Uc1ControlEntryBBScenariosPgk

[1]) 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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



Derived Use Case Scenario BB_Uc1Sc1 Nominal

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

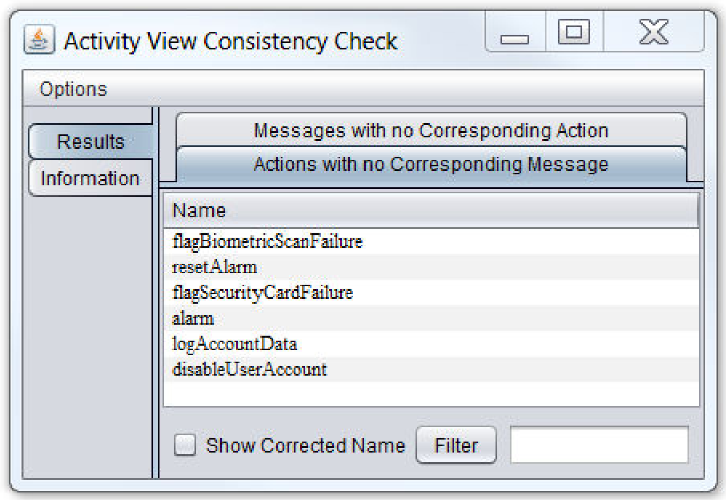
Activity View Consistency Check

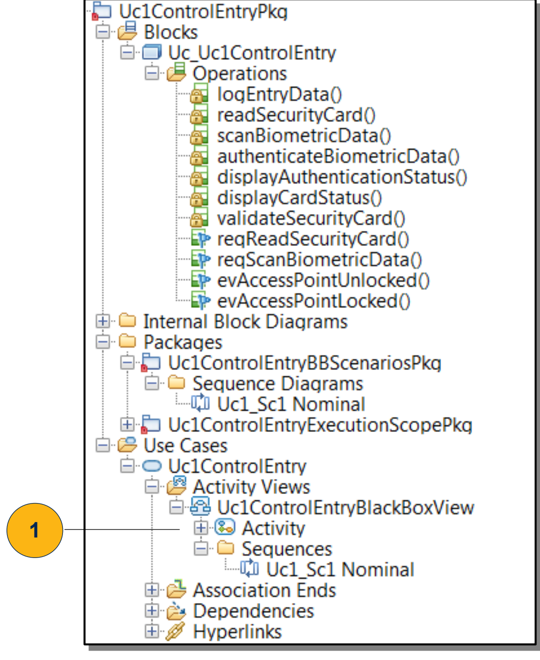
The consistency between actions of the black-box Activity Diagram and the operations in the derived use case scenarios may be checked by means of the of the SE-Toolkit feature ***Perform Activity View Consistency Check.***

For the selected Activity View the feature checks whether

- 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

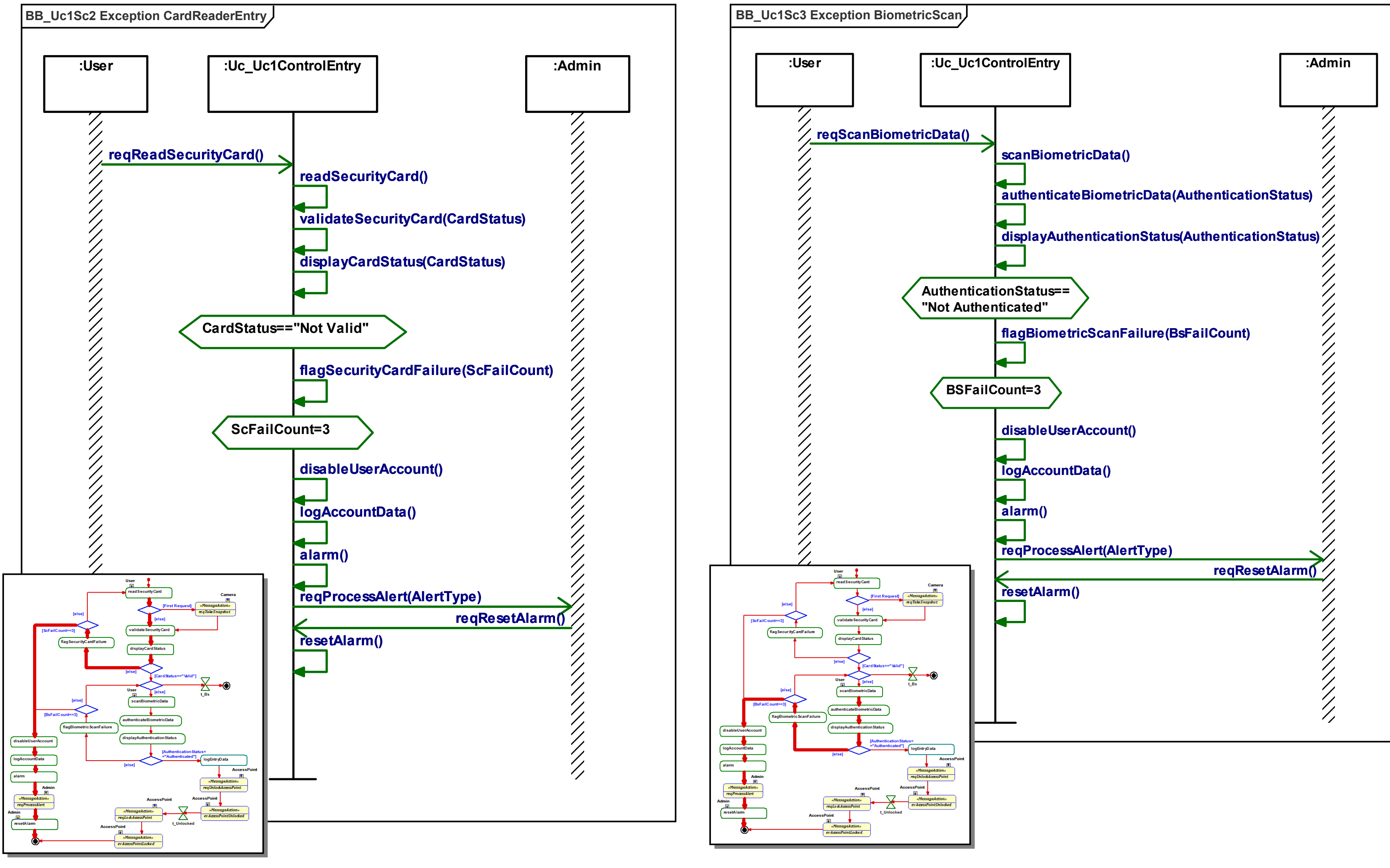
1 Right-click Uc1_ControlEntryBlackBoxView > Activity and select *SE-Toolkit > Perform Activity View Consistency Check*.





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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



Derived Use Case Scenario BB_Uc1Sc2 Exception CardReaderEntry

Derived Use Case Scenario BB_Uc1Sc3 Exception BiometricScan

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

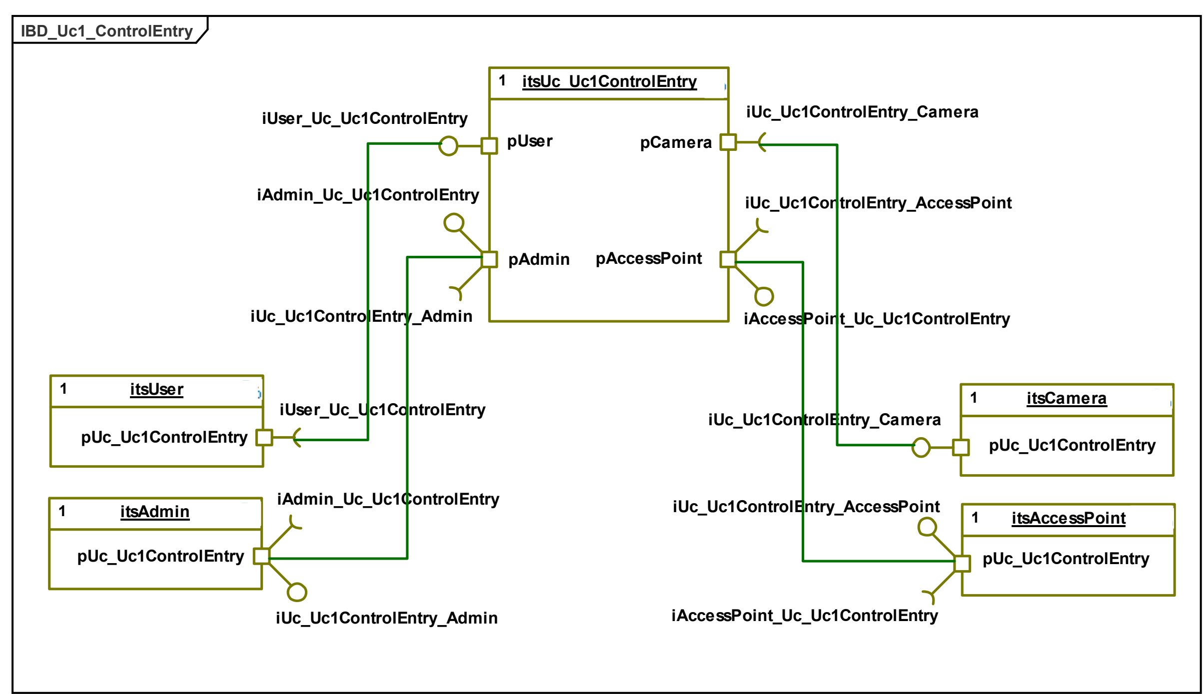
4.4.1.4 Definition of Ports and Interfaces

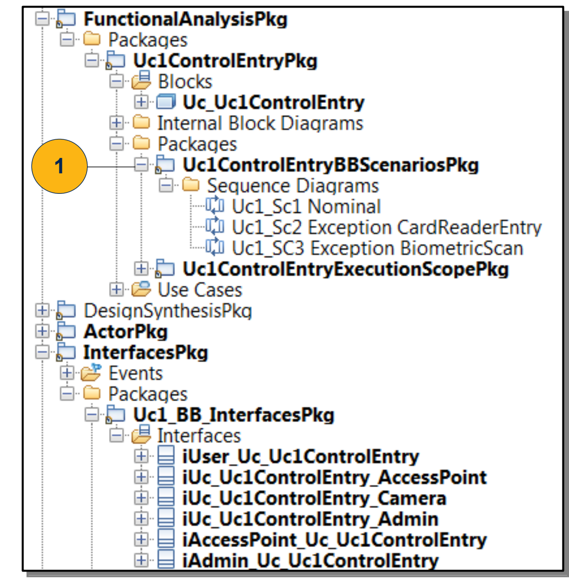
The definition of ports and associated interfaces is automated in *Rhapsody* by means of the SE-Toolkit feature **Create *Ports And Interfaces***. Pre-condition: All messages and operations in the sequence diagrams are *realized*.

Naming convention for ports: **p**<*Target Name*>
Interface names are referenced to the sender port.
Naming convention: **i**< *Sender* >_< *Receiver* >

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

2. Connect ports either manually or right-click in the IBD and select *SE-Toolkit > Connect Ports*





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

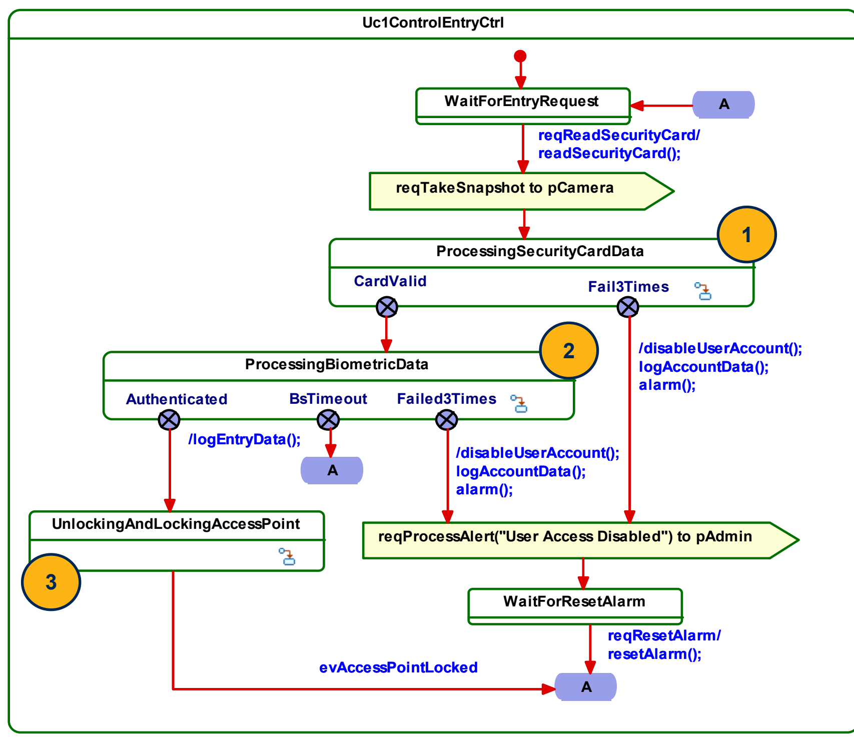
Internal Block Diagram IBD_Uc1_ControlEntry with Ports and Interfaces

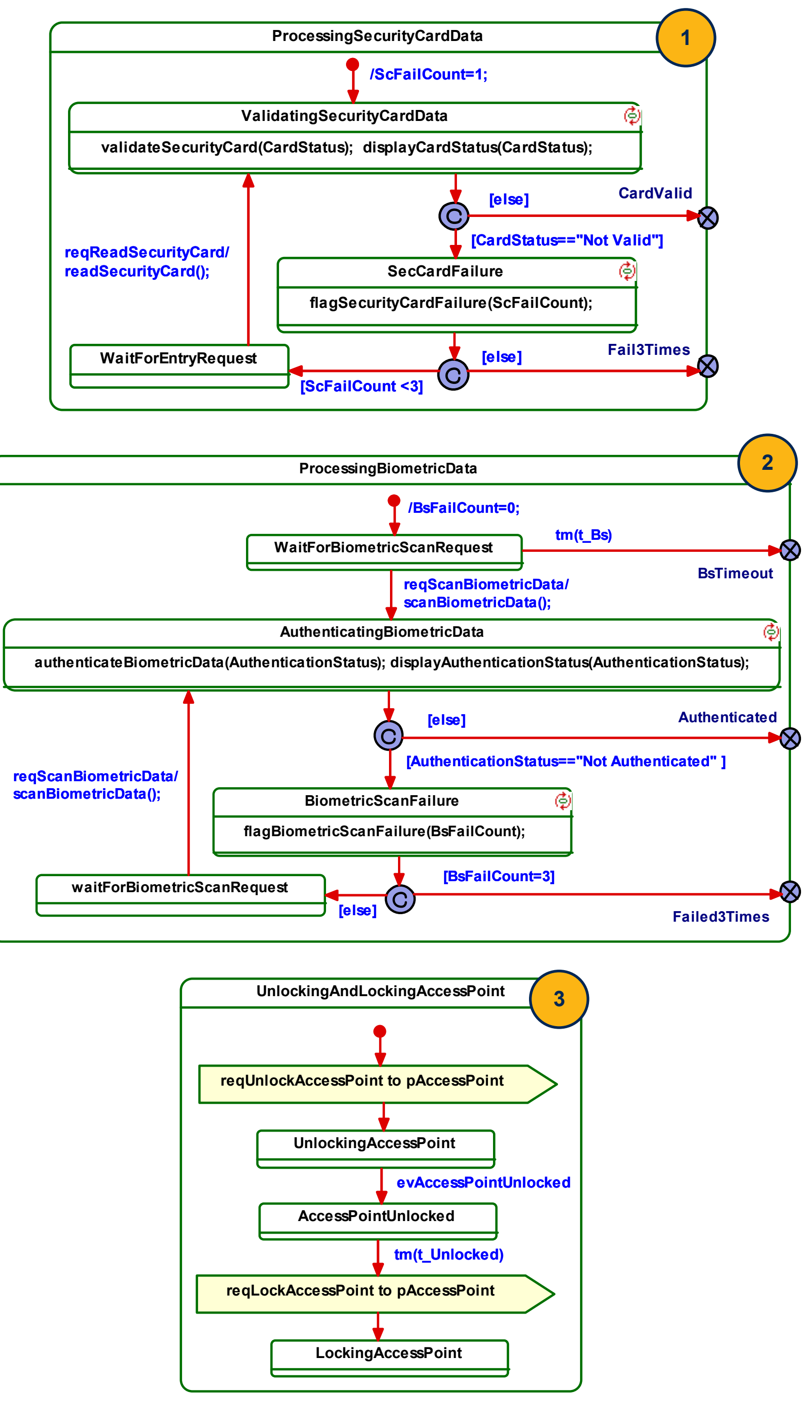
For readability reasons it is recommended not to show the interface names in the diagram. Deselect in each block the Display Option *Show Port Interfaces.*

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.4.1.5 Definition of Use Case Behavior

The state-based behavior of the use case block is described by a *Statechart Diagram*. 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.





A statechart should be hierarchically structured. This allows the reuse of behavior-patterns in later phases (e.g. Use Case Realization).

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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

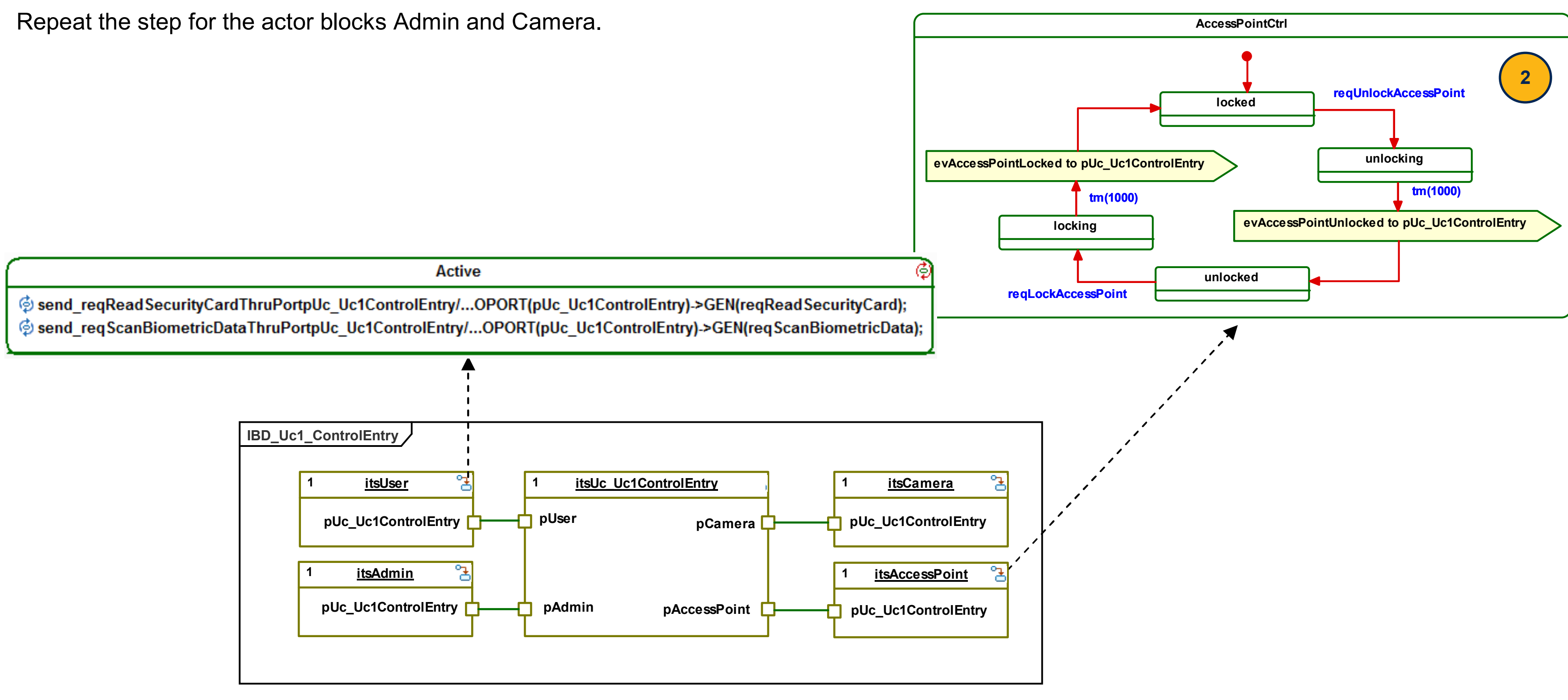
In the use case Internal Block Diagram right-click the User block and select ***SE-Toolkit / Create Test Bench***.

This toolkit feature captures the User behavior in one state (**Active**) using MOORE syntax (= action in state). and includes already the capability to run model execution via Webify.

Repeat the step for the actor blocks Admin and Camera.

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

1. In the use case Internal Block Diagram right-click the AccessPoint block and select *Class / New Statechart.*
2. Capture manually the actor behavior in a state machine.



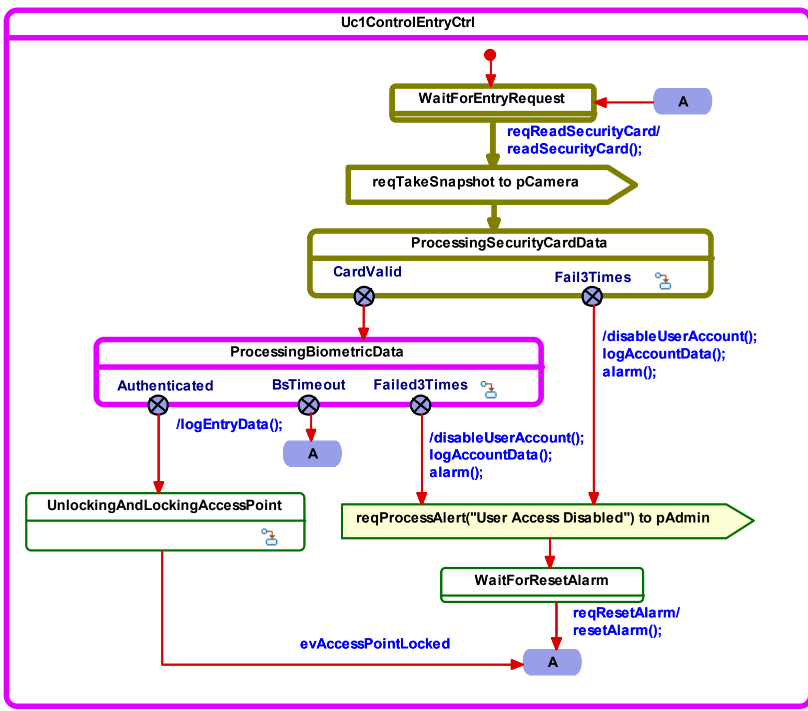
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.4.1.6 Use Case Model Verification

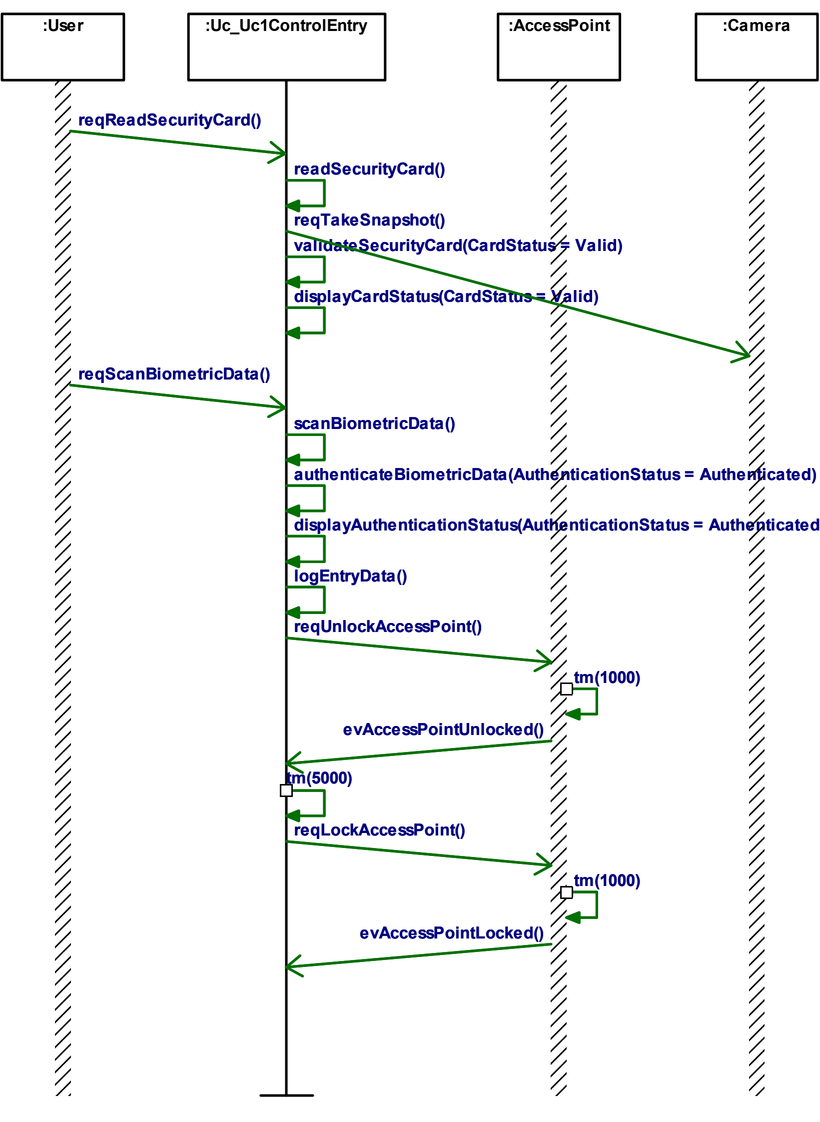
The Uc1ControlEntry model is verified through model execution on the basis of the captured use case scenarios. The correctness and completeness analysis is based on the visual inspection of the model behavior.

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 Statechart Diagram (Uc1ControlEntryCtrl)

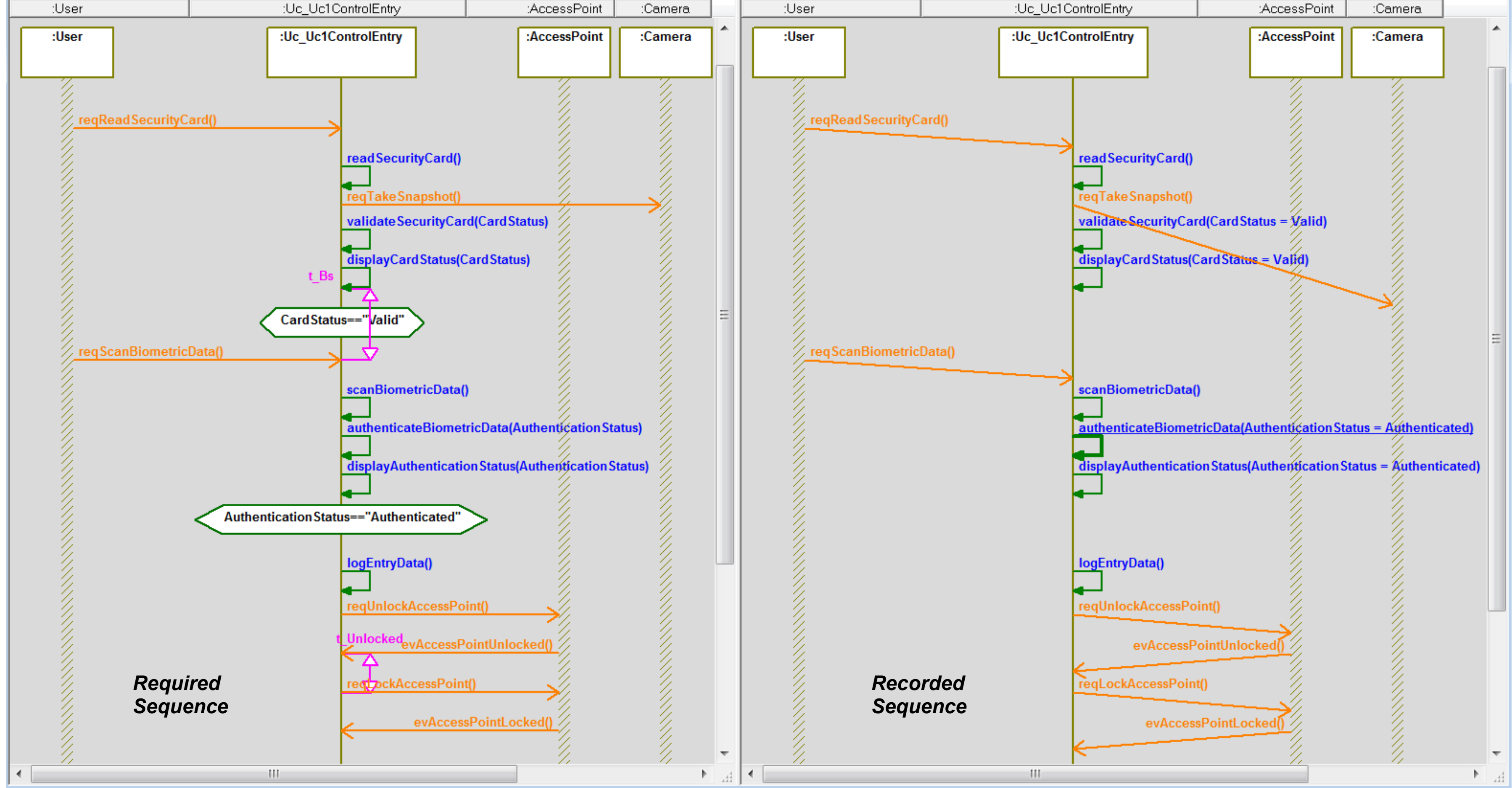


Animated Sequence Diagram BB_Uc1Sc1 Nominal

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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.

Arrow Color	Name Color	Description
Green	Blue	Msg matches in both SD
Pink	Pink	Msg is missing in the other SD
Green	Pink	Msg has different arguments in the other SD
Orange	Orange	Msg arrives at a different time in the other SD
Gray	Gray	Msg was excluded from comparison



Sequence Diagram Compare: Scenario BB_Uc1Sc1 Nominal

<u>NOTE</u>: Timeout Arrows were intentionally deselected via Preference Settings for Animated Sequence Diagrams

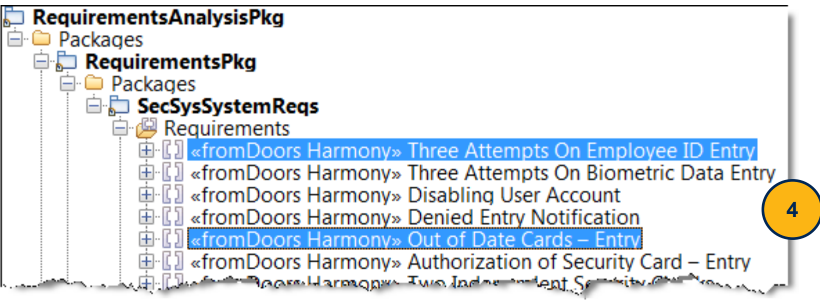
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

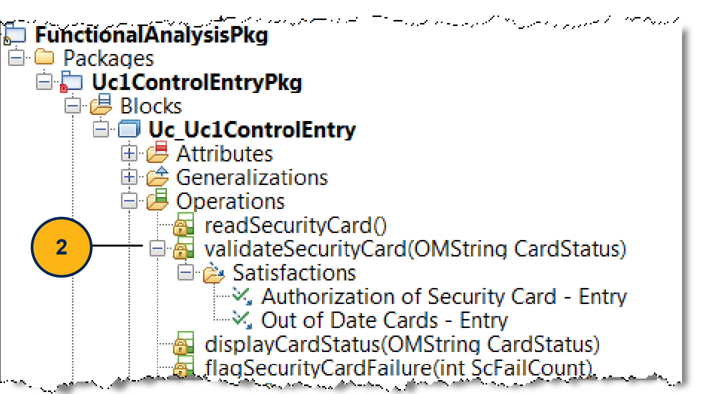
4.4.1.7 Linking Model Properties to Requirements

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

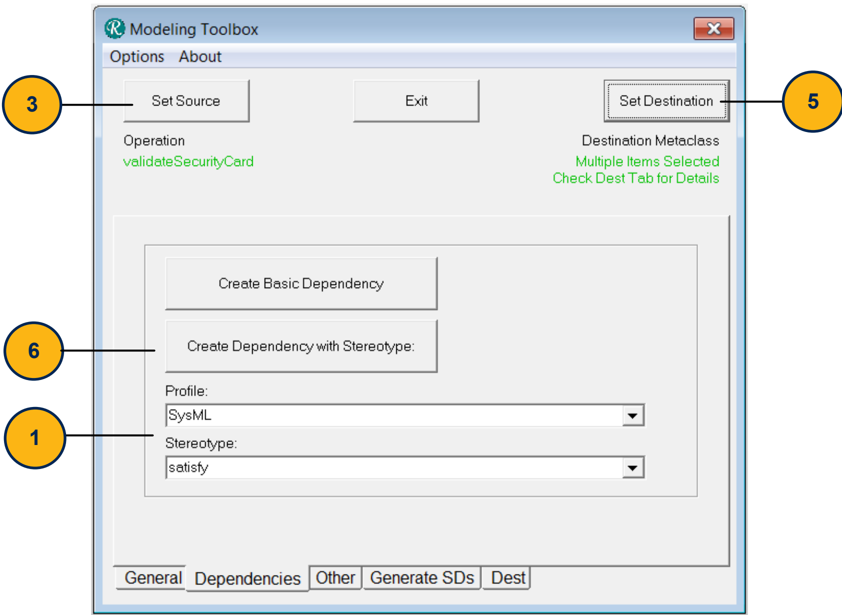
- directly in the browser using the SE-Toolkit feature ***Create Dependency***, or
- graphically, in a Requirement Diagram.

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.





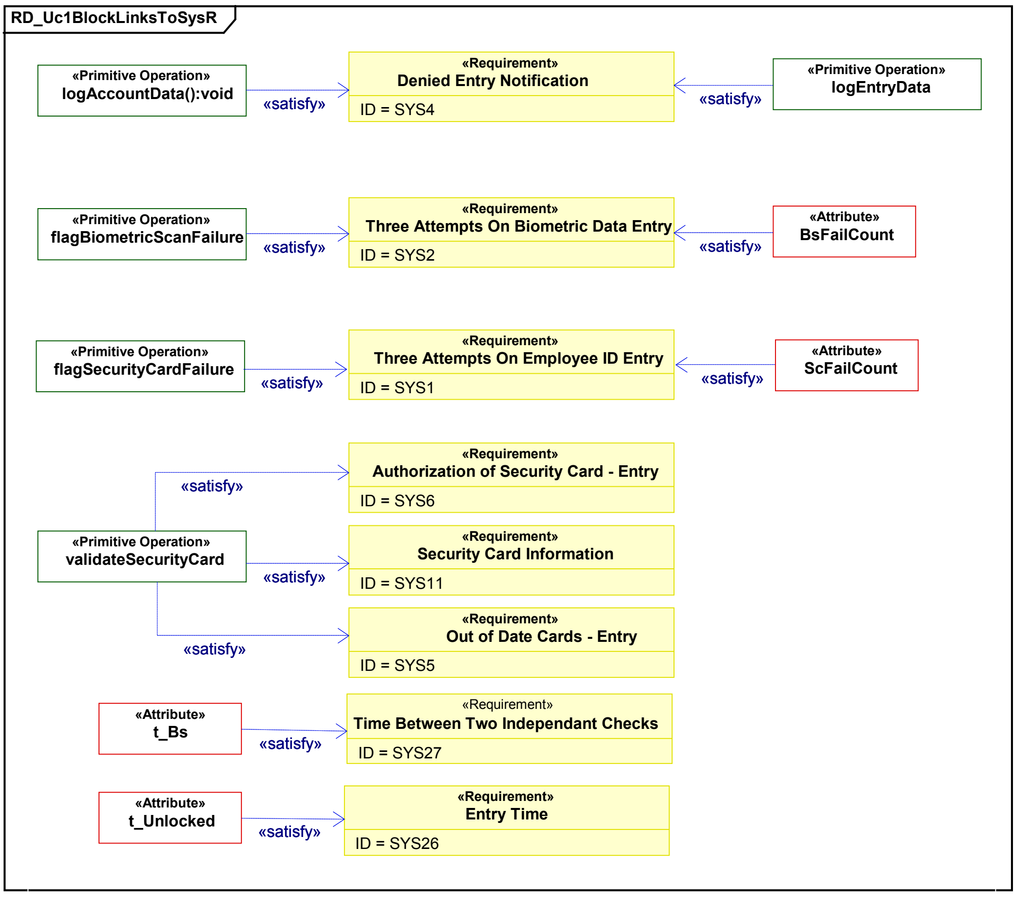
1. In the Tools Menu select *Tools > SE-Toolkit > Modeling Toolbox*
   In the dialog box select *Dependencies*.
   Select Profile: *SysML*
   Select Stereotype: *satisfy*.
2. In the system block Uc_Uc1ControlEntry select the property(s) you want to link to a system requirement.
3. In the Modeling Toolbox dialog box click *Set Source.*
4. In the SystemRequirementsPkg select the relevant system requirement(s).
5. In the ModelingToolbox dialog box click *Set Destination.*
6. In the Modeling Toolbox dialog box click *Create Dependency with Stereotype*



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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*



Uc1ControlEntry Model Properties Mapped to System Requirements (Excerpt)

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

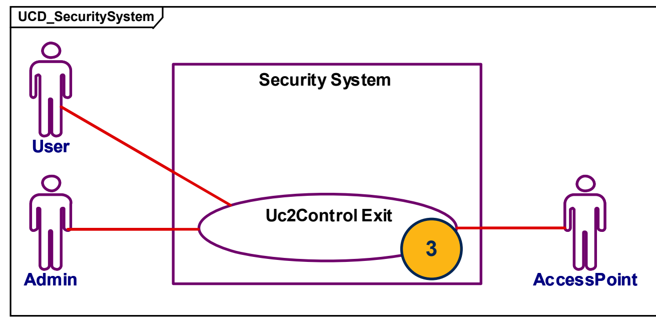
4.4.2 Uc2ControlExit

4.4.2.1 Definition of Model Context

The elaboration of of the use case Uc2ControlExit will be performed in a separate Rhapsody project **Uc2ControlExit**. The modeling starts with the import of the relevant information from the Rhapsody project **SecSys_RA** into the new project (ref. Section 4.4.1.1).

1. *Add to Model as Unit* the packages
   - UseCaseDiagramsPkg.sbs,
   - RequirementsPkg.sbs

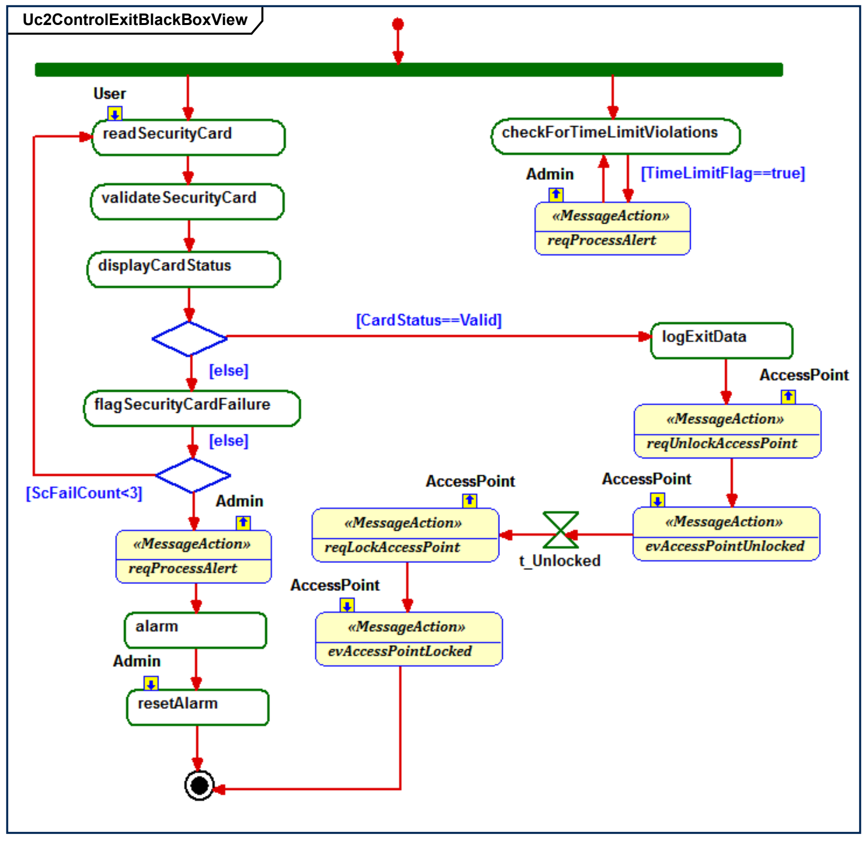
2. In the imported use case diagram UCD_SecuritySystem you may *Delete from Model* the the use case Uc1ControlEntry and the actor Camera.



3. Right-click use case Uc2Control Exit and select ***SE-Toolkit / Create System Model From Use Case.***

4.4.2.2 Definition of Functional Flow

Similar to the steps outlined in Section 4.4.1.2 the functional flow of the use case is elaborated in the Activity Diagram Uc2ControlExitBlackBoxView that was created by the toolkit feature *Create System Model From Use Case*.



Uc2ControlExit Functional Flow (Black-Box View)

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.4.2.3 Derivation of Black-Box Use Case Scenarios

The combined nominal and exception sequences are created from the black-box activity diagram by means of the SE-Toolkit feature ***Create New Scenario From Activity Diagram*** (ref. Section4.4.1.3). The toolkit automatically stores the derived Sequence Diagram in the Uc2ControlExitBBScenariosPkg.

NOTE: The Interaction Ocurences / Operand Separators as well as the Condition Mark were added manually after the generation of the Sequence Diagram

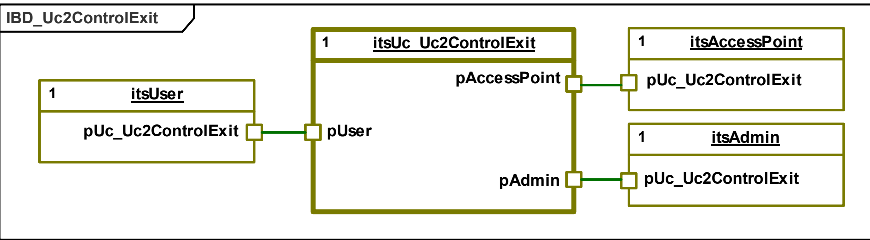
BB_Uc2Sc1 Nominal and Exception
:User
:Uc_Uc2ControlExit
:Admin
:AccessPoint
parallel
loop
checkForTimeLimitViolations(TimeLimitFlag)
opt
[TimeLimitStatus==true]
reqProcessAlert(AlertType)
reqReadSecurityCard()
readSecurityCard()
validateSecurityCard(CardStatus)
displayCardStatus(CardStatus)
alt
[CardStatus=="Not Valid"]
flagSecurityCardFailure(ScFailCount)
ScFailCount==3
reqProcessAlert(AlertType)
alarm()
reqResetAlarm()
resetAlarm()
[CardStatus=="Valid"]
logExitData()
reqUnlockAccessPoint()
t_Unlocked
evAccessPointUnlocked()
reqLockAccessPoint()
evAccessPointLocked()

Derived Use Case Scenario BB_Uc2Sc1 Nominal and Exception

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.4.2.4 Definition of Ports and Interfaces

1. Right-click the package Uc2_ControlExit_BBScenarios and select *SE-Toolkit > Create Ports And Interfaces.*
2. Connect ports either manually or right-click in the IBD and select *SE-Toolkit > Connect Ports*

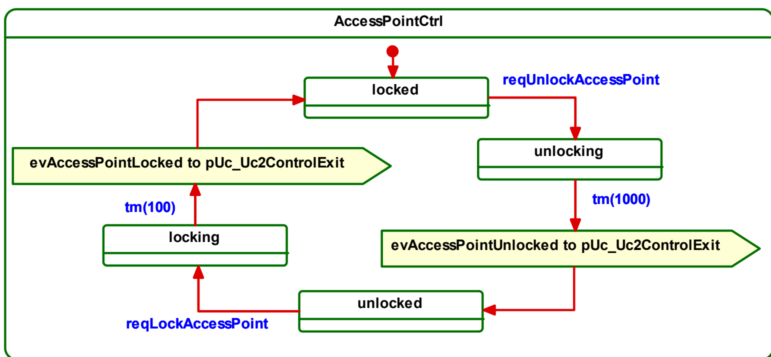


IBD of Use Case Model Uc2ControlExit with generated Ports and Interfaces

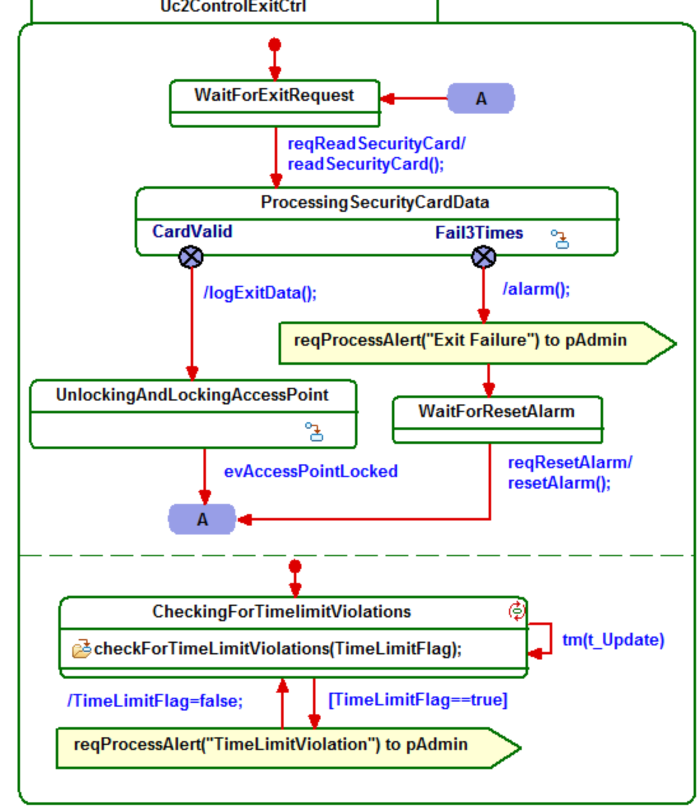
4.4.2.5 Definition of Use Case Behavior

The behavior of the actors User and AccessPoint are generated by means of the SE-Tookit feature ***Create Test Bench*** (ref. Section 4.4.1.5.). The behavior of the actor Accesspoint is manually captured in a Statechart Diagram.

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.



State-based Behavior of Actor AccessPoint

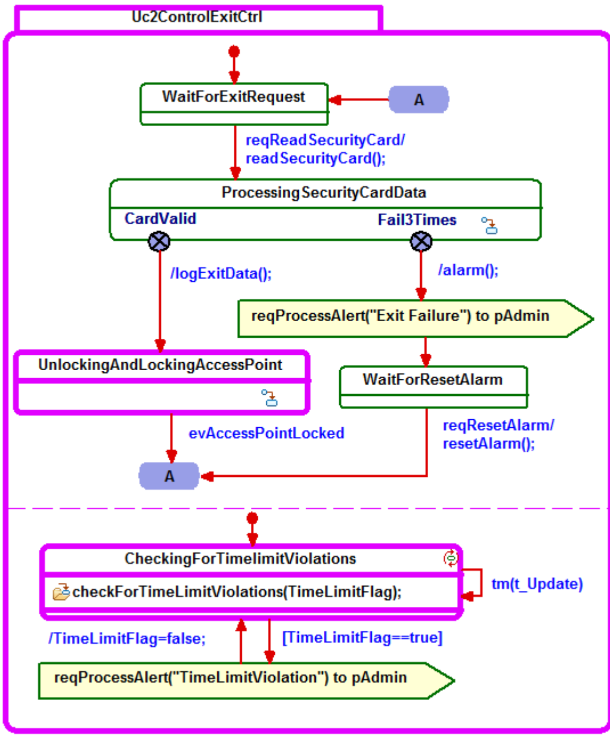


State-based Behavior of Use Case Block Uc_Uc2ControlExit

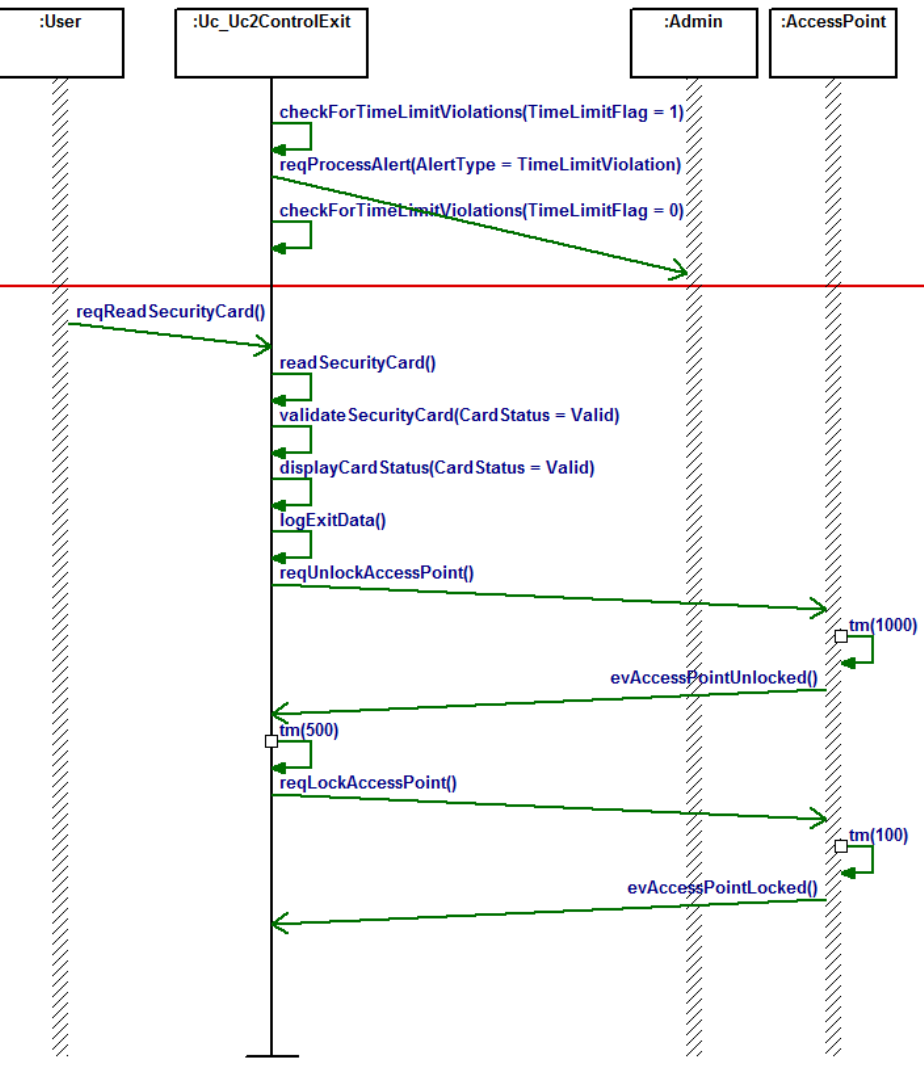
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.4.2.6 Use Case 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



Animated Sequence Diagram BB_Uc2Sc2 Nominal

4.4.2.7 Linking Model Properties to Requirements

In order to assure that all Uc2 allocated functional and performance requirements are considered, traceability links from the Uc2 block properties to the system requirements are established using a *satisfy* dependency (ref. Section 4.4.1.7).

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5 *Design Synthesis*

4.5.1 Architectural Analysis (Trade-Off Analysis)

The focus of the Architectural Analysis is on the determination of a system decomposition 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.

As outlined in Section 4.1 Architectural Analysis is performed in a separate project **SecSys_AA**. The elaborated system architecture captured in the Block Definition Diagram BDD_SecuritySystem and IBD_SecuritySystem in the ArchitecturalDesignPkg, will be common in all subsequent realized use case models even when only a subset of the system blocks will be addressed in the individual use case realization.

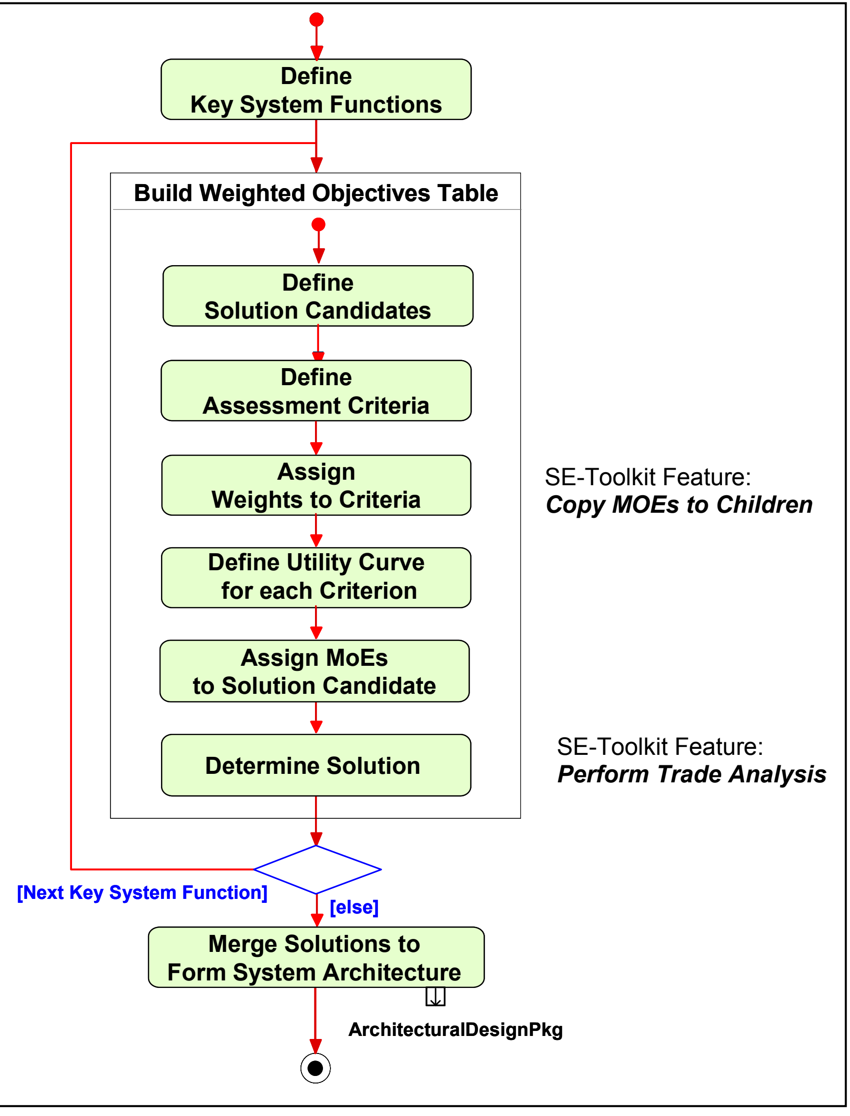


Fig. 4-4 Workflow in the Architectural Analysis Phase and its Support through the Rhapsody SE-Toolkit

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.1.1 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.

Step1: 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:

ReadCardInfomation:

- readSecurityCard
- displayCardStatus
- alarm
- resetAlarm

CaptureBiometricData:

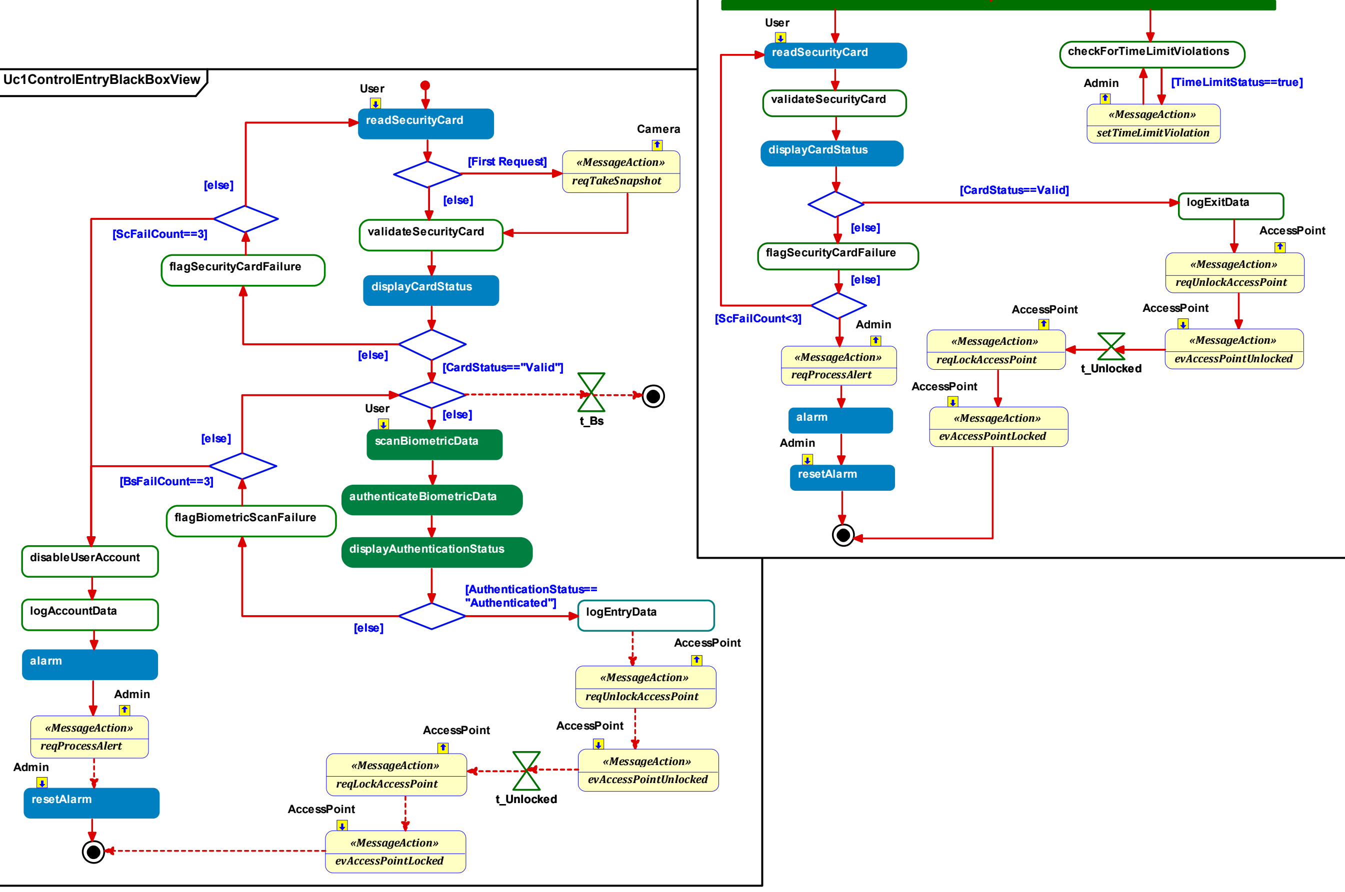
- scanBiometricData
- authenticateBiometricData
- displayAuthenticationStatus

ControlSecSys:

- validateSecurityCard
- flagSecurityCardFailure
- flagBiometricCheckFailure
- disableUserAccount
- logAccountData
- logEntryData
- logExitData
- checkForTimelimitViolations
- unlockAccesPoint
- lockAccessPoint

Step2: 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.



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.1.2 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**

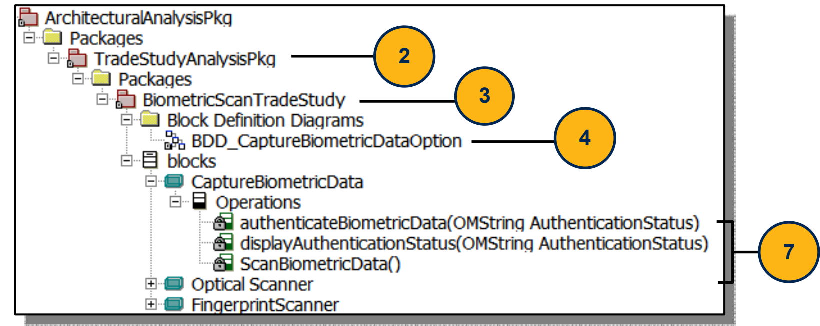
3. In the TradeStudyAnalysisPkg create a package **BiometricScanTradeStudy**

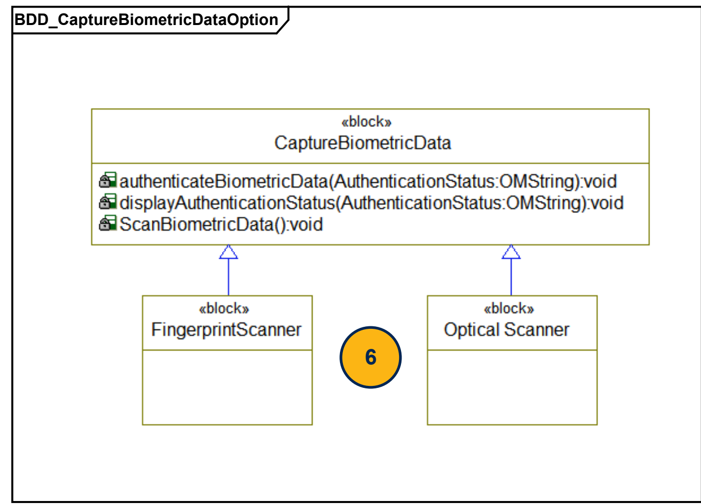
4. In the BiometricScanTradeStudy package create a Block Definition Diagram called ***BDD_CaptureBiometricDataOptions***

5. In the BiometricScanTradeStudy create the following blocks
   - *Capture Biometric Data*
   - *Optical Scanner*
   - *FingerprintScanner*

6. Move the blocks onto BDD_CaptureBiometricDataOptions and join them together using inheritance associations.

7. In the block CaptureBiometricData manually add the Uc1ControlEntry operations that are associated with the key system function CaptureBiometricData. This shows what the OpticalScanner and FingerprintScanner should be capable of.





© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

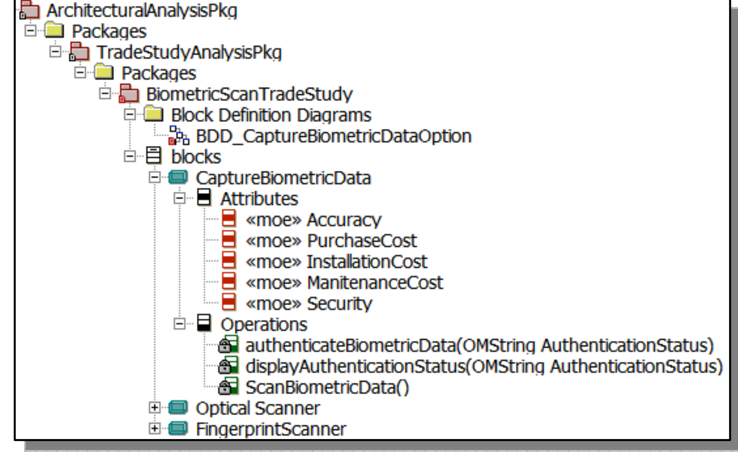
4.5.1.3 Definition of Assessment Criteria

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

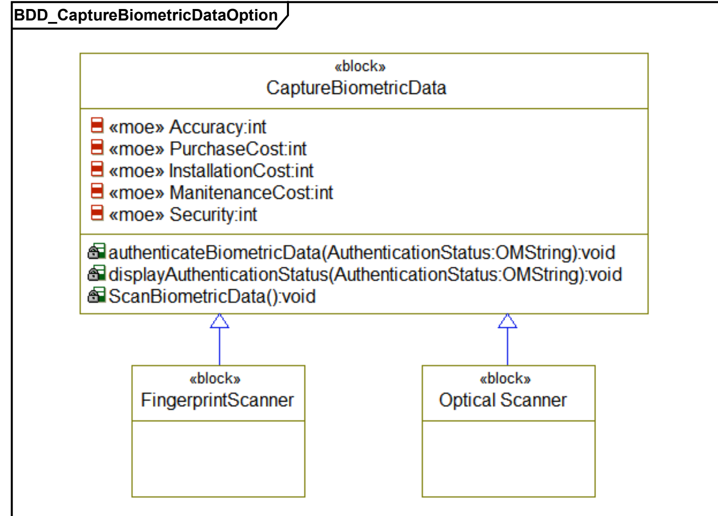
Assessment criteria are normally subjective but can also be very specific. A *subjective* target could be low cost. A *specific* target could be a precise measure of accuracy i.e. +/- 0.1 mm. In this case study the assessment criteria are a mixture of both.

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>>.



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.1.4 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

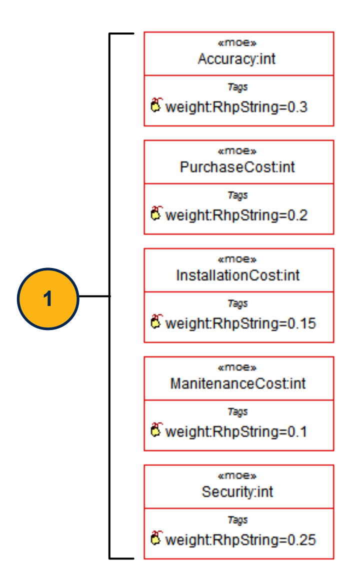
- Accuracy: 0.30
- Security : 0.25
- Purchase Cost: 0.20
- Installation Cost: 0.15
- Maintenance Cost: 0.10

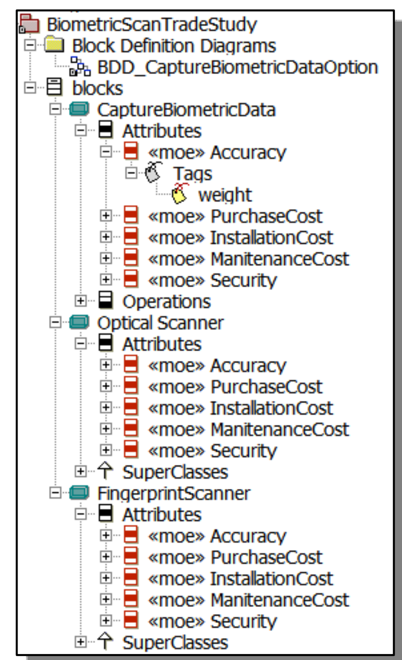
These values are represented in the model by a tag called *weight* attached to each of the <<moe>> attributes.

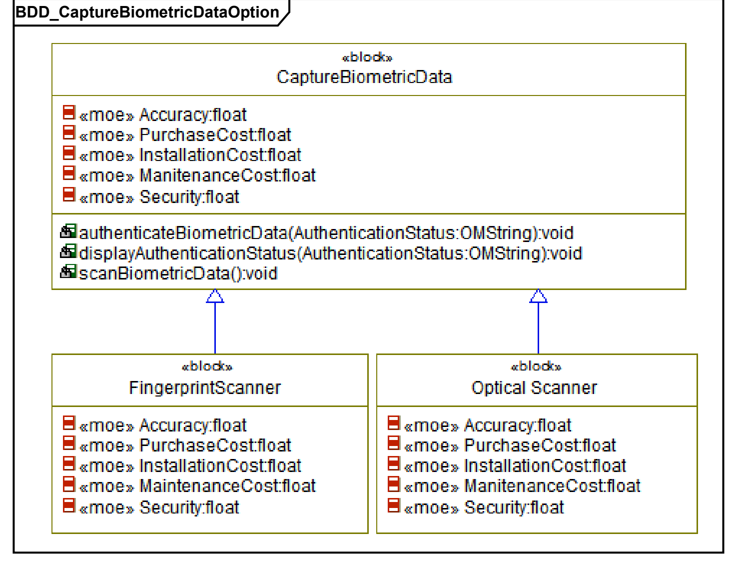
1 In each <<moe>> attribute select the tab *Tags* and add the appropriate value.

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.*





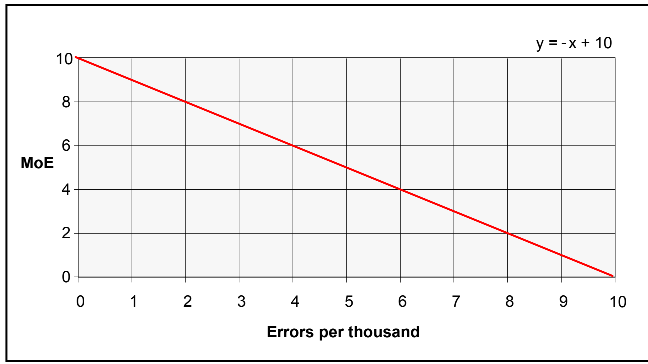


© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.1.5 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

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

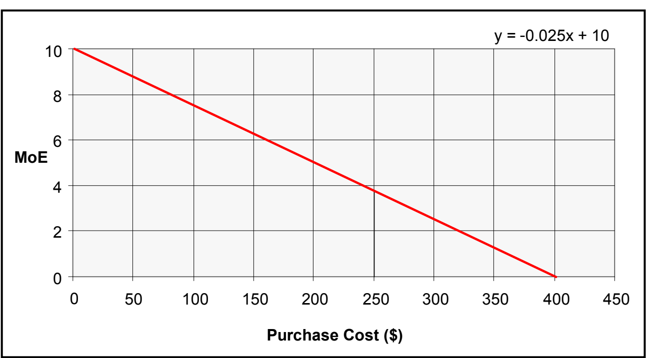
MoE=-(MoE range/target range)+MOE range

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

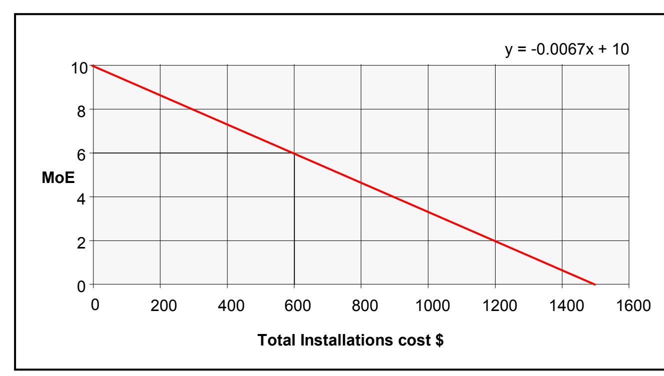
Purchase cost MoE=-0.025*Purchase Cost + 10 .



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*installation cost +10



Installation Cost Utility Curve

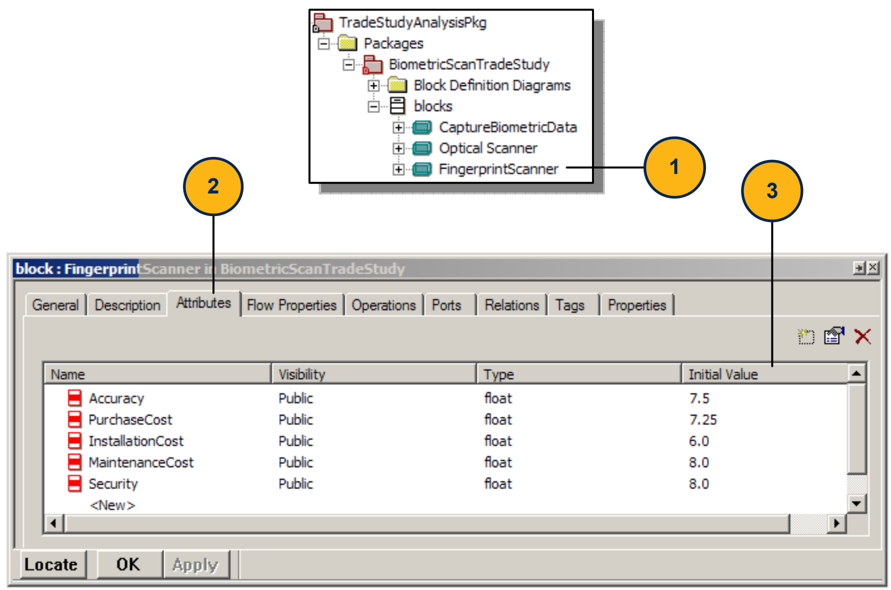
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

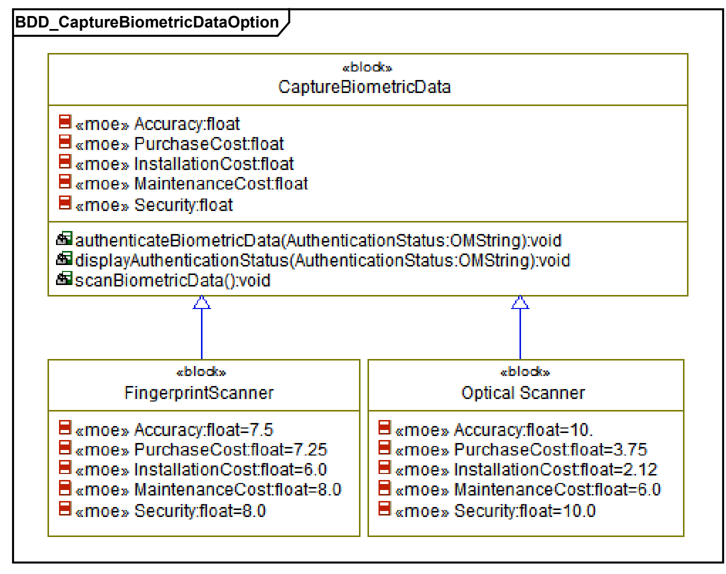
4.5.1.6 Assigning Measures of Effectiveness (MoE) to each Solution

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 finger print 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 am maintenance MoE of 8.0 and optical scanners and maintenance MoE of 6.0.

1. In the browser select a block representing one of the solutions and opend its features.
2. Select the attribute tab.
3. Select the attribute to be edited and in the *Initial Value* field enter the expected value.
4. Select and edit each attribute in turn.

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



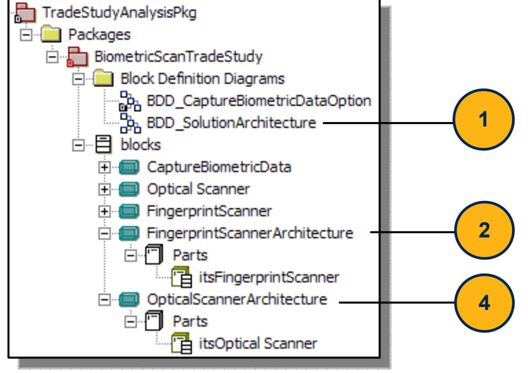


© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.1.7 Determination of Solution

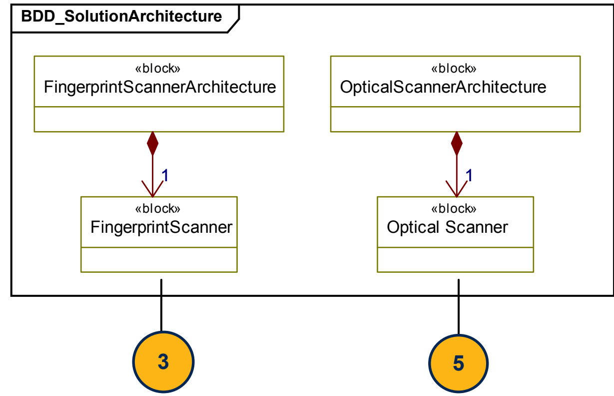
Once each of the key functions has a number of possible solutions with MoEs assigned to them, it is possible to combine the various solutions in order to determine the optimum solution for the architecture.

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 **FingerprintScannerArchitecture**.
3. Drag on the FingerprintScanner block and using the decomposition relationship make it part of the FingerprintScannerArchitecture.
4. Create a block **OpticalScannerArchitecture**.
5. Drag on the OpticalScanner block and using the decomposition relationship make it part of the OpticalScannerArchitecture.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

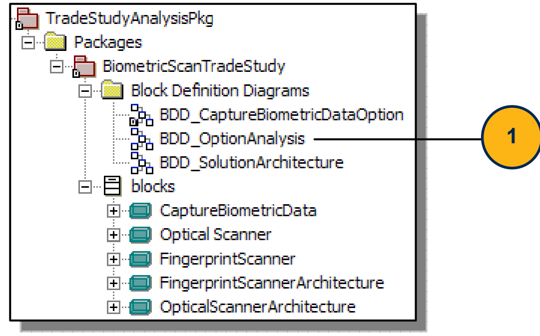
Step 2: Perform Weighted Objectives calculation

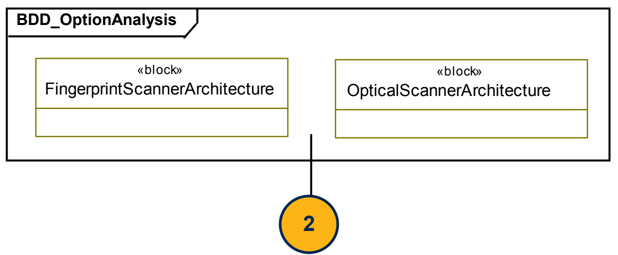
Once the possible solution architectures are in place, the analysis to determine the best solution from the presented options can be carried out. The means of doing this analysis is the *Weighted Objectives Calculation*. It is used to determine the solution for a particular function. It consists of multiplying 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**.
2. Drag on the blocks OpticalScannerArchitecture and the FingerprintScannerArchitecture.
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.





		FingerprintScannerArchitecture		*OpticalScannerArchitecture*	
	weight	value	WV	value	WV
CaptureBiometricData.Accuracy	*0.3*	7.5	2.25	10	3
CaptureBiometricData.PurchaseCost	*0.2*	7.25	1.45	3.75	0.75
CaptureBiometricData.InstallationCost	*0.15*	6	0.9	2.12	0.318
CaptureBiometricData.MaintenanceCost	*0.1*	8	0.8	6	0.6
CaptureBiometricData.Security	*0.25*	8	2	10	2.5
			7.4		7.168

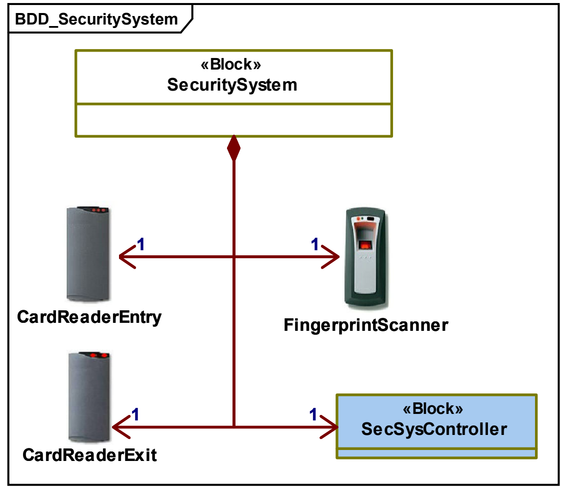
Rhapsody Generated Weighted Objectives Table (Excel)

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

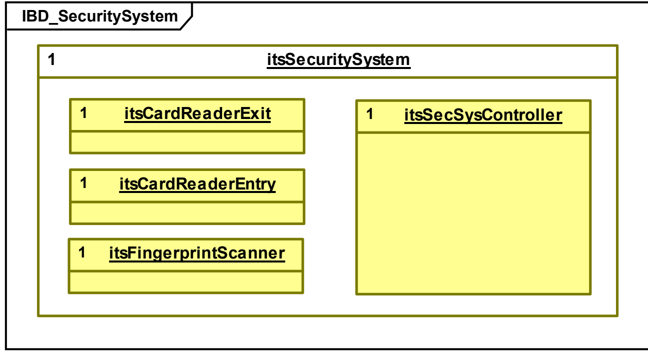
4.5.1.8 Documentation of the Solution in the ArchitecturalDesignPkg

In the DesignSynthesisPkg create a package **ArchitecturalDesignPkg**.

The elaborated 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.**



Block Definition Diagram BDD_Security System

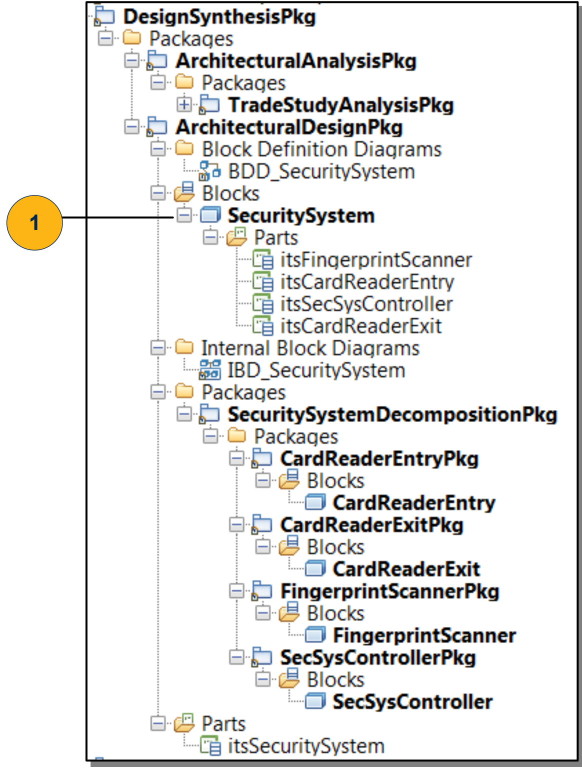


Internal Block Diagram IBD_Security System

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

It is recommended to standardize the structure of the ArchitecturalDesignPkg. 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***:

1. Right-click the SecuritySystem block, select *SE-Toolkit > Create Sub Packages*



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2 Architectural Design

Fig. 4-5 shows the architecturall design workflow in the case study. The architectural design is performed for each use case of an iteration by transitioning from the black-box view to the white-box view – also referred to as **Use Case Realization** (ref. *Fig. 4-6*).

Once all use cases of an iteration are realized, they are merged in the **Integrated System Architecture Model**.

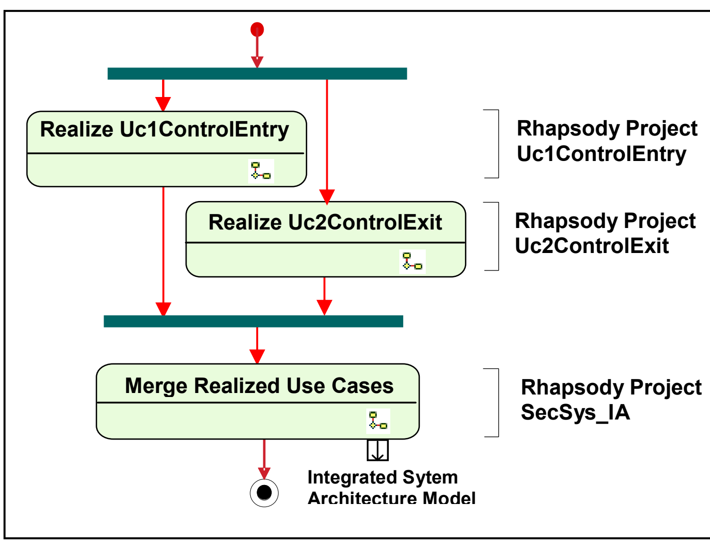


Fig. 4-5 Architectural Design Workflow and Associated Rhapsody Projects

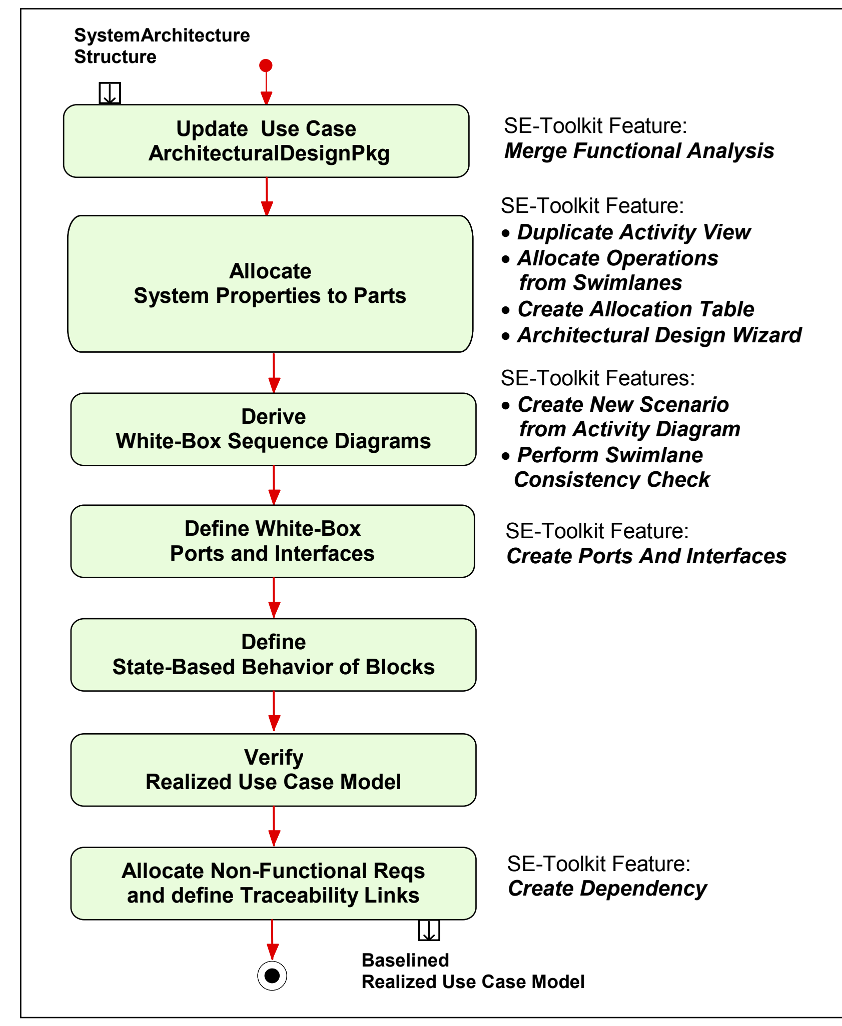


Fig. 4-6 Use Case Realization Workflow and its Support through the Rhapsody SE-Toolkit

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.1 Use Case Realization Uc1ControlEntry

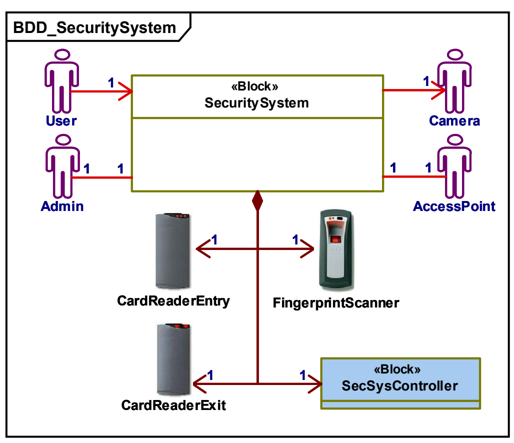
4.5.2.1.1 Update of the ArchitecturalDesignPkg

The update of the ArchitecturalDesignPkg in the Uc1ControlEntry model project structure will be performed in three steps.

Step 1: Import the ArchitecturalDesignPkg from SecSys_AA project

1. In the Rhapsody main menue select *File > Add to Model*, navigate to the SecSys_AA project and double-klick *SecSys_AA.rpy*

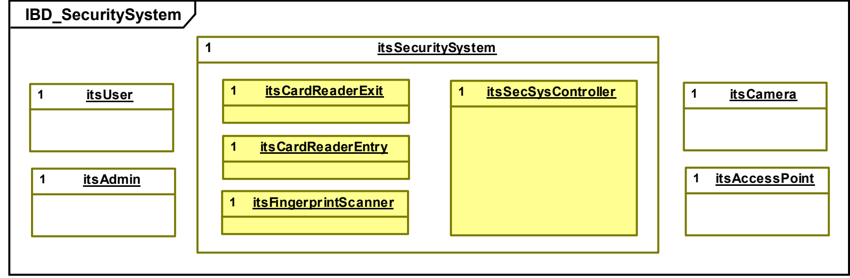
2. In the dialog box tick *As unit* and select ArchitecturalDesignPkg.sbs



Updated BDD_SecuritySystem

Step 2: Update the imported BDD and IBD

As in the SecSys model only the system architecture was captured in the BDD_SecuritySystem and IBD_SecuritySystem, the imported diagrams need to be updated w.r.t the use case associated actors.

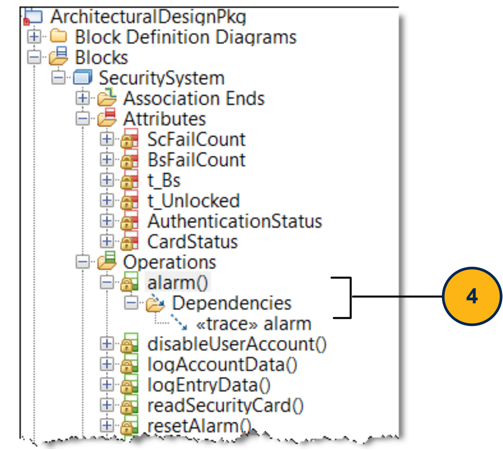


Updated IBD_SecuritySystem

Step 3: Copy/paste the events, operations and attributes from the use case block **Uc_Uc1ControlEntry** in the FunctionalAnalysisPkg into the system block **SecuritySystem** in the ArchitecturalDesignPkg.

3. Right-click the block SecuritySystem and select *SE-Toolkit > Merge Functional Analysis*

4. The copies are traced back to the origins.



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.1.2 Allocation of System Block Properies to Parts

4.5.2.1.2.1 Allocation of Operations to Parts

The allocation of operations to the parts of the system block is elaborated graphically (*White-Box Activity Diagram*). The Black-box use case activity diagram is partitioned into swim lanes, each of which corresponds to a part of the decomposed system block (case study: CardReader_Entry, 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_Uc1.**

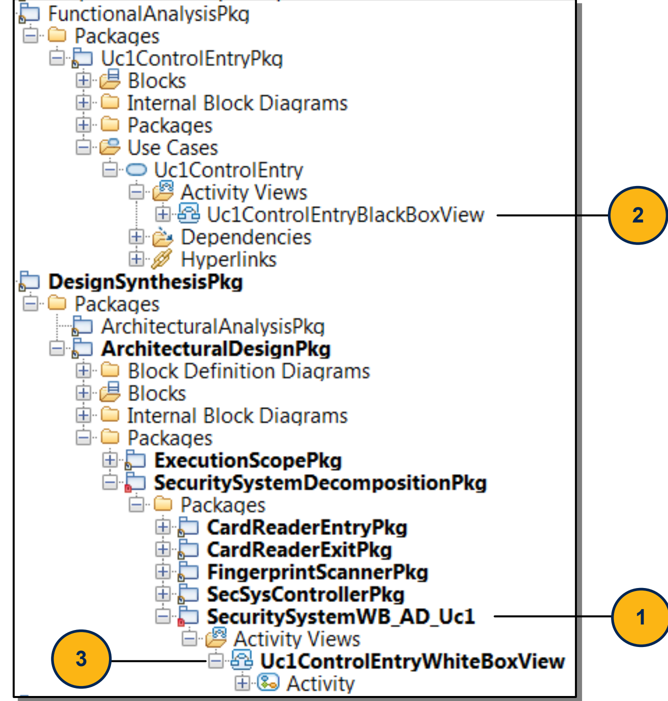
2. In the FunctionalAnalysisPkg > Uc1ControlExitPkg right-click Uc1ControlEntryBlackBoxView and select *Duplicate Activity View*.

3. Rename the copied Activity View to **Uc1ControlEntryWhiteBoxView** and move it into the SecuritySystemWB_AD_Uc1 package.

4. In the category Uc1ControlEntryWhiteBoxView partition the Activity Diagram (**Activity**) into swimlanes:
   - CardReader_Entry,
   - FingerprintScanner and
   - SecSysController.

5. Allocate blocks via drag and drop on swimlane headlines

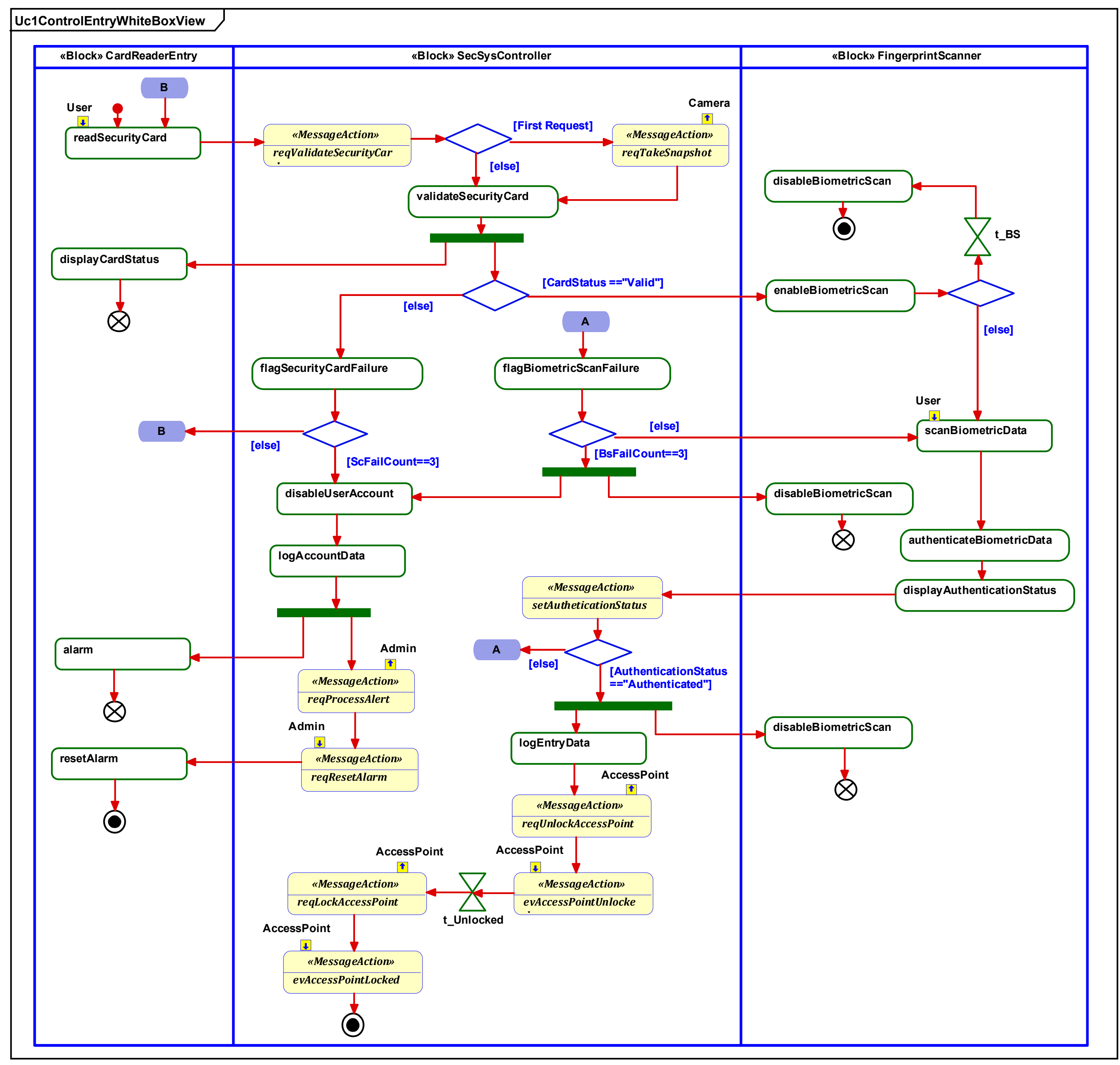
6. Allocate actions to the respective swim lanes.



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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



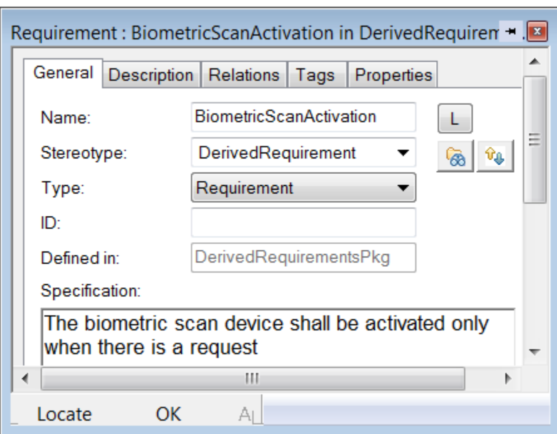
White-Box Activity Diagram Uc1ControlEntry

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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:

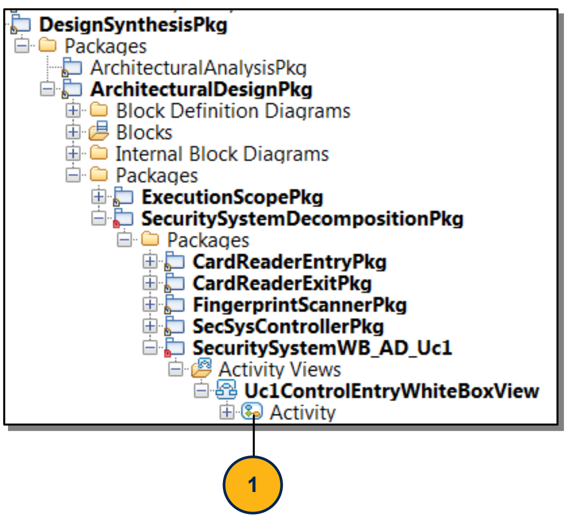
enableBiometricScan and
disableBiometricScan

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



Summarizing the Allocation of Operations

The allocation of operations to the subsystems may be summarized in an Excel spreadsheet by means of the SE-Toolkit feature ***Create Allocation Table***.



1. Right-click *Activity* in SecuritySystemWB_AD_Uc1 > ActivityViews > Uc1ControlEntryWhiteBoxView.
2. Select *SE-Toolkit > Create Allocation Table*

FingerprintScanner	SecSysController	CardReader_Entry
scanBiometricData	validateSecurityCard	alarm
authenticateBiometricData	flagBiometricScanFailure	displayCardStatus
enableBiometricScan	disableUserAccount	readSecurityCard
disableBiometricScan	flagSecurityCardFailure	resetAlarm
displayAuthenticationStatus	logEntryData	
	logAccountData	

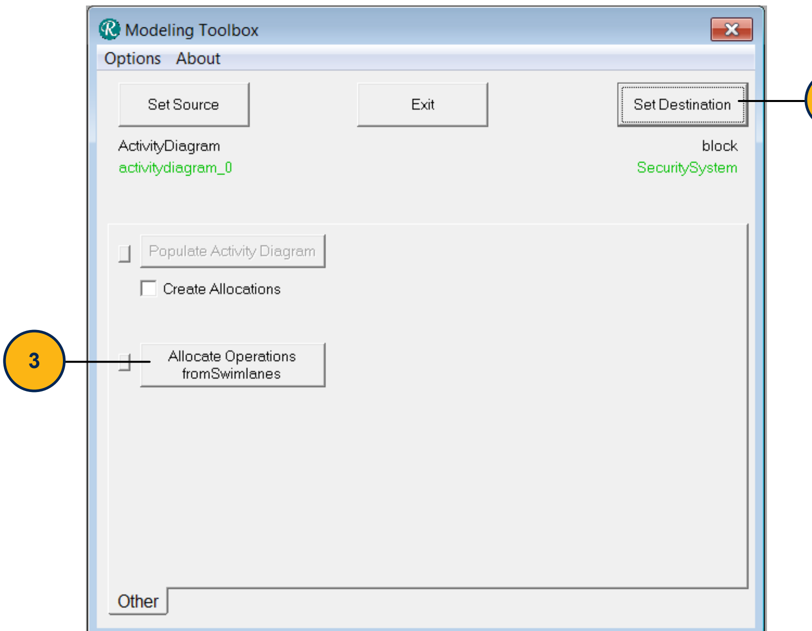
Allocation Table of Uc1ControlEntryWhiteBoxView (Excel Spreadsheet)

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

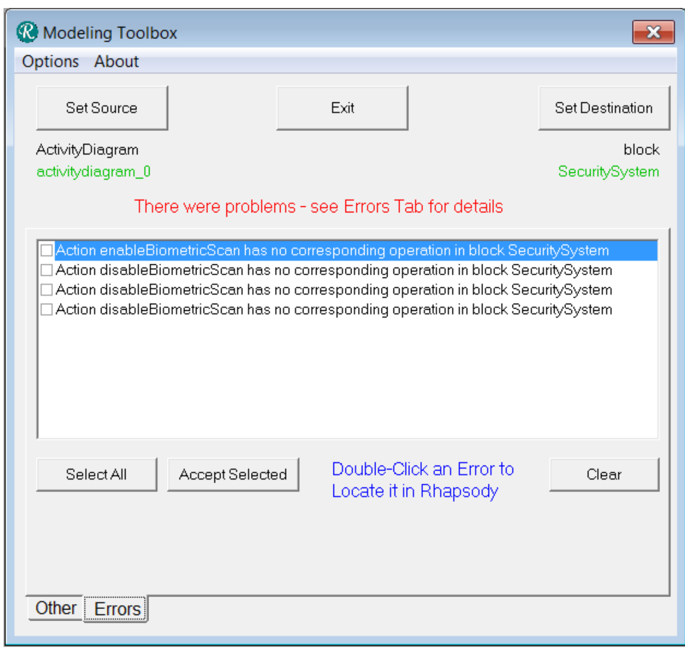
Formalizing the Allocation of Operations

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. Right-click *Activity* in SecuritySystemWB_AD_Uc1 > ActivityViews > Uc1ControlEntryWhiteBoxView and select *SE-Toolkit > Allocate Operations from Swimlanes.*
2. In the ArchitecturalDesignPkg select system block SecuritySystem and click *Set Destination*
3. In the Modeling Toolbox dialog box click *Allocate Operations from Swimlanes*



The reason for the error messge 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:

4. In the dialog box *Select All,*
5. click *Accept Selected.*

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.1.2.2 Allocation of Attributes and Events to Parts

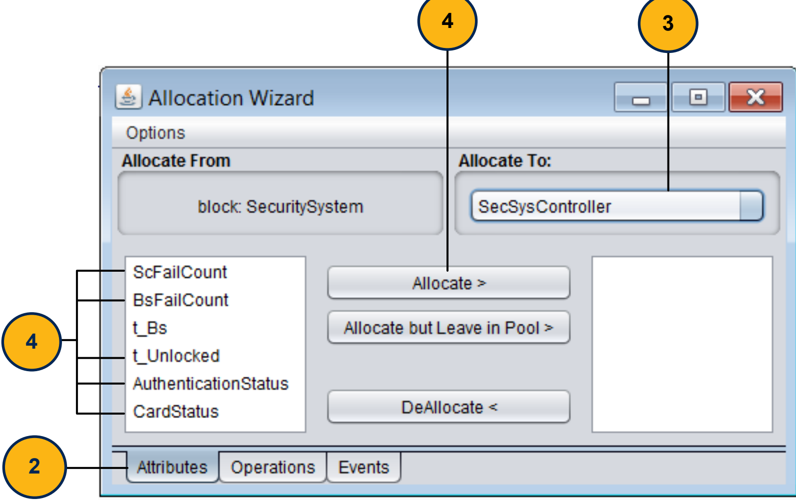
The allocation of SecuritySystem block attributes and receptions (events) to the subsystems is performed by means of the SE-Toolkit feature ***Allocation Wizard***.

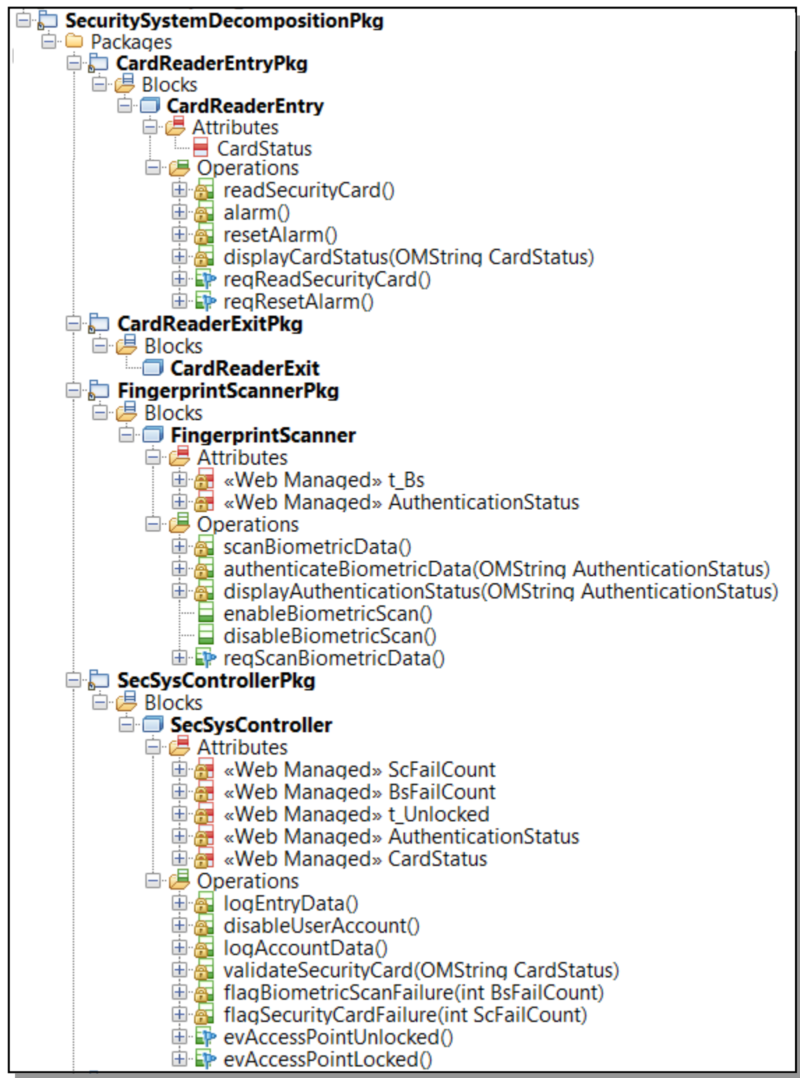
<u>NOTE</u>: This SE-Toolkit feature may also be used for allocating operations.

1. In the ArchitecturalDesignPkg right-click SecuritySystem block, select *SE-Toolkit > Allocation Wizard*
2. In the dialog box select *Attributes*
3. In the *Allocate To* drop-down menue select SecSysController.
4. In the *Allocate From* window select attribute(s) and click *Allocate*

   <u>NOTE</u>: If an element needs to be allocated to more than one subsystem, select *Allocate but Leave in Pool*

Repeat step 2 - 4 for the allocation of events.





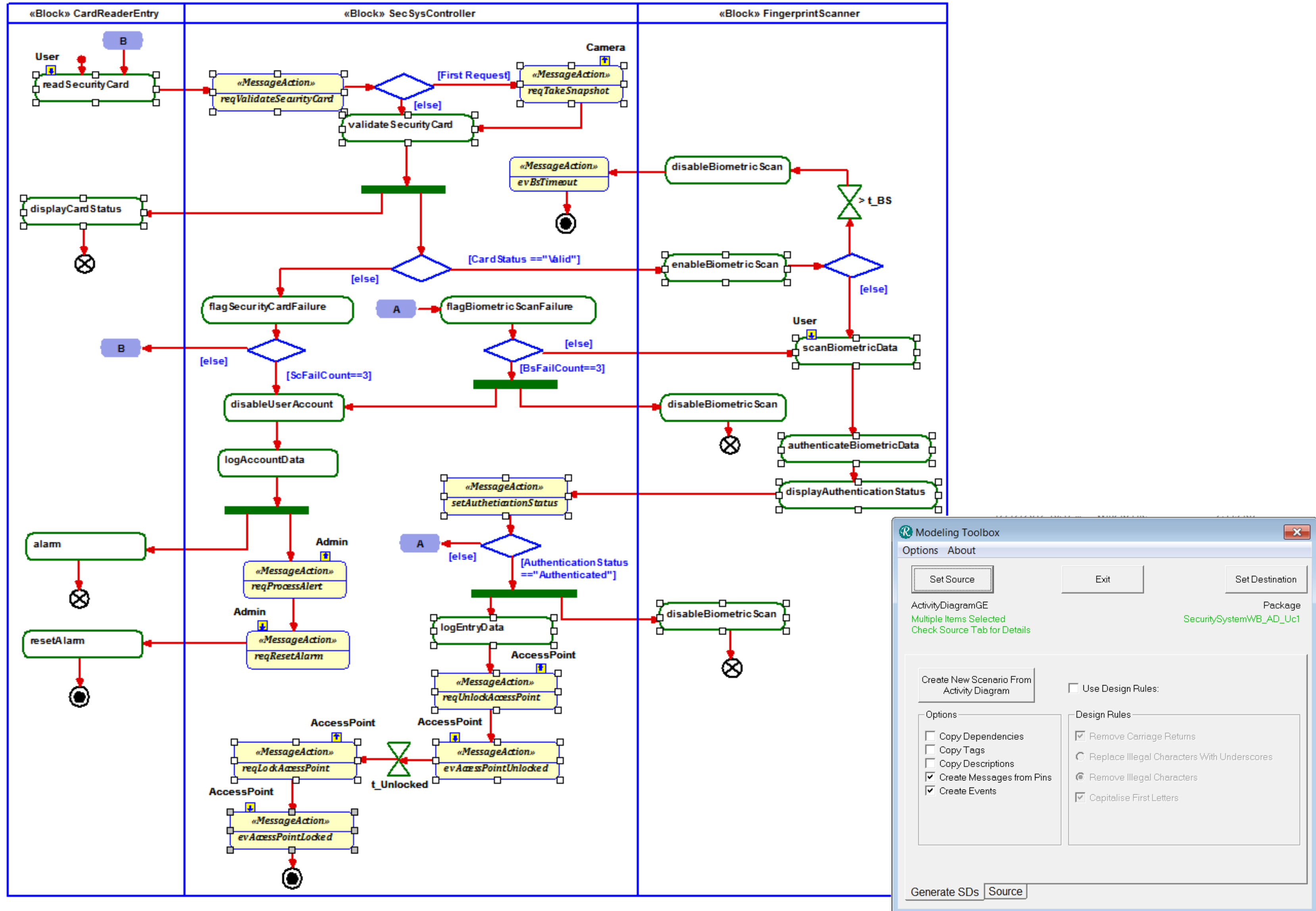
Allocated Attributes, Operations and Events

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

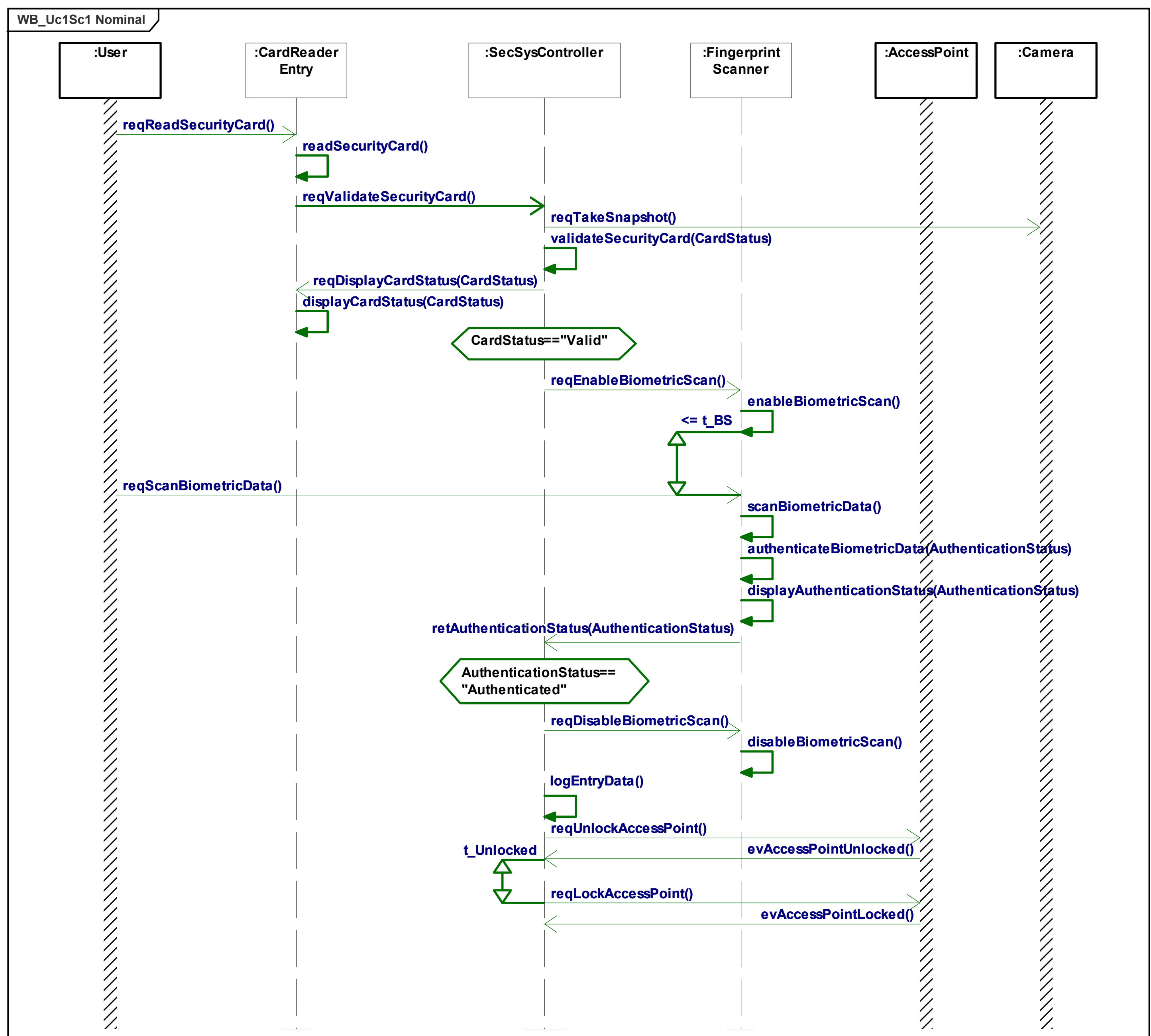
4.5.2.1.3 Derivation of White-Box Sequence Diagrams

White-box scenarios are derived from the white-box activity diagrams by means of the SE-Toolkit feature ***Create New Scenario From Activity Diagram.***

In the SecuritySystemDecompositionPkg create a package **SecuritySystemWB_SD_Uc1** and follow the steps outlined in Section 4.4.1.3.

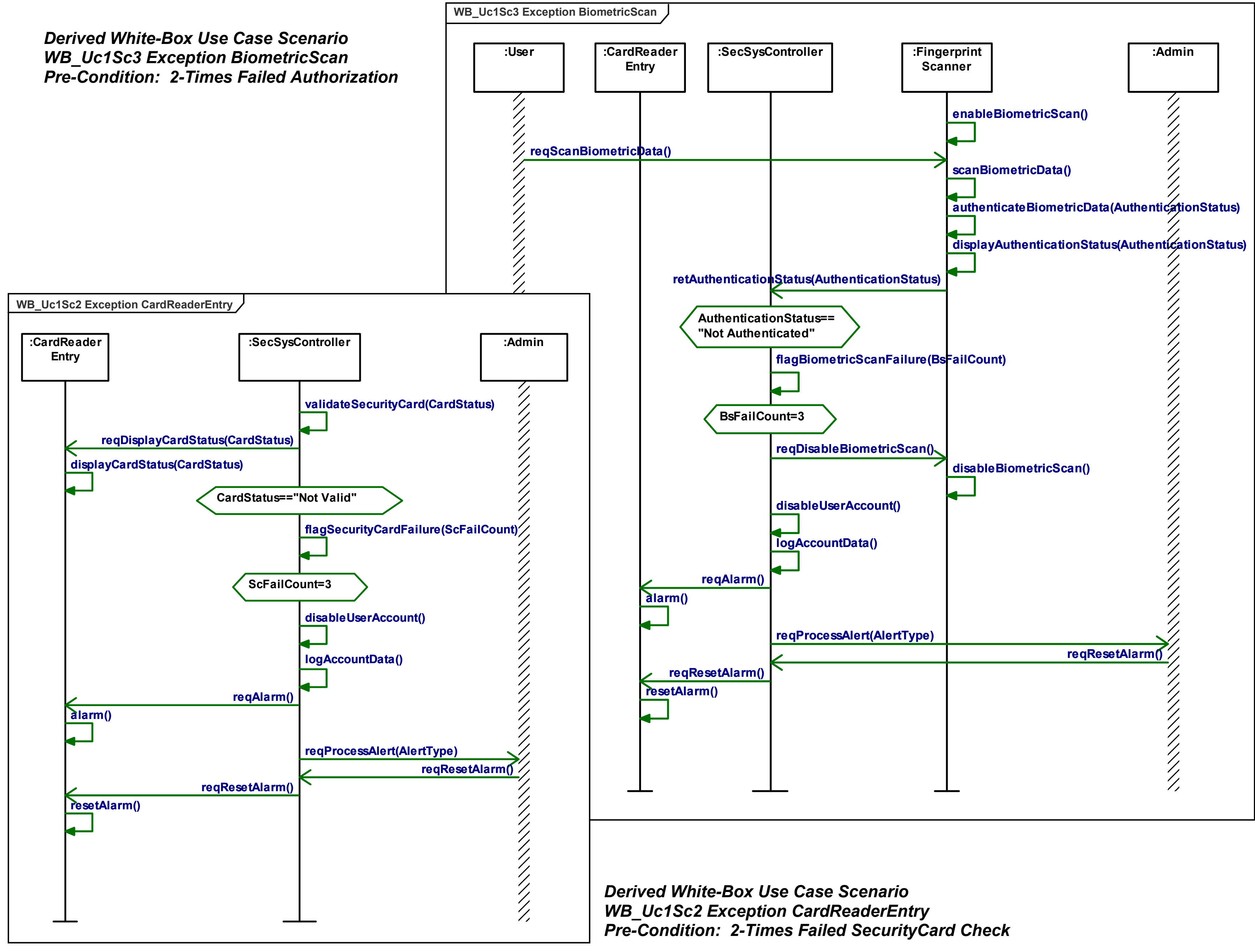


© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



Derived White-Box Use Case Scenario WB_Uc1Sc1 Nominal

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.1.4 Definition of Ports and Interfaces

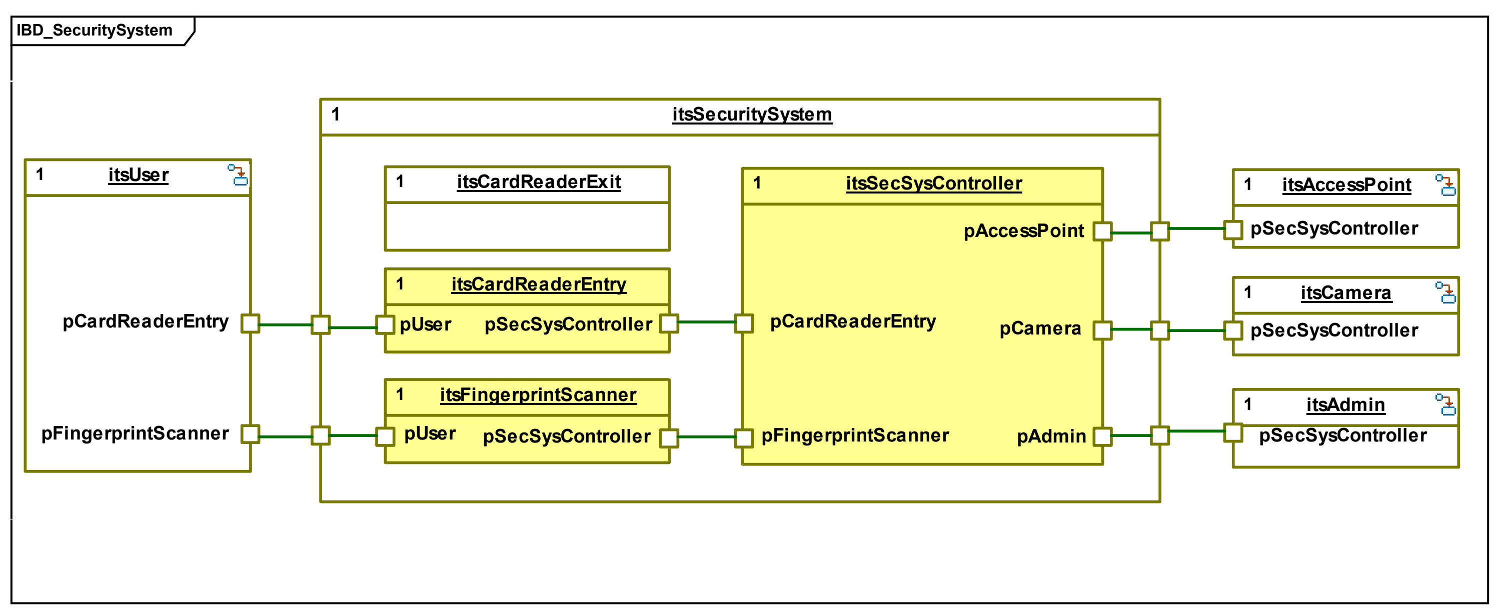
Once all black-box use case scenarios are decomposed into white-box scenarios, the resulting subsystem ports and interfaces are defined by means of the SE-Toolkit feature ***Create Ports And Interfaces***.

1. In the *ArchitecturalDesignPkg* right-click the package SecuritySystemWB_UcSD and select *SE-Toolkit > Create Ports And Interfaces*.

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

2. Manually add Delegation Ports and associated interfaces.

3. Connect ports either manually or right-click in the IBD and select *SE-Toolkit > Connect Ports.*



Internal Block Diagram of the Realized Use Case Uc1ControlEntry

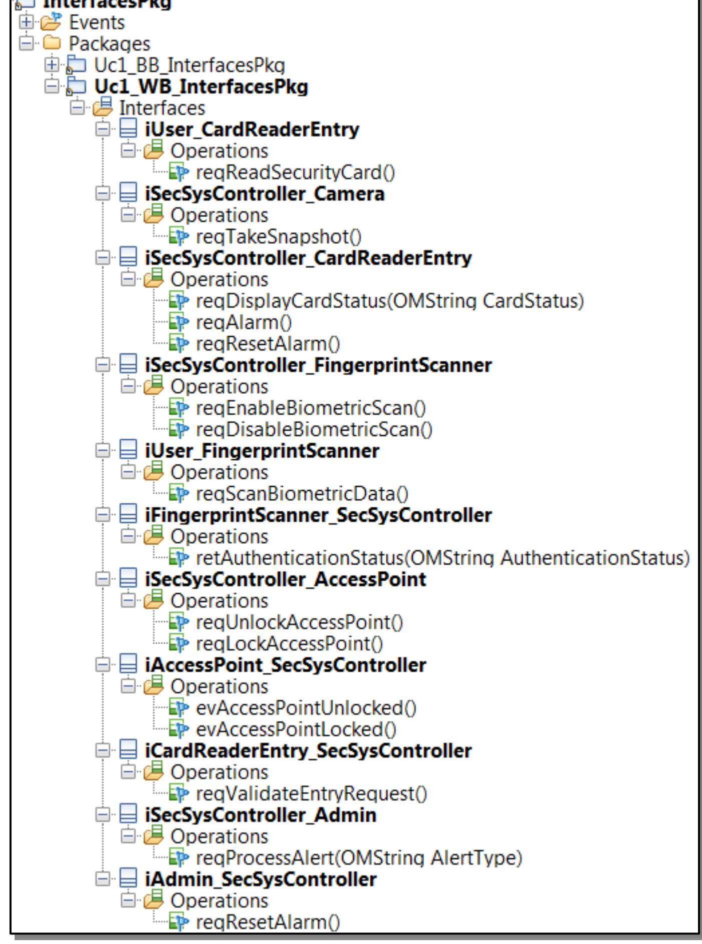
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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 (UML/SysML *required* interfaces) are located in the row of that node and inputs (UML/SysML *provided* interfaces) are in the column of that node. The diagram below depicts the N^2 chart of the realized use case Uc1ControlEntry.

The N^2 chart is generated by means of the SE-Toolkit feature ***Generate N2 Matrix***:

In the ArchitecturalDesignPkg right-click the internal block diagram IBD_SecuritySystem and select *SE-Toolkit > Generate N2 Matrix*.



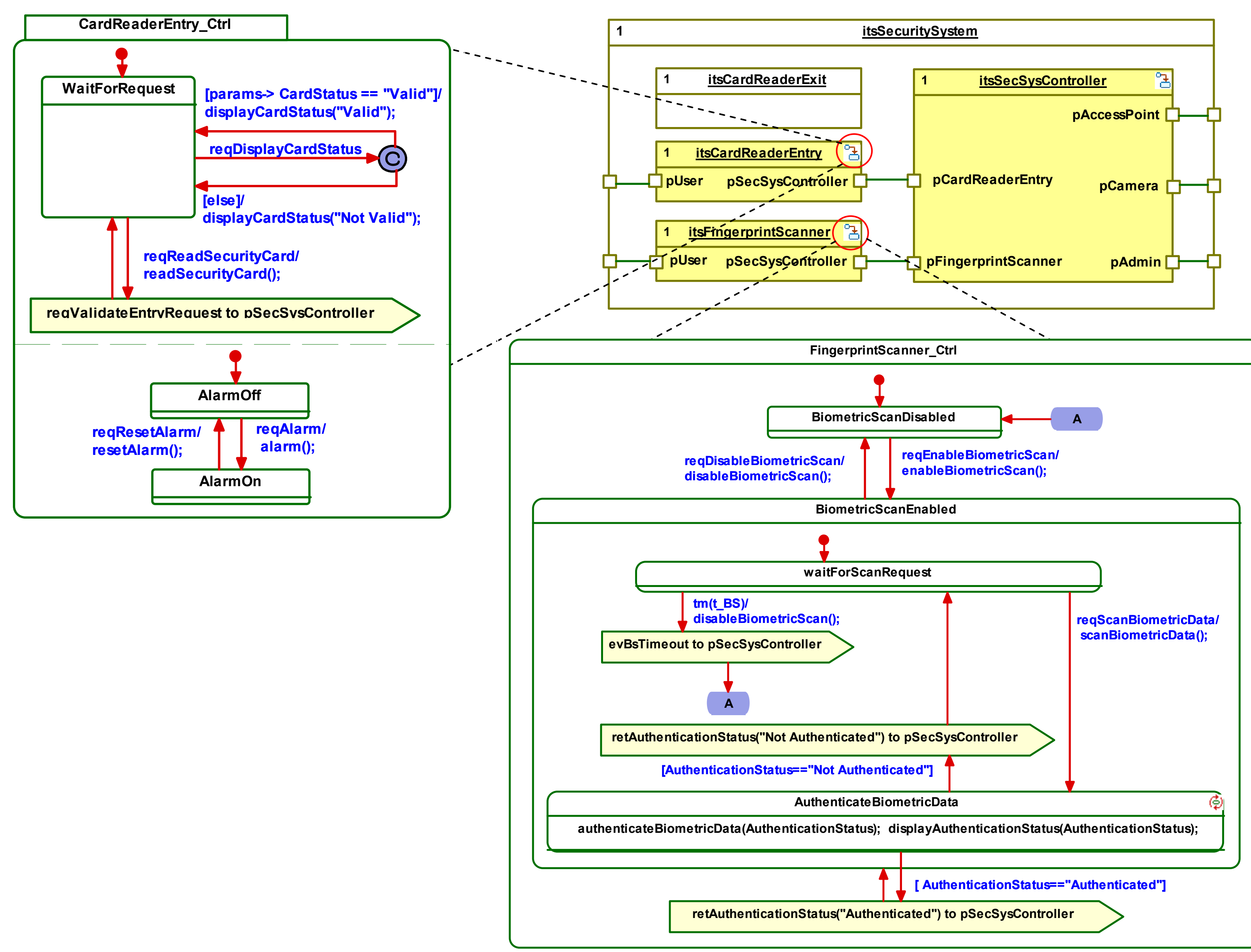
	User	Admin	Camera	AccessPoint	CardReaderEntry	FingerprintScanner	SecSysController
User	X				iUser_CardReaderEntry	iUser_FingerprintScanner	
Admin		X					iAdmin_SecSysController
Camera			X				
AccessPoint				X			iAccessPoint_SecSysController
CardReaderEntry					X		iCardReaderEntry_SecSysController
FingerprintScanner						X	iFingerprintScanner_SecSysController
SecSysController		iSecSysController_Admin	iSecSysController_Camera	iSecSysController_AccessPoint	iSecSysController_CardReaderEntry	iSecSysController_FingerprintScanner	X

N^2 Chart of the Realized Use Case Uc1ControlEntry

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.1.5 Definition of Realized Use Case Behavior

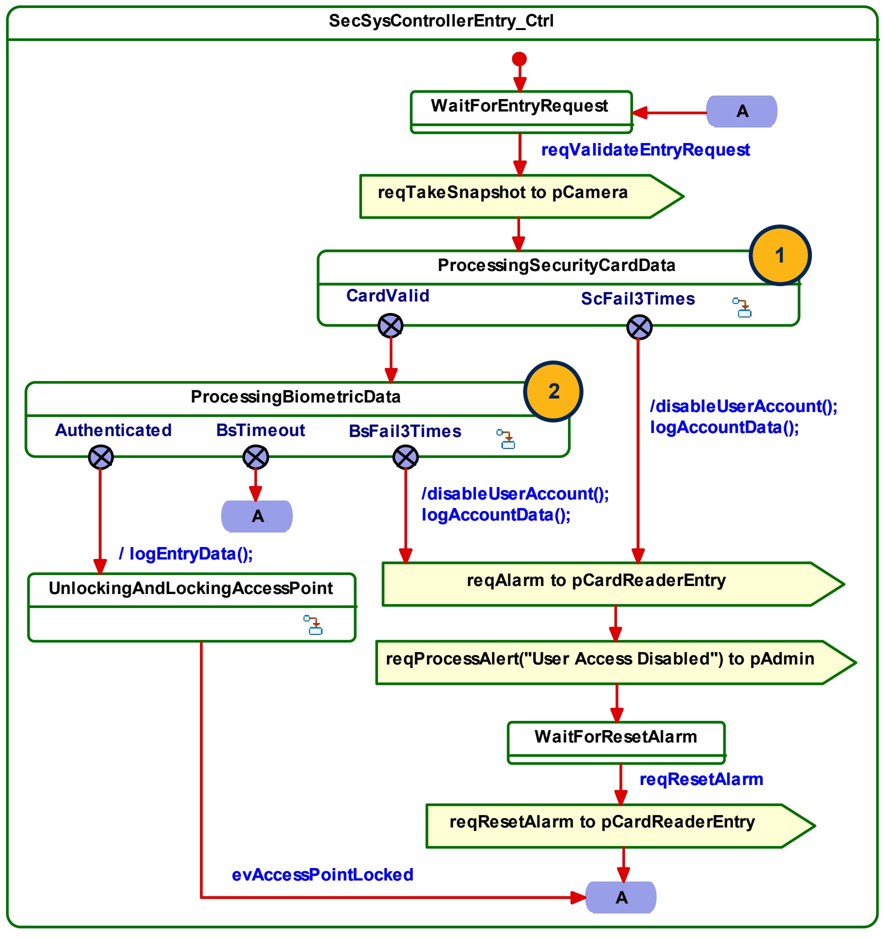
State-based Behavior of the CardReaderEntry Block and FingerprintScanner Block

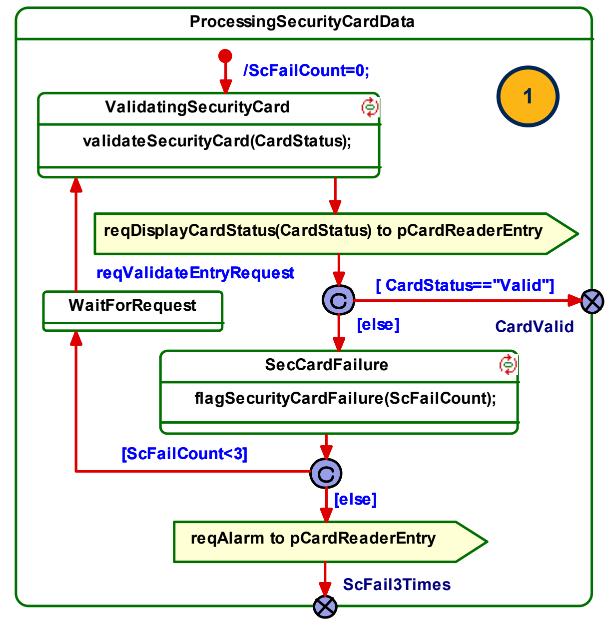


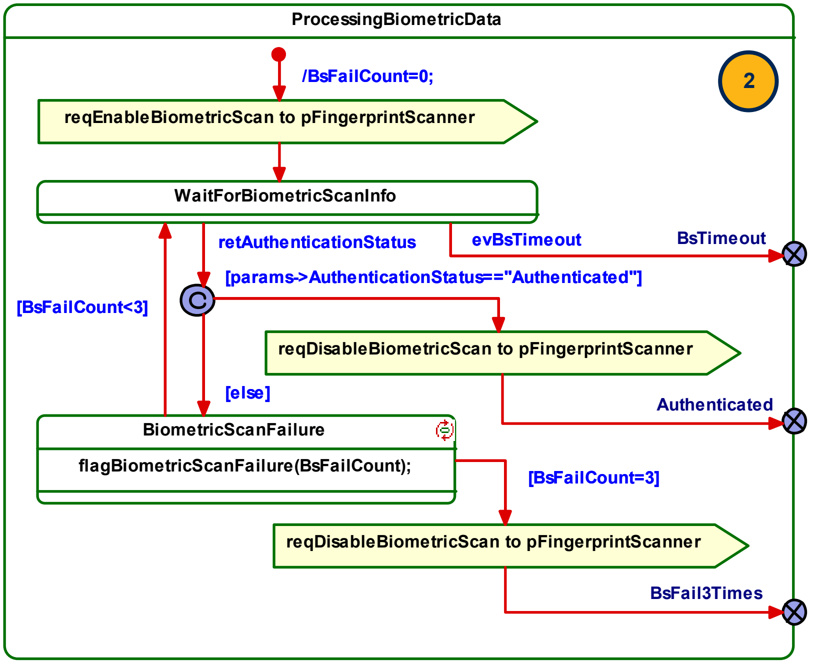
 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

State-based Behavior of the SecSysController Block

Note the reuse of behavior patterns in the statechart diagram of the SecSysController block. The statecharts *ProcessingSecurityCardData* and *ProcessingBiometricData* are extended copies of the ones used in the black-box Uc1ControlEntry use case block. The statechart *UnlockingAndLockingAccessPoint* is an unchanged copy.







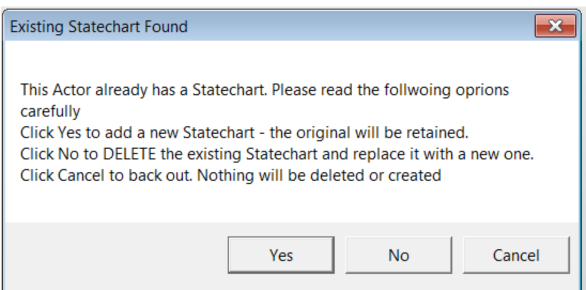
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Behavior of the Actor Blocks

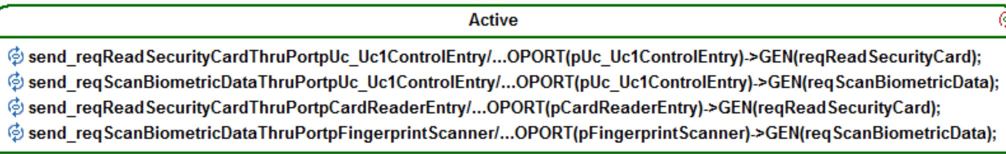
Taking into consideration the communication via the additional actor port in the User block and Adminstrator block, both behaviors need to be extended by applying the SE-Toolkit feature *Create Test Bench* (ref. Section 4.4.1.5). No change is needed for the actor Camera.

Example: Actor block User:

1. In the Internal Block Diagram IBD_SecuritySystem right-click the block User and select *SE-Toolkit > Create Test Bench.*



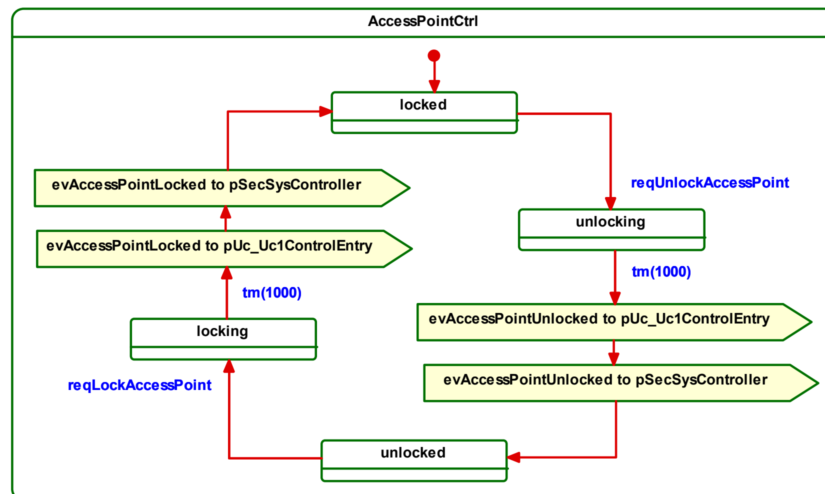
2. Select *No*



***Extended Behavior of the Actor Block* User**

3. Repeat the steps 1-2 for the actor block Admin.

The state-based behavior of the actor block AccessPoint needs to be extended graphically.



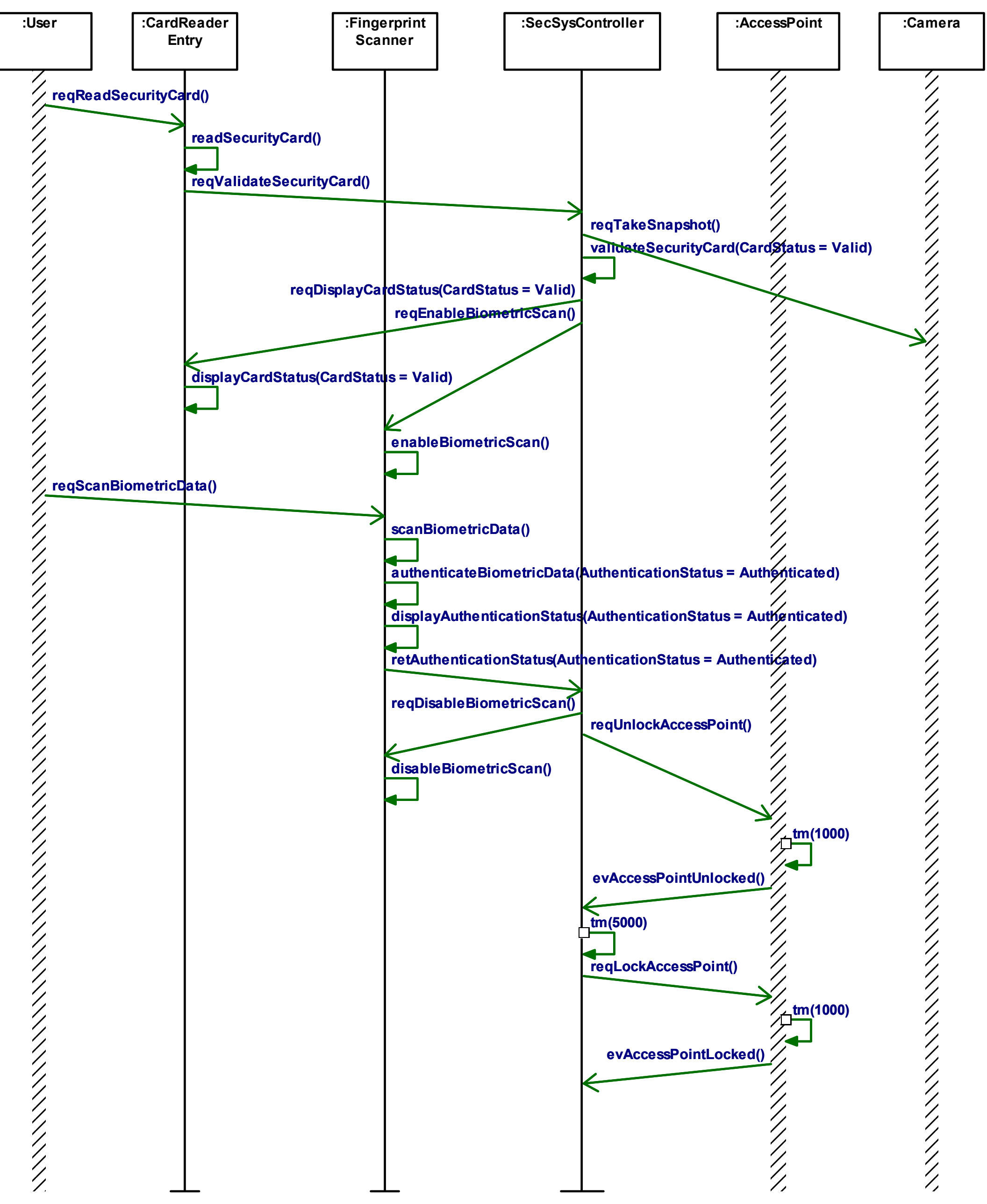
 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.1.6 Realized Use Case Verification

The realized use case model Uc1ControlEntry, 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 Statechart and Sequence Diagrams).

4.5.2.1.7 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 final step in the use case realization taskflow is the allocation of non-functional requirements. In order to assure that all use case related 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.



Animated Sequence Diagram WB_Uc1Sc1 Nominal)

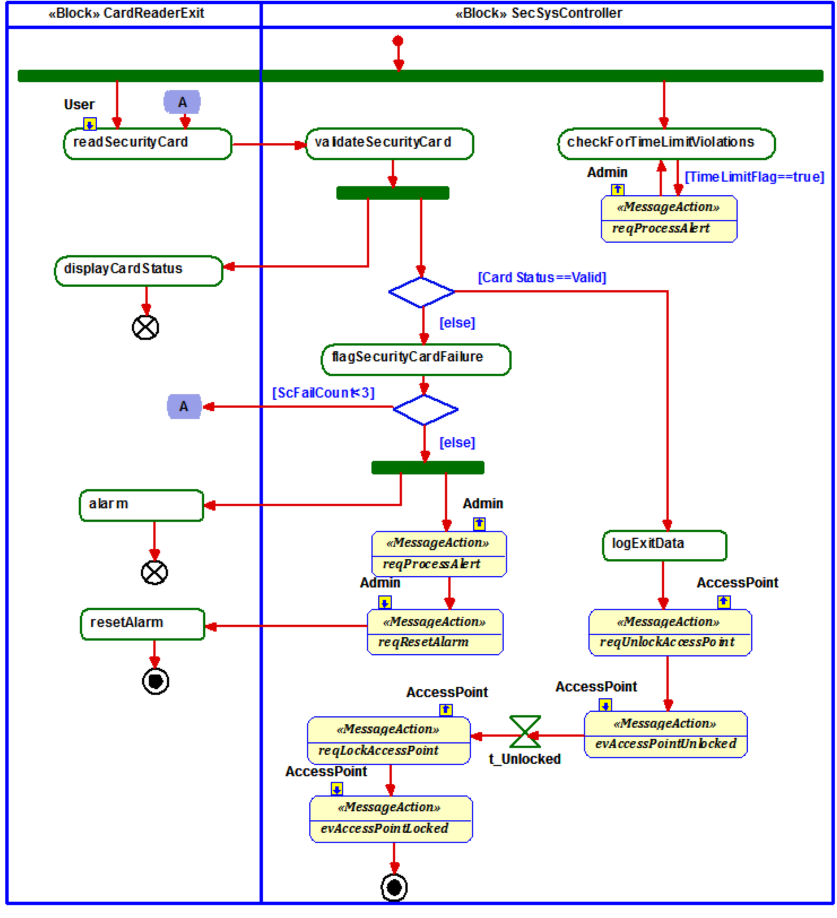
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.2 Use Case Realization Uc2ControlExit

4.5.2.2.1 Update of the ArchitecturalDesignPkg

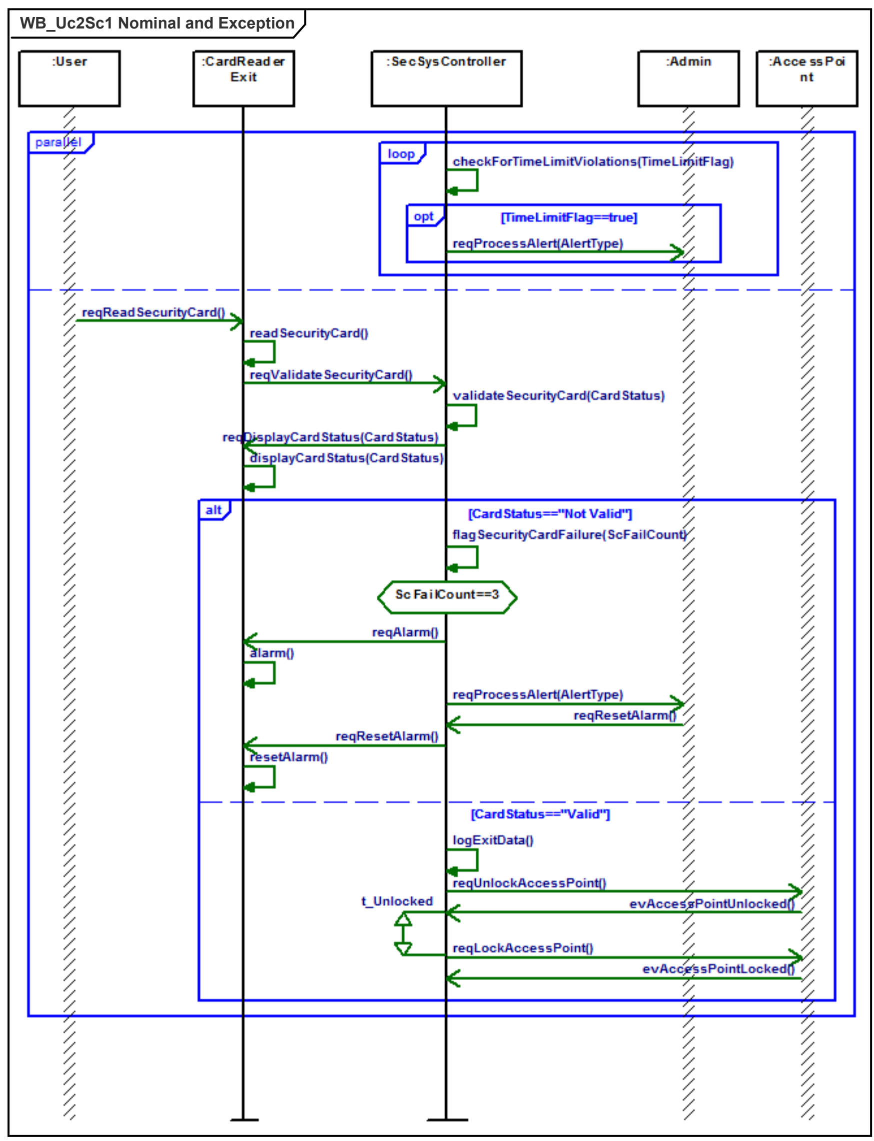
The steps to be performed are similar to the ones described in Section 4.5.2.1.1.

4.5.2.2.2 Allocation of System Block Properies to Parts



White-Box Activity Diagram Uc2ControlExit

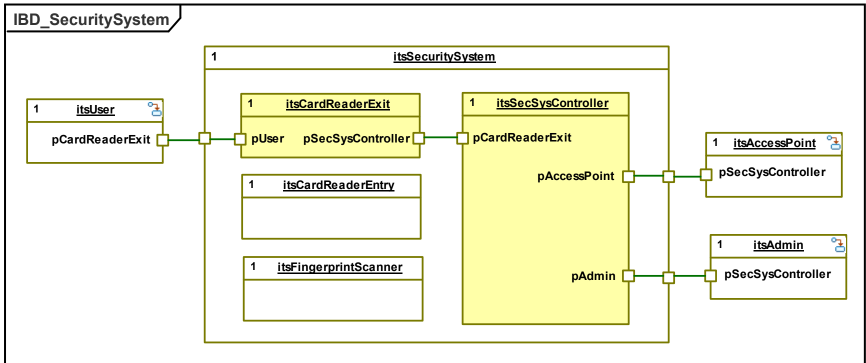
4.5.2.2.3 Derivation of White-Box Sequence Diagrams



White-Box Use Case Scenario WB_Uc2Sc1 Nominal and Exception

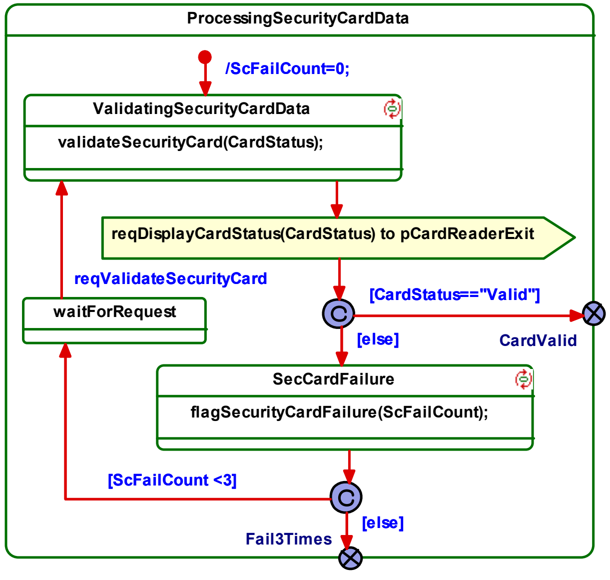
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

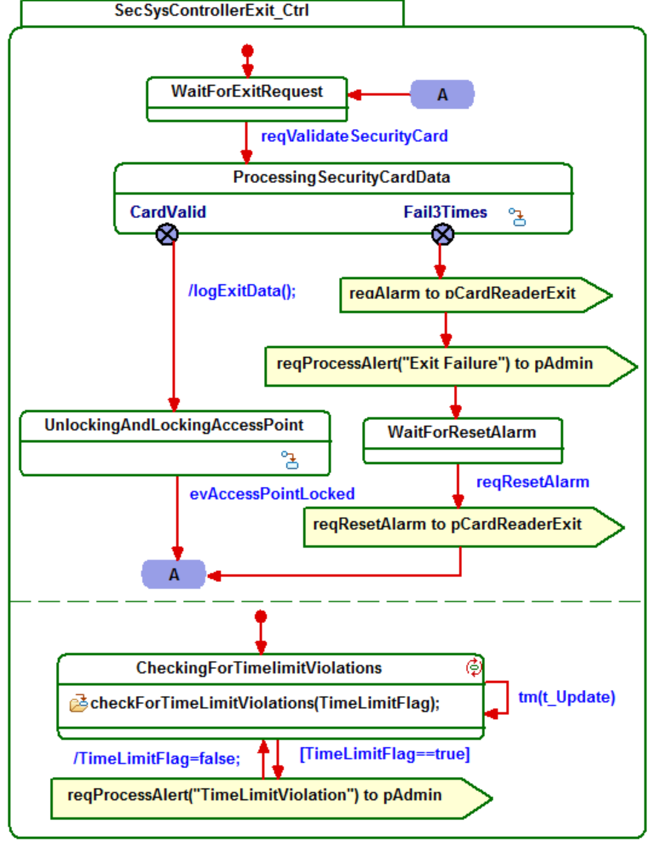
4.5.2.2.4 Definition of Ports and Interfaces



IBD of the Realized Use Case Model Uc2ControlExit with generated Ports and Interfaces

4.5.2.2.5 Definition of Realized Use Case Behavior





Note the reuse of behavior patterns in the statechart diagrams. The statecharts *SecSysControllerExit_Ctrl* and *ProcessigSecurityData* are extended copies of the ones used in the black-box Uc2ControlExit use case. The statechart *UnlockingAndLockingAccessPoint* is an unchanged copy.

Similar to the steps described in Section 4.5.2.2.5, the behavior of the actor blocks *User* and *Adminstrator* needs to be extended by applying the SE-Toolkit feature ***Create Test Bench***. The state-based behavior of the actor block AccessPoint needs to be extended graphically.

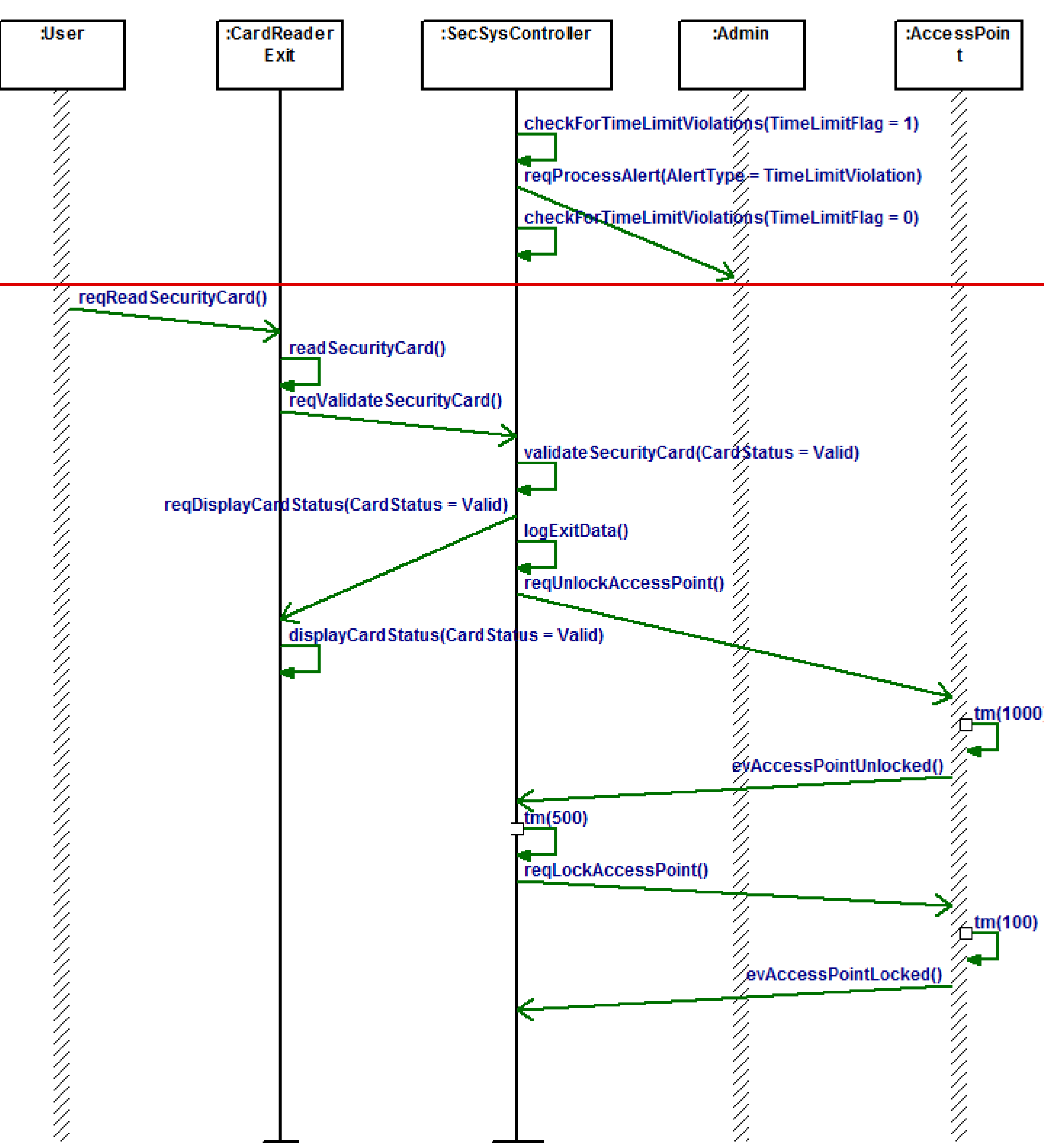
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.2.6 Realized Use Case Verification

The realized use case model Uc2ControlExit, 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 Statechart and Sequence Diagrams).

4.5.2.2.7 Allocation of Non-functional Requirements

The final step in the use case realization taskflow is the allocation of non-functional requirements. In order to assure that all use case related 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.



Animated Sequence Diagram WB_Uc2Sc1 Nominal

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.3 Integrated Use Case Realization

The final task in the architectural design phase is the *Integrated Use Case Realization*, i.e.the merger of the realized use case models in the *Integrated System Architecture Model*.

Before merging a realized use case model, care must be taken that in the two models all operation names and associated system requirements links are unique, i.e.

- If two operations with different names describe the same functionality and are linked to the same system requirement, the names need to be harmonized.
- If two operations have different names and describe different functionality but are linked to the same requirement, the system requirement needs to be split. For the child requirements respective trace links have to be established.
- If two operations have the same name and are linked to the same requirement but describe different functionality, the names need to be modified and the system requirement split accordingly. For the child requirements respective trace links have to be established. In the case of changes the realized use case model needs to be baselined accordingly.

Fig. 4-7 shows the Integrated Use Case Realization workflow.
The first step is the creation of *a Rhapsody Harmony* compliant project **SecSys_IA**. The realized (WB) use case model Uc1ControlEntry was chosen as first contributor and imported into this project.

It is important to keep in mind, that the subsequent integration of realized use cases essentially focuses on the integration of the respective architectural components. Once imported, additional steps are needed to enable the collaboration (ref. *Fig. 4-8*).

Regarding the import of the realized use case model Uc2_ControlExit, the relevant information will be captured in a separate *Rhapsody* project **Uc2ControlExit_HandOff**. It should be noted, that this project is only a temporary project, to be used only for the integration.

The use cases collaboration as well as the correctness and completeness of the integrated system architecture model will be verified through model execution.

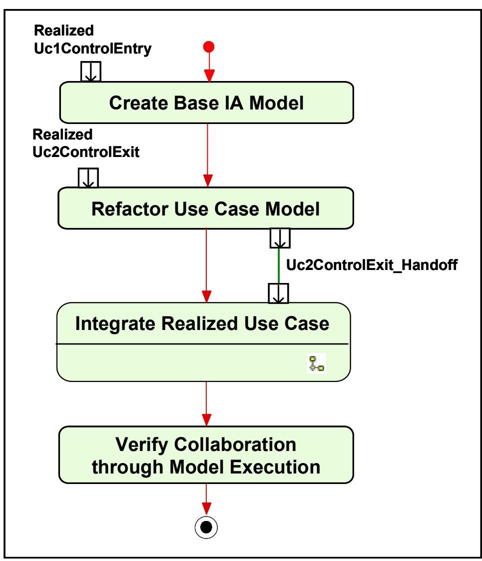


Fig. 4-7 Integrated Use Case Realization Workflow

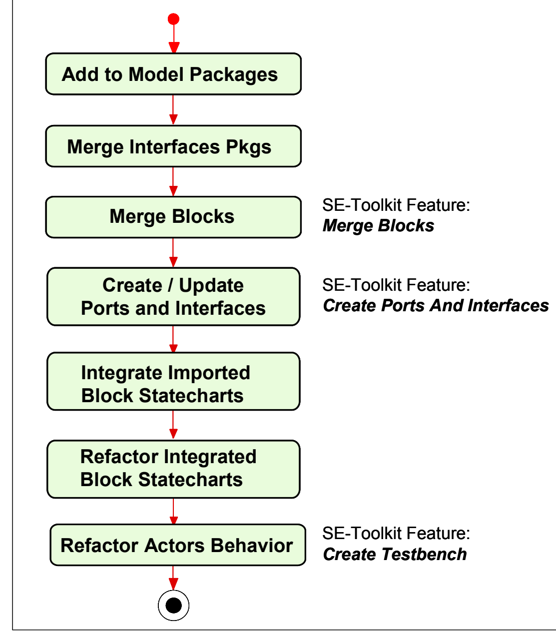


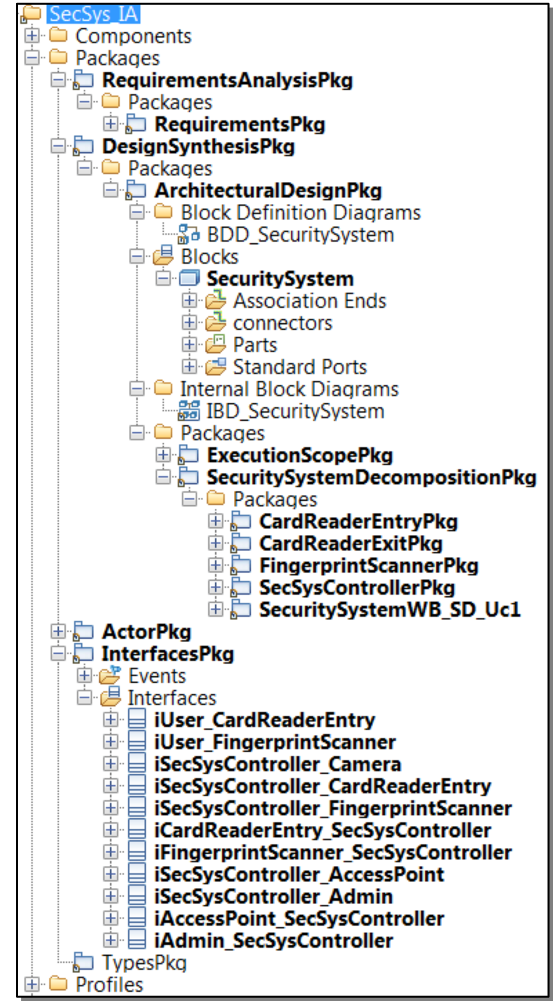
Fig. 4-8 Use Case Integration Task Flow and its Support through the Rhapsody SE-Toolkit

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.3.1 Creation of Base IA Model

In the case study the realized use case model Uc1ControlEntry_AD was chosen as the base *Integrated System Architecture Model*.

1. Create a Harmony compliant project **SecSys_IA**
2. In the RequirementsAnalysisPkg *Delete from Model* the UseCaseDiagramsPkg
3. *Delete from Model* the FunctionalAnalysisPkg
4. *Add to Model As unit* from the Rhapsody project Uc1ControlEntry the packages
   - ActorPkg.sbs
   - ArchitecturalDesignPkg.sbs
   - InterfacesPkg.sbs
5. *Delete from Model* all attributes and operations of the block SecuritySystem
6. In the InterfacesPkg
   - *Delete from Model* the Uc1_BB_InterfacesPkg
   - Move the interfaces in the Uc1_WB_Interfaces into the InterfacesPkg and *Delete from Model* the empty Uc1_WB_InterfacesPkg.
7. In the SecuritySystemDecompositionPkg *Delete from Model* the package SecuritySystemWB_AD_Uc1
8. In the ActorPkg *Delete from Model* all functional analysis related actor ports (pUc_Uc1ControlEntry).

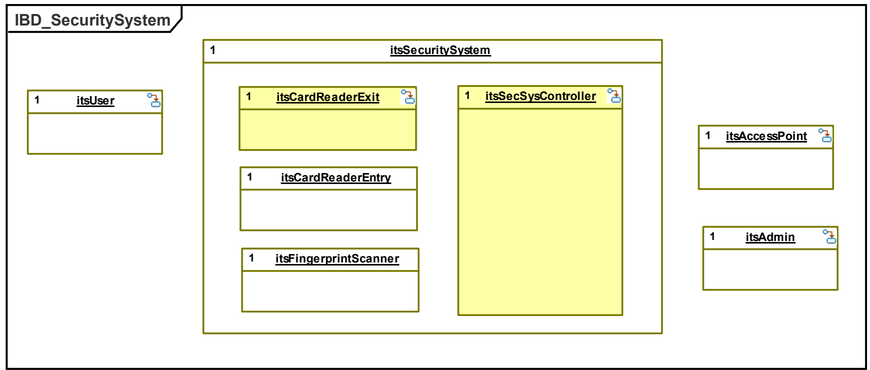


***Rhapsody Project Structure of* SecSys_IA Model**

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

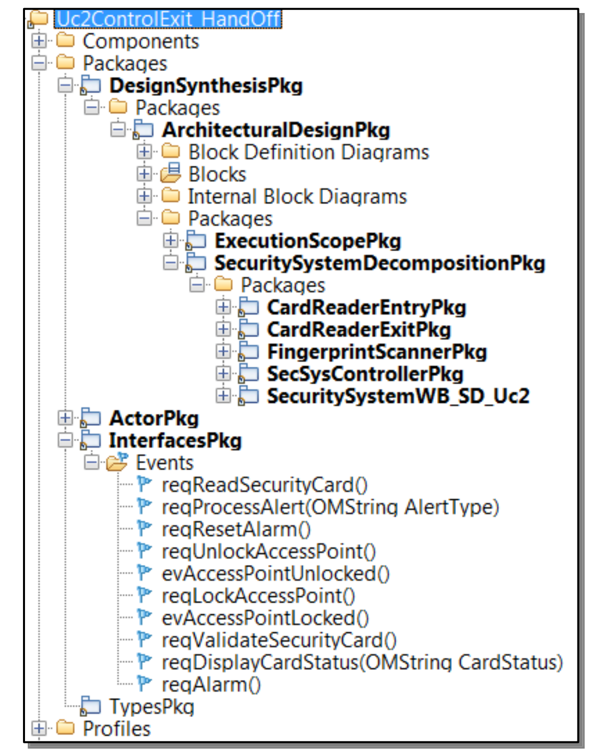
4.5.2.3.2 Configuring Realized Use Case Model Handoff

As mentioned in Section 4.5.2.3 only a subset of the realized use case Uc2ControlExit – i.e. the components (blocks) of the respective system architecture - will be integrated into the SecSys_IA model. For this purpose a specific handoff model needs to be configured.



***IBD of the HandOff Model* Uc2ControlExit_HandOff**

1. Create a Harmony compliant project **Uc2ControlExit_HandOff**
2. *Delete from Model*
   - RequirementsAnalysisPkg and
   - FunctionalAnalysisPkg
3. *Add to Model* from the Rhapsody project Uc2ControlExit the packages
   - ActorPkg.sbs
   - ArchitecturalDesignPkg.sbs
   - InterfacesPkg
4. In the InterfacesPkg *Delete from Model* the packages Uc2_BB_InterfacesPkg and Uc2_WB_InterfacesPkg
5. In the SecuritySystemDecompositionPkg *Delete from Model* the package SecuritySystemWB_AD_Uc2
6. In the IBD_SecuritySystem *Delete from Model* all ports and connections



***Rhapsody Project Structure of* Uc2ControlExit_HandOff**

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.3.3 Integration of Realized Use Case

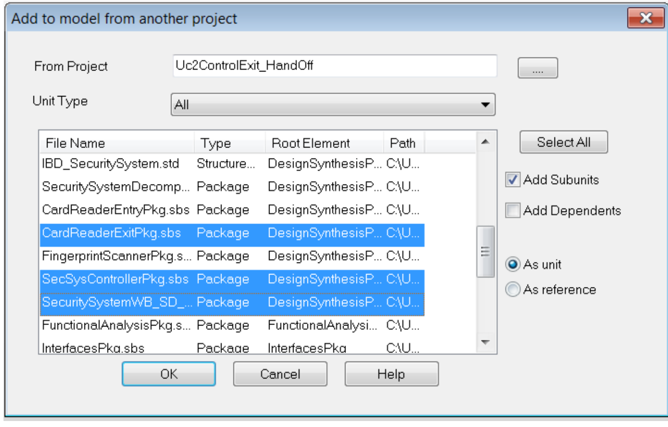
There are two concepts to integrate elements of two models

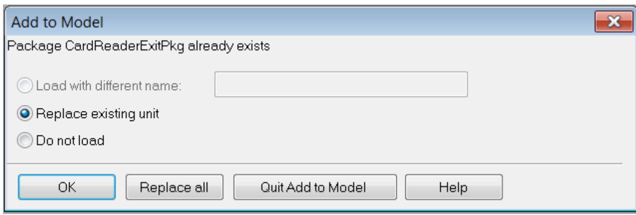
- Addition or replacement of model elements (Rhapsody feature *Add to Model*) or
- Combination of model elements (Rhapsody tool *Diff/Merge*)

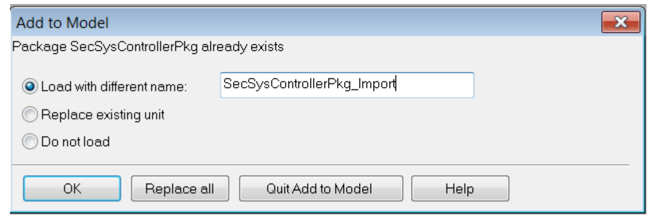
Step 1: Add to Model Packages

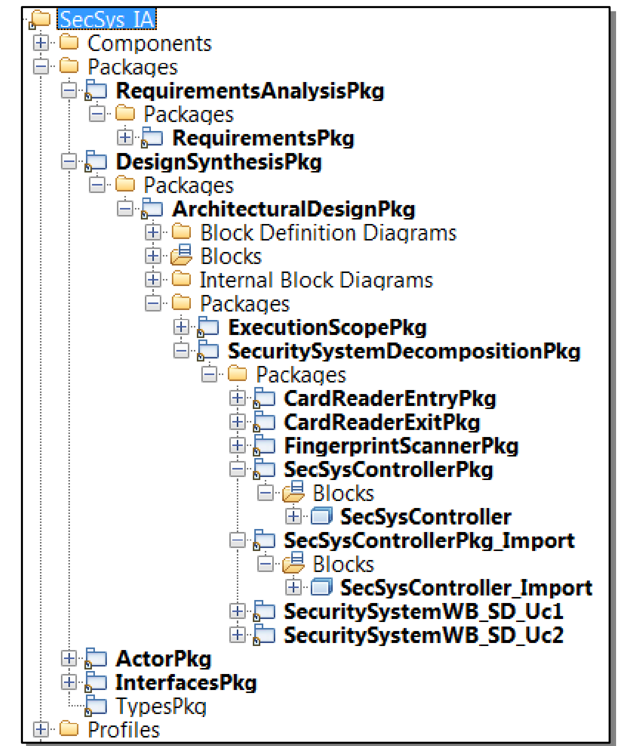
The following packages of the Uc2ControlExit_HandOff model are integrated into the SecSys_IA model using the *Add to Model As Unit* feature:

- CardReaderExitPkg.sbs,
- SecSysControllerPkg.sbs,
- SecuritySystemWB_SD_Uc2.sbs,









In the package SecSysController_Import rename of the block **SecSysController** to **SecSysController_Import**

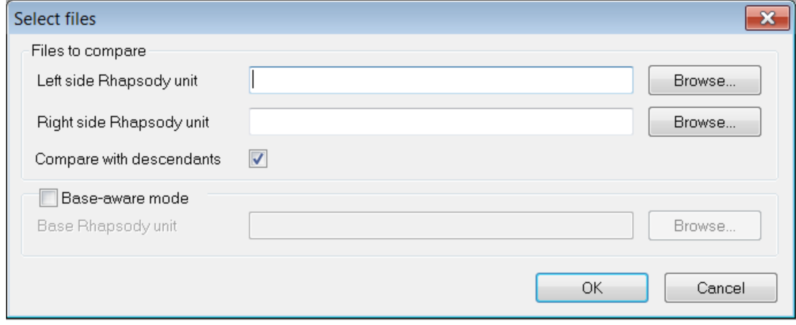
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Step 2: Merge Interfaces Packages

Interfaces packages of two models are merged by means of the ***Rhapsody Diff/Merge*** tool.

Preparation:
Create a Rhapsody project with an empty *InterfacesPkg.*
Save the model as **EmptyModel**.

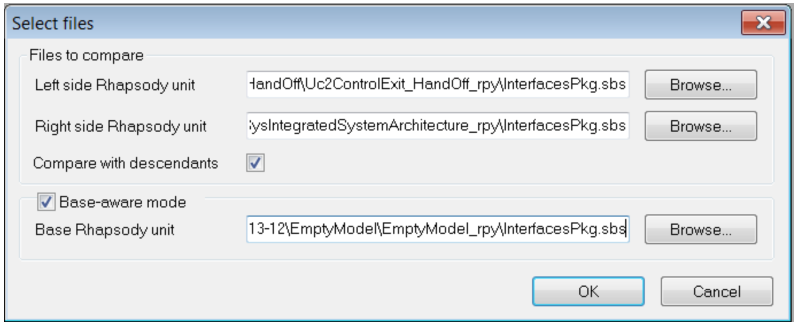
1. Launch ***Rhapsody Diff/Merge*** and from the menue select *File > Compare*



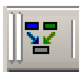
2. Select as *Left side Rhapsody unit* the **InterfacesPkg.sbs** in the **Uc2ControlExit_HandOff** model as

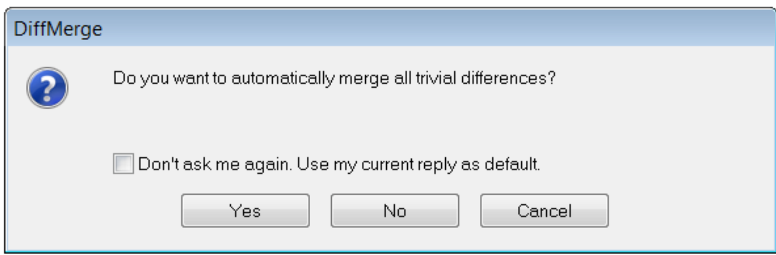
3. Select as *Right side Rhapsody unit* the **InterfacesPkg.sbs** in the **SecSys_IA** model

4. Check the Base-aware check-box and select the **InterfacesPkg.sbs** in the **EmptyModel** as Base Rhapsody Unit



5. Check the differences: There shall only be elements (interfaces and events) added or missing. If there are changes, they need to be resolved manually before continuing.

6. Click the *Start* button to start the merge process: 



Answer *Yes* to automatically merge all trivial differences. All differences will be resolved automatically

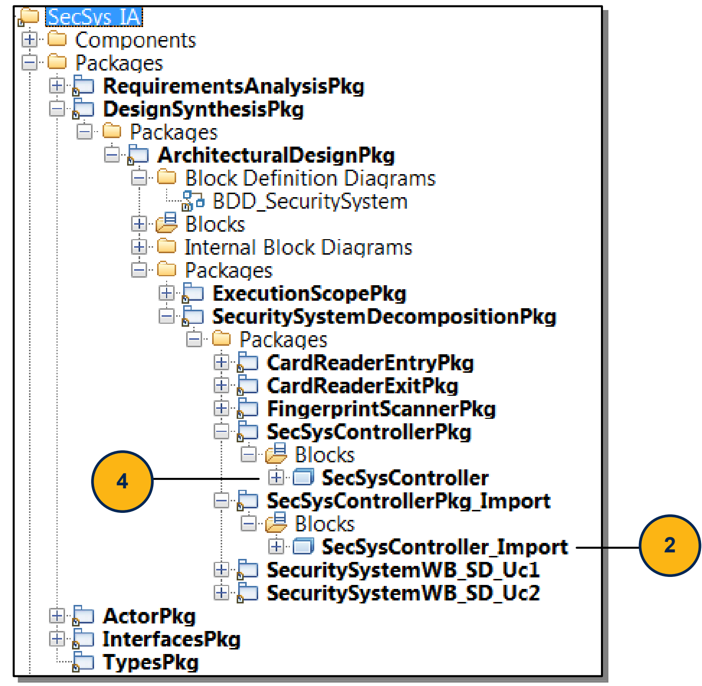
7. From the menue select: *File > Save merge as…*
Overwrite the **InterfacesPkg.sbs** of the **SecSys_IA** model.

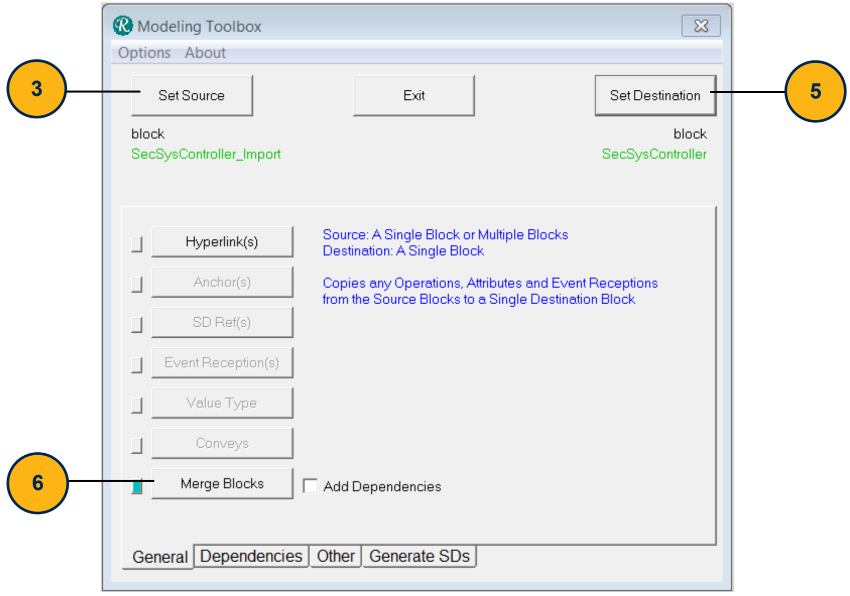
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Step 3: Merge Blocks

Operations, receptions and attributes of the block **SecSysControllerImport** are merged with the block **SecSysController** by means of the SE-Toolkit feature ***Merge Blocks***.

1. In the Tools Menu select *Tools > SE-Tookit > Modeling Toolbox > General*
2. In the SecSysControllerPkg_Import select block SecSysController_Import
3. In the Modeling Toolbox dialog box click *Set Source*
4. In the SecSysControllerPkg select block SecSysController
5. In the Modeling Toolbox dialog box click *Set Desination*
6. In the Modeling Toolbox dialog box click *Merge Blocks*





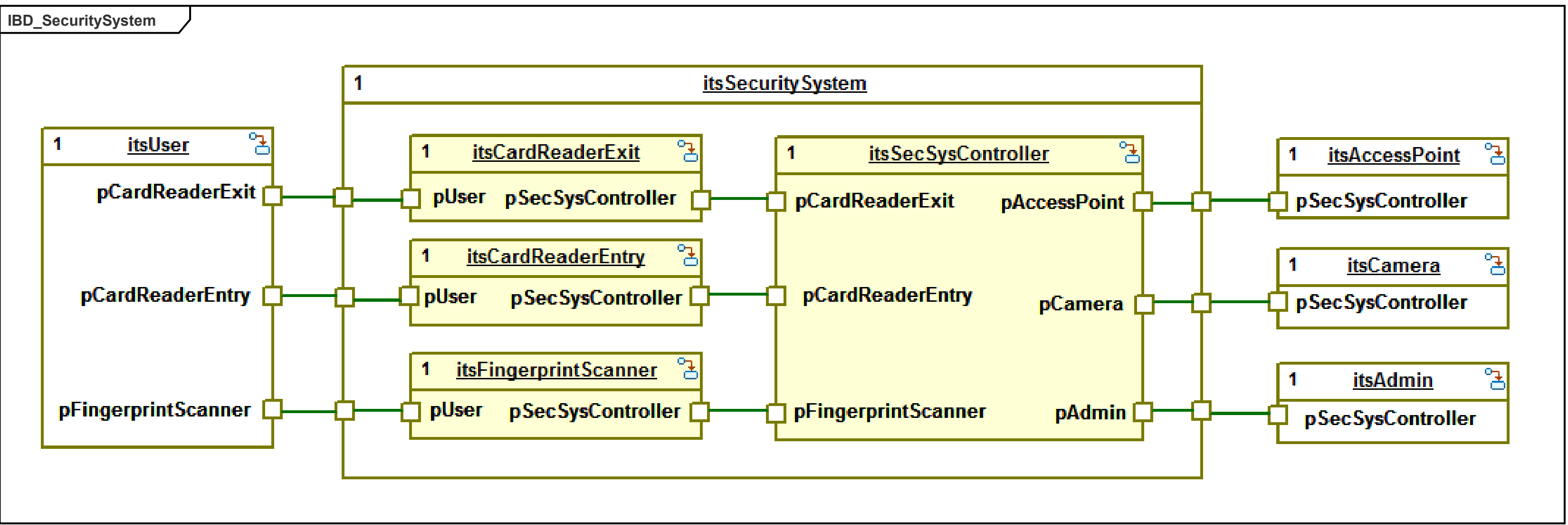
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Step 4: Create/update Ports and Interfaces

The ports and interfaces of the SecSys_IA model are created/updated by means of the SE Toolkit feature ***Create Ports And Interfaces***

NOTE: In the imported Sequence Diagram package update the lifeline names and check the autorealization status of messages.

1. Right-click SecuritySystemWB_SD_Uc2 and select ***SE-Toolkit > CreatePorts And Interfaces.***
2. Manually add Delegation Ports and associated interfaces
3. Manually connect ports



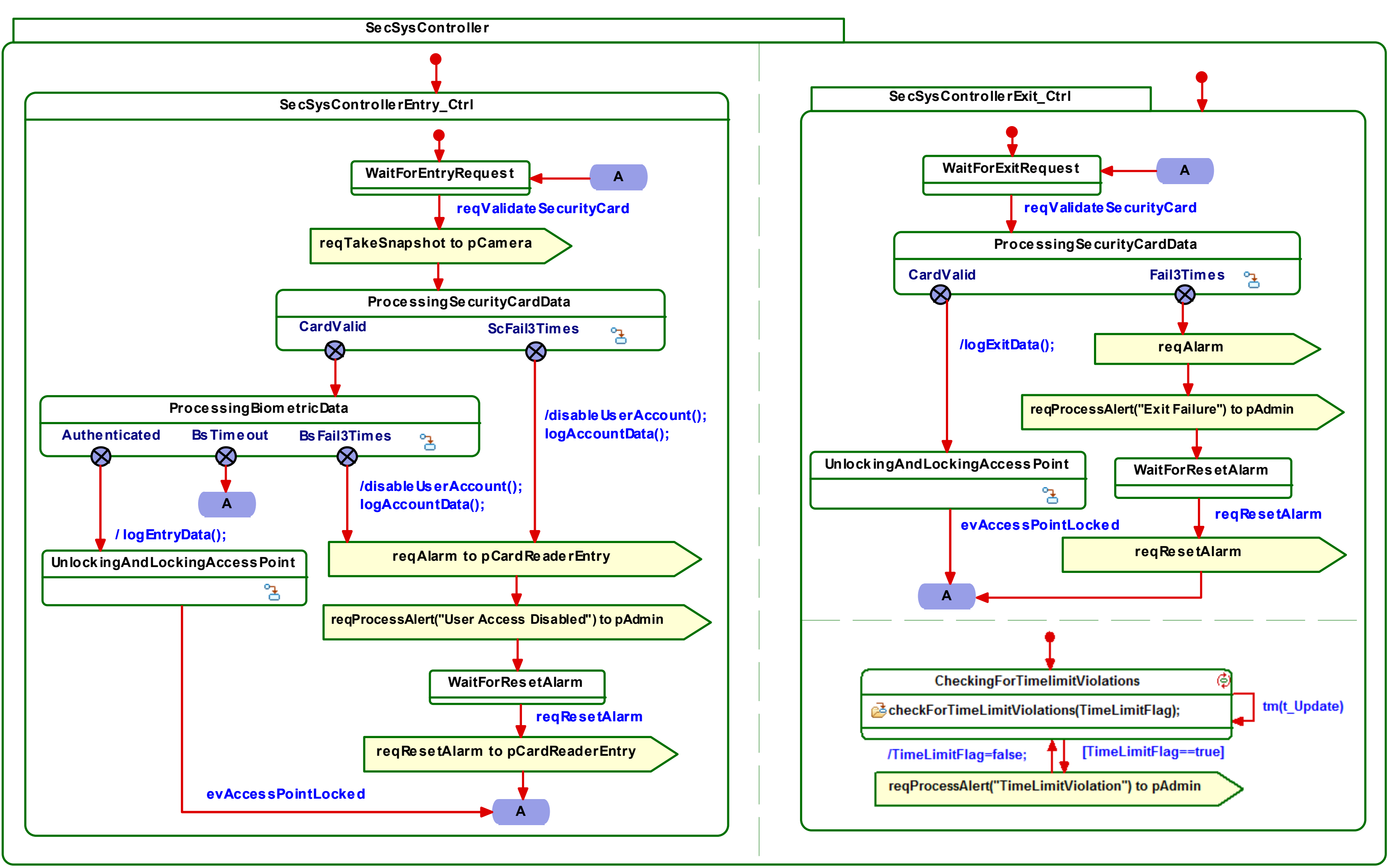
IBD of the updated SecSys_IA Model

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Step 5: Integrate Imported Block Statechart

1. Copy & paste the Statechart of the imported block into the Statechard of the merged block and make it a concurrent state.

2. Once the Statechart of the imported block is copied into the merged block statechart, *Delete from Model* the imported block package.



***Integrated State-based Behavior of Merged Block* SecSysController**

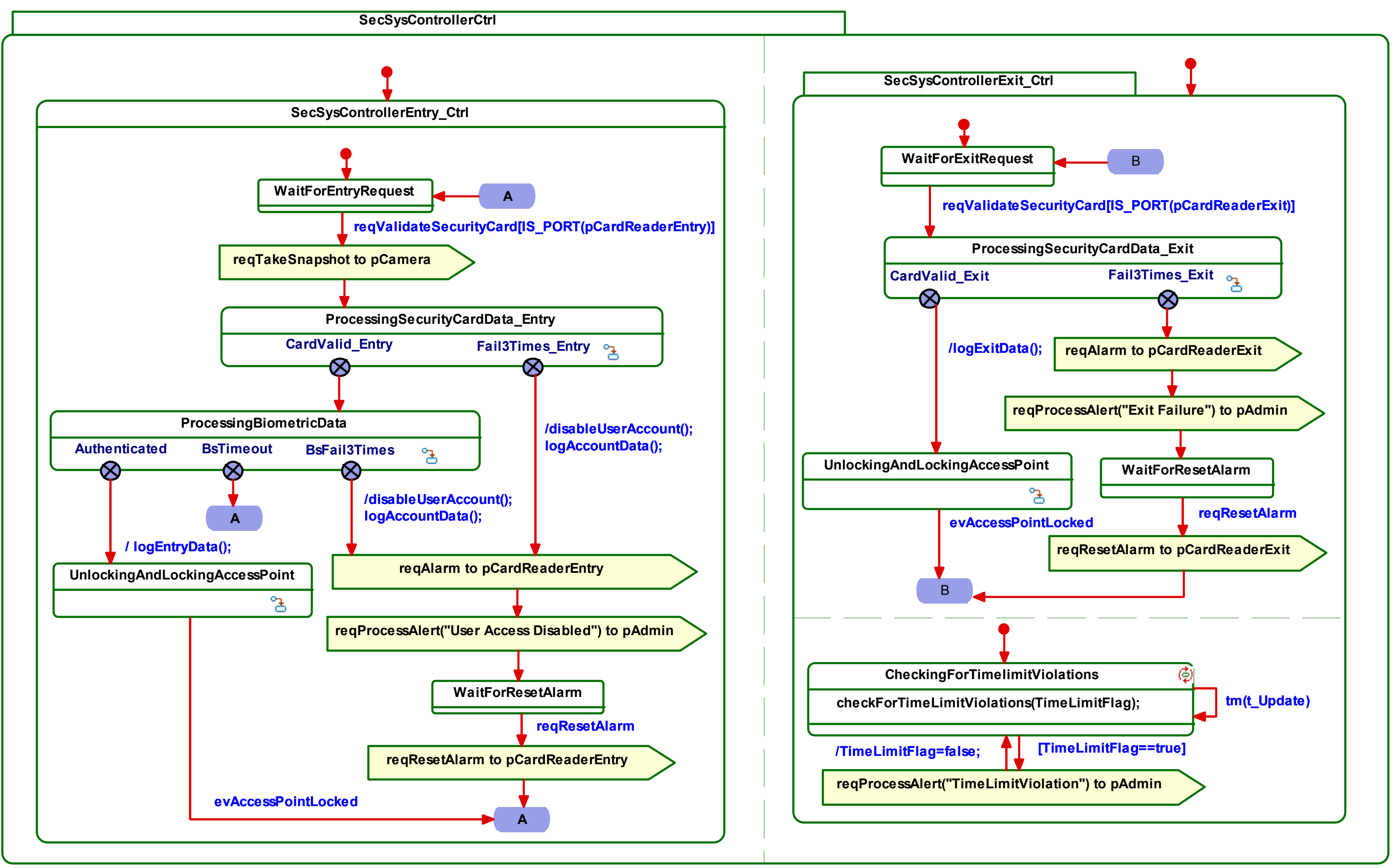
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Step 6: Refactor State-base Behavior

Statechart SecSysController_Ctrl Alternative 1

1. In the SecSysController_Exit_Ctrl Statechart and its Sub-Statecharts update the Send Actions and change the label of the connectors *A* and the EnterExit Points *Fail3Times and CardValid.*

2. Differentiate between the Entry request and Exit request

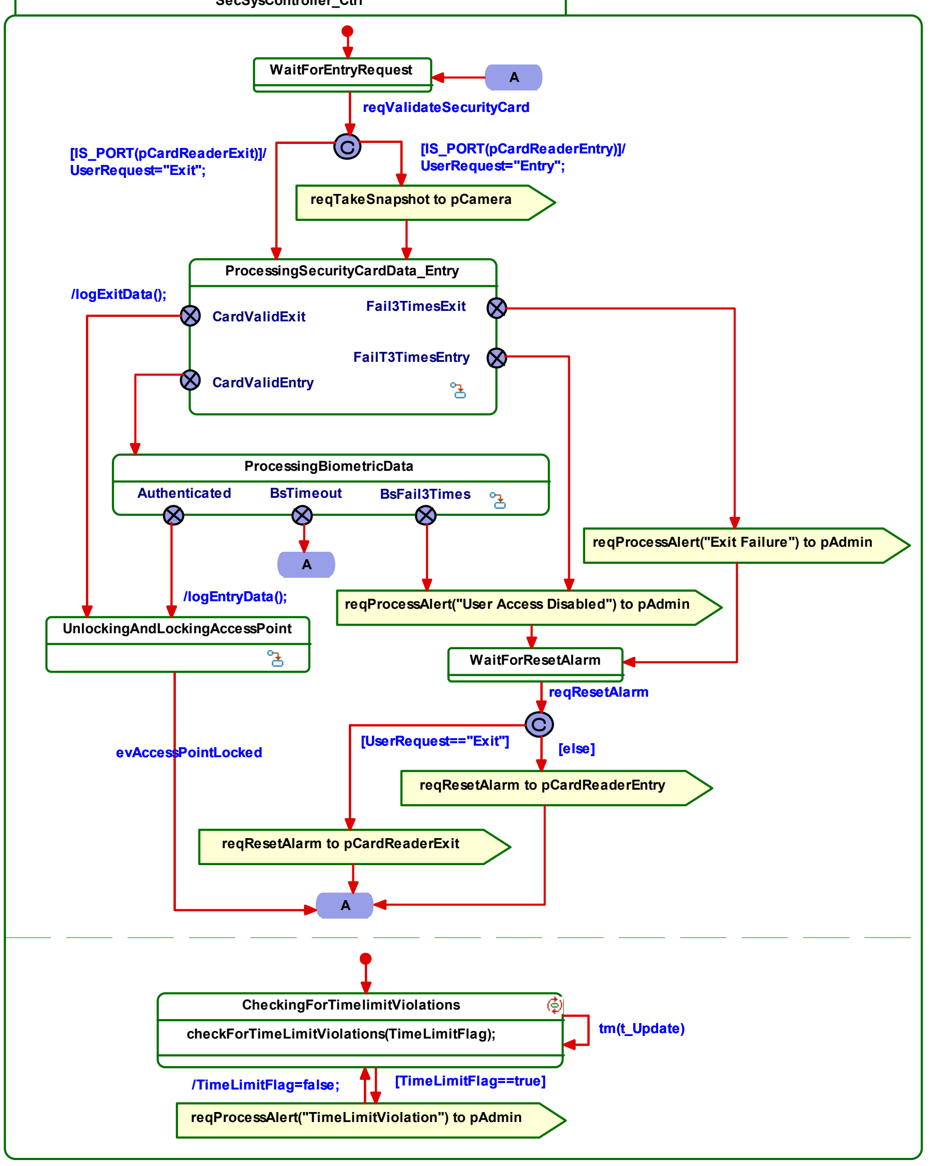


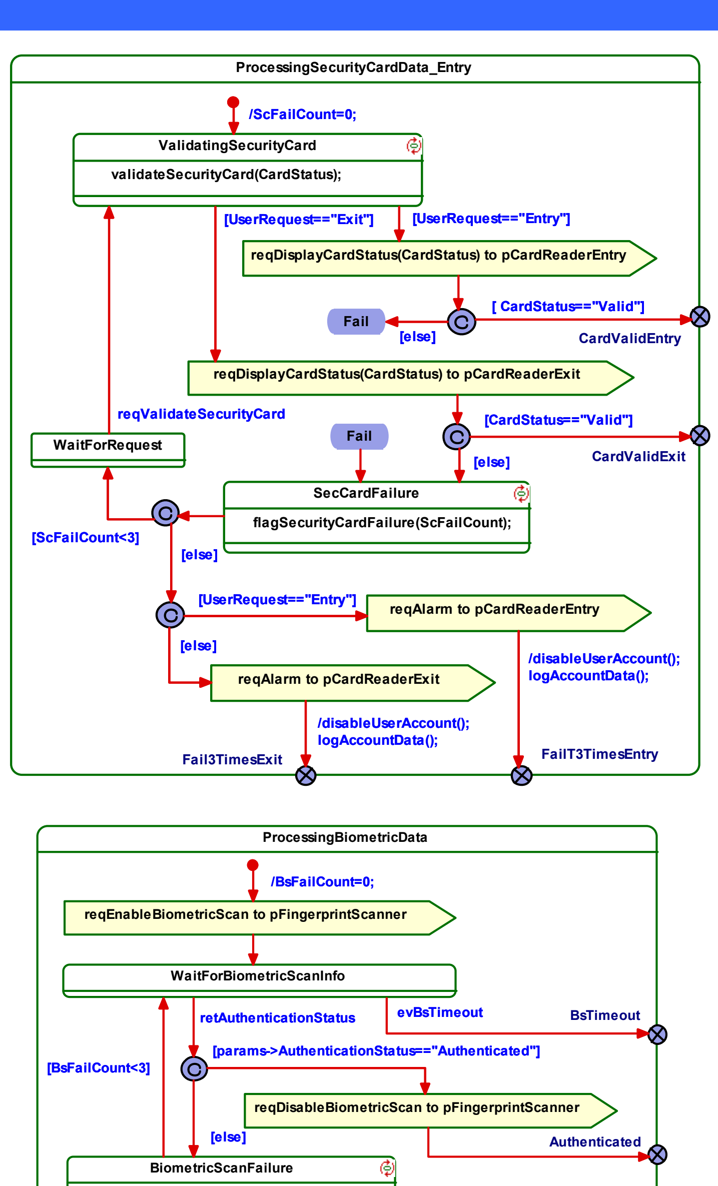
Refactored State-based Behavior of Block SecSysController (Alternative 1)

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Statechart SecSysControllerCtrl Alternative 2

An alternative strategy to refactor the integrated statebased behavior of the SecSysController block is to manually merge the concurrent processes SecSysControllerEntry_Ctrl and SecSysControllerExit_Ctrl.



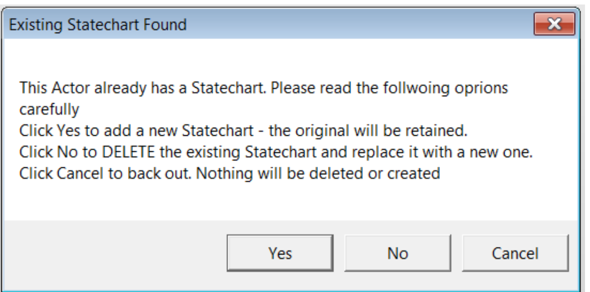


© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

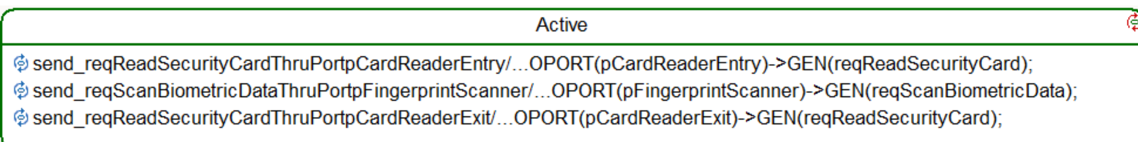
Actor Behavor

The behavior of the actor User needs to extended w.r.t. the additional request via the port pCardReaderExit.. The extension is performed by meas of the TE-Toolkit feature ***Create Testbench***.

1 In the Internal Block Diagram IBD_SeuritySystem right-click the User block and select *SE-Toolkit > Create Test Bench.*

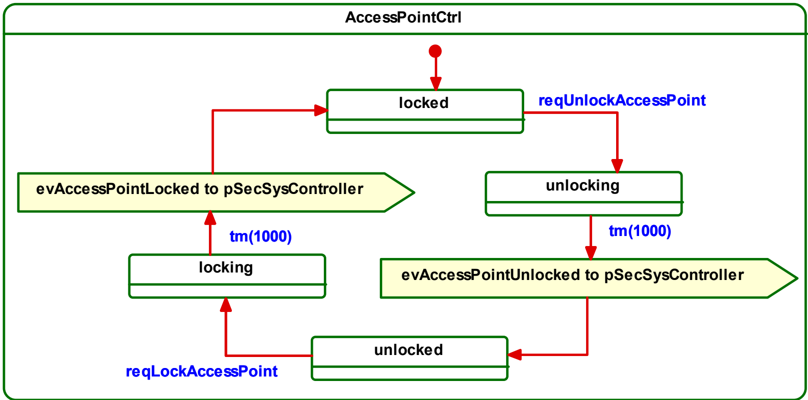


2 Select *No*



Extended Behavior of the Actor Block **User**

The state-based behavior of the actor block AccessPoint needs to be updated graphically.



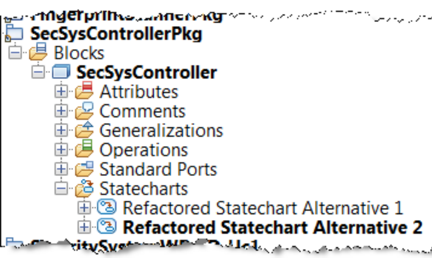
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

4.5.2.3.4 Verification of Use Cases Collaboration

The collaboraton of the merged realized use case models in the Integrated 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).

NOTE: The model verification should cover both alternative statecharts of the SecSysController block.

Right-click the statechart and select *Set As Main Behavior* and re-generate code.



:User
:CardReader Exit
:CardReader Entry
:Fingerprint Scanner
:SecSysController
:Admin
:AccessPoint
:Camera

Entry
reqReadSecurityCard()
readSecurityCard()
reqValidateSecurityCard()
reqTakeSnapshot()
validateSecurityCard(CardStatus = Valid)
reqDisplayCardStatus(CardStatus = Valid)
reqEnableBiometricScan()
displayCardStatus(CardStatus = Valid)
enableBiometricScan()
reqScanBiometricData()
scanBiometricData()
authenticateBiometricData(AuthenticationStatus = Authenticated)
displayAuthenticationStatus(AuthenticationStatus = Authenticated)
retAuthenticationStatus(AuthenticationStatus = Authenticated)
reqDisableBiometricScan()
logEntryData()
reqUnlockAccessPoint()
disableBiometricScan()
evAccessPointUnlocked()
reqLockAccessPoint()
evAccessPointLocked()

Exit
reqReadSecurityCard()
readSecurityCard()
reqValidateSecurityCard()
validateSecurityCard(CardStatus = Valid)
reqDisplayCardStatus(CardStatus = Valid)
logExitData()

SecSys_IA Verification Animated Sequence Diagram

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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. As outlined in Section 2.2.4, the hand-off to the subsequent development is an executable model derived from the baselined Integrated System Architecture Model.

1. Create a Harmony compliant Rhapsody project and name it **SecSysController_HandOff**

2. *Delete from Model* the

   - FunctionalAnalysisPkg and
   - UseCaseDiagramsPkg

3. *Add to Model As unit* from the **SecSys_RA** project the RequirementsPkg.sbs

4. *Add to Model As unit* from the **SecSys_IA** project

   - SecSysControllerPkg.sbs
   - SecuritySystemWB_SD_Uc1.sbs
   - SecuritySystemWB_SD_Uc2.sbs
   - InterfacesPkg.sbs

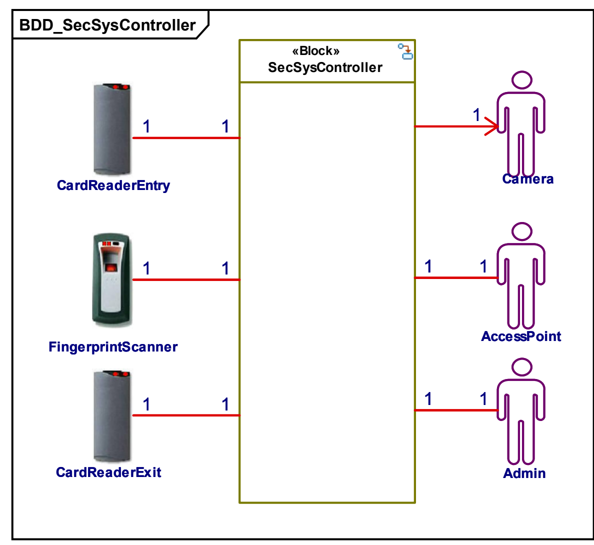
5. As the interfaces will be re-generated in this workflow, *Delete from Model* in the InterfacesPkg all interfaces.

6. Move the SecSysControllerPkg and the Sequence Diagram packages into the DesignSymthesisPkg.

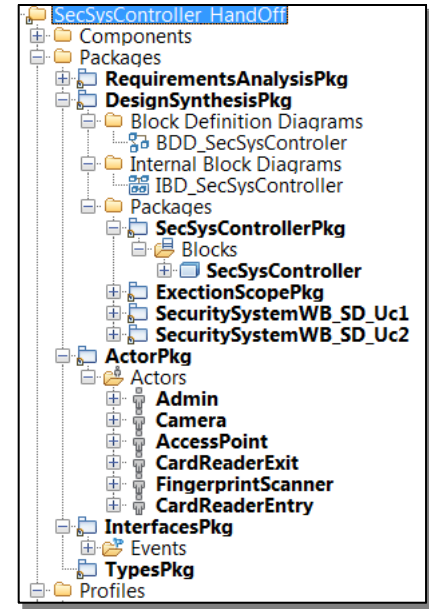
7. Manually add the following actors to the ActorPkg:

   - Admin
   - Camera
   - AccessPoint
   - CardReaderEntry
   - CardReaderExit
   - FingerprintScanner

8. Create a BDD_SecSysController and an IBD_SecSysController



BDD of the SecSysController_HandOff Model



Project Structure of the SecSysController_HandOff Model

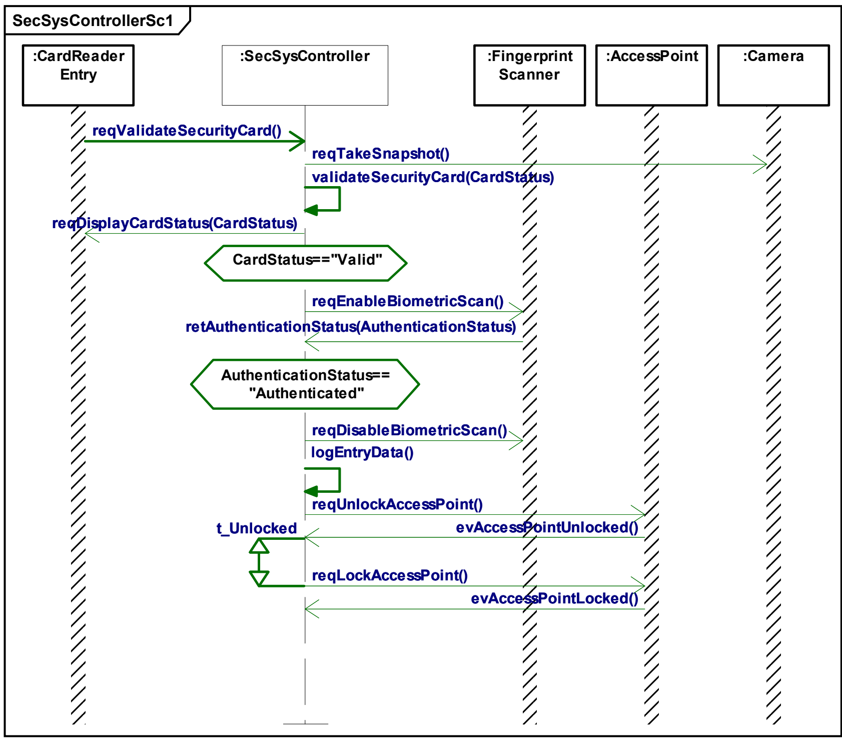
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

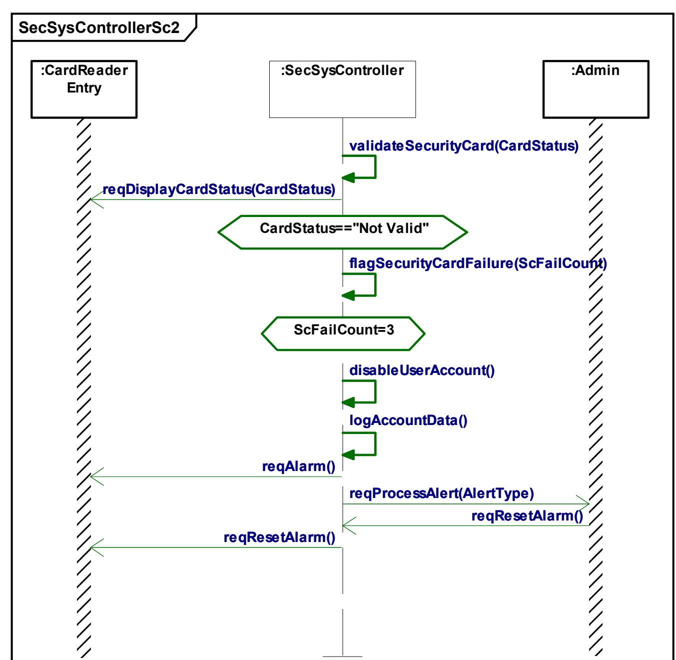
Taking into consideration the new system scope, update the Sequence Diagrams in the packages

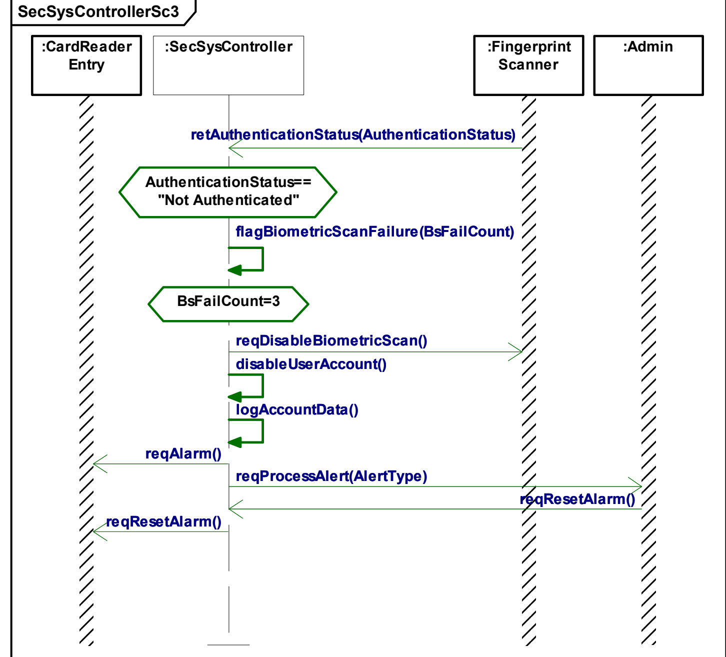
- SecuritySystemWB_SD_Uc1 and
- SecuritySystemWB_SD_Uc2

NOTE: Do not forget to autorealize the message to the actors.

It is recommended to rename the updated Sequence Diagrams and to move them into a new package (ref. below).







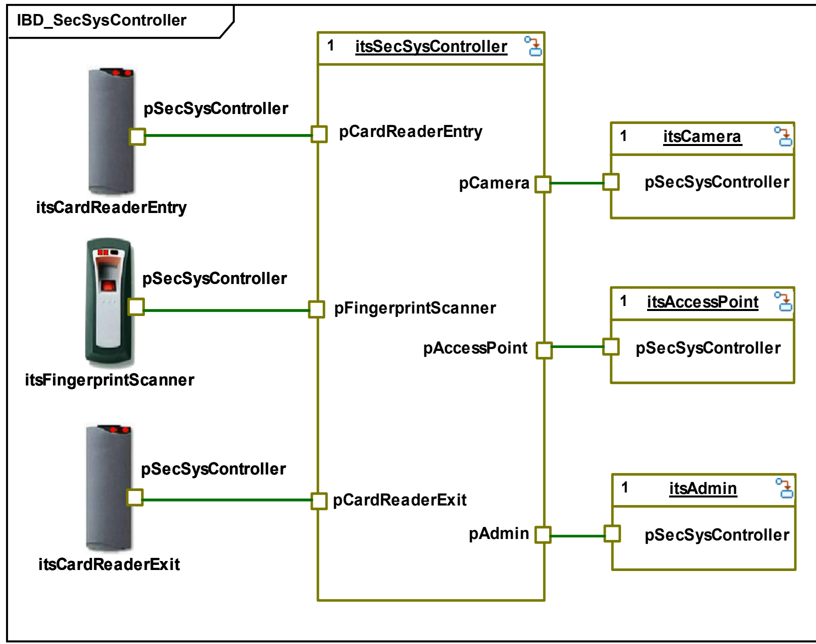
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

10 Based on the updated Sequence Diagrams, create ports and interfaces by means of the respective SE-Toolkit feature.

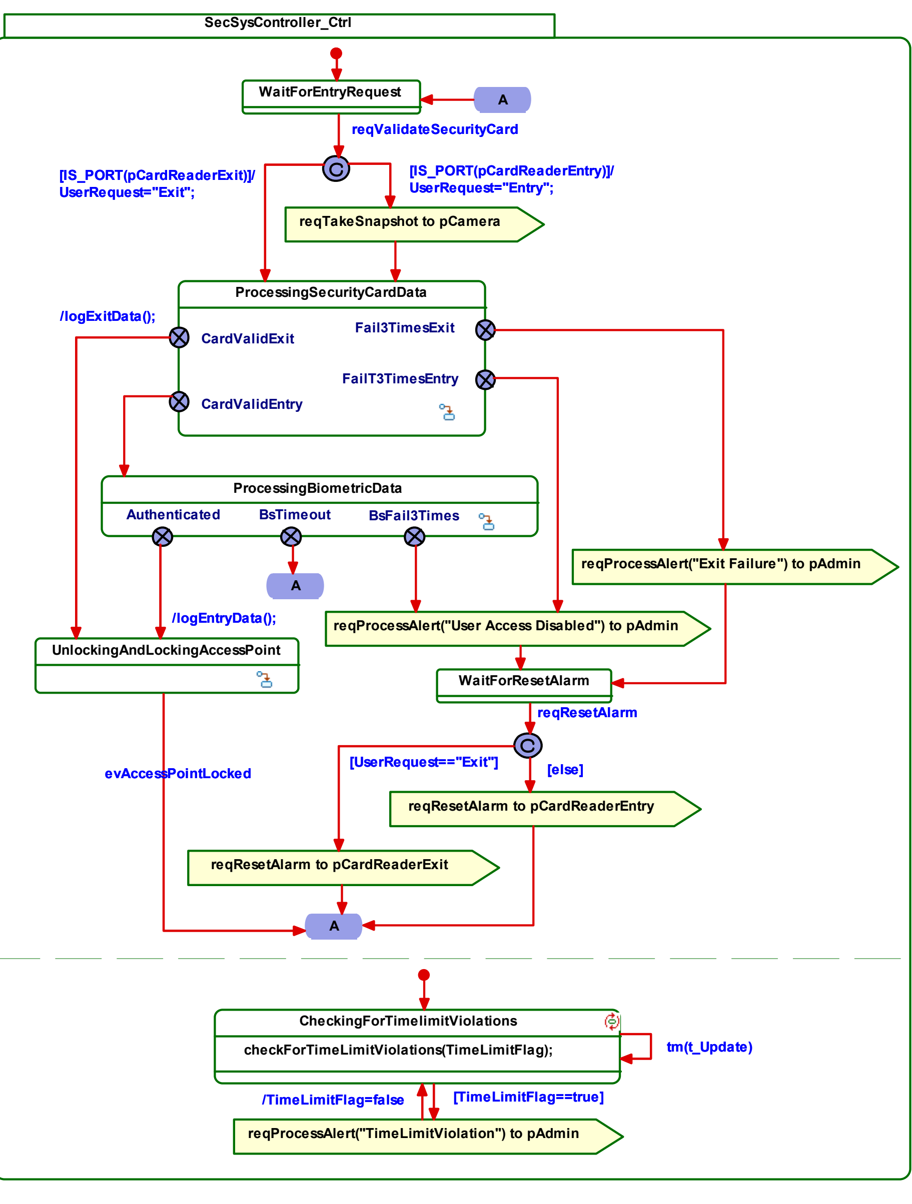
11 Capture by means of the SE-Toolkit feature **Create Testbench**. the behavior of the actor blocks

- CardReader Entry
- FingerprintScanner
- CardReaderExit
- Camera
- Administrator

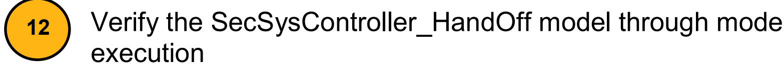
The state-based behavior of the actor block AccessPoint describe graphically (ref. SecSys_IA).



Populated IBD of the SecSysController_HandOff Model



State-based Behavior of the SecSysController_HandOff Model.
For Sub-Statecharts refer to the IA Model (Section 4.5.2.3)



12 Verify the SecSysController_HandOff model through model execution

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Systems Requirements Coverage of the SecSysController_HandOff Model

NOTE: The table below was generated by means of the Rational Publishing Engine (RPE)

ID	System Requirement	Requirement Type	Satisfied by
SYS1	**Three Attempts On Employee ID Entry** Upon entry the user shall be allowed three attempts on card identification.	Functional	ScFailCount flagSecurityCardFailure
SYS2	**Three Attempts On Biometric Data Entry** Upon entry the user shall be allowed three biometric data entries.	Functional	BsFailCount flagBiometricScanFailure
SYS3	**Disabling User Account** After three failed attempts at card identification or biometric data entry the user account shall be disabled.	Functional	disableUserAccount
SYS4	**Denied Entry Notification** Any denied access attempt shall be logged and account details sent to the administrator.	Functional	logAccountData logEntryData reqProcessAlert
SYS5	**Out of Date Cards** Out of date cards shall deny entry and invalidate the card.	Functional	validateSecurityCard
SYS6	**Authorization of Security Card – Entry** Access to the secure area shall only be allowed with a valid security card.	Functional	CardStatus
SYS7	**Two Independent Security Checks** Secure areas shall be protected by two independent security checks.	Functional	SecSysController
SYS8	**Alarm – Entry** On a denied entry an alarm signal shall be raised.	Functional	reqAlarm
SYS9	**Employee ID Card Identification – Entry** Entry shall be protected by a security check based upon employee ID.	Functional	validateSecurityCard
SYS10	**Visualization of Security Card Check Status – Entry** The user shall be visually informed about the status of his/her ID card check.	Functional	reqDisplayCardStatus
SYS11	**Security Card Information** Security cards only contain the employee name and ID and will be renewed yearly.	Functional	validateSecurityCard
SYS12	**Visualization of Biometric Data Check Status** The user shall be visually informed about the status of his/her biometric data check.	Functional	retAuthenticationStatus
SYS13	**Approval of Biometric Data** The user shall not be allowed access unless his/her biometric data are recognized.	Functional	AuthenticationStatus
SYS14	**Biometric Scan** Entry to the secure areas shall be protected by a second independent security check, based upon biometric data.	Functional	reqEnableBiometricScan
SYS15	**Image Capture** An image shall be taken of any person, at the initial attempt, when trying to access a secure area.	Functional	reqTakeSnapshot

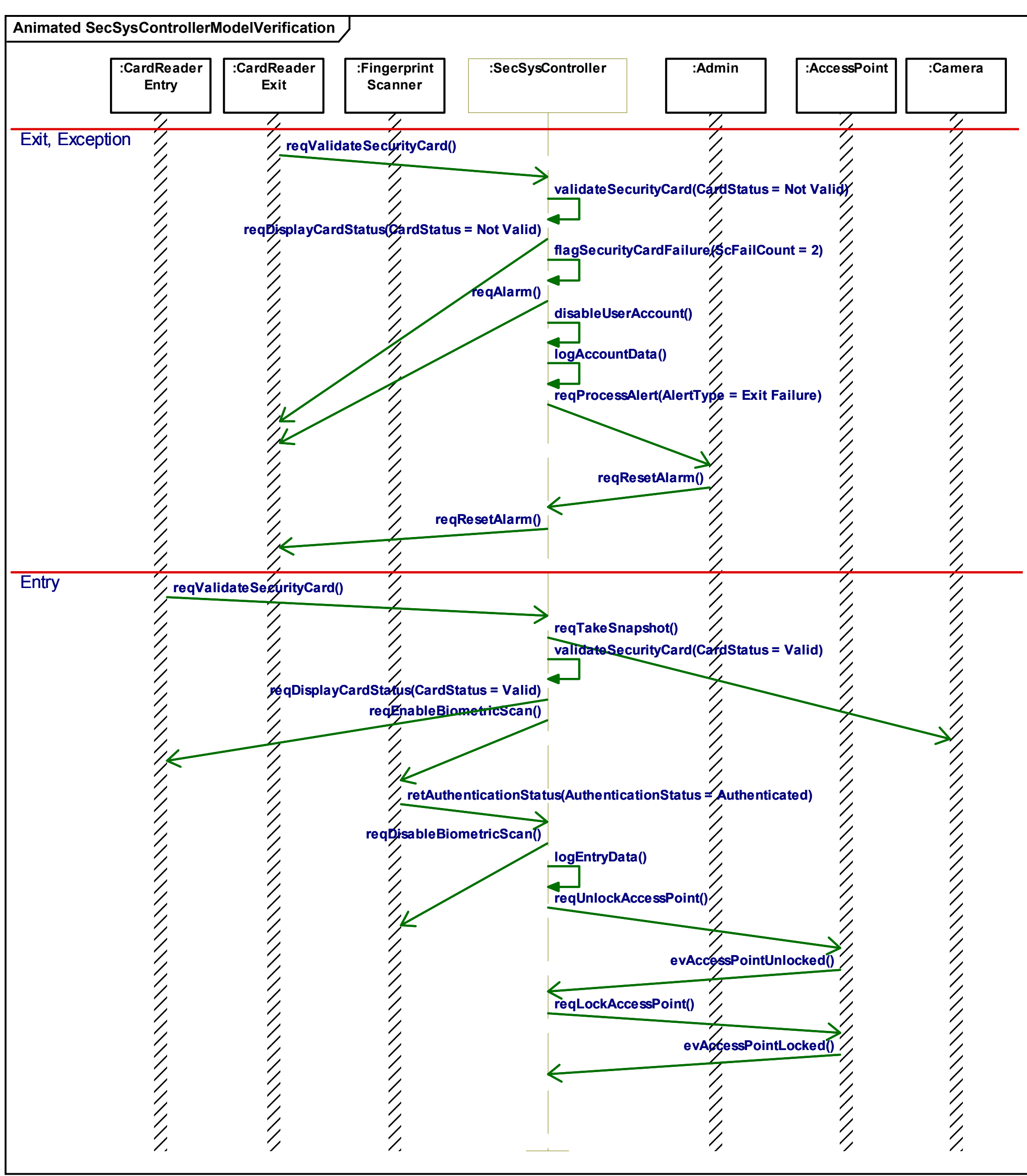
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

System Requirements Coverage of the SecSysController_HandOff Model (cont'd)

ID	System Requirement	Requirement Type	Satisfied by
SYS16	**Three Attempts On Employee ID Exit** Upon exit the user shall be allowed three attempts on card identification.	Functional	flagSecurityCardFailure
SYS16	**Three Attempts On Employee ID Exit** Upon exit the user shall be allowed three attempts on card identification.	Functional	flagSecurityCardFailure
SYS17	**Time Limit Violation** An alarm shall notify if a person stays longer than 10 hours in the secure area.	Functional	checkForTimeLimitViolations
SYS18	**Denied Exit Notification** The administrator shall be notified about any denied exit. The notification shall include user account details.	Functional	logExitData reqProcessAlert
SYS19	**Alarm – Exit** On a denied exit an alarm signal shall be raised.	Functional	reqAlarm
SYS20	**Employee ID Card Identification – Exit** Exit shall be protected by a security check based upon employee ID.	Functional	validateSecurityCard
SYS21	**Visualization of Security Card Check Status – Exit** The user shall be visually informed about the status of his/her ID card check.	Functional	reqDisplayCardStatus
SYS24	**Authorization of Security Card – Exit** The user shall not be allowed to exit until the security card has been successfully authorized.	Functional	CardStatus
SYS25	**Entry Time** The user shall be given sufficient time to enter the secure area.	Non-Functional	t_Unlocked
SYS26	**Time Between Two Independent Checks** The time between the two independent security checks shall not exceed a configurable period.	Non-Functional	evBsTimeout
SYS27	**Processing User Request** The system shall only process one user at a time.	Non-Functional	SecSysController
SYS28	**Biometric Data Storage** Biometric data shall be stored in the system database and not on the security card.	Non-Functional	SecSysController
SYS29	**Time Recording** The time a user spends in a secure area shall be recorded.	Non-Functional	SecSysController logExitData
SYS30	**Exit Time** The user shall be given sufficient time to exit the secure area.	Non-Functional	SecSysController
SYS31	**Automatic Securing the Secure Area – Entry** Once the user has entered the secure area the system shall automatically secure itself.	Functional	evAccessPointLocked
SYS32	**Automatic Securing the Secure Area – Exit** Once the user has exited the secure area the system shall automatically secure itself.	Functional	evAccessPointLocked
SYS33	**Configuration of Entry and Exit Time** The time to enter and exit the secure area shall be customizable.	Non-Functional	SecSysController

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Verification of the SecSysController_HandOff Model through Model Execution



NOTE:
Timeout events intentionally are not shown in this diagram.

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

6 Appendix

A1 *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 and communication 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.

A1.1 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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A1.2 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

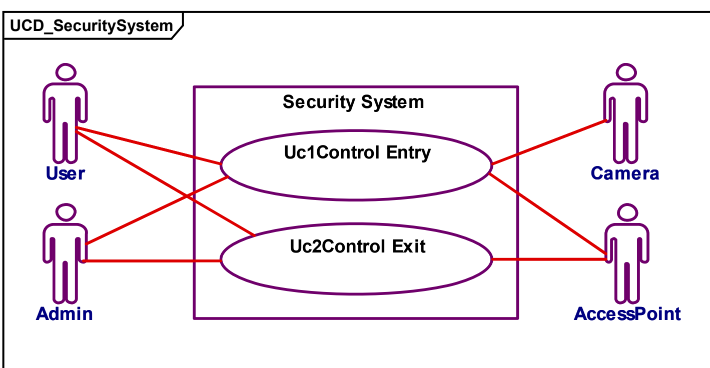
Use Case: A use case defines the system context. Name use cases using verbs that describe their ultimate goal.

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.



Use Case Diagram

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**<*Nr*> <*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**<*Nr*>**_UC**<*Nr*> <*Use Case Name*>.
- <u>*Note:*</u> Use case names may have spaces
- The use case name shall start with a verb.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A1.3 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 subsytems 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 re-usable 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.

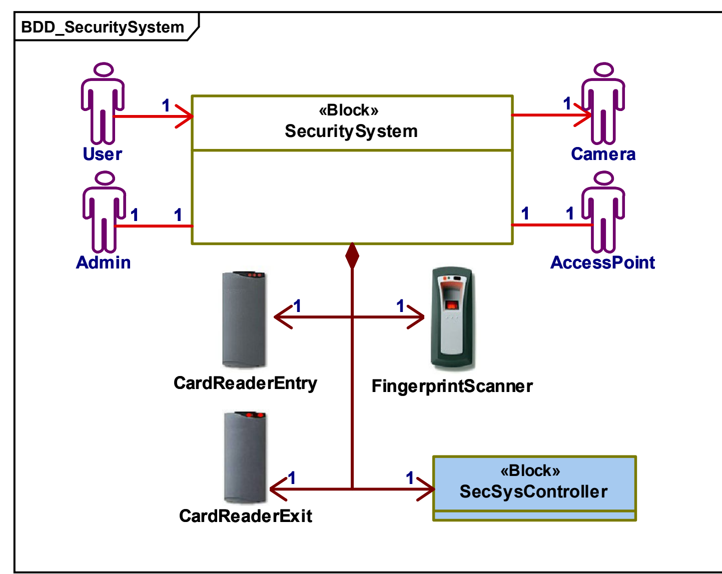
Association: Represents a bidirectional relationship between system blocks and actors.

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.



Block Definition Diagram

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.

Naming Conventions

The name of a block definition diagram should have the pre-fix "**BDD_**".

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A1.4 Internal Block Diagram

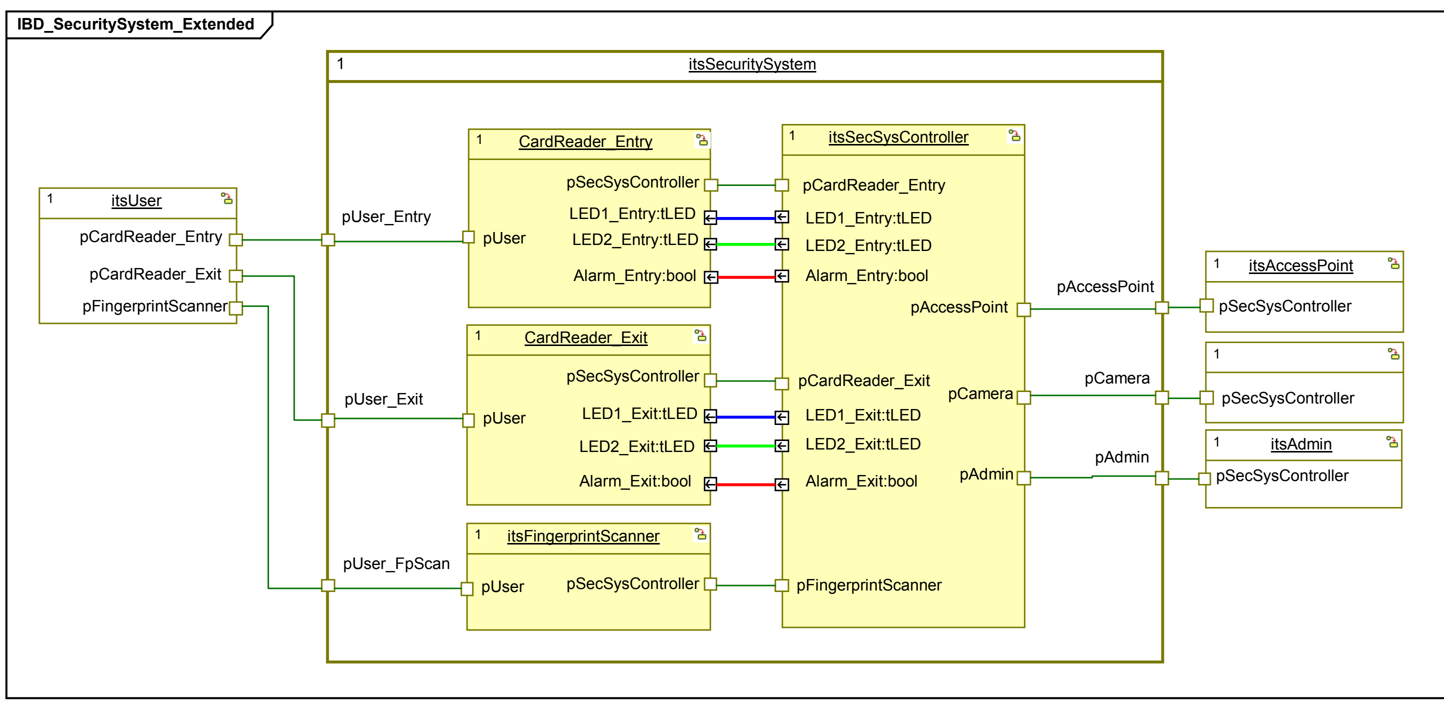
The SysML *Internal Block Diagram* shows the realization of the system structure defined in the Block Definition Diagram. It is comprised of a set of nested *parts* (i.e. instances of the blocks) that are inter-connected via ports and connectors.

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.



Internal Block Diagram

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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 focussed 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*>

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A1.5 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.

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 subactivities.

NOTE: It is recommended not to use subactivities. 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.

Call Behavior: References to another activity diagram as an activity.

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.

Merge Node: Routes each input received to the output. Unlike the Join Node it does not require tokens on all its inputs before offering them on its output flow.

Swim Lane Frame: Draws a frame around the entire set of activitie 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.

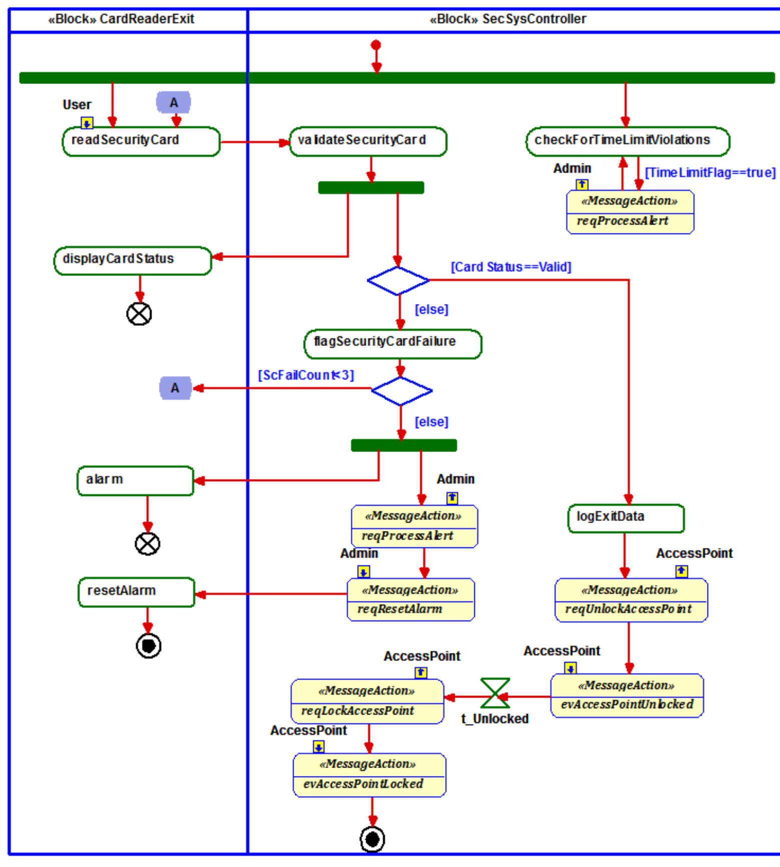
Decision Node: A condition connector splits a single control flow into multiple branches, each containing a guard. The guards on each branch should be orthogonal conditions, though they do not need to cover all possibilities. An “else” guard should be added to provide a default branch when no other guards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

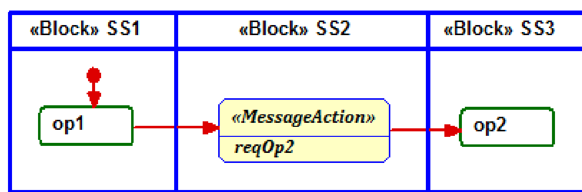
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

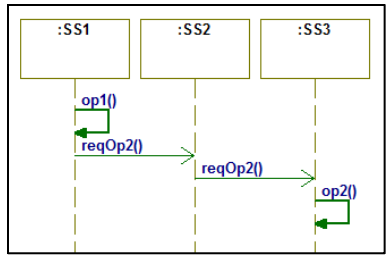


Activity Diagram

Guidelines and Drawing Conventions

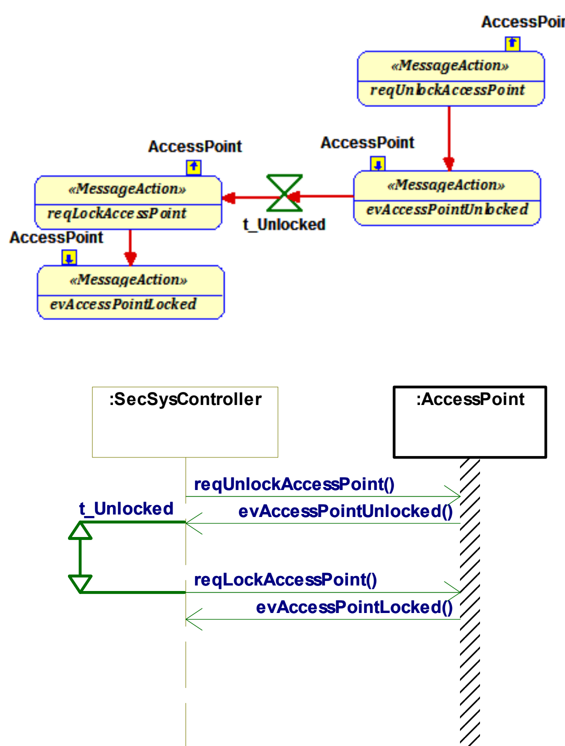
- In *Harmony for Systems Engineering*, the activity diagram is used exclusively to describe the functional flow through a use case. Therefore, select the activity diagram mode “”Analysis”.
- Document the pre-conditions in the respective tag of the diagram.
- Actor swim lanes should not be used. The link of an activity to the actor should be described through action pins, stereotyped *<<ActorPIN>>>.*
- When performing an operational 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 action 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 A3A6 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 action, drag that activity diagram from the browser onto the diagram. This creates a *call behavior* action that links to the external activity diagram.
- Generally, an action should correspond to an operation to be performed in the associated block. Exception: Actions stereotyped <<*Message Action*>> which describe the reception or transmittion of a message, e.g.





© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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

Naming Conventions

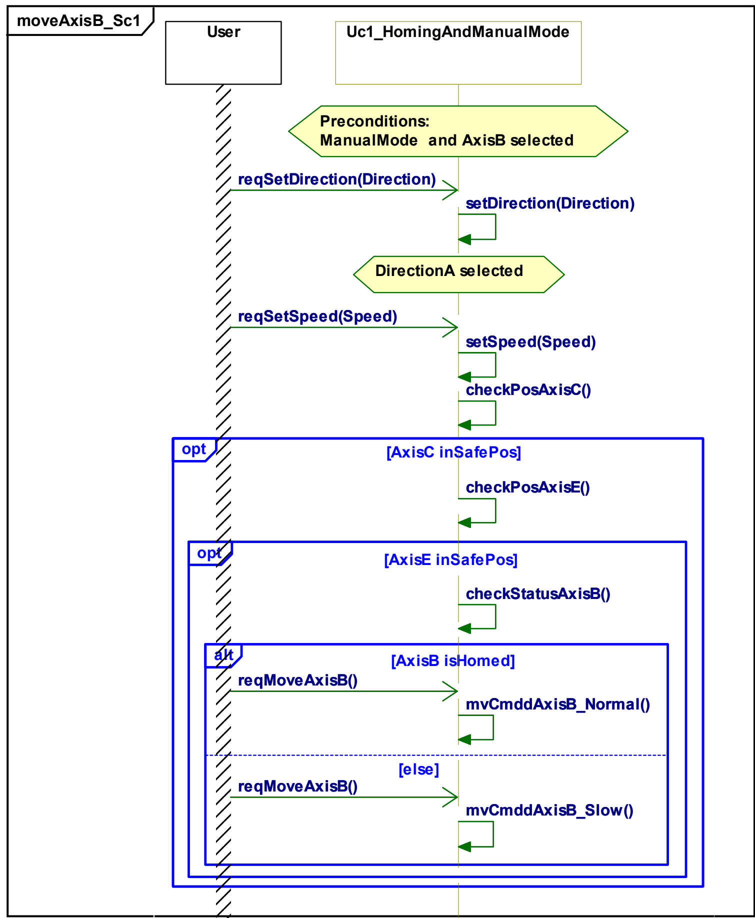
- The diagram shall have the associated use case name in plain text at the top of the diagram.
- Activity names shall start with a verb, beginning with a lover case letter, and map directly to the names of operations on system blocks.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A1.6 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.



Sequence Diagram

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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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:

Name	Description
req<*Service*>	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
ret<*Service*>Status	Used to provide results of a service (operation) back to the requester. Example: retAuthenticateBiometricDataStatus
ev<*Event*>	Used to send a notification of change Example: evAccessPointLocked

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A1.7 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.

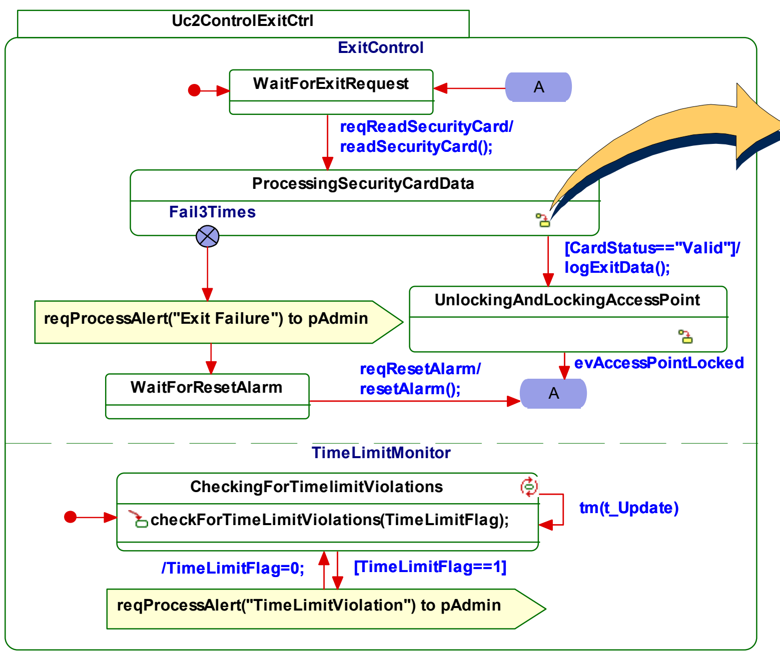
Junction Connector: A junction connector helps manage diagram complexity by combining several incoming transitions into a single outgoing transition.

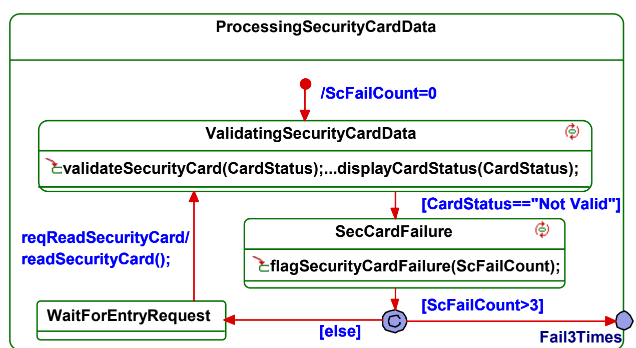
Diagram Connector: A diagram connector helps manage diagram complexity by allowing jumping to different sections of the statechart diagram to avoid line crossing.

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

Send Action State: Graphical representation of a send signal action.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.





Statechart Diagram

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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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”.

A1.8 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. A1-2 shows the properties of a project-specific profile that supports the modeling guidelines outlined in the previous sections. *Tab. A1-1* depicts the definition of element tags that was added to the profile in order to support the documentation.

Tag	Applicable To	Type
PreCondition	Use Case Sequence Diagram Primitive Operation	String
PostCondition	Use Case Sequence Diagram Primitive Operation	String
Constraint	Use Case Sequence Diagram Primitive Operation	String

Tab. A1-1 Project-Specific Tags Defined in a Profile

Property	Value
Activity_diagram>Transition>line_style	rectilinear_arrows
Activity_diagram>DefaultTransition>line_style	straight_arrows
Statechart>Transition>line_style	rectilinear_arrows
Statechart>DefaultTransition>line_style	straight_arrows
Statechart>CompState>ShowCompName	false
SequenceDiagram>General>HorizontalMessageType	Event
SequenceDiagram>General>SelfMessageType	PrimitiveOperation
SequenceDiagram>General>ShowwAnimStateMark	false
ObjectModelGe>Actor>ShowName	Name_only
ObjectModelGe>Class>ShowName	Name_only
ObjectModelGe>Object>ShowName	Name_only
ObjectModelGe>Inheritance>line_style	rectilinear_arrows
ObjectModelGe>Depends>line_style	rectilinear_arrows
ObjectModelGe>Class>ShowPorts	false
ObjectModelGe>Class>ShowPortsInterfaces	false
UseCaseGe>Actor>ShowName	Name_only
UseCaseGe>UseCase>ShowName	Name_only

Tab. A1-2 Project-Specific Properties Defined in a Profile

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A2 *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 an example the simplified use case Uc2*ControlExit.*

Fig. A2-1 depicts the black-box activity diagram 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 actions joined by transitions and condition connectors.

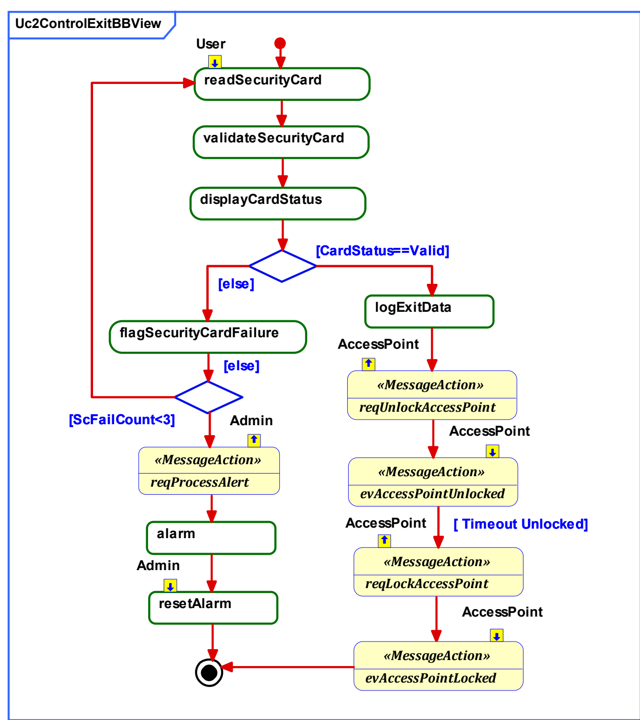


Fig. A2-1 Use Case Uc2_ControlExit Black-Box Activity Diagram

Fig. A2-2 shows the black-box sequence diagram that was 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.

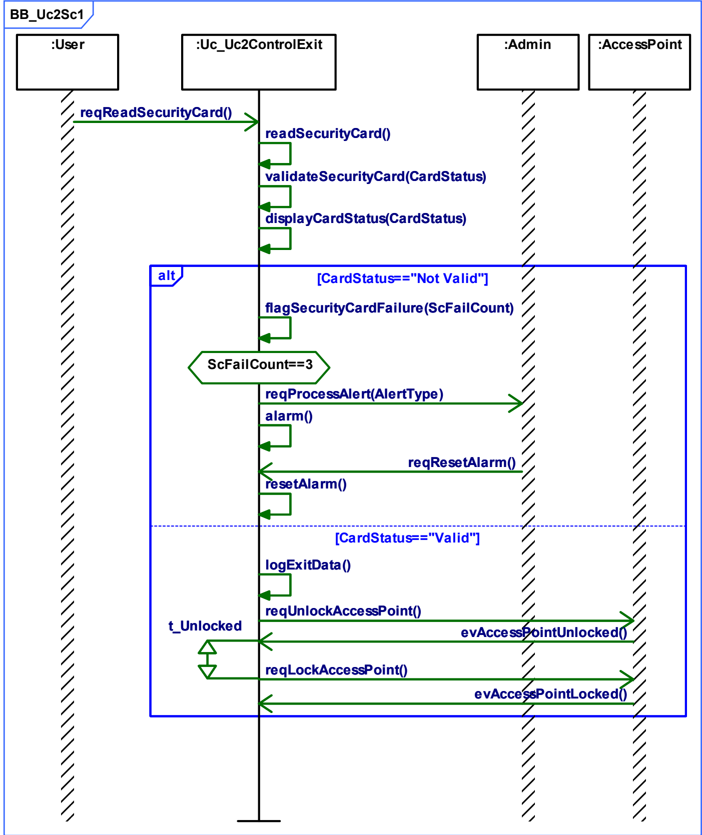


Fig. A2-2 Use Case Scenario Sc1 derived from BB-Activity Diagram

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Step1: 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. A2-2*) 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. A2-2*: *t_Unlocked*)), create in the statechart diagram a wait state with a name that describes the actual system status (*Fig. A2-3: AccessPointUnlocked*).

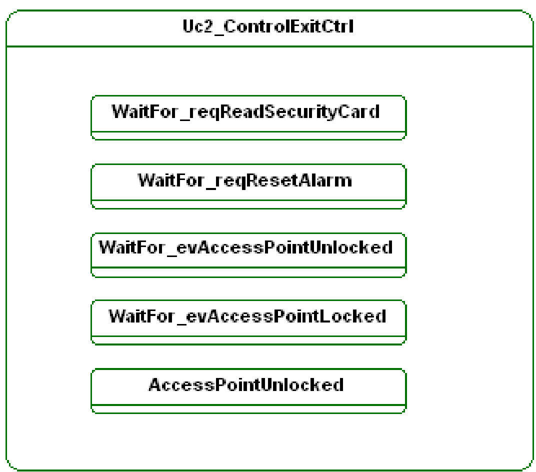


Fig. A2-3 Wait States of Uc2_ControlExit

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.
<u>*NOTE*</u>: Besides the output-relevant action, an action-state may also have additional context-related actions allocated to it (*Fig. A2-4*: action state *ValidatingSecurityCard)*.

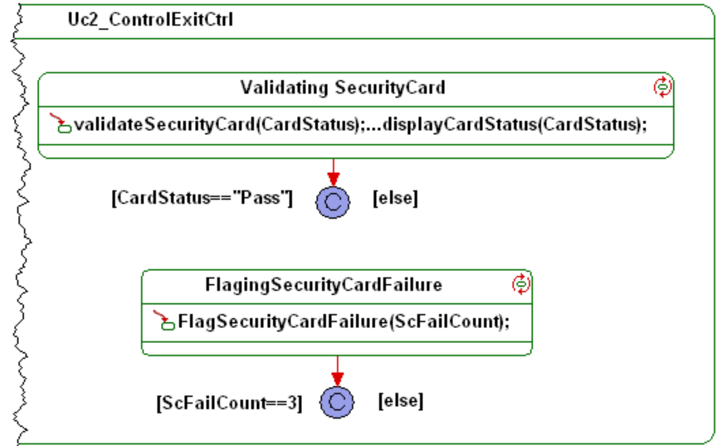


Fig. A2-4 Action States of Uc2 ControlExit

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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. A2-5*), add respective actions to the default connector.

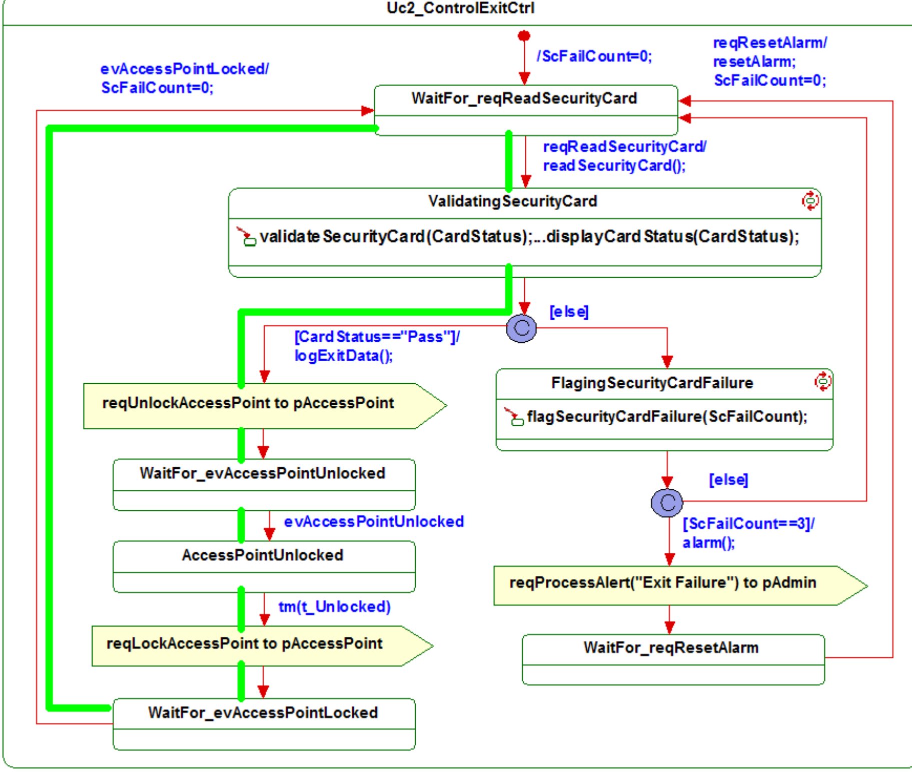


Fig. A2-5 Flat Statechart of Uc2_ControlExit
(Sequence for CardStatus ==“Pass”)

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 use case sequence diagrams.

Select a use case scenario. Replicate the scenario in the statechart diagram:
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 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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Step 3: Structure the Statechart hierarchically

Step 3.1: Identify state hierarchies

Once the flat statechart is verified, look for ways to structure 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 **Error! Reference source not found.** 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. A2-6*). 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.

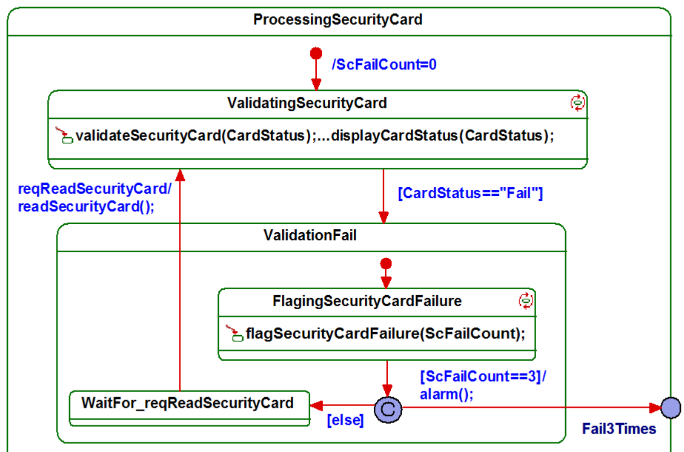


Fig. A2-6 Composite State ProcessingCardData

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. A2-5*) 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*).

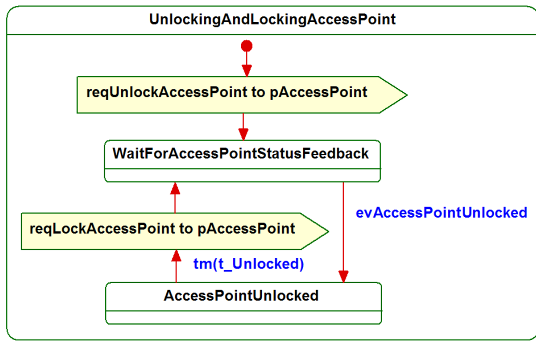


Fig. A2-7 Composite State UnlockingAndLockingAccessPoint

States in the flat statechart *Fig. A2-5*, that relate to the access point control can be aggregated into the composite state *UnlockingAndLockingAccessPoint*, as shown in *Fig. A2-7*. 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.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Fig. A2-8 shows the final structure of the top-level statechart of the use case Uc2ControlExit.

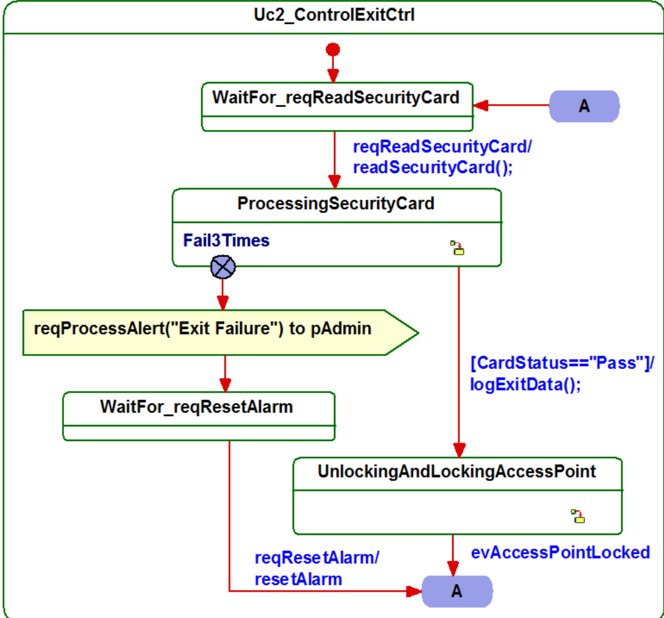


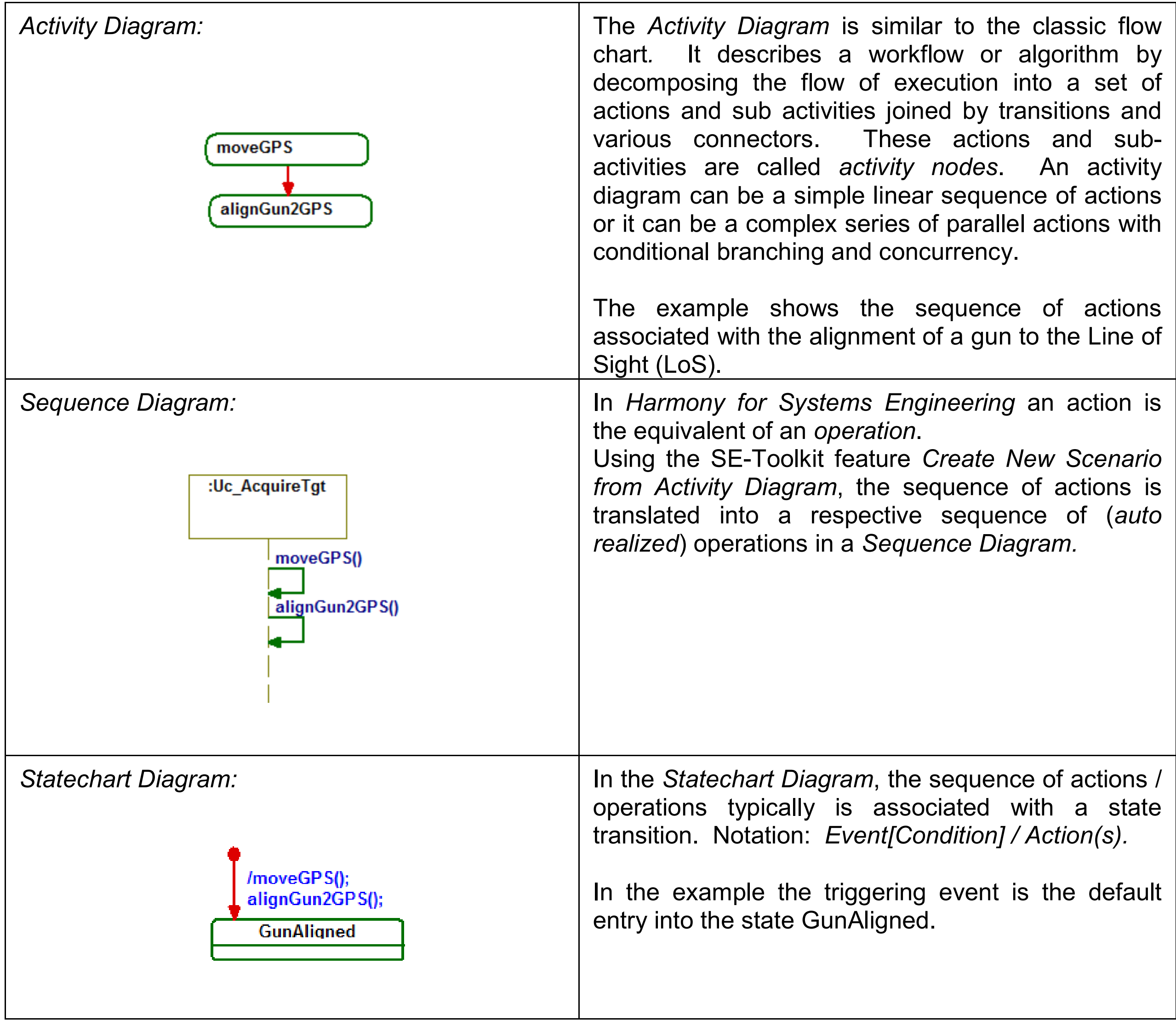
Fig. A2-8 Top-Level Statechart of Uc2ControlExit

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.

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

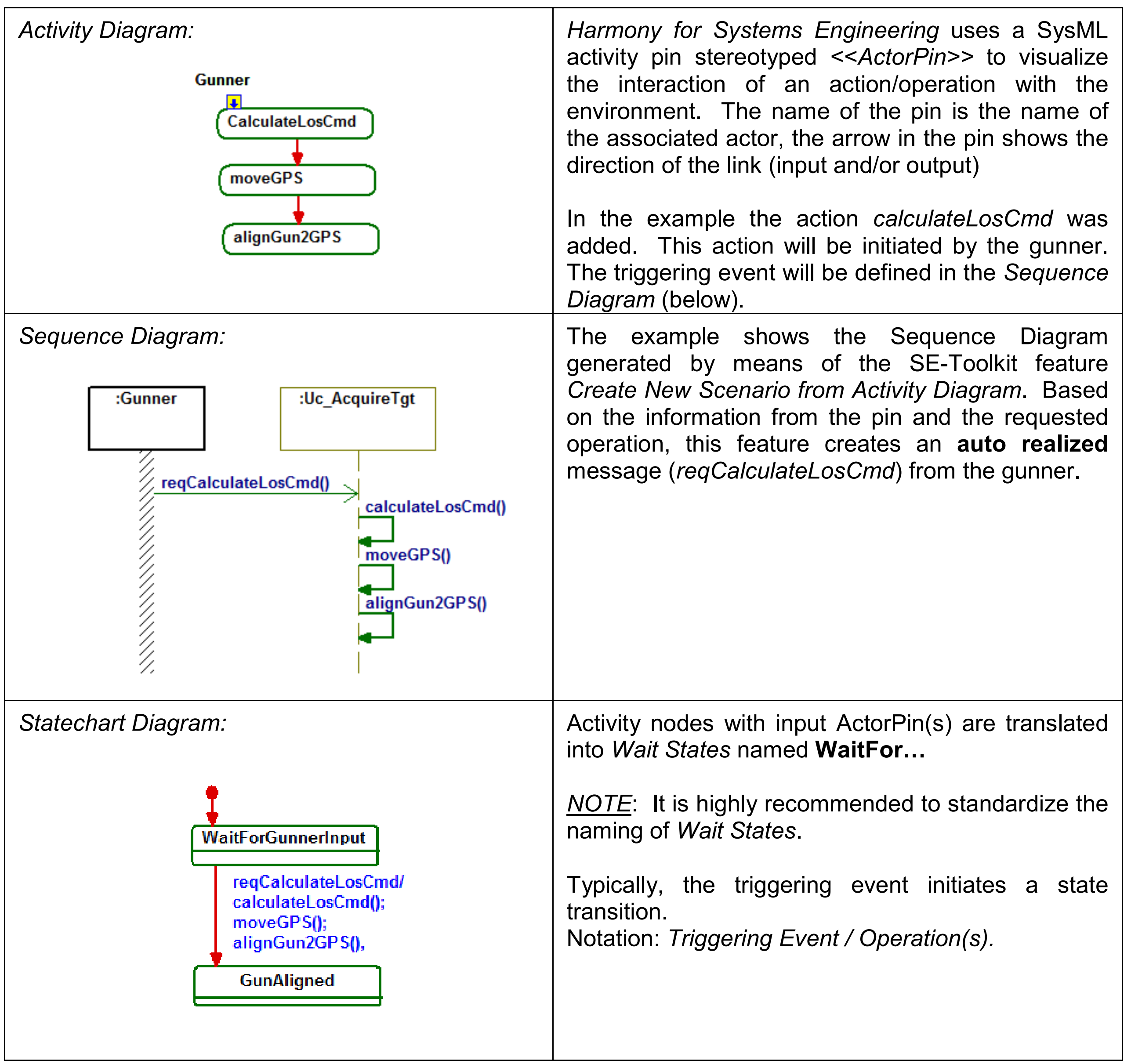
A3 *Usage of Activity Diagram Information in the SE Workflow*



Activity Diagram:	The *Activity Diagram* is similar to the classic flow chart. It describes a workflow or algorithm by decomposing the flow of execution into a set of actions and sub activities joined by transitions and various connectors. These actions and sub-activities are called *activity nodes*. 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. The example shows the sequence of actions associated with the alignment of a gun to the Line of Sight (LoS).
Sequence Diagram:	In *Harmony for Systems Engineering* an action is the equivalent of an *operation*. Using the SE-Toolkit feature *Create New Scenario from Activity Diagram*, the sequence of actions is translated into a respective sequence of (*auto realized*) operations in a *Sequence Diagram.*
Statechart Diagram:	In the *Statechart Diagram*, the sequence of actions / operations typically is associated with a state transition. Notation: *Event[Condition] / Action(s).* In the example the triggering event is the default entry into the state GunAligned.

Reuse of the UML/SyML Activity Node Information

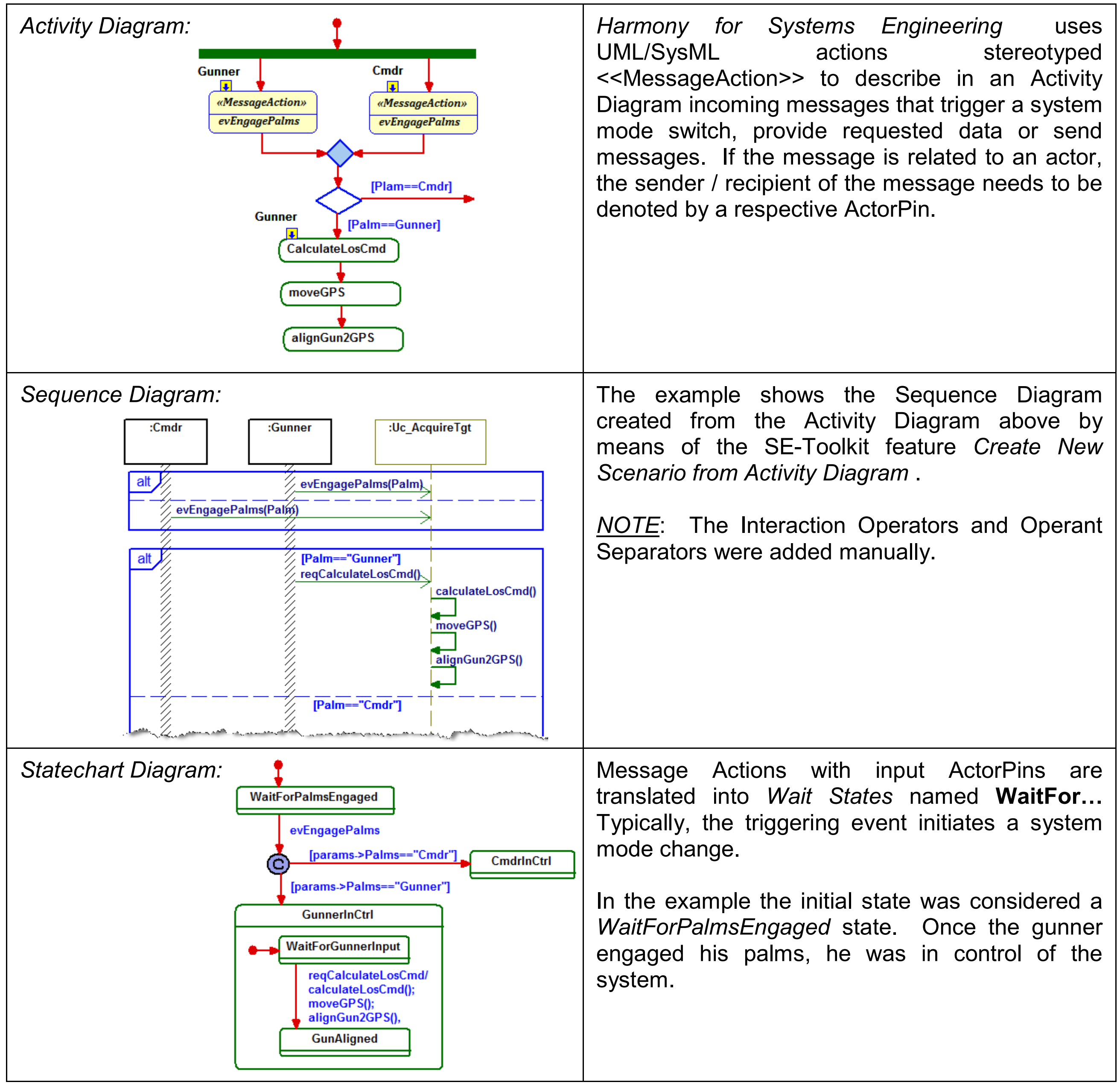
© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



Activity Diagram:	*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 (input and/or output) In the example the action *calculateLosCmd* was added. This action will be initiated by the gunner. The triggering event will be defined in the *Sequence Diagram* (below).
Sequence Diagram:	The example shows the Sequence Diagram generated by means of the SE-Toolkit feature *Create New Scenario from Activity Diagram*. Based on the information from the pin and the requested operation, this feature creates an **auto realized** message (*reqCalculateLosCmd*) from the gunner.
Statechart Diagram:	Activity nodes with input ActorPin(s) are translated into *Wait States* named **WaitFor…** _NOTE_: It is highly recommended to standardize the naming of *Wait States*. Typically, the triggering event initiates a state transition. Notation: *Triggering Event / Operation(s).*

Reuse of Activity Node with ActorPins Information (cont'd)

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



Harmony for Systems Engineering uses UML/SysML actions stereotyped <<MessageAction>> to describe in an Activity Diagram incoming messages that trigger a system mode switch, provide requested data or send messages. If the message is related to an actor, the sender / recipient of the message needs to be denoted by a respective ActorPin.

The example shows the Sequence Diagram created from the Activity Diagram above by means of the SE-Toolkit feature *Create New Scenario from Activity Diagram* .

NOTE: The Interaction Operators and Operant Separators were added manually.

Message Actions with input ActorPins are translated into *Wait States* named **WaitFor…** Typically, the triggering event initiates a system mode change.

In the example the initial state was considered a *WaitForPalmsEngaged* state. Once the gunner engaged his palms, he was in control of the system.

Reuse of Message Action Information (cont'd)

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A6 *Rhapsody Action Language*

This section provides a brief introduction to the action language applied in the *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: 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

X=1	(Sets X equal to 1)
X=Y	(Sets X equal to Y)
X=X+5	(Adds 5 to X)
X=X-3	(Subtracts 3 from X)
X=X*4	(Multiplies X by 4)
X=X/2	(Divides X by 2)
X=X%5	(Sets X to the remainder of X divided by 5)
X++	(Increments X by 1)
X--	(Decrements X by 1)

Printing

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

cout << “The value of X is “ << X << endl;

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

The value of X is 5

Comparison Operators

<table>
<tr><td>X==5</td><td>(X equal to 5)</td></tr>
<tr><td>X!=5</td><td>(X not equal to 5)</td></tr>
<tr><td>X<3</td><td>(X less than 3)</td></tr>
<tr><td>X<=3</td><td>(X less than or equal to 3)</td></tr>
<tr><td>X>4</td><td>(X greater than 4)</td></tr>
<tr><td>X>=4</td><td>(X greater than or equal to 4)</td></tr>
<tr><td>X>2 && X<7</td><td>(X greater than 2 **and** X less than 7</td></tr>
<tr><td>X<2 || X==7</td><td>(X less than 2 **or** X equal to 7)</td></tr>
</table>

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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:

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

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

```
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:

```
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:

```
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:

```
go();
```

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

```
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:

```
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:

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

NOTE: The "**OPORT**" keyword can be used in place of "**OUT_PORT**".

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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:

```
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:

```
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”:

```
if (IS_IN(Accelerating))
```


© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A5 *Change Request-driven System Design Approach*

The chance for systems engineers to be involved in the design of a completely new system is rare. Mostly, systems engineers have to deal with modifications or extensions of an existing (*legacy*) system. Typically, the changes are based on requirements specified in form of textual Change Requests (CR). Although accompanied by descriptive documents such as Concepts of Operations (CONOPS), further analysis is needed to assess the impact of change requests on the existing system architecture.

This section describes by means of a generic example, a model-based change request-driven system design approach aimed at the early validation of customer requirements by means of executable models.

Essentially, the outlined workflow follows the MbSE workflow documented in the previous paragraphs. The only *essential* difference is the hand-off to the subsequent subsystem development teams, i.e. to HW, SW and Test. As in this case the executable *Change Request (CR) System Architecture Model* only defines the allocation of change request related functional/non-functional requirements to the legacy system architecture Configuration Items (CI), the resulting impact on respective CIs has to be elaborated by the Integrate Product Team (IPT) as a follow-up activity.

The benefits of the Change Request-driven System Design approach are:

- Improved understanding of customer requirements up-front in the system design and
- Support of system impact analysis in order to allow early submission of Change Proposals.

Requirements Analysis

The requirements of the generic change request are grouped in two use cases (CR_Uc1, CR_Uc2). *Fig.A5-1* depicts the resulting use case diagram.

NOTE: The association between the legacy system CIs and the use case have to be **unidirectional.**

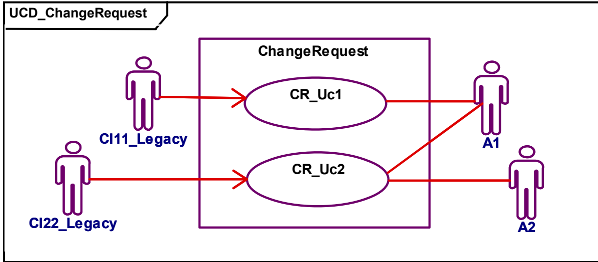


Fig.A5-1 Use Case Diagram of the Change Request Case Study

Functional Analysis

The functional flow of each use case is captured in a black-box activity diagram (*Fig.A5-2*).

NOTE: The "use case story" must be self-contained. It may include functionality that is implemented in the legacy system. In the later design phase (ref. "Hand-off to the Integrated Product Team") respective redundancies will be filtered out.

Following the functional analysis workflow, use case black-box scenarios are derived from the respective use case black-box activity diagram. These scenarios are the basis from which use case block ports and interfaces were defined. Eventually, based on the information from the activity diagram and the sequence diagrams, the state-based behavior of the use case block is captured in a statechart diagram. Each use case model is then verified through model execution.

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

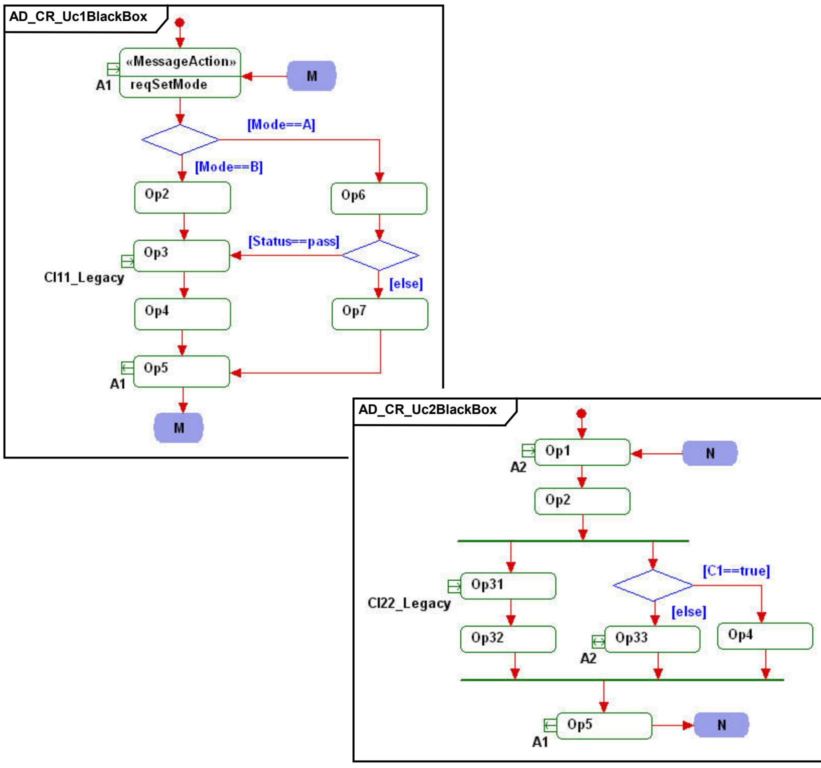


Fig.A5-2 Black-Box Activity Diagrams of Use Cases CR_Uc1 and CR_Uc2

Architectural Analysis

The objective of the Architectural Analysis phase – also referred to as the Trade Study phase – is to elaborate an architectural concept that best satisfies the CR related set of functional and performance requirements. In collaboration with the Integrated Product Team (IPT), different architectural concepts are evaluated based upon a set of criteria that are weighted according to their relative importance. It is beyond the scope of this section to go into details of the trade study.

The lowest level of architectural decomposition to be captured in the CR system architecture model is the node level – also referred to as configuration item (CI) level (*Fig.A5-3*).

<u>*NOTE*</u>: The CR system structure captures only the “delta” architecture, i.e. those CIs that are involved in the design process either as actors in the use cases (CI11, CI22) or as a location for the change request (CI211). CI3 was added to the legacy system architecture as an additional component.

At the lowest level a CI consists of a legacy (black-box) part. Optionally, this CI will contain the allocated change request related functional/non-functional requirements.

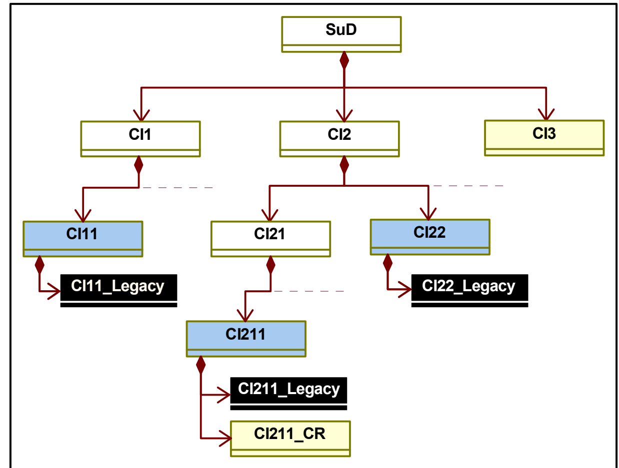


Fig.A5-3 Case Study: CR System Architecture

Architectural Design

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Fig.A5-4 shows the workflow and the associated artifacts in the CR related Architectural Design phase. Two types of models are created in this phase:

- the *Realized CR Use Case Model(s)* and
- the *CR System Architecture Model*.

The Realized CR Use Case Model is the white-box view of the use case model that was created in the previous Functional Analysis phase. The CR System Architecture Model is the aggregate of all Realized CR Use Case Models.

Fig.A5-5 and *Fig.A5-6* show the created SysML artifacts in the case study.

<u>NOTE</u>: For readability reasons, the names of the delegation ports are not shown in the IBDs.

Once the correctness and completeness of the realized CR use case models are verified through model execution, they are merged in the common CR System Architecture Model. *Fig.A5-7* shows the BDD and IBD of this model. The collaboration of the different realized CR use case models is verified through model execution on the basis of the previously captured UC white-box sequence diagrams.

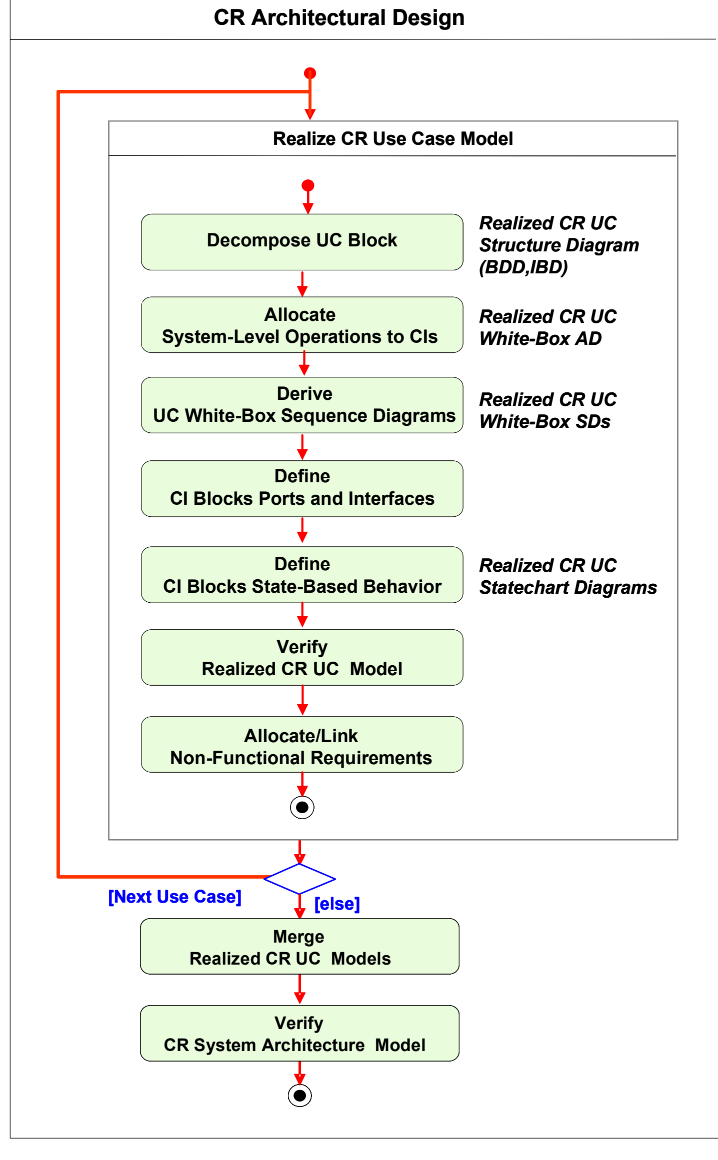
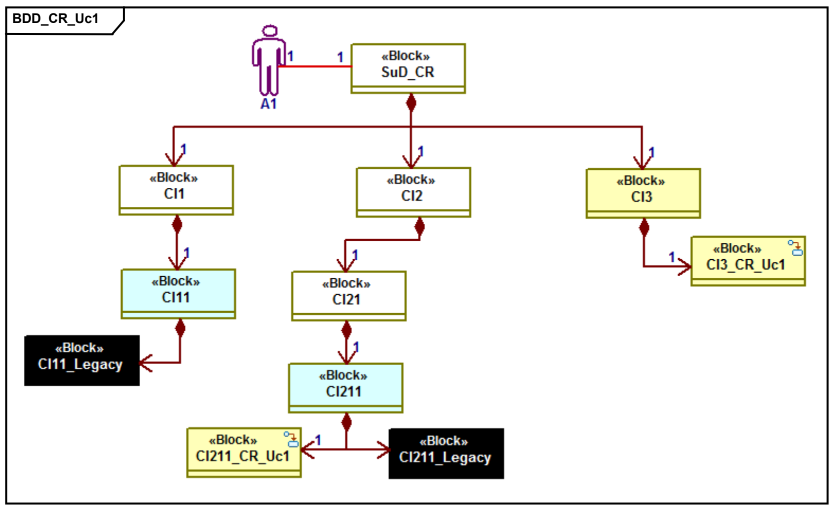
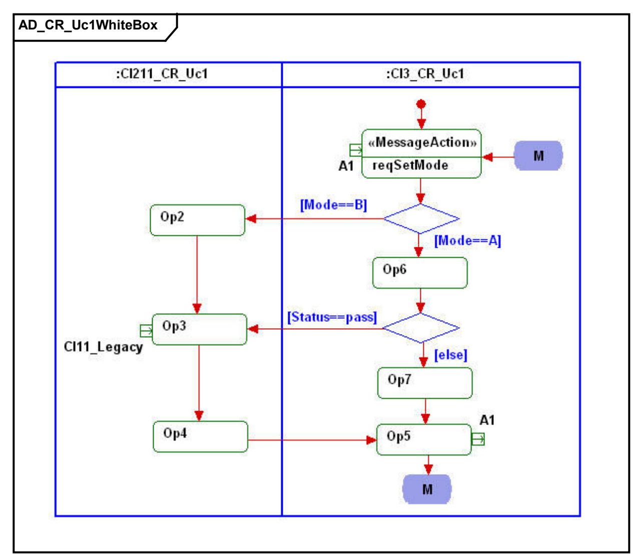


Fig.A5-4 Workflow in the Architectural Design Phase

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.





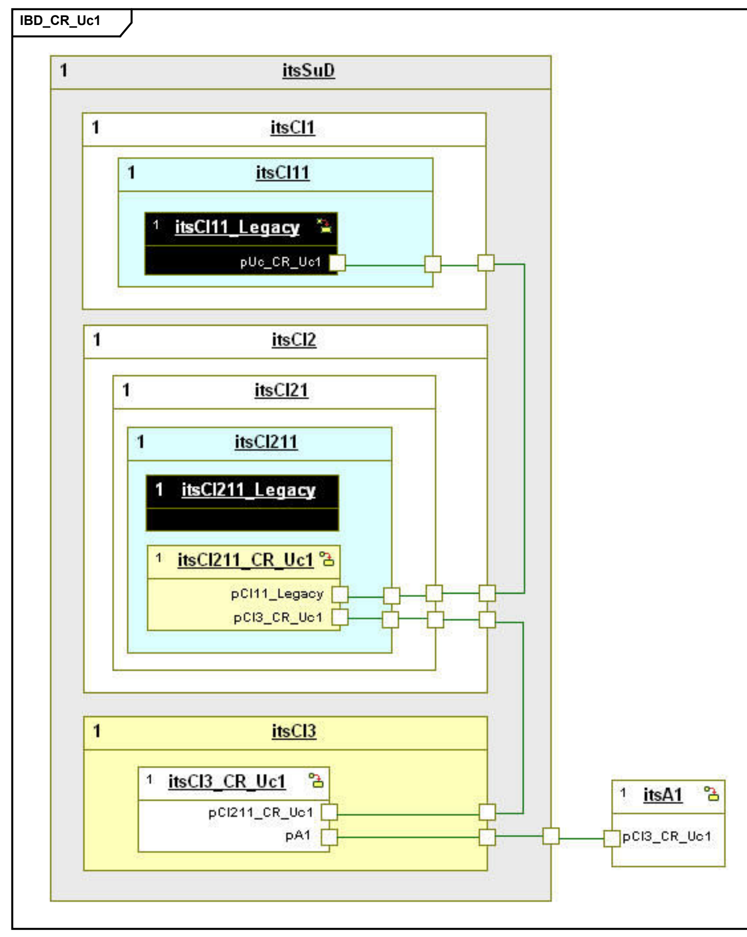
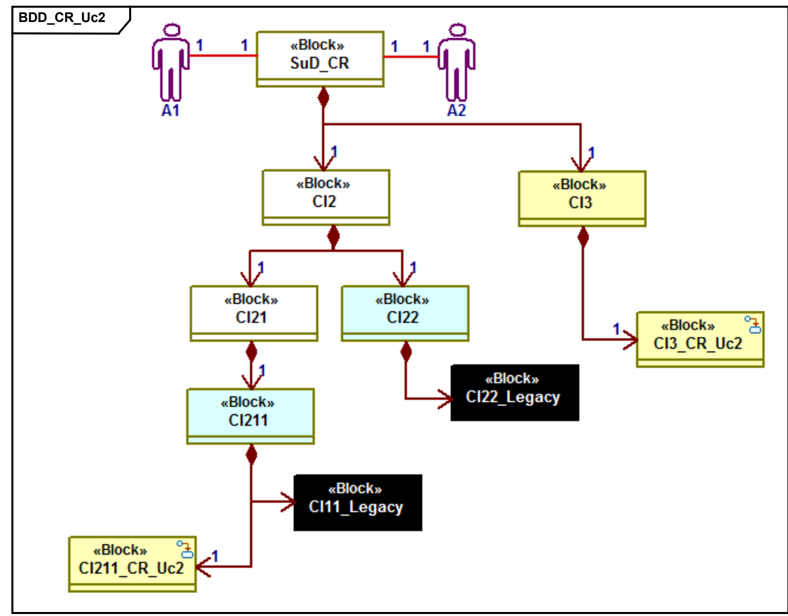
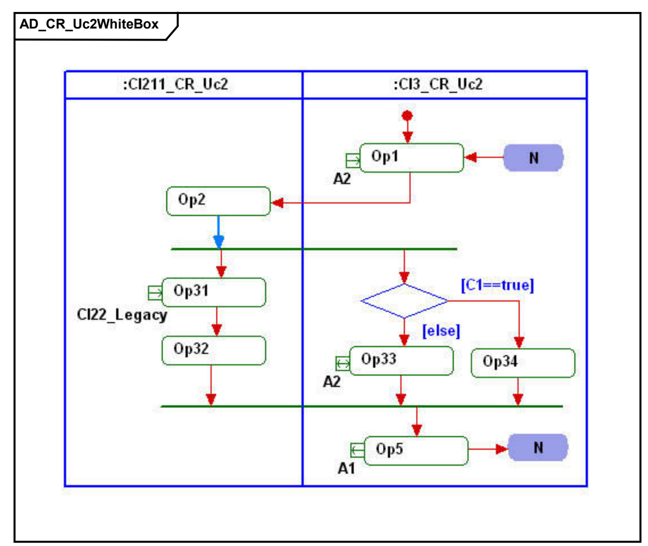


Fig.A5-5 Realized Use Case Model CR_Uc1

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.





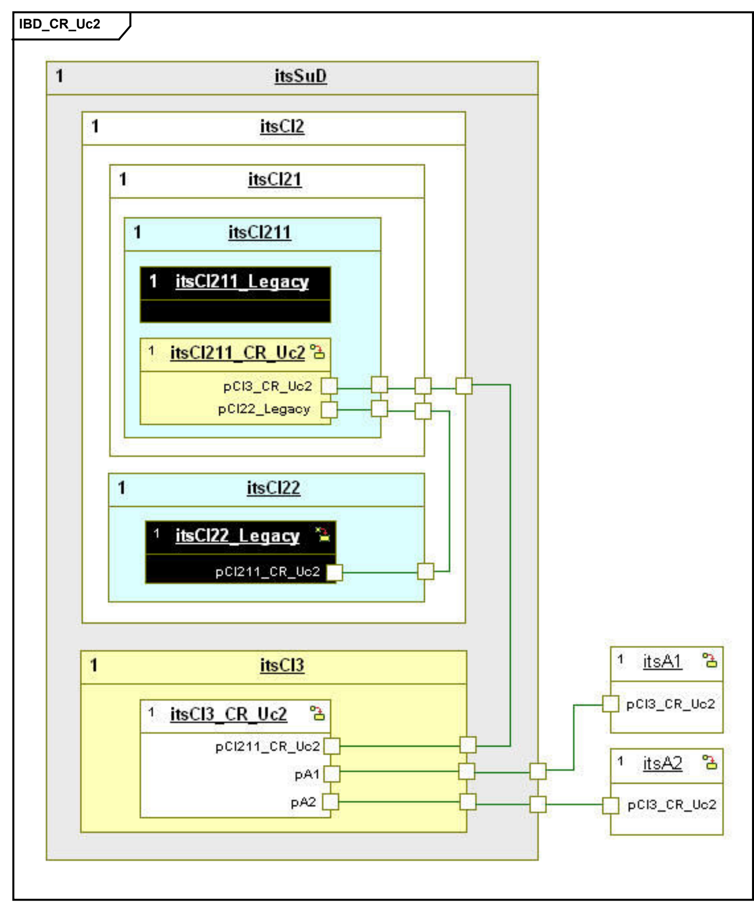
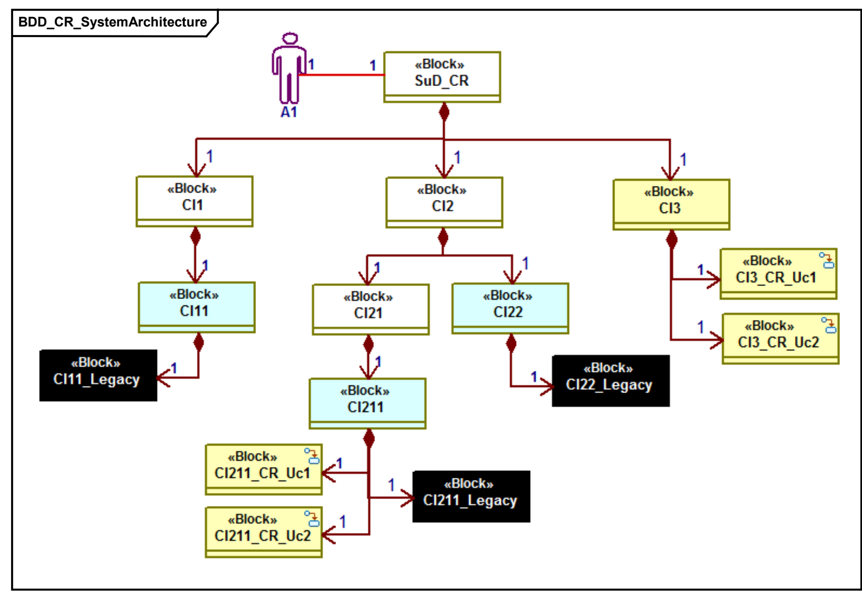


Fig.A5-6 Realized Use Case Model CR_Uc2

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



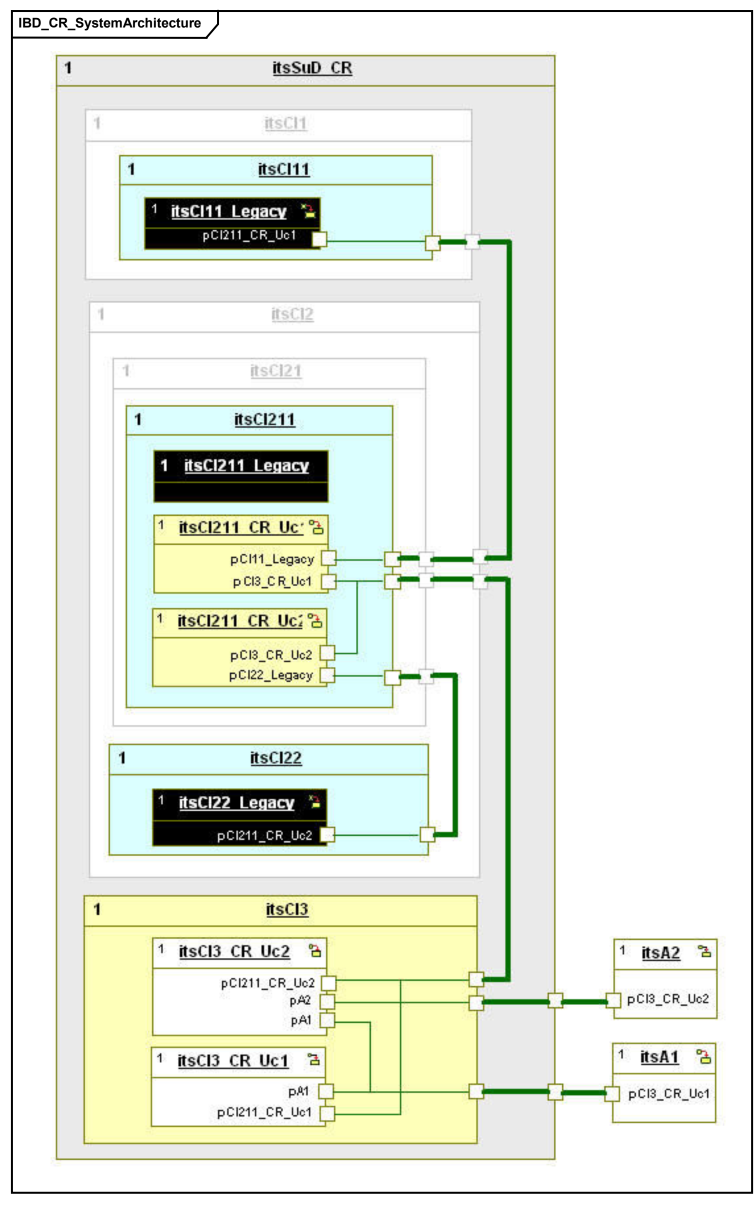


Fig.A5-7 CR System Architecture Model

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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. *Fig.A5-8* depicts the N^2 chart of the CR system architecture elaborated in the case study.

<u>NOTE</u>: In the N^2 chart the SuD_CR colum/row describes the logical system-level interfaces.

CI11	iCI11_Legacy_Uc_CR_Uc1					
	CI211		iCI211_CR_Uc1_CI3_CR_Uc1 iCI211_CR_Uc2_CI3_CR_Uc2			
	iCI22_Legacy_CI211_CR_Uc2	CI22				
	iCI3_CR_Uc1_CI211_CR_Uc1 iCI3_CR_Uc2_CI211_CR_Uc2		CI3	iCI3_CR_Uc1_A1		
			iA2_CI3_CR_Uc2 iA1_CI3_CR_Uc1	SuD_CR		iCI3_CR_Uc1_A1
				iA2_CI3_CR_Uc2	A2	
				iA1_CI3_CR_Uc1		A1

Fig.A5-8 N^2 Chart of the Case Study CR System Architecture

Hand-off to the Integrated Product Team

In this case study, the level of the architectural decomposition and associated requirements allocation is the CI-level. This constraint defines the hand-off to the subsequent hardware/software development. As outlined in the previous paragraphs, each CI at the lowest level of the CR System Architecture Model consists of a black-box legacy part and (optionally) of a change request related part which contains the allocated functional and non-functional requirements. "Harmonizing" the two parts and partitioning them into HWCIs and/or CSCIs are considered tasks to be performed by the *Integrate Product Team* (IPT).

This chapter details the hand-off artifacts to the IPT. Essentially, the hand-off addresses three types of changes to a legacy system architecture. These three types are elaborated here:

Add additional Ports/Interfaces to the Legacy CI

Fig.A5-9 depicts an example from the case study described in the previous paragraph. In this case, the legacy CIs just provided required information. No changes with regard to the CI functionality are involved.

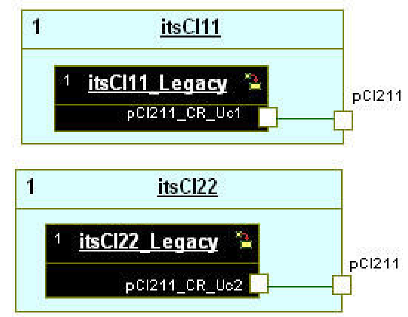


Fig.A5-9 Adding additional Ports/Interface(s) to a Legacy CI

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

Add new Functionality and Port/Interfaces to the Legacy CI

Fig.A5-10 shows an example from the case study described in the previous paragraph. In this case, a subset of requirements that were verified/validated through respective use case models is allocated to the CI in use case-related blocks.

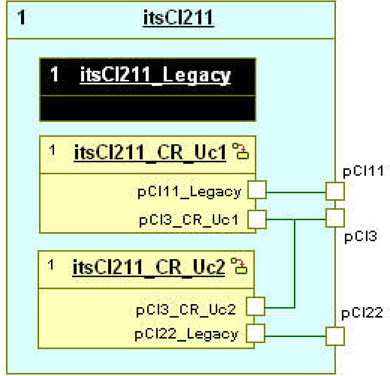


Fig.A5-10 Adding New Functionality and associated Ports/Interfaces to a Legacy CI

These CR blocks then are linked to CIs of the CR system architecture via respective ports and interfaces. As mentioned in the Functional Analysis paragraph, some of the identified operations in these blocks may address functionality already implemented in the black-box part of the CI. It will be the task of the IPT to filter-out respective redundancies.

Add a new CI to the Legacy System Architecture

Fig.A5-11 depicts an example from the case study described in the previous chapter. In this case, a subset of requirements that were verified/validated through respective use case models, is allocated to the new CI in use case-related blocks. These blocks then are linked to CIs of the CR system architecture via respective ports and interfaces

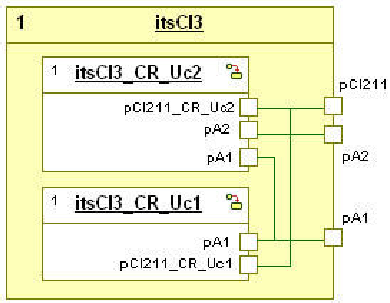


Fig.A5-11 Adding a new CI to the Legacy System Architecture

In any of these cases, the individual CI hand-off will be composed of:

- The baselined *executable* CI model
- The definition of CI-allocated operations, including links to the associated system functional and performance requirements. The allocated operations may be grouped in separate system use case related blocks.
- The definition of CI ports and *logical* interfaces. If a CI is sub-structured into use case related CR blocks, respective internal ports and associated interfaces are included.
- The definition of CI behavior, captured in a statechart diagram. If a CI is sub-structured into use case related CR blocks, the state-based behavior is split accordingly.
- Test scenarios – captured in sequence diagrams - derived from system-level (i.e. white-box) use case scenarios, and
- CI-allocated non-functional requirements

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A6 *Using Model-Based Testing for the Verification of Hand-Off Models*

Overview of the Rhapsody TestConductor and Rhapsody Automatic Test Generation

The Rational *Rhapsody TestConductor* Add-On (TC) solution is a SysML/UML-compliant model-based testing environment for real-time embedded systems and software. By analyzing a model, TC can help build the test context automatically, and test cases can be described as sequence diagrams. TC automatically converts them into executable test procedures where the “inputs” to the system under test are driven from the test case scenario and so are the resulting messages that need to be observed. Hence, the verification steps to create test architectures, to specify executable test cases, and to execute the test cases is largely automated. Furthermore, since model-based testing enables to continuously test against requirements, this solution can aid in reducing specification time and costs while helping to improve system quality.

The Rational *Rhapsody Automatic Test Generation* Add-On (ATG) solution offers a superior capability: by analyzing a model, ATG automatically generates test scenarios that drive the model through many paths with a goal of helping to maximize the coverage of the model. The automatically generated test scenarios are in the form of sequence diagrams similar to the ones a human tester would specify with the Rhapsody sequence diagram editor. Hence, ATG generated the test cases that can be executed using TC.

In a model-driven system development environment, the key artifact of the hand-off from systems engineering to subsystem development is executable models. The Harmony/SE Deskbook recommends an interactive verification using model execution, including model animation, and a visual comparison of the “as-is” behavior regarding the expected behavior. This approach pays-off only if the costly incremental and iterative verification of the hand-off model can be highly automated. Integration test scenarios shall be part of each composed subsystem hand-off package. ATG can be applied to automatically generate such integration test scenarios. Then, TC can be used to verify a developed subsystem against the requirements and to verify, that changes in the executable model do not lead to regressions in the model.

The following sections provide an overview about how TC and ATG can be applied for the verification of hand-off models using, as an example, the SecSysController subsystem hand-off model elaborated in Section 5 of the Deskbook. A more detailed step-by-step description can be found in the video “TestConductor Tutorial for the Verification of Harmony/SE Hand-off Models” [5].

ATG and TC Harmony/SE Workflow

Fig.A6-1 outlines the main workflow. The goal is to generate and execute tests in order to achieve a highly automated verification of the hand-off model.

Several activities have to be performed within this verification workflow:

- a hand-off model is analyzed and test scenarios are generated with ATG
- the generated test scenarios have to be manually reviewed to verify correctness regarding the initial requirements
- test scenarios are automatically converted into test cases ready for execution
- test cases can be executed with TC, and
- additional test cases can be added to the test suite to enhance it

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

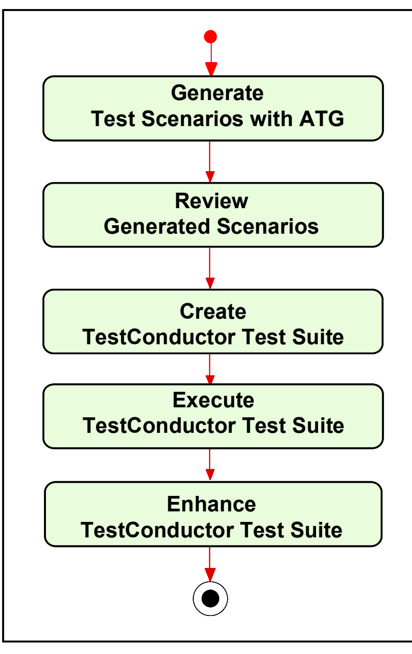


Fig.A6-1 ATG and TC Workflow

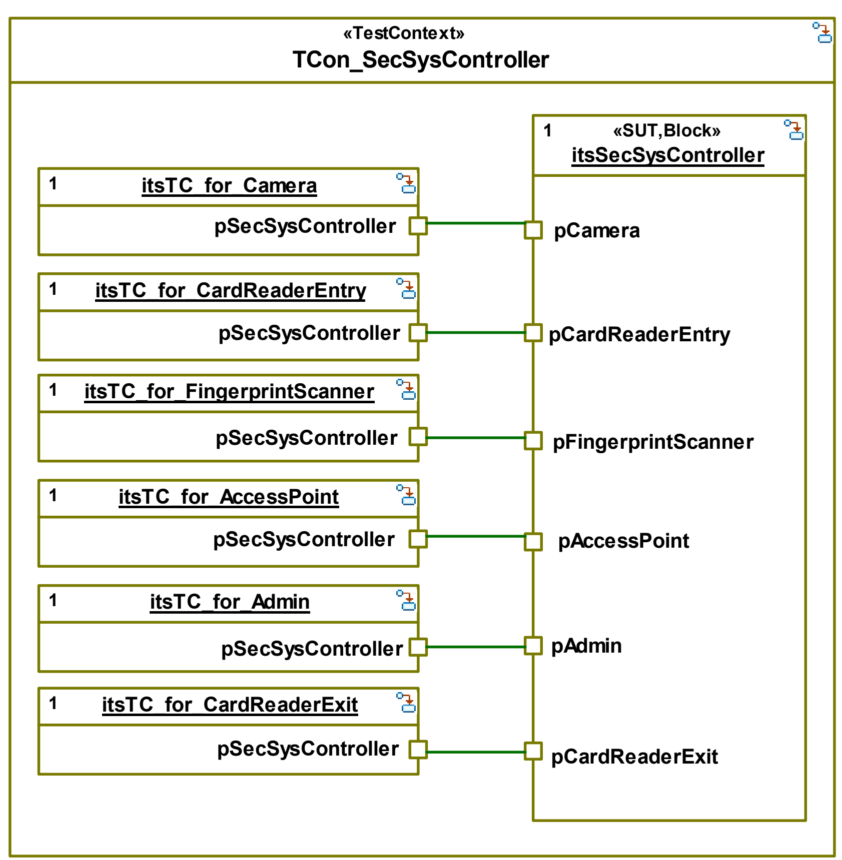


Fig.A6-2 TestArchitecture for SecSysController

Generate Test Scenarios with ATG

ATG is applied on the SecSysController hand-off model. Since it is an executable model of the given requirements, it is the perfect model to derive test cases for further testing activities. With ATG, the objective is to generate a set of scenarios including inputs to the model and expected outputs from the model that sufficiently cover the whole SecSysController behavior. The SecSysController block is selected to be the System Under Test (SUT). TC automatically creates a test architecture including the needed test actors connected to the SUT, as shown in *Fig.A6-2*.

The created test architecture contains an instance of the block SecSysController, which is stereotyped as SUT. It also contains six auto-generated test actors which are connected to the six ports of the SecSysController, respectively. Leveraging from this test architecture, ATG automatically generates the test scenarios. ATG triggers the test actors to automatically send input messages to the SUT via the ports and records these messages. Likewise, the observable reactions of the SUT (i.e. messages from the SUT to the test actors via the ports are recorded.

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

As described in the Deskbook, the SecSysController block is an executable model of the requirements for the SecSysController subsystem of the Security System case study. The model elements have an explicit connection to the requirements through a <<satisfy>> dependency. Verification of the subsystem model against the subsystem requirements can be performed with integration test scenarios. The set of integration test scenarios shall be sufficiently rich enough to execute all parts of the executable model in order to ensure a proper verification of the model. While ATG generates test scenarios, it also measures the achieved model element coverage (i.e. state coverage, transition coverage, and operation coverage). ATG could, for example, generate a test scenario that traverses the states and transitions of the main SecSysController statechart, as illustrated with green color in *Fig.A6-3*.

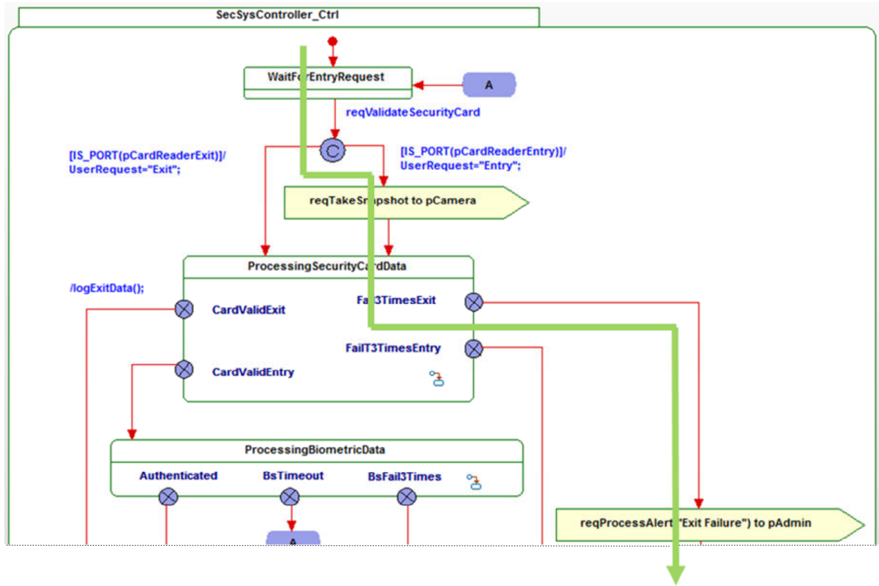


Fig.A6-3 Sample Test Scenario

Obviously, more than one test scenario is necessary to achieve sufficient coverage of the model. ATG terminates the automatic test scenario generation when all model elements are traversed. *Fig.A6-4* summarizes the information about the achieved SecSysController model coverage status after ATG finishes the scenario generation.

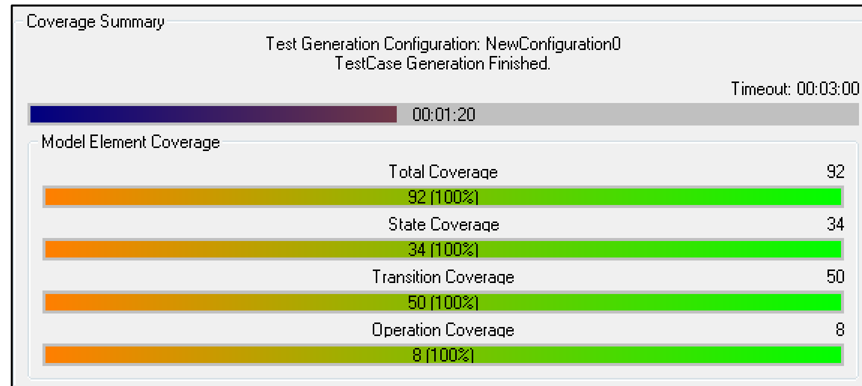


Fig.A6-4 ATG Model Coverage Overview

The SecSysController model contains 34 states, 50 transitions and 8 operations, in total 92 model elements. As shown in Fig.A6-4, the 92 model elements are covered with test scenarios. ATG can cover several states and transitions with one test scenario, as shown in the statechart *Fig.A6-3*. Hence, ATG computes a minimal set of necessary test scenarios, and adds them to the hand-off model. In this case, just 14 test scenarios were needed to achieve 100% model coverage.

Review Generated Scenarios

It is important to review the generated test scenarios in order to verify their correctness against the initial requirements. Hence, the review of the ATG generated scenarios can be considered to be another cross-check that verifies that the hand-off model indeed meets its requirements. If a scenario is approved, it can be moved into a new folder to collect the approved scenarios. An example of a reviewed and approved sample scenario can be seen in *Fig.A6-5.*

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

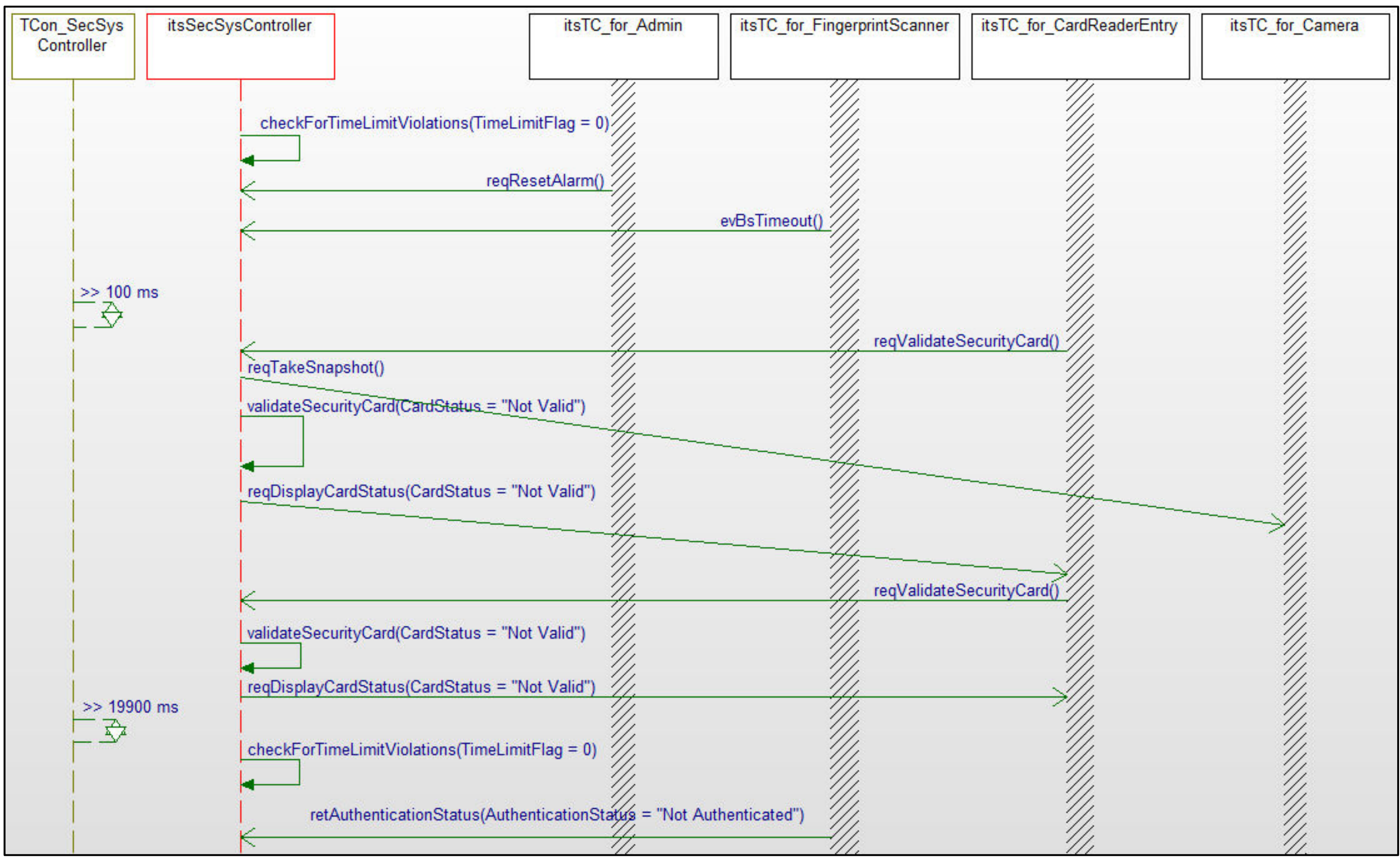


Fig.A6-5 Reviewed and Approved Test Scenario Generated by ATG

The lifelines in this test scenario represent (from left to right) the test context, the SUT, and four of the six test actors of the SecSysController which are involved in this particular test scenario: Admin, FingerprintScanner, CardReaderEntry, and Camera.

Additional information not visible in this scenario view: 1 operation, 7 states, and 4 transitions, are covered with this single test scenario.

Create TestConductor Test Suite

The reviewed and approved scenarios are used to create executable test cases for TC. This activity is fully automated. Such test cases can be used with TC to verify the hand-off model, especially after changes, enhancements, or fixes, have been made. A Test Case is a model element that is visible in the browser underneath the Test Context. Each Test Case references a Test Scenario which specifies the details of a test case.

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

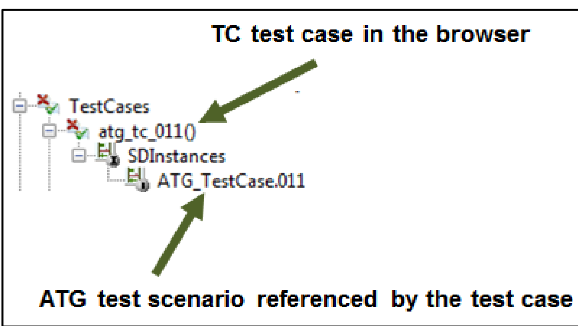
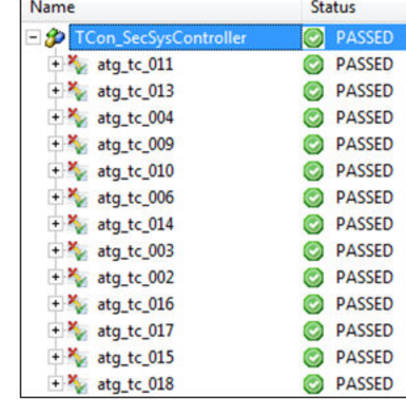


Fig.A6-6 ***TC Test Case in the Browser Referencing a ATG Test Scenario***

For each approved test scenario, TC creates a test case. 14 TC test cases are created for the SecSysController block and added to the test architecture TCon_SecSysController. As part of the hand-off package for SecSysController, they can be handed-off to subsystem development.

Execute and Review TestConductor Test Suite

Single or multiple test cases can be executed automatically in order to verify that the black-box behavior of the hand-off model is as expected, even after changes and enhancements have been made. The generated test execution report contains information with passed or failed verdicts, respectively.



Name	Status
TCon_SecSysController	PASSED
atg_tc_011	PASSED
atg_tc_013	PASSED
atg_tc_004	PASSED
atg_tc_009	PASSED
atg_tc_010	PASSED
atg_tc_006	PASSED
atg_tc_014	PASSED
atg_tc_003	PASSED
atg_tc_002	PASSED
atg_tc_016	PASSED
atg_tc_017	PASSED
atg_tc_015	PASSED
atg_tc_018	PASSED

Fig.A6-7 Test Case Execution with TC Leads to Passed Results

In addition, TC can measure model coverage during test case execution and generate a report. As explained earlier in this section, one single test case may cover many states, transitions, and operations. Accumulated model element coverage is computed when executing several test cases.

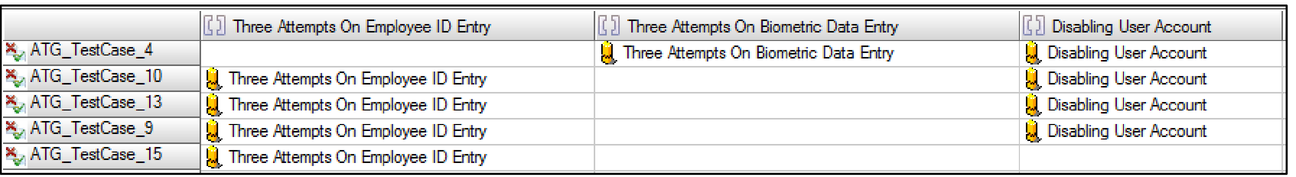
StateChart: Refactored Statechart Alternative 2

covered	ROOT.SecSysController_Ctrl	State
covered	ROOT.SecSysController_Ctrl.state_2	State
covered	ROOT.SecSysController_Ctrl.state_2.WaitForEntryRequest	State
covered	ROOT.SecSysController_Ctrl.state_2.sendaction_1	State
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData	State
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData.ROOT.ProcessingSecurityCardData.ValidatingSecurityCard	State
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData.ROOT.ProcessingSecurityCardData.sendaction_9	State
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData.ROOT.ProcessingSecurityCardData.WaitForRequest	State
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData.ROOT.ProcessingSecurityCardData.SecCardFailure	State
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData.ROOT.ProcessingSecurityCardData.sendaction_1	State
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData.ROOT.ProcessingSecurityCardData.sendaction_2	State
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData.ROOT.ProcessingSecurityCardData.sendaction_3	State
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData.2	Transition
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData.6	Transition
covered	ROOT.SecSysController_Ctrl.state_2.ProcessingSecurityCardData.0	Transition

Fig.A6-8 shows an excerpt of the achieved model element coverage of the SecSysController model. It visualizes in green color the states and transitions that have been executed. Red color would indicate that a model element is not executed by the test suite. In our example, everything is green because all model elements are covered.

Fig.A6-8 Model Coverage Report after Test Case Execution

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.



	Three Attempts On Employee ID Entry	Three Attempts On Biometric Data Entry	Disabling User Account
ATG_TestCase_4		Three Attempts On Biometric Data Entry	Disabling User Account
ATG_TestCase_10	Three Attempts On Employee ID Entry		Disabling User Account
ATG_TestCase_13	Three Attempts On Employee ID Entry		Disabling User Account
ATG_TestCase_9	Three Attempts On Employee ID Entry		Disabling User Account
ATG_TestCase_15	Three Attempts On Employee ID Entry		

Fig.A6-9 Test / Requirements Coverage Overview (excerpt)

As mentioned earlier, Harmony/SE recommends explicitly linking model elements to the requirements through <<satisfy>> dependencies. As ATG and TC know the relation between generated test cases and model elements, the coverage of the requirements associated with the generated tests can be reported.

Fig.A6-9 shows an excerpt of such a report. On the left side the generated test cases are listed. The top line shows the requirements. The entries in the cells of the table indicate wheather a test case contributes to the verification of a requirement. For instance, test case *ATG_TestCase_4* does not contribute to the verification of requirement *Three Attempts On Employee ID Entry*. But it contributes to the verification of requirements *Three Attempts On Biometric Data Entry* and *Disabling User Account.*

Enhance TestConductor Test Suite

When a systems engineer changes, improves, or enhances the system model, additional test cases are needed to perform a thorough test of the model. These test cases may be added using the Rhapsody sequence diagram editor. The test will then be part of the whole test suite and can also be executed with TC, thus also contributing to the complete pass/fail results as well as to the model requirements coverage.

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.

A7 *Rhapsody SE-Toolkit (Overview)*

	SE-Toolkit Feature		Description
1	Modeling Toolbox	1.1 Add Hyperlink(s)	Adds a hyperlink from the source(s) to the destination(s).
		1.2 Add Anchor(s)	Adds an anchor from the source(s) to the destination(s)
		1.3 Add SD Ref(s)	Adds selected sequence diagram(s) as *Referenced Sequences* to the use case.
		1.4 Add Event Reception(s)	Adds receptions of the chosen events to the target interface.
		4.5 Add Value Typr	Maps the seleted value type to the selected unit. Tags of the value type are populated from the unit.
		1.6 Merge Blocks	Copies any operations, receptions, and attributes from the source blocks to a single destination block.
		1.7 Create Dependency	Creates dependencies between model elements.
		1.8 Populate Activity Diagram	For each reflexive message on the selected sequence(s) an action is created on the selected activity diagram
		1.9 Allocate Operations from Swimlanes	Copies operations allocated to a swimlane in a White-Box Activity Diagram into the relevant sub-system block.
		1.10 Create New Scenario from Activity Diagram	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

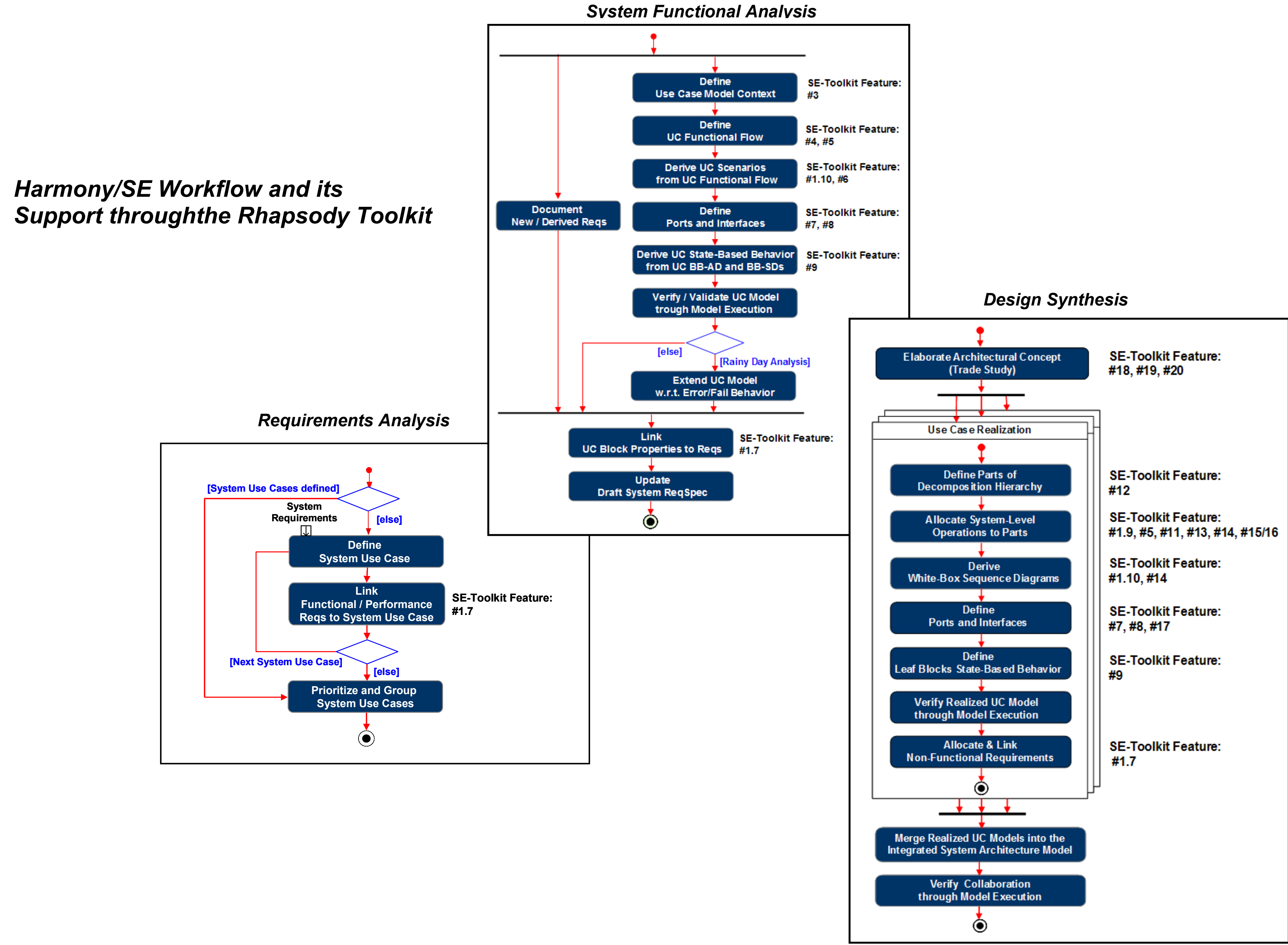
Rhapsody SE-Toolkit Features

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.

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

Rhapsody SE-Toolkit Features cont'd

© Copyright IBM Corporation 2006, 2010. All Rights Reserved.


Harmony/SE Workflow and its
Support throughthe Rhapsody Toolkit
System Functional Analysis
Define
Use Case Model Context
SE-Toolkit Feature:
#3
Define
UC Functional Flow
SE-Toolkit Feature:
#4, #5
Derive UC Scenarios
from UC Functional Flow
SE-Toolkit Feature:
#1.10, #6
Document
New / Derived Reqs
Define
Ports and Interfaces
SE-Toolkit Feature:
#7, #8
Derive UC State-Based Behavior
from UC BB-AD and BB-SDs
SE-Toolkit Feature:
#9
Verify / Validate UC Model
trough Model Execution
[else]
[Rainy Day Analysis]
Extend UC Model
w.r.t. Error/Fail Behavior
Link
UC Block Properties to Reqs
SE-Toolkit Feature:
#1.7
Update
Draft System ReqSpec
Requirements Analysis
[System Use Cases defined]
System
Requirements
[else]
Define
System Use Case
Link
Functional / Performance
Reqs to System Use Case
SE-Toolkit Feature:
#1.7
[Next System Use Case]
[else]
Prioritize and Group
System Use Cases
Design Synthesis
Elaborate Architectural Concept
(Trade Study)
SE-Toolkit Feature:
#18, #19, #20
Use Case Realization
Define Parts of
Decomposition Hierarchy
SE-Toolkit Feature:
#12
Allocate System-Level
Operations to Parts
SE-Toolkit Feature:
#1.9, #5, #11, #13, #14, #15/16
Derive
White-Box Sequence Diagrams
SE-Toolkit Feature:
#1.10, #14
Define
Ports and Interfaces
SE-Toolkit Feature:
#7, #8, #17
Define
Leaf Blocks State-Based Behavior
SE-Toolkit Feature:
#9
Verify Realized UC Model
through Model Execution
Allocate & Link
Non-Functional Requirements
SE-Toolkit Feature:
#1.7
Merge Realized UC Models into the
Integrated System Architecture Model
Verify Collaboration
through Model Execution

7 References

[1] OMG SysML Specification 1.3, June 2012,
http://www.sysml.org/specs

[2] Bruce Powel Douglass, “The Harmony Process: "The Development Spiral”.
Telelogic Whitepaper 2006

[3] Bruce Powel Douglass, Mats Goethe,
“IBM Rational Workbench for Systems and Software Engineering”, IBM Redpaper, 2010
http://www.redbooks.ibm.com/redpapers/pdfs/redp4681.pdf

[4] “Engineering Design Methods: Strategies for Product Design”
Nigel Cross, Wiley, 1989

[5] BTC Embedded Systems AG,
Rhapsody TestConductor Tutorial for the Verification of Harmony/SE Hand-off Models”, Online video, 2014
http://pic.dhe.ibm.com/infocenter/rhaphlp/v8/index.jsp?topic=%2Fcom.btc.tcatg.user.doc%2Ftopics%2Fcom.btc.tcatg.user.doc.html

 © Copyright IBM Corporation 2006, 2010. All Rights Reserved.