Rational Statemate From Code to Concept

Automotive Approach to Using Rational Statemate:

White Paper by Hans-Peter Hoffmann, Ph.D.

Chief Methodologist for Systems Design

Contents

Model-based System Development

This white paper guides Automotive Industry systems engineers and software engineers to use IBM® Rational® Statemate® for the following:

- Systems design automation tool
- Software design
- Model-based software development process

Besides the methodological aspect of the integrated process, “best practice” advice is given which enables a seamless transition “from concept to code.”

It should be noted that the purpose of this document is to supplement the existing training courses offered. It is assumed that the reader has a basic knowledge of the Rational Statemate and an understanding of the basic elements and syntax of the Rational Statemate “language.” The following are the major topics covered in the ratepayer:

- The **Model-based System Development** section describes the model-based system development process as it is typically applied in the Automotive Industry. It also defines where in this process the tools Rational Statemate and MicroC should be used. It also addresses the car manufacturer/supplier trade-off from the process and tools point of view.
- The **Requirements Capture and System Design with Rational Statemate** section describes details of the functional analysis and modeling paradigm that should be adopted for requirements capture and systems design either by means of Rational Statemate or - to a lesser extend - by means of MicroC. Driven by the need to harmonize and standardize body electronic designs cross-platform, many car manufacturers follow a “feature-based approach” in their system designs. The role of Rational Statemate in this process is outlined in detail, especially with respect to a seamless transition to the S/W development using MicroC.
- The **Model Verification and Validation** section gives an overview on the model verification and validation process supported by Rational Statemate and - to a lesser extend - by MicroC. Test strategies and test vector gene-ration are emphasized for the later use during the software design stage with MicroC.
- **Style Guidelines and Best Practices** summarizes the common set of supported language elements. In addition, formal guidelines and “best practice” advice for the usage of the language of Rational Statemate are given. These “best practices” are based on many years of modeling experience in aerospace/defense and automotive projects and have proved to

significantly enhance the readability of specifications developed using Rational Statemate.

System Development in the Automotive Industry

It has long been recognized that ambiguous and inconsistent requirements are the primary cause of design errors in system designs. This is not helped by the fact that most specifications are still produced as paper documents and then subject to an inadequate review process. A rigorous review might catch many of these errors, but all too often the review is cursory due to lack of time or experience. Written requirement specifications are often completed after the design as a documentation exercise, often containing errors and ambiguities.

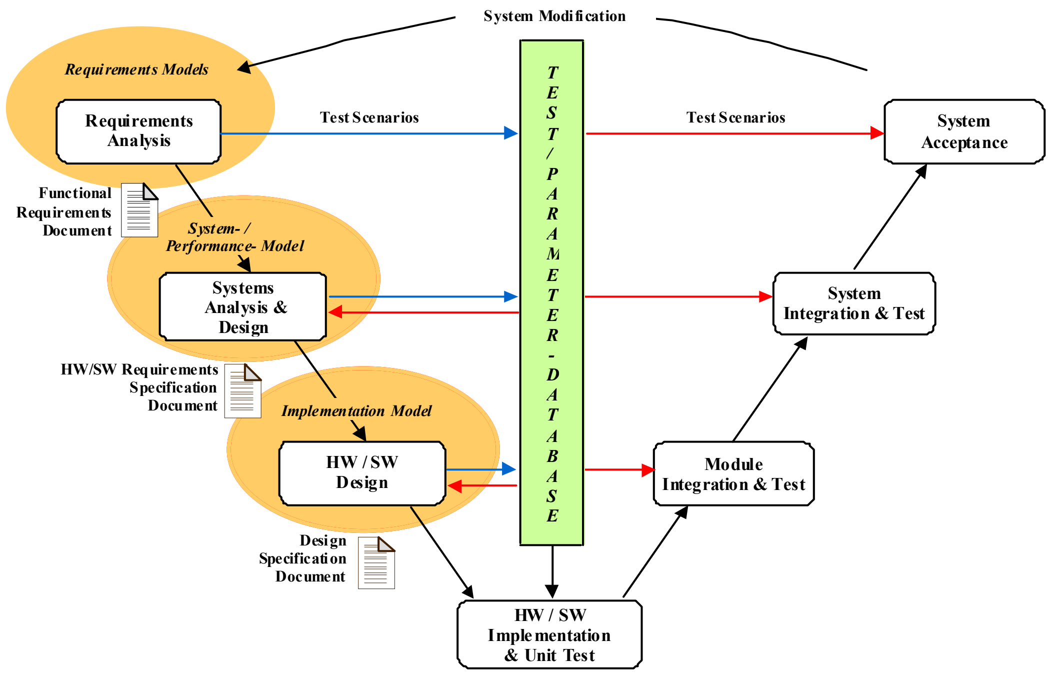
Model-based system development moves away from the written specification approach towards a dynamic representation of the system, which is constantly being reviewed as it is constructed.

Through the model-based approach, design errors are detected much earlier in the development process where the cost to fix them is much less. Customer change requests can be more efficiently assessed and responded, thus providing more timely feedback.

The greatest benefit from a model-based approach is increased communication, not only between the engineering disciplines but also between the technical and non-technical parties involved in the system development process; thus supporting Concurrent Engineering. This is possible due to the inherent ability to produce models at different levels of abstraction thereby avoiding the detailed overload that often occurs when data is passed between different domains.

But modeling should not be seen as a stand-alone task. It should be embedded within the overall system development process.

A widely used and accepted process model is the “V” Development Lifecycle illustrated in the following figure.


System Modification
Requirements Models
Requirements Analysis
Test Scenarios
Test Scenarios
System Acceptance
Functional Requirements Document
System- / Performance- Model
Systems Analysis & Design
TEST / PARAMETER - DATABASE
System Integration & Test
HW/SW Requirements Specification Document
Implementation Model
HW / SW Design
Module Integration & Test
Design Specification Document
HW / SW Implementation & Unit Test

The left leg of the V describes the top-down design process over time while the right leg of the V corresponds to the bottom-up synthesis path over time, where smaller components are integrated to build the complete product.

It is important to note the creation of test data all along the design path (left leg). With this approach it is no longer good enough to say that the system shall meet a particular requirement. Rather, it is necessary to also define how the requirement shall be tested. As can be seen from the above figure, these test data can be re-used either at the next design level or during the bottom-up integration.

In a model-based development process test-data are derived directly from the simulation of respective models. These data are input to or later exported from a Test/Parameter Database, e.g. an ORACLE database.

To understand the model-based design process further, the left-hand side of the “V” is illustrated in more detail in the following figure. It describes a specific approach applied in the Automotive Industry - the so-called “Feature-based Design Process.”

In modern body electronics development, functional requirements are captured and analyzed by means of a special class of models called “feature models.” These models describe either one basic functionality like window control, mirror control, etc., or an ensemble of basic functionalities (“feature interaction models”) like seat positioning / seat heating / seat venting. Feature (interaction) models are purely functional and do not reveal implementation details. They are designed as re-usable objects (“Feature Library Components”), that allow an easy “plug & play” design of new body electronics systems.

The results of the Feature Analysis Phase are:

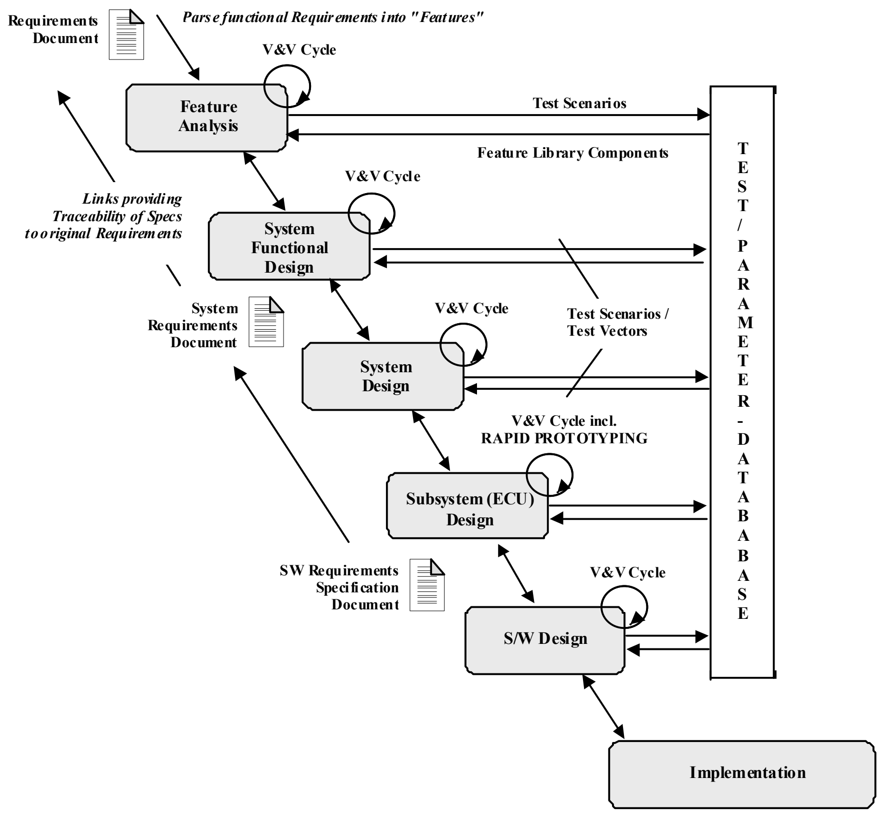
- Validated vehicle-specific feature (interaction) models.
- Test scenarios derived from respective requirements.

Both are re-used in the following System Functional Design Phase.

The integration of all vehicle-specific feature (interaction) models into one common model defines the Conceptual System Model. It is the basis for the overall Verification & Validation (V&V) of systems requirements in the System Functional Design Phase. V&V will be performed through model execution. The respective test scenarios form the basis of the later validation tests of the synthesized system architecture.

Sometimes a “Soft Prototype”, automatically generated from the Conceptual Model, together with a graphical user interface is used as a first proof of concept - or for marketing presentations.

The Automotive “Feature-based System/Software Design


Requirements Document
Parse functional Requirements into "Features"
V&V Cycle
Feature Analysis
Test Scenarios
Feature Library Components
TEST/PARAMETER-DATABASE
V&V Cycle
Links providing Traceability of Specs to original Requirements
System Functional Design
V&V Cycle
System Requirements Document
Test Scenarios / Test Vectors
System Design
V&V Cycle incl. RAPID PROTOTYPING
Subsystem (ECU) Design
SW Requirements Specification Document
V&V Cycle
S/W Design
Implementation

The results of the Functional System Design Phase are:

- The validated System Requirements Documentation derived from the Conceptual Model.
- High-level test scenarios proving that all requirements were fulfilled.

The Conceptual System Model is the entry point to the System (Architectural) Design Phase. At this stage of the development, the system is first partitioned into subsystems (i.e. ECUs) and then the feature (interaction) models or respective sub-functions are distributed among them. Although the resulting model is still pure functional, the partitioning will be determined almost by implementation considerations, e.g.:

- Computational resource utilization.
- Hardware requirements of a specific function.
- Optimization of overall system communication in terms of wiring harness and EMI.
- Best fault tolerance or graceful degradation.
- Ability to diagnose system failures.

In some cases, architectural design criteria might also be derived from additional Performance Models. These models are based on queuing theory and typically used for throughput analysis and identification of potential communication bottlenecks.

The subsystem partitioning and feature/function parsing is an iterative process. The particular architectural system design will be verified by means of test scenarios previously used for the verification of the Conceptual Model. For each ECU the respective I/O will be recorded.

The final step in the System (Architectural) Design Phase will be the definition of the hardware attributes of the logical interface, i.e. which information will be hardwired and which will be provided via bus.

The results of the System (Architectural) Design Phase are:

- The Architectural System Model with validated logical subsystem (i.e. ECU) interfaces.
- A definition of the hardware interface attributes (hardwired signals/bus).
- Logical test vectors for each subsystem (i.e. ECU), derived from the high-level test scenarios of the Conceptual Model verification.

Based on the particular hardware interface definition, additional I/O-functionality has to be added to each ECU model of the Architectural System Model in the Subsystem Design Phase (e.g. in case of hardwired input: switch debouncing, low pass filtering, etc.). As the respective Architectural Subsystem Model is still purely functional, the later ECU Operating System / Scheduler will be excluded, i.e. all functions will “run in parallel.” If the ECU is to be connected via the bus, the respective bus protocol will not be modeled. Rather, only the bus-related logical interface (“Communication Matrix”) will be defined and verified based on test vectors derived in the previous System (Architectural) Design Phase.

At this stage of the development Rapid Prototyping might be used as an additional means of validation. A “Soft Prototype” - automatically generated from a particular Architectural ECU Model - might be rehosted to a Rapid Prototyping Hardware platform for “on-board” and/or “man-in-the-loop” validation of the design concept prior to the implementation.

In the case of Model-based Software-development the results of the Systems Analysis and Design Phase (see the figure) are:

- The validated SW Requirements Specification derived from the Architectural System / Sub-system Design Model.
- Test vectors for each ECU, derived from the high-level test scenarios of the Conceptual Model validation.

Normally the Architectural Subsystem Model is the entry point to the Software Design Phase. Now target specifics, the Operating System/Scheduler, the sequencing and timing of functions, and the bus protocol have to be considered. The respective Implementation Model essentially becomes a design model for the code implementation. It is validated by re-using test patterns generated during the previous development stages.

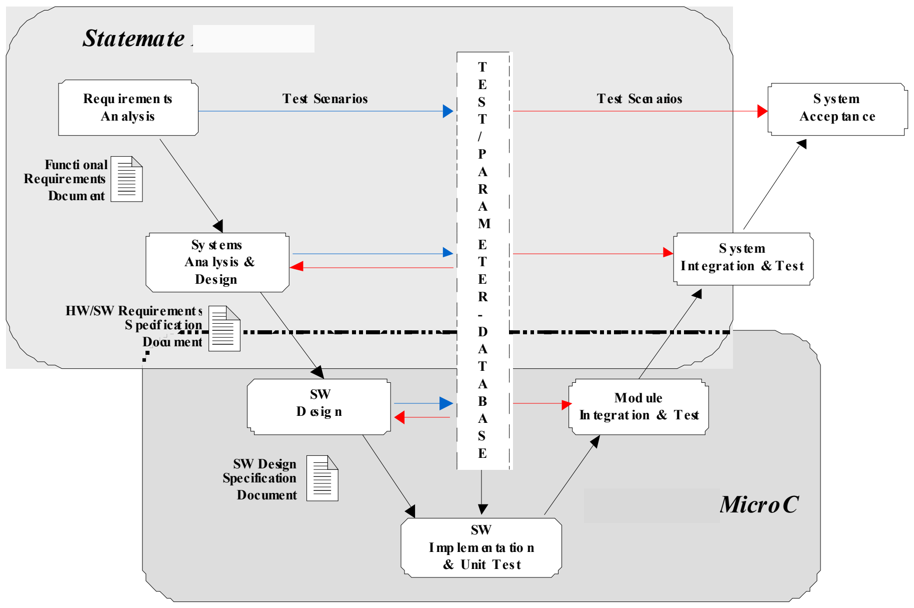
However, the transition to the Software Design Phase is not distinct. It might also start from the Architectural System Model, leaving the I/O processing and the verification and validation through Rapid Prototyping as an implementation task to the Software Design. It should be defined with close co-operation between those responsible for the implementation, such as sub-contractors or members of a software department. Nevertheless, wherever the transition within the process takes place, the deliveries should be model-based with respective test vectors included for cross-validation.

Rational Statemate in a Model-Based Software Development Process

To enable Model-based software development in the Automotive Industry, this product provides a tool that allows a seamless transition from requirements capture to code implementation (“From Concept to Code”): Rational Statemate. The tool follows a model-based function driven paradigm, facilitating a seamless “integrated development process.” The following figure shows the model-based software development process.

Rational Statemate should be used for requirement capture and systems analysis / design, while the Rational Statemate code generator is a software design tool with the capability of generating automatically production C-Code from the design. Requirements Capture and Systems Design with Rational Statemate.

Rational Statemate in a Model-Based SW Development Process

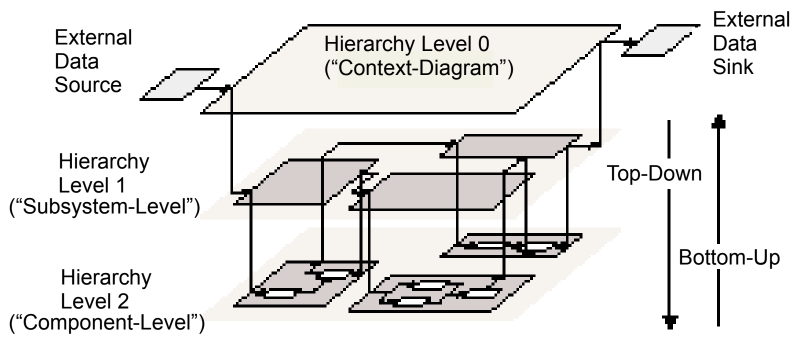

Statemate
Requirements Analysis
Test Scenarios
Test Scenarios
System Acceptance
Functional Requirements Document
Systems Analysis & Design
System Integration & Test
HW/SW Requirements Specification Document
TEST/PARAMETER-DATABASE
SW Design
Module Integration & Test
SW Design Specification Document
MicroC
SW Implementation & Unit Test

Requirements Capture and System Design with Rational Statemate

Functional System Design Methodology Roadmap

Functional system design is based upon the principle of Functional Decomposition. Functional decomposition breaks down complex systems into a hierarchical structure of simpler parts. It ensures that the role of the identified sub-functions is clear and distinct from the other sub-functions.

Functional Decomposition

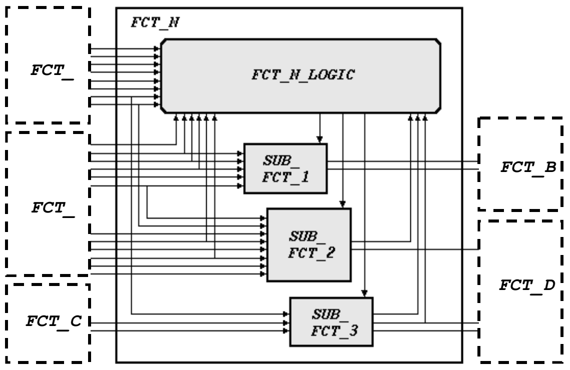

External Data Source
Hierarchy Level 0 ("Context-Diagram")
External Data Sink
Hierarchy Level 1 ("Subsystem-Level")
Top-Down
Bottom-Up
Hierarchy Level 2 ("Component-Level")

Hierarchy Level 0 is the starting point for a Top-Down system approach. It defines the system context, i.e. the system boundary, the external data sources and sinks ("External / Environment Activities"), and the respective information flows.

Hierarchy Level 1 describes the system in terms of high-level functions or subsystems with their logical interfaces. They should be encapsulated and re-usable entities.

Hierarchy Level 2 is the functional decomposition of the respective functional modules or subsystems identified at hierarchy level 1. Starting at this level, state-based behavior might be added through layers of additional decomposition (see the following figure), describing the relationship between the identified functional blocks and respective system states.

Typical Rational Statemate Structure of Hierarchy Level 2 with State-based Behavior Embedded in the Control Activity



Experience shows that the system functionality should be completely defined at hierarchy level 2. This does not mean that the system is completely specified at this level. Rather, the structural outline of the system should be captured in the same way as a table of contents shows the outline of a document.

Typically after decomposition level 2 - latest after decomposition level 3 - the system design process (i.e. the definition of the high-level architecture) might be started.

The outlined top-down system approach is the most frequently used in systems design. But there are also approaches, which start at hierarchy level 2 (“Component-Level”) and then iteratively design the system bottom-up / top-down. An example for this is the feature-based design approach described in the **Feature-based System Design Approach** section.

Architectural System Approach

The classical functional top-down approach, outlined in the previous section, is based on the assumption that the system is being designed from scratch with no legacy influences and with complete design control over the partitioning of functionality.

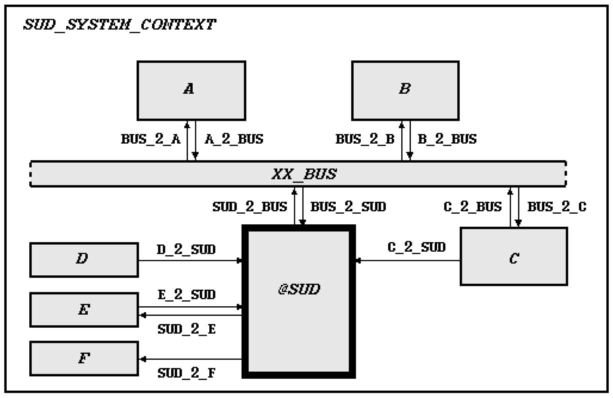
In practice, many applications require the design of additional system or sub-system functions embedded within an existing architecture. This places constraints on the designer, which need to be fully understood before the actual design can take place.

In the following sections, an equipment-related generic architectural system approach will be presented.

Capturing the System Context ("Extended System Context")

Like in the classical functional decomposition approach, the generic architectural approach starts with the definition of the system context. In this case it is called the "Extended System Context" because it captures the operational interfaces between data sources/sinks and the System Under Design (SUD) together with the respective hard-ware interface attributes (hardwired signals/bus, see the following figure). This is needed because I/O data processing will form an essential part of the SUD top-level functions

Architectural System Approach: Hierarchy Level "0"
("Extended System Context")



By starting from the extended system context level, Rapid Prototyping and the later integration of the different subsystems to the overall system model will also be facilitated.

Identifying SUD relevant Data Sources / Sinks

Experience shows that schematics of the overall physical system architecture should be used for the identification of the SUD-relevant data sources/sinks and the respective hardware interfaces.

Data Modeling

The extended context diagram should be restricted to the data flow as far as the SUD is concerned. The communication between the different external data sources and sinks is excluded.

The type of communication - via bus and/or direct lines (Hardwired I/O) - is not only graphically depicted in the diagram, but also reflected in the labels of the particular information flows.

Modeling Hardwired I/O

Hardwired I/O is represented in the extended context diagram as information flows between the SUD and its external data sources and data sinks with labels using the syntax: <source>_2_<sink> (e.g. E_2_SUD, SUD_2_E in the **Architectural System Approach: Hierarchy Level "0" ("Extended System Context")**) figure.

Note

At this stage it is not necessary to define the elements of the respective information flows. Instead, you can define them when the system behavior is captured (typically starting at hierarchy level 2).

Modeling Bus Communication

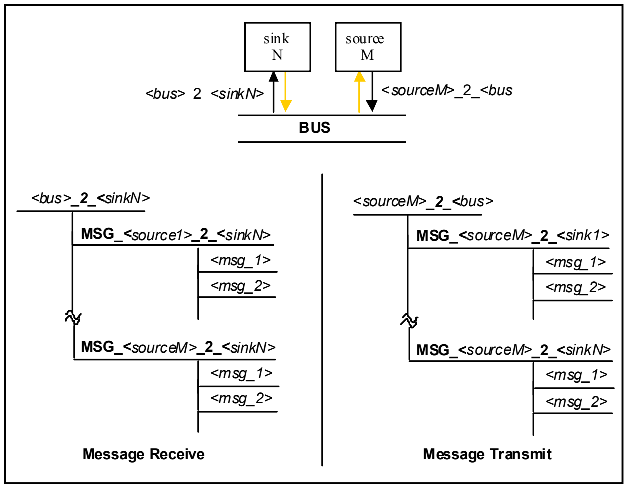
If bus behavior is neglected (i.e. "Functional Model"), the bus is graphically represented by a data store.

If bus behavior is needed (e.g. in order to simulate message loss or delays) the data store should be replaced by an activity with a respective behavior.

In general, the bus communication between data sources and data sinks is captured by two information flows as illustrated in the following figure:

- A flow between the data source and the bus, labeled <source_2_<bus> and
- A flow between the bus and the respective data sink, labeled <bus_2_<sink>.

Modeling Bus Communication



Both flows are decomposed into further information flows ("message containers") which, by their labels explicitly identify the data source and sink, as well as the fact that this flow contains bus messages: i.e. MSG_<source>_2_<sink>.

Message containers have the structure of a record, with each field of the record specifying a particular component of the message. If bus nodes share messages from a specific data source, these messages have to be defined in the respective message containers.

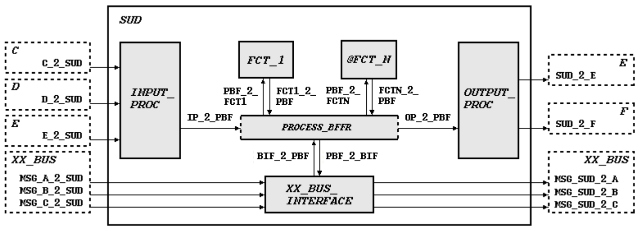
Note

At this stage it is not necessary to define the elements of the "message containers." Instead, you can define them when the system behavior is captured (typically starting at hierarchy level 2).

Generic Top-Level SUD Structure

The proposed generic top-level SUD structure depicted in the following figure is the most frequently used in control unit designs with Rational Statemate.

Generic Top-Level Structure for ECUs



At this level the SUD is partitioned into hardwired/bus I/O-functions and application related main system functions (Firmware, Diagnostics, Control Algorithm, etc.).

Note

No initialization functions are represented at this level. Such functions are considered to be an implementation detail

The functionality captured in the input processing block (INPUT_PROC) could be: signal filtering, switch debouncing, transformation of electrical signals to physical or logical values, etc.

Details of bus protocols are not considered in a functional model. Therefore the receive/transmit functionality within the bus interface is confined to checking for changes on the bus and mapping them to internal variables, or formulating the messages for output onto the bus on changes of respective internal variables; e.g:

- Bus Receive:

  ch(MSG_11.AUX) / KL_R := MSG_11.AUX;
- Bus Transmit:

  ch(STORE_MEM1_ACTIVE) / MSG_78.STORE_MEM1_ACTIVE;

The data-store (PROCESS_BFFR) in the generic structure might be interpreted either as a data repository or as an "internal bus." In the latter case the communication between the functions connected to the "internal bus" should be modeled as outlined in **Modeling Bus Communication**.

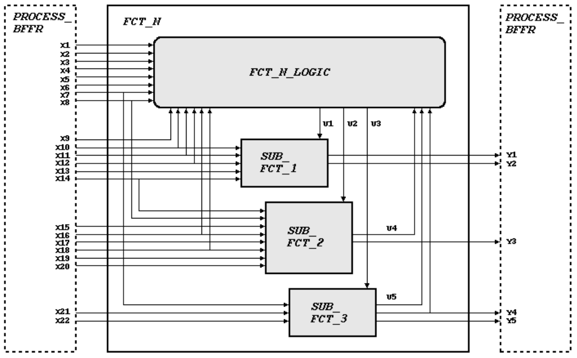
Note

At this stage it is not necessary to define elements of the internal information flows. Instead, you can define them when the system behavior is captured (typically starting at hierarchy level 2).

Application Software Modules

The following figure shows the typical structure of an application software module at decomposition level 2 The function blocks depict the algorithm structure, while the related state-based system behavior is captured in the Control Activity.

Architectural System Approach: Typical Structure of an Application S/W Module at Decomposition Level 2



Note

Starting at this level the information flows defined in the higher hierarchies should be "filled" with the respective details "bottom-up." This ensures that only data are defined, which are actually used within the model.

Feature-based System Design Approach

Driven by the need to harmonize and standardize body electronic design cross-platform, many car manufacturers follow a feature-based approach. In this approach, functional requirements are mapped to a special class of re-usable objects called features. The integrated ensemble of respective feature models describes the overall system functionality and forms the basis for the subsequent bottom-up design process.

In the following paragraphs, the feature-based design process is outlined in detail, especially with respect to a seamless transition to the software design using in MicroC.

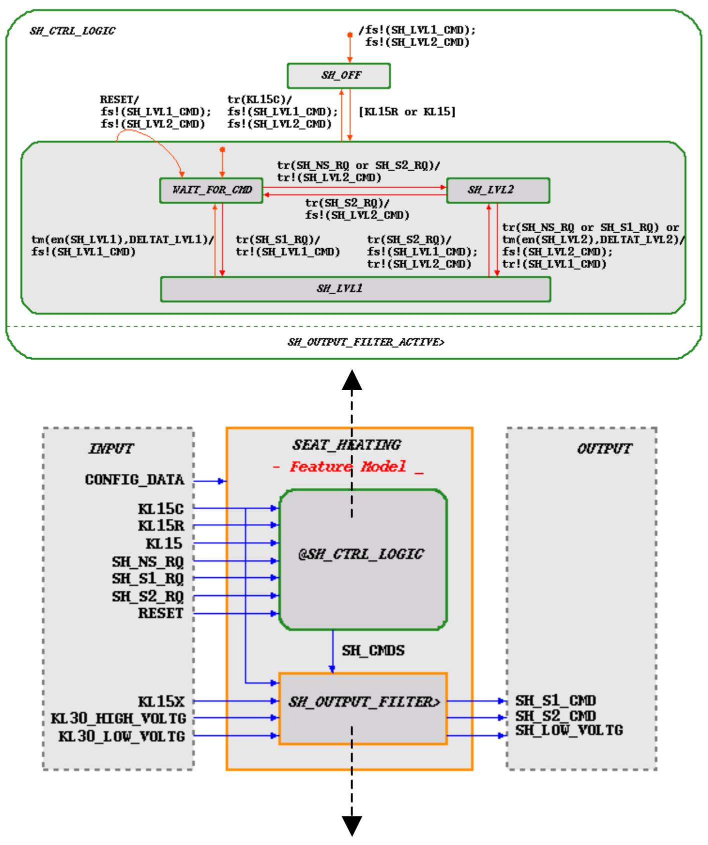
Development of Feature Library Components1

The underlying idea of a feature-based design in body electronic development is that such systems can be described by means of standardized modules with design-specific interactions. Thus, the identification of such basic functions and the development of respective feature models are a prerequisite prior to any feature-based design. Such models typically are archived as re-usable components in a Feature Library. Examples of feature models in Body Electronic Systems are:

- Power Window
- Power Mirror
- Seat Positioning
- Seat Heating, etc.

The **Feature Model: Seat Heating** figure depicts the structure of a Seat Heating feature model. Feature models are functional models, i.e. they do not reveal implementation details. Their top-level structure corresponds to the hierarchy level 2 structure outlined in **Functional System Design Methodology Roadmap** respectively in the **Typical Rational Statemate Structure of Hierarchy Level 2 with State-based Behavior Embedded in the Control Activity** figure. Feature models are generic and have purely logical interfaces.

Feature Model: Seat Heating



```
started/
SH_S1_CMD:=SH_LVL1_CMD;
SH_S2_CMD:=SH_LVL2_CMD;
SH_LOW_VOLTG:=KL30_LOW_VOLTG;;
--
ch(SH_LVL1_CMD) or ch(SH_LVL2_CMD) or
fs(KL15C or KL15X or
KL30_HIGH_VOLTG or KL30_LOW_VOLTG)/
if not KL15C and not KL15X and
   not KL30_HIGH_VOLTG and
   not KL30_LOW_VOLTG then
      SH_S1_CMD:=SH_LVL1_CMD;
      SH_S2_CMD:=SH_LVL2_CMD;
      SH_LOW_VOLTG:=KL30_LOW_VOLTG
end if;;
--
tr(KL15C or KL15X or
KL30_HIGH_VOLTG or KL30_LOW_VOLTG)/
fs!(SH_S1_CMD);
fs!(SH_S2_CMD);
SH_LOW_VOLTG:=KL30_LOW_VOLTG;;
```

An ensemble of basic feature models with specific interactions can be grouped within a macro, called a feature interaction model. In such a model, the feature interactions are decoupled from the feature blocks and described in an additional functional block called an arbitrator. Arbitrators might be interpreted as "intelligent switches." The **Feature Interaction Model: Seat Controller** figure shows a Seat Controller module as an example of a feature interaction model. It combines the following basic features:

- Seat Positioning
- Seat Heating
- Seat Venting

The Seat Feature Arbitrator describes the specific behavior of the Seat Heating in this configuration:

- When the Seat Positioning is active, the Arbitrator sets the actual Seat Heating commands to false.
- Normally, when in SH_LVL1, the Seat Heating is switched OFF after a specified time

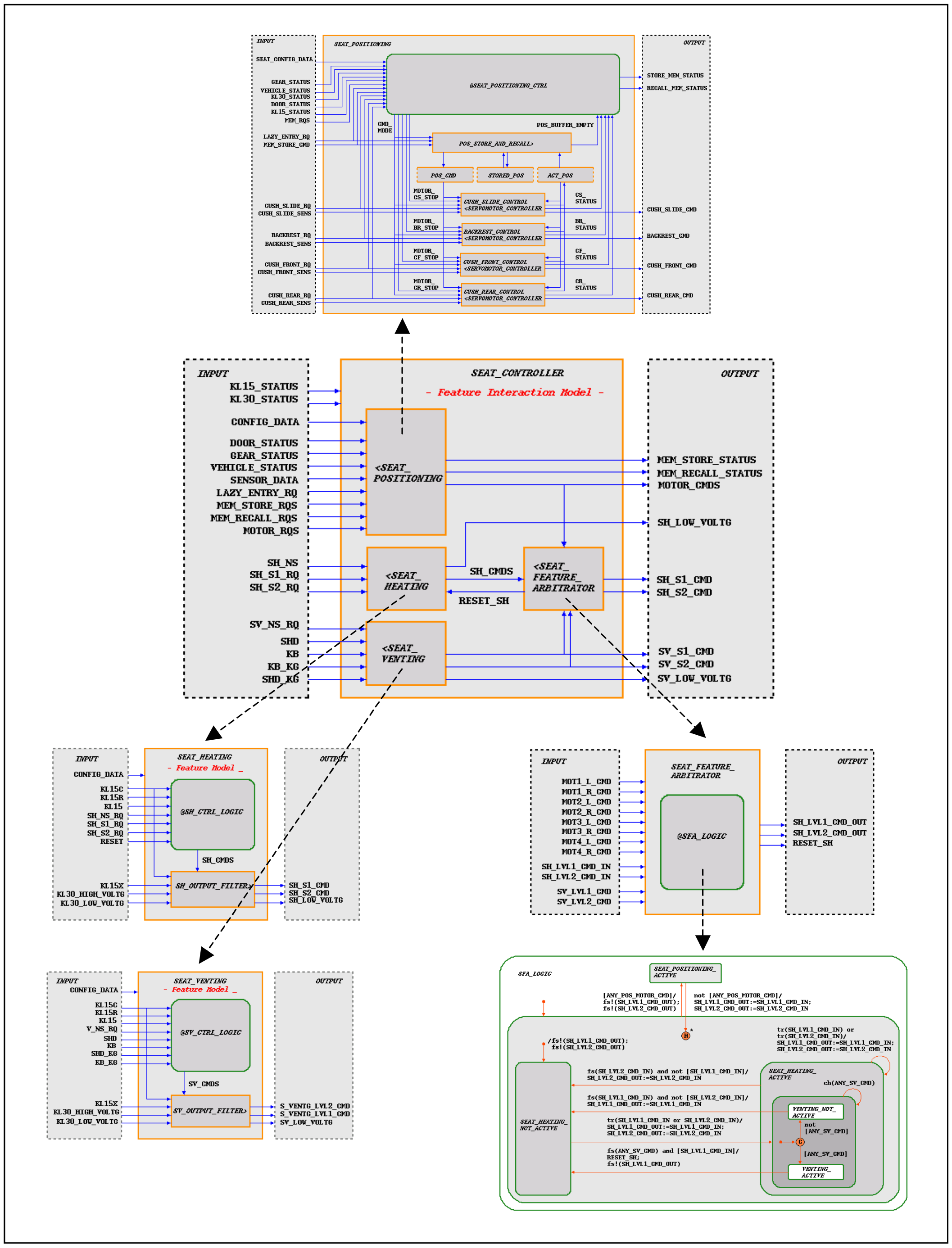
Refer to the **Feature Model: Seat Heating** figure. If the Seat Venting was ON and is switched OFF during this time, the Seat Heating will also be switched OFF. The Arbitrator then shall generate a respective RESET event and send it to the Seat Heating module.

Feature/feature interaction models are verified and validated through simulation. Test scenarios are derived from respective requirements.

Note

Not only the Feature Interaction Model itself but also its top-level functions (feature models and the arbitrator) shall be generic, thus enabling an easy partitioning to different ECUs (refer to **System Partitioning**).

Feature Interaction Model: Seat Controller

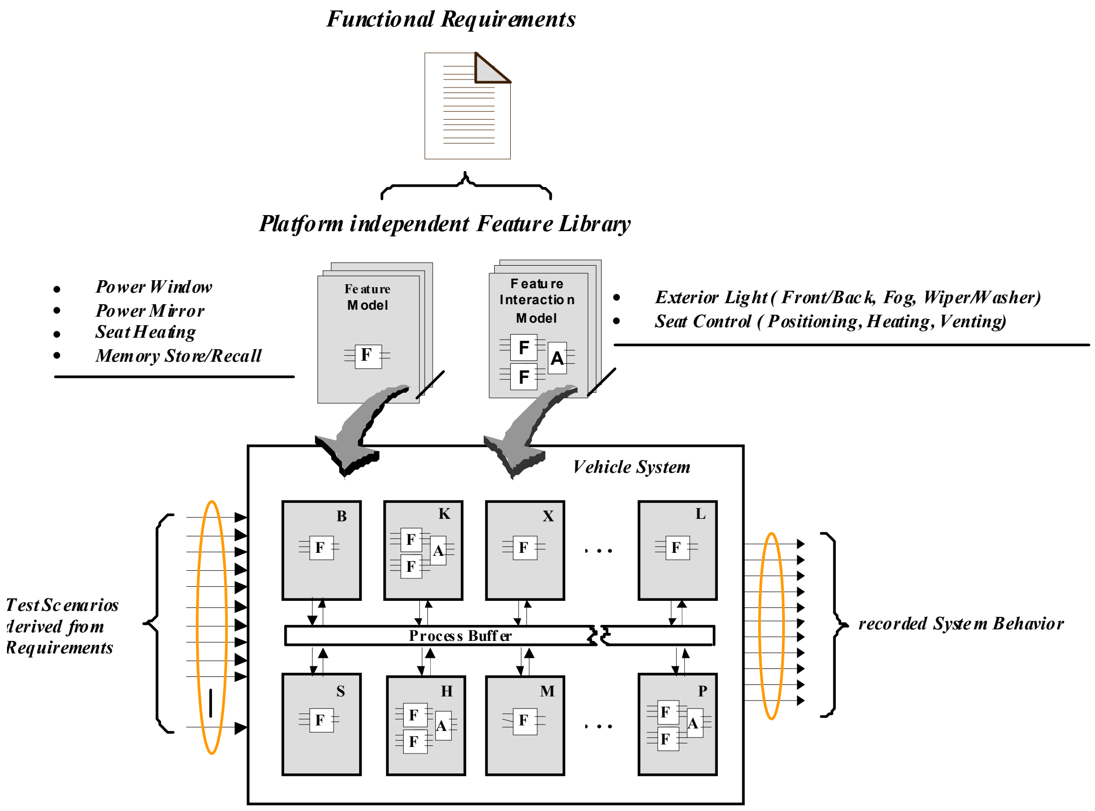


Building the Conceptual Model

The feature-based system design process follows an iterative bottom-up/top-down approach. It starts at functional decomposition level 2 (Component-Level). See the **Functional Decomposition** figure).

As a first step the functional requirements are mapped to features described either by means of basic feature models or feature interaction models from the feature library. These validated models then are integrated in a common model - the Conceptual System Model. See **Capturing Vehicle-Specific Features in a Conceptual System Model** figure.

Capturing Vehicle-Specific Features in a Conceptual System Model


Functional Requirements
Platform independent Feature Library
Power Window
Power Mirror
Seat Heating
Memory Store/Recall
Feature Model
Feature Interaction Model
Exterior Light (Front/Back, Fog, Wiper/Washer)
Seat Control (Positioning, Heating, Venting)
Vehicle System
B
K
X
L
Process Buffer
S
H
M
P
F
A
Test Scenarios derived from Requirements
recorded System Behavior

Note

At this stage it is not necessary to visualize in detail the communication between the individual feature/feature interaction models. The Process Buffer is used symbolically as a means for the interconnection. The broadcasting mechanism is used for external/internal communication.

The benefits derived from the Conceptual System Model are:

- Early proof of concept.
- Easy definition of system-level test scenarios through the purely logical external interface.

The Conceptual System Model is verified and validated through simulation. Test scenarios are derived from the overall system requirements. The recorded test vectors together with the respective system responses form the basis of the later validation of the synthesized system architecture, refer to **System Partitioning**

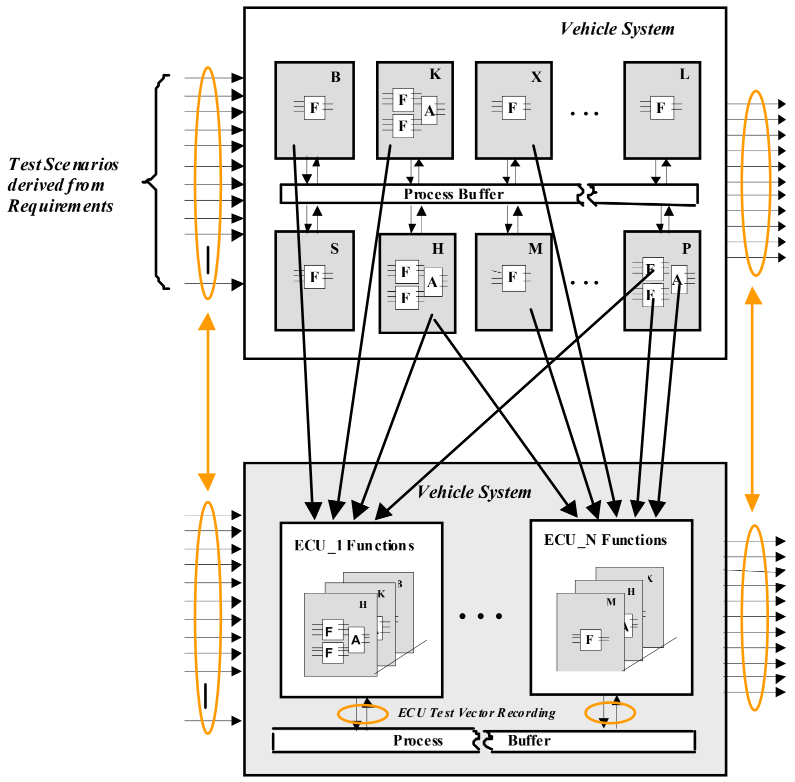
System Partitioning

The next step in the feature-based design process is the architectural system design. The system is partitioned into subsystems (i.e. ECUs) and then bottom-up the feature/feature interaction models or respective sub-functions are distributed among them. See the **System Partitioning, Parsing of Features/Feature Sub-Functions, and Validation of logical Interfaces** figure. Although the resulting model is still purely functional, the partitioning is determined almost by implementation considerations, e.g.:

- Computational resource utilization
- Hardware requirements of a specific function
- Optimization of overall system communication in terms of wiring harness and EMI
- Best fault tolerance or graceful degradation
- Ability to diagnose system failures

In some cases, architectural design criteria might also be derived from additional Performance Models. These models are based on queuing theory and typically used for throughput analysis and identification of potential communication bottlenecks. Rational Statemate should not be used for this kind of analysis. Rather, you can use dedicated tools such as SES Workbench.

System Partitioning, Parsing of Features/Feature Sub-Functions, and Validation of logical Interfaces



Note

At this stage it is not necessary to visualize in detail the communication between the individual ECUs. The Process Buffer is used symbolically as a means for the interconnection. The broadcasting mechanism is used for external/internal communication.

The subsystem partitioning and feature/function parsing is an iterative process. The particular architectural system design is verified by means of the test scenarios previously used for the verification / validation of the Conceptual Model. For each ECU the respective I/O is recorded.

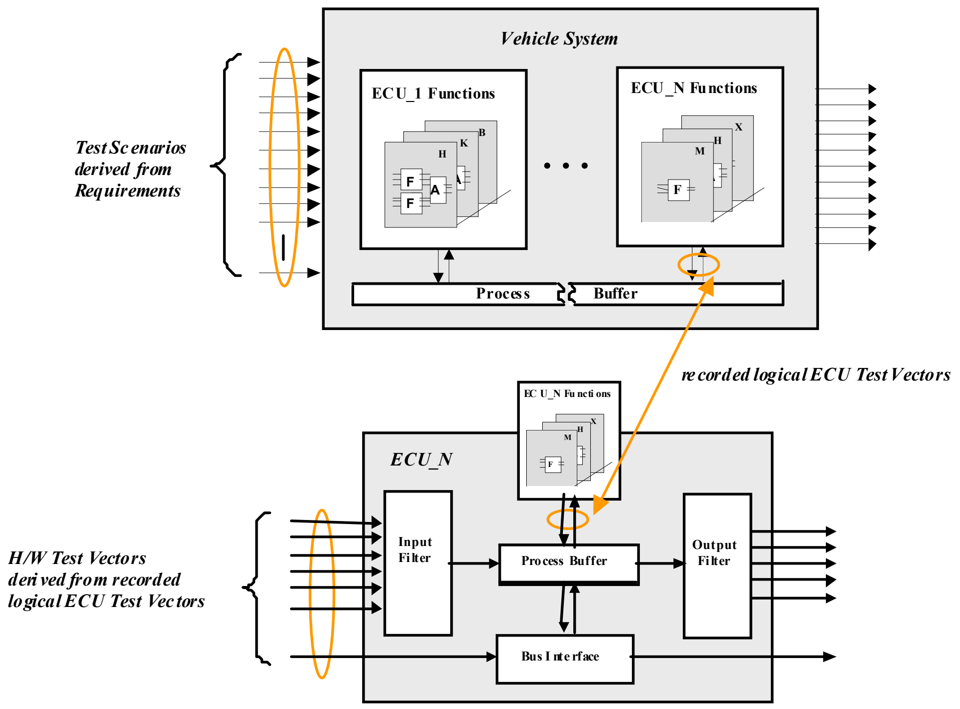
Adding Hardware Design Aspects

The final step in the system architectural design phase is the definition of the hardware attributes of the logical ECU interfaces, i.e. which information will be hardwired and which will be provided via data bus. This top-down design process can be achieved in two ways:

- By adding respective attributes to the logical interface variables, or
- By modeling the respective ECU as described in Generic Top-Level SUD Structure (refer to the Functional ECU Model with H/W Interfaces for Rapid Prototyping in Rational Statemate figure), thus enabling a validation though Rapid Prototyping.

Principally, both approaches allow a seamless transition to the software design by means of the tool MicroC. Which of the outlined approaches is appropriate depends on the system design validation criteria in the OEM / Supplier environment. Either the OEM engineer wants to validate the hardware design. In this case the engineer uses the Rational Statemate Rapid Prototyping feature or transfers the hardware design validation to the supplier. In this case, Rapid Prototyping is performed directly on the target by means of the MicroC tool based on the logical interface variable attributes.

Functional ECU Model with H/W Interfaces for Rapid Prototyping in Rational Statemate


Vehicle System
ECU_1 Functions
ECU_N Functions
Test Scenarios derived from Requirements
Process
Buffer
recorded logical ECU Test Vectors
ECU_N Functions
ECU_N
H/W Test Vectors derived from recorded logical ECU Test Vectors
Input Filter
Process Buffer
Output Filter
Bus Interface

Model Verification and Validation

In the following, *verification* is defined as an activity to confirm the fulfillment of requirements. *Validation* is defined as an activity to confirm that the specific intended use of the model is accomplished.

The verification and validation (V&V) process should always follow the same top-down process as the model design refer to **Functional System Design Methodology Roadmap**), i.e. the model should be verified and validated before proceeding to the next level of decomposition.

The following summarizes the steps to follow in the V&V process.

Step 1. Verification and Validation on Chart Level:	• Visually inspect the chart. • Run Check Model on the chart. • Interactively simulate the chart.

Step 2. Verification and Validation on Module/ Sub-System and System Level:	• Identify the normal operations of the system from the requirements. • Simulate the normal operations of the chart, record simulation script files, record input and output patterns, and use trace files to examine state coverage in the waveform viewer. • Modify the recorded script files and create additional test scripts if necessary. • Playback the newly created script files, record input and output patterns, and use trace files to examine state coverage in the waveform viewer. • Identify the boundary cases and failure conditions to test. • Simulate the boundary cases and failure conditions, record simulation script files, record input and output patterns, and use trace files to examine state coverage in the waveform viewer. • Modify the recorded script files and create additional test scripts if necessary. • Playback the newly created script files, record input and output patterns, and use trace files to examine state coverage in the waveform viewer. • Generate Prototype code from the model and ensure the generated code behaves the same as the model by comparing the output stimulus pattern with the output pattern from simulation.

Note

On the system level, the V&V efforts should focus on the communication among the parts of the system, and not attempt to duplicate all of the chart level testing.

Model Inspection

Several errors in the model can be detected by visual inspection. While this can be done on-line, it is more convenient to work from a hardcopy produced by printing the chart with the “With Elements' Dictionary” option enabled.

Common visual checks are:

- Identifying unresolved elements,
- Checking the scope of resolved elements
- Identifying any potential non-determinism when exiting a state with multiple exiting transitions.

Visual inspection should be done by the creator of the chart or by another engineer familiar with the chart(s) being inspected.

Model inspection should also include the "cleanup" of the design. (Refer to the **Style Guidelines and Best Practices** section.) It involves e.g. the definition of actions in order to make charts more readable, redrawing flows and transitions that cross unnecessarily, ensuring that actions are consistently placed either on transition labels or in static reactions, checking for items that can be relocated into the Global Definition Set, and compacting the chart hierarchy.

Static Model Analysis ("Check Model")

The Check Model feature is used to statically check the model for syntax errors, incomplete usage of language semantics, and inconsistent usage of model elements.

Check Model runs approximately 400 tests against the model, and generates a report of all errors found. The user can customize the tests as needed to disable some of the tests performed by Check Model. Disabling tests will be necessary to eliminate the reporting of error messages caused by model inconsistencies that result when intentional deviations from the standard graphical representations are taken. If you want tests beyond those supported by Check Model, they can often be created using a DGL (Document Generation Language) or a Dataport program. These custom tests must be run outside of Check Model.

Dynamic Model Analysis (Simulation)

Following the static analysis of the model, it is necessary to dynamically exercise the model to verify and validate that the system functions correctly. These tests typically start at decomposition level 2, when behavior (e.g. by means of Statecharts, Mini-Specs, Truth Tables) is captured.

Test vectors and tests scenarios are derived directly from the written requirements. If during modeling additional requirements ("derived requirements") are formulated, they have to be documented and the model has to be checked against them.

The model can be exercised in an interactive mode where the user injects stimuli into the model, controls the simulation of the system functions, and observes the response from the model. The code generated from the model in the interactive mode is an interpreter code, thus allowing to debug the system by stepping through it forward and backwards.

In addition, the model can be executed in a batch mode by means of a script written in the Simulation Control Language (SCL). It is based on the Rational Statemate Action Language used to describe the behavior of the model. A simulation script can also be generated from a playback file, recorded while interactively simulating the model. It might be used as the baseline test that can easily be modified to quickly create a set of tests that later become part of a suite of regression tests.

Note

SCL scripts are test programs re-usable only in the Rational Statemate environment. For the later re-use of test scenarios outside the Rational Statemate tool (e.g. in MicroC), test vectors should be generated from the SCL scripts.

While executing the model in either interactive or batch mode, a waveform display of the model elements can be used to capture the history of the changes in the model. These changes might also be captured textually via a trace file. After completion of the simulation, the respective trace file can be displayed as a waveform for easier analysis. An important use of the trace files and/or Waveform Viewer is to check for test coverage. In cases where the trace files are very big, a testbench chart should be used to record only the model information needed for the coverage analysis.

Additionally, testbench charts and graphical panels are needed for model verification and validation.

Testbench charts are used to accomplish following tasks:

- Modeling the system's environment in a closed loop (refer to **Closed Loop Testing**)
- Inducing system disturbances (e.g. for FMEA purposes).
- Monitoring for undesirable hazards and error conditions.
- Providing a translation scheme between a graphical panel and the system.
- Recording the values of model elements in a data file.

Graphical panels are used to visualize the functionality of the model without having to look at the charts that describe the system's behavior. They should be considered as another view of the SUD, in order to communicate system understanding at a higher level of abstraction. This is particular useful when talking with marketing departments, suppliers or managers. Often, panels are used as a Graphical User Interface (GUI) to the system, but they are also valuable as a graphical testing interface to the system.

Note

Graphical panels in Rational Statemate should not be viewed as photo-realistic. They are primarily engineering views. Much time can be wasted trying to produce a GUI with a high degree of realism. If photo-realism is a requirement, then a dedicated tool should be used.

The dynamic analysis starts with the testing of the normal operation of the system to ensure that the model functions as expected. Then boundary cases and failure mode operations should be analyzed. Ideally the designer of the model should do the normal mode tests while another engineer should be tasked with the verification and validation of boundary cases and failure mode operations.

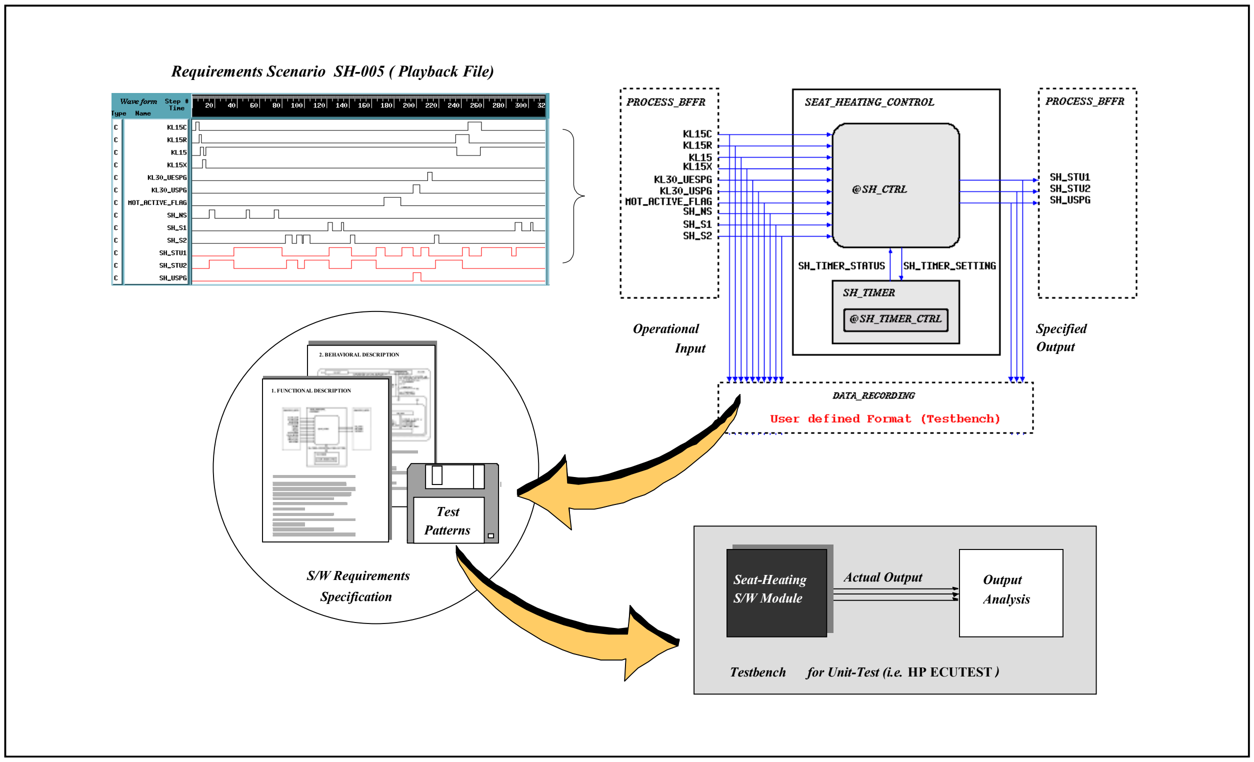
It is often helpful to hold a Peer Review in order to ensure the designer that the system requirements were correctly interpreted.

In order to perform boundary case and failure mode analysis, the respective boundary cases and system failures must be identified. A testbench chart should be created to check for incorrect operation in these modes. While the use of a testbench is not mandatory, it enables the checking of boundary cases and system failures under all operational aspects. These testbenches can also be re-used during system integration

Testbench charts might also be used to record the inputs and outputs of each module/sub-system in a format needed later in the test equipment.

The following figure shows an example from an automotive application:

Test Pattern Generation and Re-Use of Tests ("Unit Tests" Automotive)



In a first step the SUD is executed in playback mode and trace files / waveforms are generated for coverage analysis. Once a scenario is approved for later re-use, the respective (SCL-) file is re-played for data recording via the testbench chart for later re-use in the test hardware HP ECUTEST. The recorded test vectors are part of the model documentation (i.e. SRS).

Closed Loop Testing

For model verification and validation it is not sufficient to analyze the System Under Test (SUT) in an open-loop mode alone. As most systems interact with their environment closed-loop they also have to be tested closed-loop. Two approaches are suggested for closed loop testing:

One is, to use testbench charts. As long as the behavior of data sources and sinks identified in the context diagram (refer to **Capturing the System Context ("Extended System Context")** and **Building the Conceptual Model**) can be described by state-machines, the loop can be closed through respective testbench charts applying the mechanism of Broadcasting.

Note

For the validation of the interfaces within the system, the testbench charts should only generate operational inputs.

Another approach would be to close the loop at the extended context diagram level. Refer to **Capturing the System Context ("Extended System Context")**. Each of the identified nodes should be modeled with respect to its interaction with the SUT:

- If state-based using Statecharts.
- If time-continuous (e.g. capturing sensor / actuator dynamics) using the VisSim tool within Rational Statemate. In this case, a testbench chart should only be used to monitor and/or induce system disturbances (e.g. in an FMEA analysis).

As the extended context diagram corresponds to the later system integration layer, this approach additionally validates the operational interfaces between the SUD and its respective data sources and sinks.

Prototype Code Generation

A final test of the model is to generate prototype code from the model - including the graphical user interface - and test the executable prototype using the recorded stimuli and response files from the interactive simulation. The compiled code might be run on any OS compatible computer. The process of generating code requires Rational Statemate to perform additional tests on the model that are not needed for simulation and are not performed by Check Model.

From the V&V process point of view, it is important that code is generated before proceeding to the next level of decomposition since, as the system becomes more detailed and complex, a point might be reached where interpreted simulation is no longer viable due to speed and performance considerations. It is much easier to generate code gradually as the model is built up, when problems can be more easily identified and fixed rather than leaving the code generation for the end of the design process where problems are likely to be deep within the model.

Style Guidelines and Best Practices

In the following formal guidelines and "best practice" advice for the usage of the language of Rational Statemate will be given. They are based on many years of modeling experience by IBM consultants in aerospace/defense and automotive projects. They have proved to significantly enhance the readability of specifications developed using Rational Statemate.

Activity Chart Conventions

Graphical Settings and Drawing Preferences

Whenever the Activity Chart Editor is used, the respective activity chart should fill the full size of the monitor; also enabling a standardized format of the graphics in the automatically generated documentation.

General Settings	Grid ON with 0.25 spacing and 10 pixel trap radius: • **Flow Lines**: straight lines • **Line Width of Flow Lines**: 1
Standard size of Functional Blocks, Data Stores, External Activities, and Control Activities	Function Blocks, External Activities and Data Stores: • **Box Height**: 2 • **Box Width**: 3.5 • **Line width**: 1 Control Activities should be double the width·
Standard font and font size for Names and Labels	• **Names**: Black Courier, 12 point, bold • **Labels**: Black Courier, 12 point, bold, italic

External / Environment Activities

All External / Environment Activities should have the same size in height and width. If a bigger size is needed, the respective size should be a multiple of a pre-defined basic size.

Data sources (Inputs) shall be placed on the left side, data sinks (Outputs) shall be placed on the right side of the Activity Chart - even if their names are identical.

In addition to the External / Environment Activity name it might be useful to add textual notes next to the name to indicate their origin from the modeling point of view - e.g. TESTBENCH.

Internal Activities

All Internal Activities should have the same size in height and width. If a bigger size is needed, the respective size should be a multiple of a pre-defined basic size. There is no magic figure for the number of functions/chart. The priority is readability and this requires a clear layout of activities and associated connectivity. Off-page hierarchy ("Create Sub-Chart") should be used to maintain readability.

Internal Activities should be placed one below the other thus enabling the reader to reveal the dynamics of the system by reading the chart "top-to-bottom" and "left-to-right."

The name of the top-level activity should be identical with the chart name.

Control Activities

A Control Activity should have the same size in height and width as Internal Activities. Contrary to the guidelines for External/Internal Activities there is no fixed scaling, if a bigger size is needed.

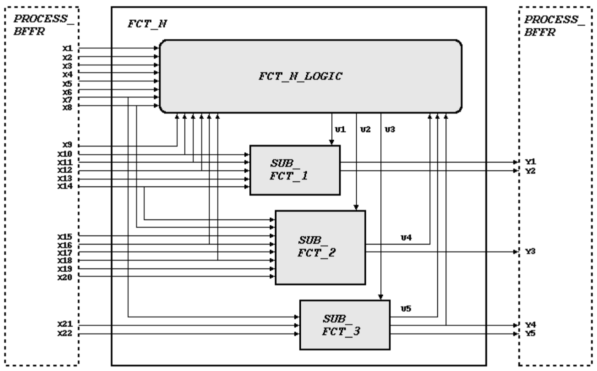
A Control Activity should be located at the top of the internal activity it resides in, and be named with the activity name appended with '_CTRL'.

Data-, Control- and Information-Flows

Flow lines, which represent either data flows or control flows, shall be drawn as solid lines. Dashed lines shall be avoided. Within Rational Statemate there is no difference in the handling of those flows and furthermore, different line styles might lead to confusion.

Principally you can use information-flows between external and internal activities as well as between internal activities. Only at the lowest level of decomposition the information flow from/to external activities should be split into its data/control elements. Flows between external activities and Control Activities generally should depict the explicit data/control information (see the following figure).

External and Internal Flow Lines



All internal flow lines shall flow in a clockwise direction, with the respective names on the left-hand side of the arrow direction. The names of external flows should be written in the external activity, not on the arrow that defines the data flow.

Junction Connectors should not be used. Use direct branching of flows instead.

Readability hint: When flows cross within a chart, only a "T" is a joint or fork while a "+" is simply a crossing line with no connection.

Data Stores

In Rational Statemate data stores do not have any impact on code generation. Therefore they can be used as an additional means of visualization:

- Data stores might represent stored information for later use (e.g. characteristic tables/ maps) or describe a buffer in the computer memory (e.g. "PROCESS_BUFFER" in the **Generic Top-Level Structure for ECUs** figure). In these "classical" applications data stores are named with the data they contain. Unlabeled flows to or from a store carry the whole data group of the store. Flows carrying subgroups of the stored data are labeled with the respective subgroup names.
- Data stores might also be used to visualize the different of communication characteristics within a system e.g. CAN BUS, J1850,… etc. For details on modeling data bus communication, refer to **Modeling Bus Communication**.

Page Connectors

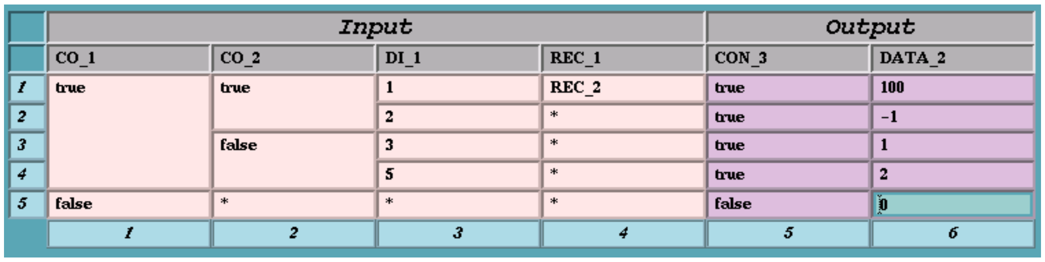
In-Page Connectors should only be used when the readability of the activity chart is disturbed by a direct connection. Off-Page Connectors should be avoided.

Combinational Assignments, Mini-Specs, Subroutines, and Truthtables

Combinational Assignments should not be used since there is no visible indication in the Properties window once combinational assignments are defined. Mini-Specs starting with “started” should be used instead.

For reading and debug purposes, mini-specs are usually no longer than one page. Also for readability reasons indents and tabs should be added to show nested loops. If the mini-spec code is longer than a screen page, it might be more suitable to implement the action line with a subroutine, described by the action language. By this the Micro Step Debugger of the simulator can be used to debug the single step of behavior. In cases where mini-specs consist of complex netted “if .. then .. else” constructs, a more suitable approach would be the use of truth tables (refer to the following figure).

Mini-Spec and Truth Table Description



	Input				Output	
	CO_1	CO_2	DI_1	REC_1	CON_3	DATA_2
1	true	true	1	REC_2	true	100
2			2	*	true	-1
3		false	3	*	true	1
4			5	*	true	2
5	false	*	*	*	false	0
	1	2	3	4	5	6

```
if CO_1 then
  if CO_2 then
    if DI_1=1 and REC_1=REC_2 then
      tr!(CON_3);
      DATA_2:=100;
    else
      if DI_1=3 then
        tr!(CON_3);
      else
        if DI_1=5 then
          tr!(CON_3);
          DATA_2:=2;
        else
          fs!(CON_3);
          DATA_2:=0
        end if;
      end if;
    end if,
  end if;
           end if
```

Sometimes Statecharts are used instead of truth tables because of their graphical debug capabilities through simulation. However, from the methodical point of view, keep in mind that, as in most cases "if .. then else" constructs might not be associated with "real" states (refer to **Describing States**).

Generic Charts, Libraries and Components

A Generic is a specific type of chart with an explicit interface ("Data Encapsulation"), enabling a modular development of components. Although both Activity Charts and Statecharts might be defined as Generics, use only generic Activity Charts in system modeling (exception: Testbench Charts).

Generics should not be seen from a model reuse perspective alone. They are also a powerful means in a team-based development. Models of the different team members should always be defined on the top-level as Generics thus forcing a clear interface definition. This facilitates the later integration into the overall system significantly. A chart can be changed at any time from regular to generic and vice versa.

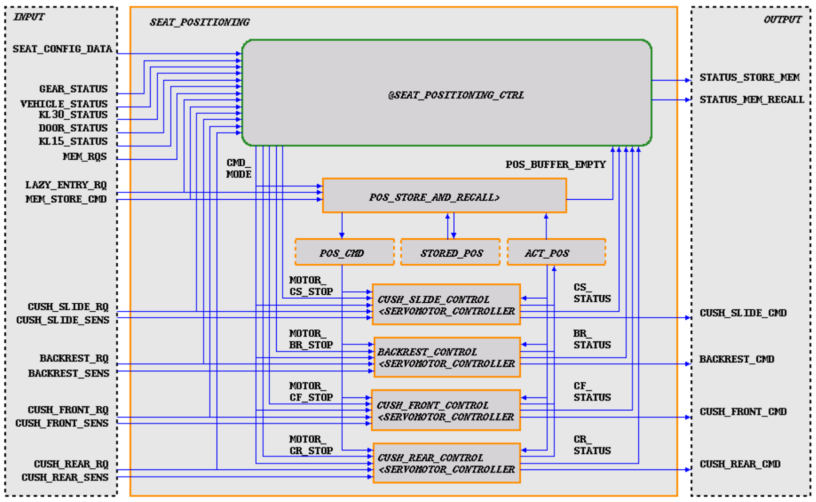
Note

Testbench Charts do not have any scope into generic charts. Therefore, modeling and model validation should be performed with the regular chart prior to its change to a Generic.

The nesting of Generics ("Generics in Generics") should be minimized even though Rational Statemate supports it. Debugging the model might become troublesome due to the difficulty in referencing parameters. Three levels depth should be sufficient.

The following figure depicts the top-level structure of a Generic Activity. Each generic formal parameter is shown graphically by a flow making it easy to determine from the diagram whether the parameter is used as an input, output or both input and output. When using large complex data structures, only those elements that are really needed should be passed instead of the entire structure.

Top Level of a Generic Chart (Seat Positioning)



External Activities of a top-level generic chart should not show any details of the origin and destination of the I/O parameters. Their names should be simply INPUT and OUTPUT.

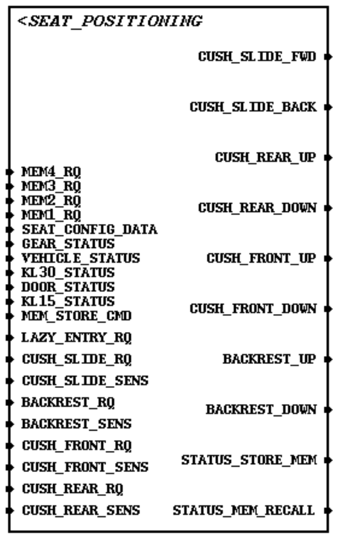
Once a generic chart is created and validated it can be archived as a component (see the **Example of a Library Component** figure) and entered into a model library.

Library Components can be used in other models by simply dragging the component from the library and dropping it into the respective chart.

Note

If a Library Component is edited in the library, any modeler who has created a dynamic link to the component, rather than just copying it sees the new change taking effect the next time the simulation is run.

Example of a Library Component



Statechart Conventions

Graphical Settings and Drawing Preferences

Whenever the Statechart Editor is used, the respective Statechart should fill the full size of the monitor; also enabling a standardized format of the graphics in the automatically generated documentation.

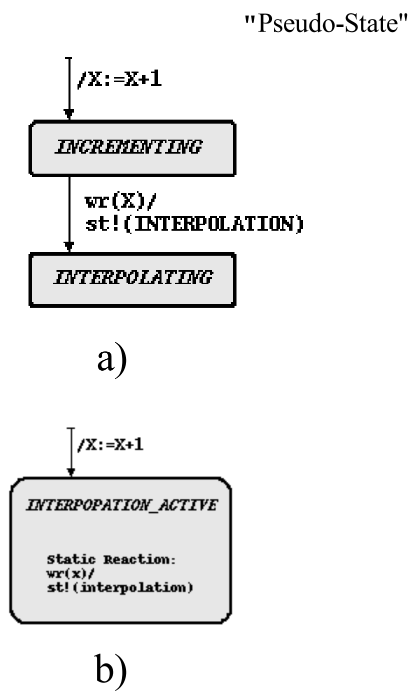
General Settings	Grid ON with 0.25 spacing and 10 pixel trap radius
Standard Size of States	• **Box Height**: 2 • **Box Width**: 3.5 • **Line Width**: 1 (Highlighted: 2)
Standard Font and Font Size for Names and Labels	• **Names**: Black Courier, 12 point, bold • **Labels**: Black Courier, 12 point, bold, italic

Describing States

Statecharts are used to describe the behavior part of the model. From a methodological point of view, they should be used to depict system states and transitions between them. As it is quite easy to translate Statecharts into code, some users apply them as graphical representations of what will later become their software code. By using Statecharts as a means for graphical programming, “system state aspects” usually are neglected. Statecharts of this kind are referred to as “Procedural Statecharts” and not used in functional system modeling.

Statecharts can be simulated. Because of the step-semantic of the Rational Statemate simulation some users tend to add artificial states either to initialize a step needed to perform an action (see the following figure) or to synchronize with concurrent processes. Statecharts should be free of such not system-related “pseudo-states.”

Capturing Simulation Steps in Statecharts



In the following a system state will be referred to as a time consuming mode of operation (e.g. WAIT_FOR_CMD or FCT_X_ACTIVE), needing a compelling event to initialize a transition to another mode of operation.

All states within a Statechart should have the same size in height and width. If a larger size is needed, the respective size should be a multiple of a pre-defined “basic” size.

State Names should describe the respective system mode of operation. Dummy names like IDLE or WAIT should be avoided. If a state is connected to a function, its name should reflect this (e.g. INTERPOLATION_ACTIVE in the **Capturing Simulation Steps in Statecharts** figure). Long names should be broken into two or more lines.

Describing Structure and Priority

States can be decomposed into sub-states using Hierarchy. This is extremely useful for adding a further level of detail to the behavior.

At the same time hierarchy can be used to represent levels of interrupt and priority to transitions.

When hierarchies are used in order to the visualize interrupt levels, the respective super-states need not be named, unless a related timeout expression is used. Obviously when hierarchies are used for state decomposition purpose, it is essential that the respective super-states be named accordingly.

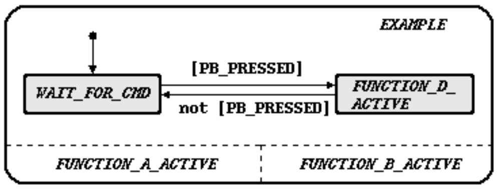
Concurrency

For readability reasons do not describe the behavior of concurrent processes by means of concurrent state machines if these processes need to by synchronized. Instead, each process should be captured by individual function blocks (Activities). The process synchronization then should be performed "hardwired."

Nevertheless, if concurrent state machines are used and the broadcasting mechanism is used for synchronization, they should be restricted for readability reasons to one chart level.

Functions (Activities), which are always active should be explicitly visualized in the control activity as concurrent processes (see the **Visualization of Concurrent Processes in a Control Activity** and the **Feature Model: Seat Heating** figures) with a respective Static Reaction (refer to **State Transitions**).

Visualization of Concurrent Processes in a Control Activity



State Transitions

Transitions between states shall be drawn as straight lines. Within a Statechart the transitions shall have a common sense of rotation (either clockwise or counter-clockwise). Transitions should neither cross each other nor states.

Labels shall be positioned on the left-hand side of the arrow direction and follow the MEALY Syntax. (Event and/or Condition / Action). The '/' should end the line prior to an action on a conditional transitions. When conditional statements must be broken to start a new line, the lines should end after the conditional operator. The actions following the slash should be allocated a single line for each action.

If the label is too long, the triggering events and/or conditions as well as the associated actions might be defined separately as Compounds with respective meaningful names.

The use of Static Reactions (MOORE Syntax) should be limited to Rational Statemate or simulation specific syntax that is unnecessary to the reader for understanding the behavior. An example for Rational Statemate specific syntax is the starting and stopping of activities. Following the state naming convention it should be obvious that when entering a state named <activity_name>_ACTIVE the respective activity is started without seeing the Rational Statemate syntax.

Note

Within / Throughout - syntax should not be used.

Default Entries

It is possible to minimize the number of Default Entries when hierarchies have been used to describe interrupts by drawing a single default transition from the top-level super-state to the default-state. In this case each of the nested super-states does not need separate entry transitions.

In-Page and Off-Page Connectors

In-Page Connectors should be used to join the same transition together, without crossing existing transitions. For readability reasons descriptive names should be chosen.

Off-Page Connectors are necessary when a transition needs to return to a previous higher level off-page chart. It is essential that the 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 might facilitate this.