Telelogic

Statemate®

Methodology and Style

IBM®

Statemate®

Methodology and Style

This edition applies to Telelogic Statemate 4.5 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Methodology

In system development, there are many different life cycle models (for example, Spiral, Waterfall, V-Process) that describe the systems design process from initial requirements through to implementation. In order to apply these guidelines, it is important to relate them to the process of system development.

With the experience gained in the field of Systems Design, it has become apparent that the V-Process model is a widely used and accepted practice for system development.

In the V-Process model, the left leg of the V basically describes the process of system analysis over time. The right leg of the V corresponds to the synthesis path over time, where smaller components are integrated to build the complete product.

The Analysis phases of the V-Process are represented as a consecutive sequence of four steps:

1. System Requirement
2. System Specification
3. System Design
4. Hardware/Software Implementation

The reason for the partitioning of the design process into these four steps is the idea that in the early stages of the design process we must not be concerned with implementation details. In the early stages of gathering the ideas for the system (System Requirements) and then formulating them into a specification (System Specification), we are simply concerned with what the system must do.

Functional decomposition then serves as a bridge between Specification and Design. Usually the higher level functions are abstract entities which express ideas that are part of the requirements. The process of decomposing such functions, in effect, describes how to accomplish each of those functions.

In the System Design phase, we start to focus on the real-time constraints on the system and the sequencing of the functions. However, we are still not concerned with which microprocessor is used nor the available memory.

In the fourth stage (HW/SW Implementation), we are now concerned with the final implementation details such as Operating System, microprocessor, and the available memory. We deal with these detailed issues secure in the knowledge that the overall system concept has been proved in the preceding steps. Thus, these four steps bring us from the conceptual system to the real product.

Then we go up the right leg of the V from HW/SW Implementation Tests, to Module Integration Tests, to System Integration Tests, and finally to System Acceptance Testing. This brings us to a very important point of the V-Process model. That is, at each stage of the analysis/design you must also perform test definition and define the test scenarios to be used for later integration and acceptance testing.

The traditional way to create test specifications is in a textual form that describes the component to be tested, applied stimulus, and expected results. However, the problem with this approach is that there is no feedback from test results to the design. At integration when a test produces a result that is different from the predicted result, it is not immediately obvious if it is the test specification that is wrong, or if the problem is in the module being tested.

Because a simulatable Statemate model exists throughout all stages of development, you can easily

- Generate these test scenarios through simulation,
- Store the stimulus and results, and then
- Use them later in the integration and acceptance tests.

Therefore it is vital that Statemate analysis is at the center of the development, driving the development process so that these tests can be accurately defined. Effectively using the analysis capabilities of Statemate will shorten the overall system development life cycle.

In the early stages of system development, two types of models are used:

- Functional Model
- Performance Model

In the Functional Model, the aim is to capture the basic structure, ideas and algorithms for the system. In a Functional Model, we are not interested in any details of the implementation. It is a common characteristic of functional models that all the activities in the top-level chart are running concurrently. Also, if there is a bus interface, we are only interested in what messages go to and from the bus. We are not interested in the details of message packing (refer to the **Generic Top-Chart Approach for Controller Designs** figure.)

The other type of model used at this stage is a Performance Model. At this level of analysis when we talk about performance models, we mean models based on queuing theory and the analysis typically involves running very large simulations to statistically analyze the performance of the system. Performance models are used to determine the throughput of the system, determine the response time of the system, and identify any bottlenecks or critical paths.

Moving down through the analysis stages, we then come to Implementation Model. The Implementation Model bridges the gap between the functional systems specification and the final software design. This practice enables the software engineers to write the software directly from the specification document produced from the Implementation Model. Hence it is very important that the software engineers are involved in creating this model.

The major change in switching from the Functional to the Implementation Model is that, in the Implementation Model we are primarily concerned with the sequencing and timing of the functions, as only one function can be active at any time (assuming a single processor system). Therefore, control activities must be added, or adapted, to specifically control the sequencing of the functions. Also each function must be modified to become self-terminating so when activated, it executes and then terminates in the same way as the final code will. As part of this Implementation Model, we also want to specify the real-time constraints on the software i.e., how long each function is allocated for execution in the final system.

Obviously there is some element of rework to move from a Functional Model, where everything is concurrent, to an implementation model, where everything is sequential. For the efficiency of the systems development process, it is important that this rework be minimized.

We must always start with the Functional Model, but it is important to define when to stop and switch to Implementation. As soon as we have validated the functional idea, we should move to the Implementation Model. It is recommended not to proceed past the second level decomposition before switching to an Implementation Model.

The Implementation Model is not appropriate in every situation. In the case of a prime and subcontractor relationship, the prime contractor may be responsible for the overall system integration and test while the subcontractor may be responsible for software development. The prime may then test the delivered integrated software against Statemate -defined integration and acceptance tests. In such a situation, the prime contractor should continue to analyze the system with functional modeling and need not concern themselves with the details of implementation.

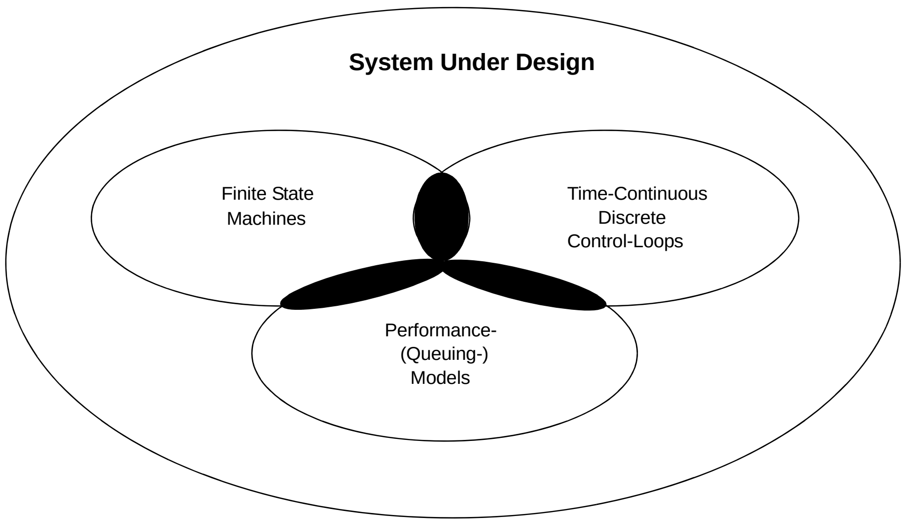
The Implementation Model can be further extended to encompass Hardware-in-the-Loop analysis. By this, the automatically generated C code prototype is connected into the system by means of a bus interface. The product can be operated with this prototype and final optimizations and testing can be performed. This Hardware-in-the-Loop analysis is really the ultimate in verification, and provides absolute confidence that the specification is correct and the system will work when built.

Basic Types of System-Models

To model a system accurately, there are typically three different domains that you must analyze:

- **Discrete Event / Modal Operation**—Finite State Machines are most commonly used for the modeling of the discrete event / modal operation of the system. The Harel Statecharts contained in Statemate offer powerful extensions to the traditional finite state machines.
- **Time-continuous / Control Loops**—Time-continuous / Discrete Control-Loop analysis is performed by mathematical models that accurately represent the behavior over time.
- **Performance**—Performance Modeling based on queuing theory is used to determine the system characteristics of throughput, response time, and resource utilization and contention. Typically the elements in the system will be modeled to a basic level of functionality to ensure the accuracy of the analysis.

System Model Domains


System Under Design
Finite State Machines
Time-Continuous Discrete Control-Loops
Performance- (Queuing-) Models

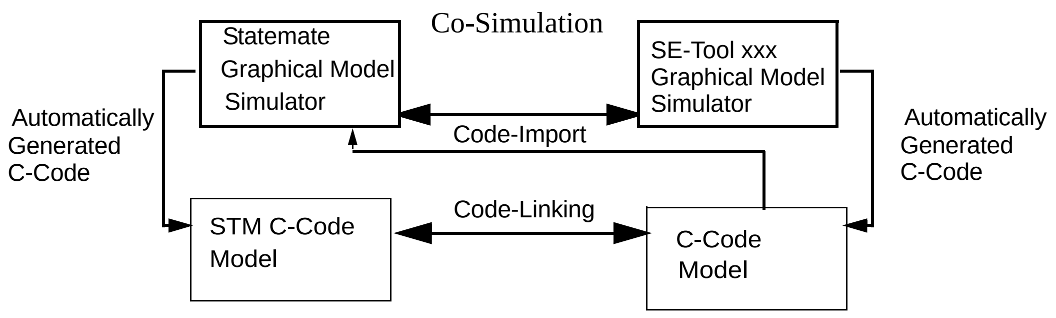
The reason the domains are shown as overlapping is that most systems under design are a combination of these different domains. Therefore, to model the system accurately, the different models must be linked to provide a complete system analysis.

This brings us to the most important role of Statemate as a tool for Systems Engineering. Through the Functional Model (Activity-charts), Statemate provides the overview of the entire system. This enables you to define the system functions and interfaces so that you can model blocks in different domains/tools. It also brings together models from different domains to provide a complete system analysis. Statemate therefore is a powerful tool for system integration and test development.

Statemate in a Systems-Engineering Environment

There are several ways of interfacing Statemate to other Systems Engineering Tools (see the **Statemate Interfaces** figure.)

Statemate Interfaces


Co-Simulation
Statemate Graphical Model Simulator
SE-Tool xxx Graphical Model Simulator
Automatically Generated C-Code
Code-Import
Automatically Generated C-Code
Code-Linking
STM C-Code Model
C-Code Model

Co-simulation is where the simulators of both Statemate and other Systems Engineering (SE) tools are running concurrently, and data is passed from one tool to the other. The advantage of this method is that you have good visibility into the design through the graphical animation and/or analysis and debugging capabilities of these simulators.

However, co-simulation trades off performance for flexible analysis and debugging capabilities as both simulators must be running. Continuous time models in particular are computation-intensive and hence cause simulations to run relatively slowly.

Code-to-code interface is another option. By generating code from both tools, you can integrate the code to provide a single executable. This provides the best execution performance, but at the cost of lesser control and visibility of what is happening in the model.

Statemate also provides a mechanism for connecting C code to the Statemate simulation. By doing this, performance is maintained at acceptable levels because the most computation-intensive parts are being run as C executables and we still have the full graphical animation and debug facilities of the simulator.

Connecting C code to the simulator is perhaps the most pragmatic approach to systems analysis and design as it allows any SE tool to be linked with Statemate without the necessity of a custom interface.

This ability to integrate external code into models makes Statemate the ideal tool for use as the systems testbench. By bringing together models from different domains/tools into a single analysis, Statemate provides a unified environment for the complete system analysis.

The Statemate Approach

The following sections provide the following information:

Two Aspects of System Description

The graphical language of Statemate provides a *stationary* description of what the system does (Activity-chart) and a *dynamic* description of when and how it does it (Statecharts).

The Activity-chart is the most important view in communicating what the system does. Activity-charts capture the functional view of the system by mapping system functions directly to activities. Hence the Activity-chart provides the static view of the main functions of the system. From the Activity-chart, the reader must be able to understand what the system does.

Although the Statemate specification is entered into a computer, the final reader is still a person. Since the goal is to communicate the specification to people, it is important to keep in mind who the readers are. For control and hardware engineers, the Activity-chart is equivalent to a Block Diagram.

The Statechart provides a behavioral view or a dynamic description of the system. The Statechart can either describe the behavior of a single activity or control the overall execution of a number of activities.

A Statechart used only for controlling when processes are activated, suspended, or stopped is known as a control activity. This means the primary role of the Statechart used as a control activity is to start and stop the activities in the Activity-chart. This has a very natural mapping for our intended readers—Software engineers.

To Software engineers, Activity-charts graphically represent the subroutines they are familiar with as they show: hierarchy, input data, and output data. The control activity is the program’s main, which controls when subroutines are called. Thus the system should be structured in functional modules that make the specification more readable and later help in code generation.

Top-Down vs. Bottom-Up

Top-down design is based on the principle of *decomposition*. Decomposition breaks down complex systems into a hierarchical structure of simpler parts. It is this hierarchy that enables us to understand, describe, and design these complex systems.Functional decomposition, as described below, is the most well known decomposition method.

Functional decomposition consists of the following steps:

1. Define the system context. This high-level view describes what is part of the system and what is external. This view also describes what information flows to and from the system under design.
2. Describe the system in terms of high-level functions and the interfaces between those functions.
3. Refine the descriptions of these high-level functions by breaking them into smaller more specific sub-functions, each of which may also be decomposed.

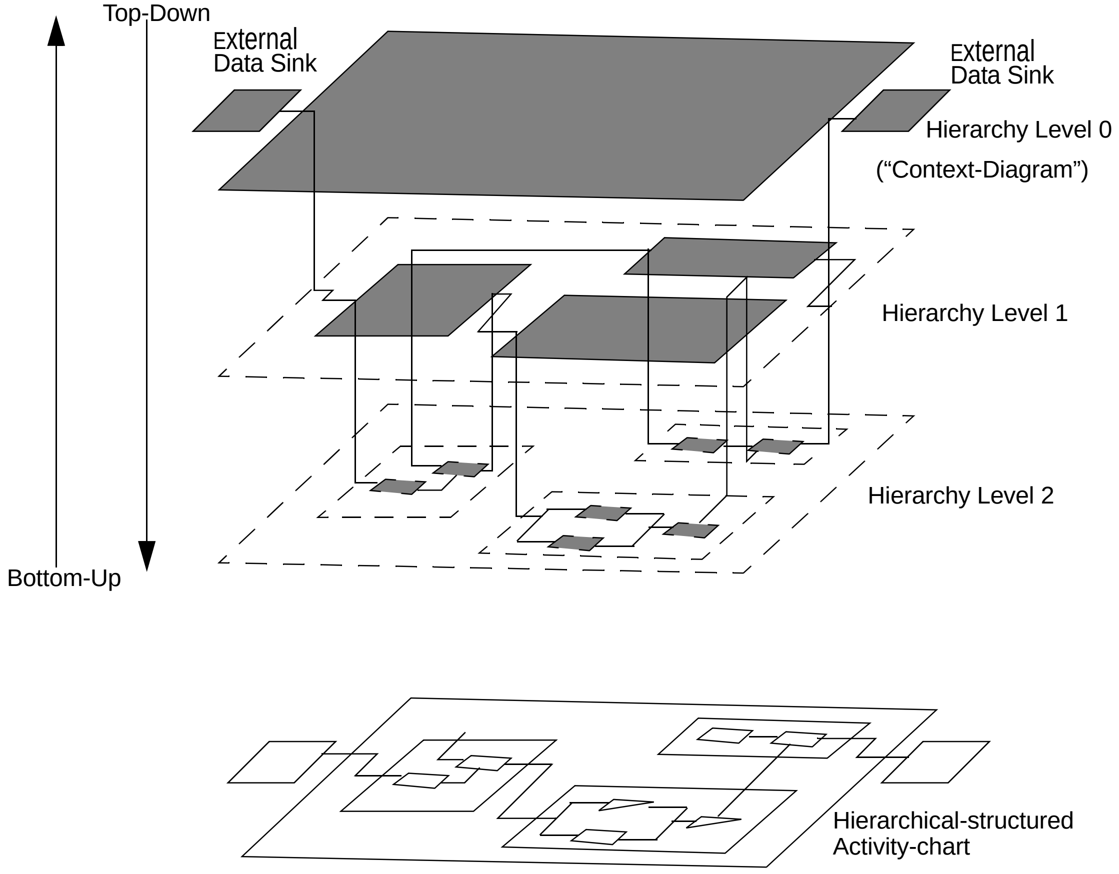
The decomposition process makes the system description easier to understand because it starts from major system functions before it describes details. The decomposition also ensures that the role of each of the identified sub-functions is clear and distinct from the other identified sub-functions, and that the function is described totally by the identified sub-functions.

Avoid Bottom-up design as this seldom results in a good system design. Bottom-up is indicative of a lack of strategic thinking and overall understanding of the system, and ends up forcing a particular implementation on the system that may be far from optimal.

However, one level of Bottom-up expansion is useful, and indeed recommended, in creating the specification. That is, based on the context diagram, we should expand one level higher in the system /network to gain a better understanding of the overall system and clarify the role of the System Under Design. Because this higher level view is not part of the specification, it is not necessary to make a detailed model but just enough to ensure we really understand our role in the overall operation.

This single level of Bottom-up expansion will assist in preparing the design for the closed loop analysis (refer to **System Validation**) by getting the Properties definitions at the correct level in the hierarchy.

Functional Decomposition



The Top-Down System Specification

This section provides the following information:

- Defining External Data-Sources / Sinks (System Level 0)
- The Generic Top-Chart Approach for Controller Designs (System Level 1)
- Functional Decomposition - System Level 2
- Functional Decomposition - System Levels > 2

Defining External Data-Sources / Sinks (System Level 0)

The first thing we must do when starting a systems design is to define the *context diagram*, as it is called in classic Structural Analysis (SA). The context diagram defines the system boundaries in terms of the interface to the external world.

Define the inputs and outputs to the system by answering the following questions:

- What is inside the system under design?
- What is outside?
- What does it interface to or control?

At this stage, you should draw the data flows between the system boundary i.e., Top Activity, and the external activities.

The Generic Top-Chart Approach for Controller Designs (System Level 1)

Experience across a wide variety of projects has shown that a generic approach can be applied to all controller designs. The generic first-level Activity-chart presents a common starting point for all systems specifications. The main purpose of this chart is to depict the following:

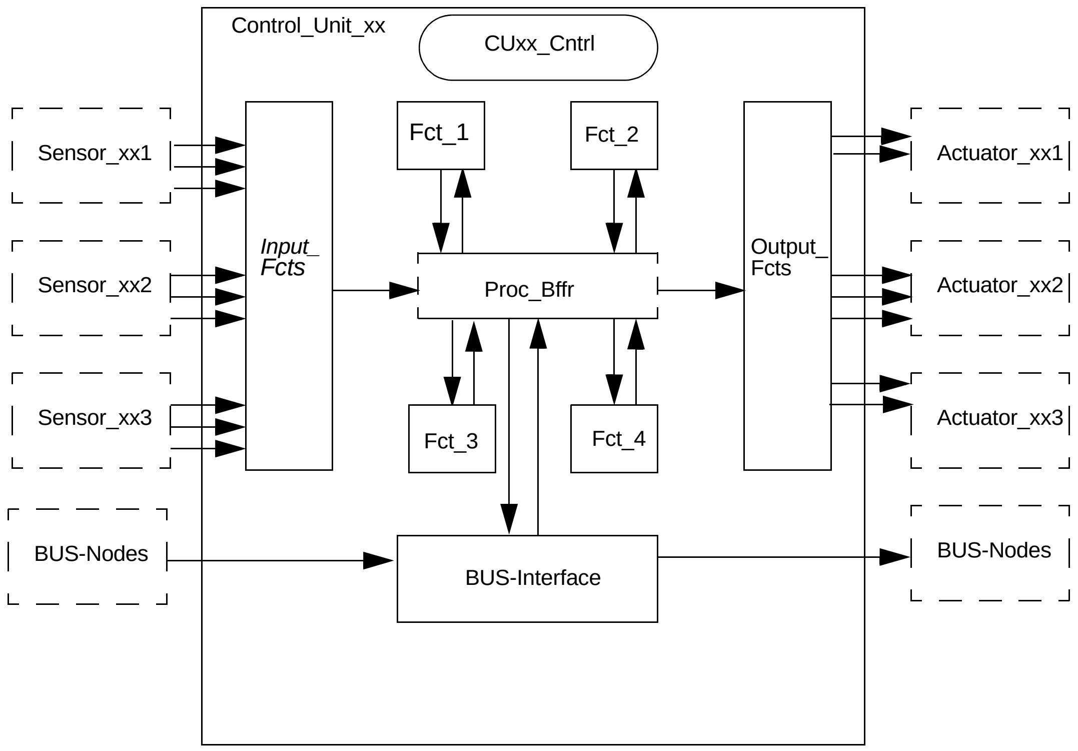
- External Sources and Sinks (refer to **Defining External Data-Sources / Sinks (System Level 0)**)
- Main System Functions
- Global Variables

After defining the external sources and sinks for the context diagram, the next job is to identify the main system blocks. The generic first-level Activity-chart consists of three common blocks that are found in every design (see the **Generic Top-Chart Approach for Controller Designs** figure.)

- Input Function
- Output Function
- Process Buffer

and some additional main system functions that vary from system to system.

Generic Top-Chart Approach for Controller Designs



The Input Function performs the transformation of electrical signals to physical/logical values. Examples of operations performed in this function include: filtering, debouncing, and conversion of electrical to logical values.

The Output function performs the opposite function of the Input Function. It converts logical / physical values to electrical signals.

Between the Input function and Output function there is a PROCESS_BUFFER data store. This is used to provide communication between the main system functions and can be interpreted either as in the classic SA approach or as an internal bus.

In addition to these three standard functions (Input Function, Output Function, and Process Buffer), this chart also shows the main system functions. These functions vary from system to system and potentially consist of all of the following:

- Initialization
- Diagnostics
- Bus-Interface
- Control Algorithms

Each system function connects to the PROCESS_BUFFER by two information flows. One information flow brings inputs into the function, the other takes outputs from the function. The reason for using the PROCESS BUFFER and information flows at this level is to keep the diagram clear and readable and avoid the “spaghetti-like” mess that normally appears in such diagrams.

From the **Generic Top-Chart Approach for Controller Designs** figure, the only part we have not yet discussed is the control activity. A control activity does not always appear in this System Level 1. In a Functional Model all the functions can be active concurrently and therefore a control activity is not necessary. As we move to an Implementation Model we must define the calling sequence of the functions, and control activities must be added or modified to represent this behavior. Effectively, the control activity corresponds to the software scheduler.

A rule, which is common to all parts of this System Level 1 structural description, is that there should be no further hierarchical decomposition on this diagram. The reason for this is to develop some consistency on how these charts look so the readers become familiar with the representation and to make the chart more readable by focusing on the main system functions.

The final role of this generic System Level 1 Activity-chart is to provide a visualization of the global variables. The global variables are the signals flowing to and from the external environment, and between the main system functions. Since these signals appear in or are defined in this top chart, the scoping rules of Statemate make them visible globally throughout the design.

Another recommendation is to use this chart to define system constants whose values need to be empirically determined or whose values can be set from simulation run to simulation run (e.g., time constants) as variables in this chart. These variable constants can then be initialized by the testbench (refer to **System Validation**)

Functional Decomposition - System Level 2

At the next level of functional decomposition, referred to as System Level 2, again some generic guidelines can be applied to the structures at this level.

For I/O functions no further generic approach is possible because of the system-dependent structure (see the **Functional Decomposition of the Input Function** figure).

For the bus interface it is possible to talk about a generic approach.

In a Functional Model, no further decomposition is required. At this stage we are only interested that a certain message is sent / received via the bus interface with a specific timing/frequency. Although these messages are actually between the bus interface and the Process_Buffer, in the analysis we can stimulate and sense them as if they went to the external world.

In the Implementation Model, the bus interface must be decomposed because now we must consider the details of the message packing/unpacking and the timing of the messages. When developing the bus model, bear in mind that the ultimate goal in analysis is the HW-in-the-loop analysis. Structure the bus model so as to make it easy to switch from the Statemate software model of the bus to using the actual system hardware (see the **Generic Bus Interface** figure).

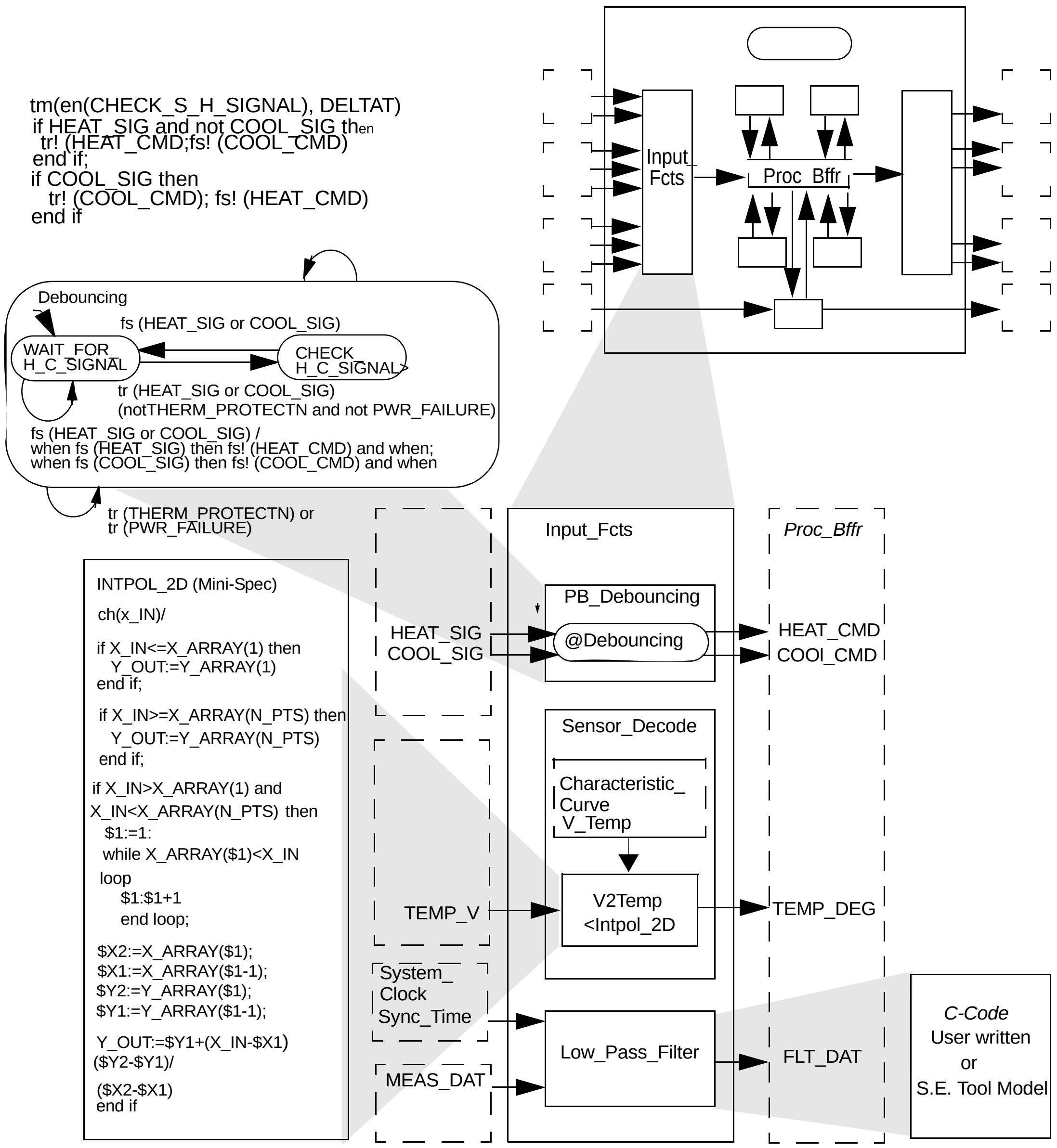
A generic approach can also be used for the System Level 2 decomposition of the control algorithms. Whereas on System Level 1 we show the main system functions, System Level 2 shows the main algorithm functions (see the **Generic Control Algorithm Approach** figure).

At this level, explicitly show each input and output instead of the single information flow used at the previous level. Then at this level you can once again see all the relevant inputs and outputs—inputs on the left and outputs on the right.

Structure this level in functional modules that later help in code generation. Also, arrange the activities to make the system readable from the structure. That is to say, all Activity-charts are read from top to bottom and left to right and the ordering of the activities should be consistent with the calling sequence in the control activity.

For readability reasons, as with System Level 1, no further hierarchical decomposition should be done on this chart.

Functional Decomposition of the Input Function

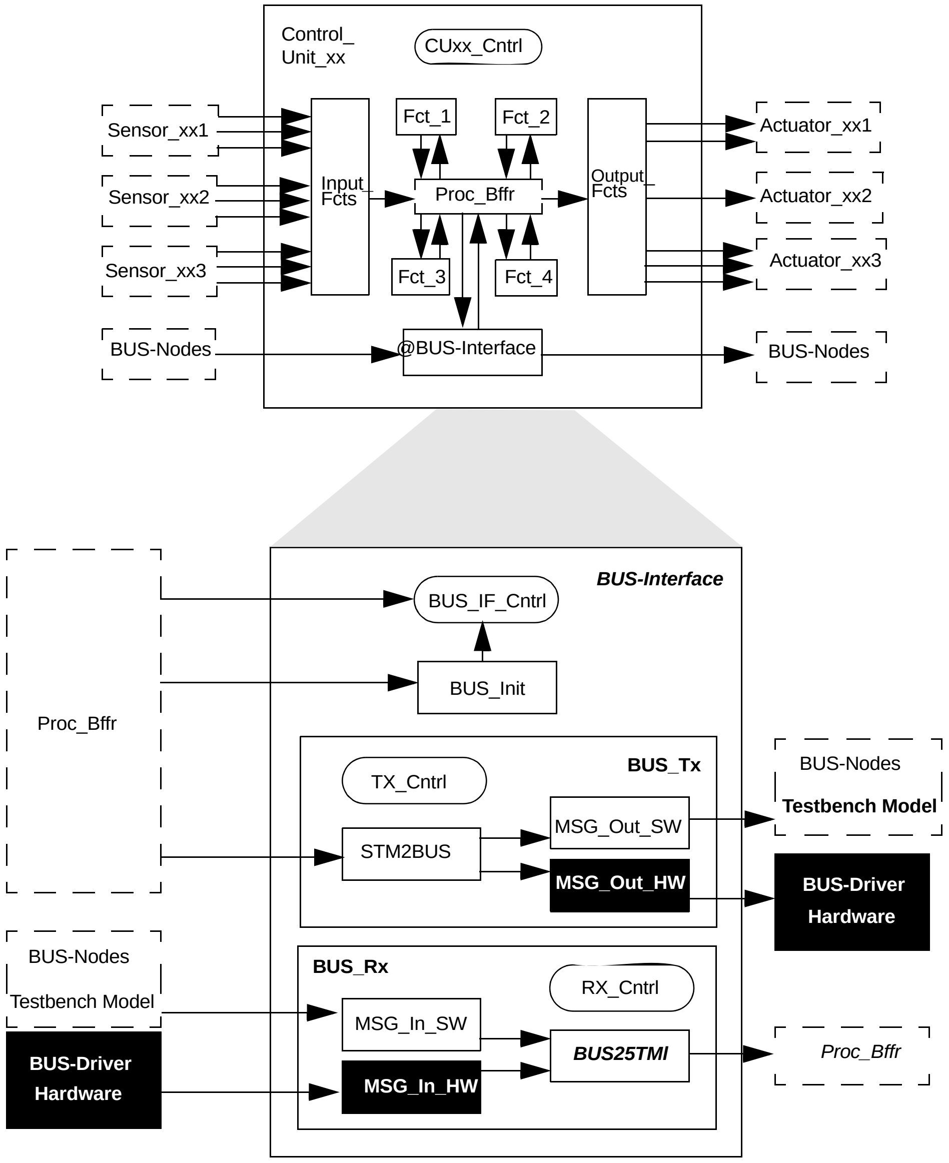


```
tm(en(CHECK_S_H_SIGNAL), DELTAT)
 if HEAT_SIG and not COOL_SIG then
  tr! (HEAT_CMD;fs! (COOL_CMD)
 end if;
if COOL_SIG then
   tr! (COOL_CMD); fs! (HEAT_CMD)
end if
```

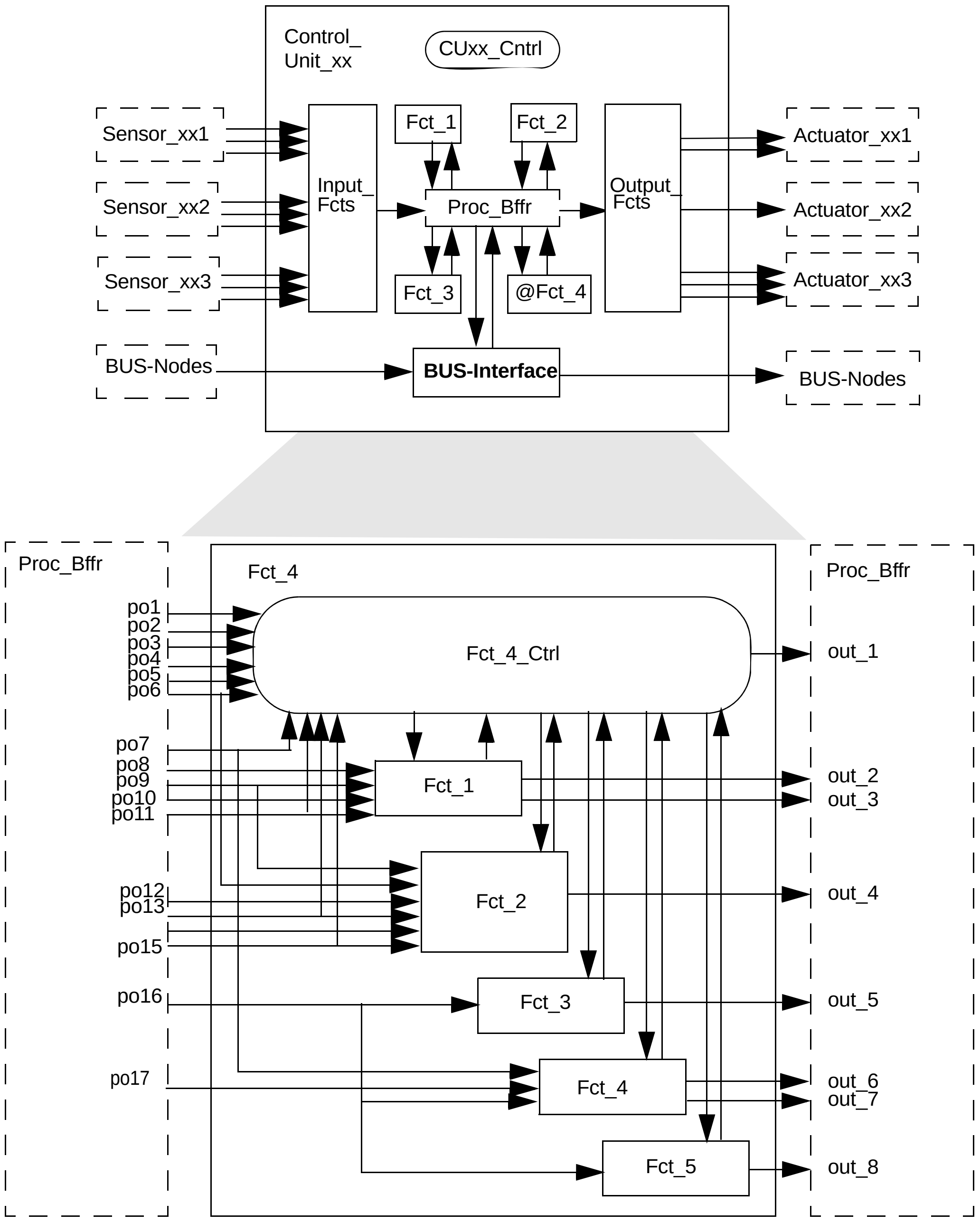
```
fs (HEAT_SIG or COOL_SIG)
tr (HEAT_SIG or COOL_SIG)
(notTHERM_PROTECTN and not PWR_FAILURE)
fs (HEAT_SIG or COOL_SIG) /
when fs (HEAT_SIG) then fs! (HEAT_CMD) and when;
when fs (COOL_SIG) then fs! (COOL_CMD) and when
tr (THERM_PROTECTN) or
tr (PWR_FAILURE)
```

```
INTPOL_2D (Mini-Spec)
ch(x_IN)/
if X_IN<=X_ARRAY(1) then
   Y_OUT:=Y_ARRAY(1)
end if;
if X_IN>=X_ARRAY(N_PTS) then
   Y_OUT:=Y_ARRAY(N_PTS)
end if;
if X_IN>X_ARRAY(1) and
X_IN<X_ARRAY(N_PTS) then
  $1:=1:
  while X_ARRAY($1)<X_IN
  loop
      $1:$1+1
      end loop;
$X2:=X_ARRAY($1);
$X1:=X_ARRAY($1-1);
$Y2:=Y_ARRAY($1);
$Y1:=Y_ARRAY($1-1);
Y_OUT:=$Y1+(X_IN-$X1)
($Y2-$Y1)/
($X2-$X1)
end if
```

Generic Bus Interface



Generic Control Algorithm Approach



Functional Decomposition - System Levels > 2

Proceeding beyond System Level 2, the structure becomes completely dependent on the system and no further generic approach can be applied.

For general guidance, remember the primary consideration is readability. From the Activity-chart, the reader should know what the function is.

Draw all second and subsequent level Activity-charts with all the inputs on the left and outputs on the right.

Structure the chart in functional modules that:

- Later help in code generation.
- Make it readable from the structure and capture the dynamics of the system.

System Validation

A characteristic of the V-Process model is that as we progress down the left leg of the V, tests must be defined at each stage to be used in the later system integration.

What this means in terms of our analysis process is that at every level of the system hierarchy, before proceeding to the next level, the model should be validated by simulation. During this simulation, the stimulus and response can then be recorded. This provides the test definition in terms of applied input and expected output to be used at integration. As part of the analysis, the code generator should be used to produce a C prototype that serves as a functional executable.

Graphical panels should be used to drive/control the simulation. The panel could be a mock up of the real user interface, but panels are also very useful for condensing information and can be used as a system monitor exhibiting values, system objects, modes, etc. Panels also serve as an excellent communications medium from designers to managers, marketing, and customers since they can show the system operation at any level of abstraction. For these reasons, time spent creating clear understandable panels is time well spent.

As all systems are part of a control loop, system validation must be performed closed-loop to ensure accuracy. Closed-loop analysis requires modeling the external environment such that if a change of output affects an input then this is modeled and simulated.

In the early stages of analysis, up to System Level 2, Statemate linear testbench models are sufficiently accurate to model the external environment and close the loop (see the **Closed-Loop Validation by Linear Testbench-Models** figure). The testbench is used to model the essential behavior of the external activities. Testbenches can use concurrency, broadcasting, and global visibility for effective testing of the system under development.

As the model becomes more detailed, generally more accurate non-linear models of the environment are required to accurately verify the system. C Code models, either hand written or generated from other SE tools, are used to model the external environment (refer to **Statemate in a Systems-Engineering Environment**)

To do this, include the C code as a basic activity in a higher level Activity-chart with the system model in the other activity as shown in the **Closed-Loop Validation by a Non-linear Plant-Model (User C-Code)** figure.

Note

In the **Closed-Loop Validation by a Non-linear Plant-Model (User C-Code)** figure, a Non-Linear Plant Model replaces the sensors and actuators in the **Closed-Loop Validation by Linear Testbench-Models** figure.

Using this Bottom-up expansion, and linking C code to a basic activity allows interactive simulation with the closed-loop C code and provides an easy transition to prototyping with generated code.

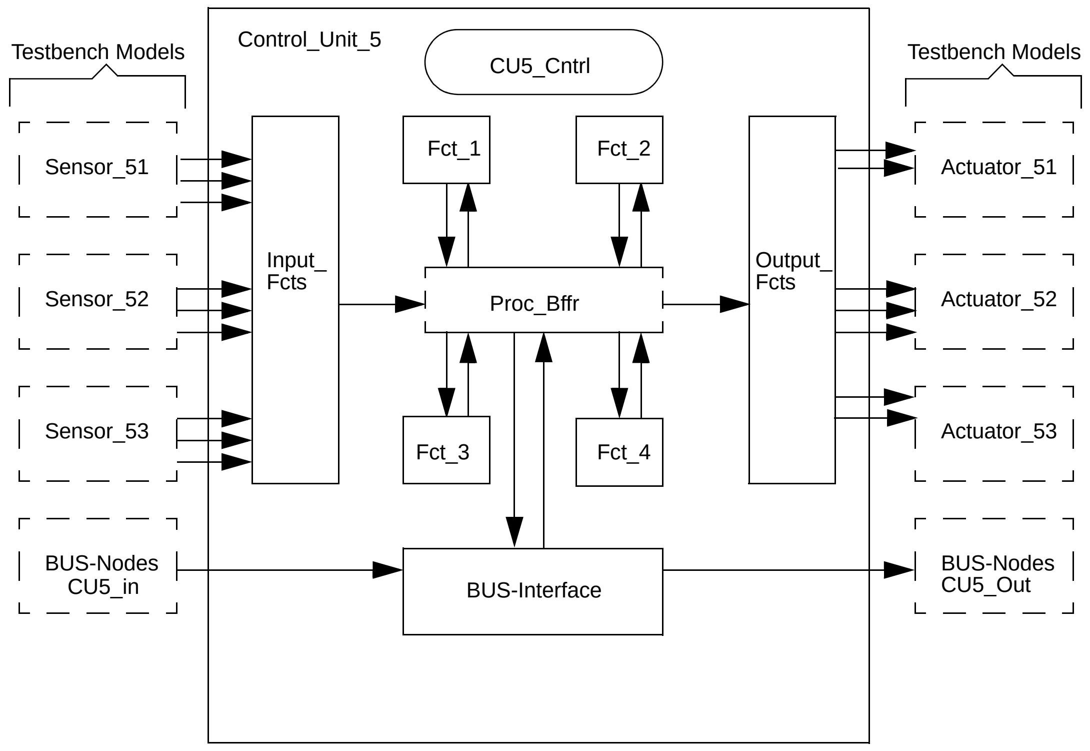
When creating these models of the external environment, be careful not to fall into the trap of spending vast resources on creating these models and forever refining them to be more detailed. The aim is to build confidence in the system model to the point where we are confidant that in switching to HW-in-the-loop analysis there will not be a catastrophic system failure. Once this confidence has been established the switch should be made to HW-in-the-loop validation.

Using the generic approach of the bus interface (refer to **Functional Decomposition - System Levels > 2**), the system model can be validated at an advanced state of system definition by connecting into the network and evaluating the algorithm in its intended environment. The product can be operated with this prototype and final optimizations and testing can be performed. This HW-in-the-loop analysis is really the ultimate in verification and provides absolute confidence that the specification is correct and the system will work when built.

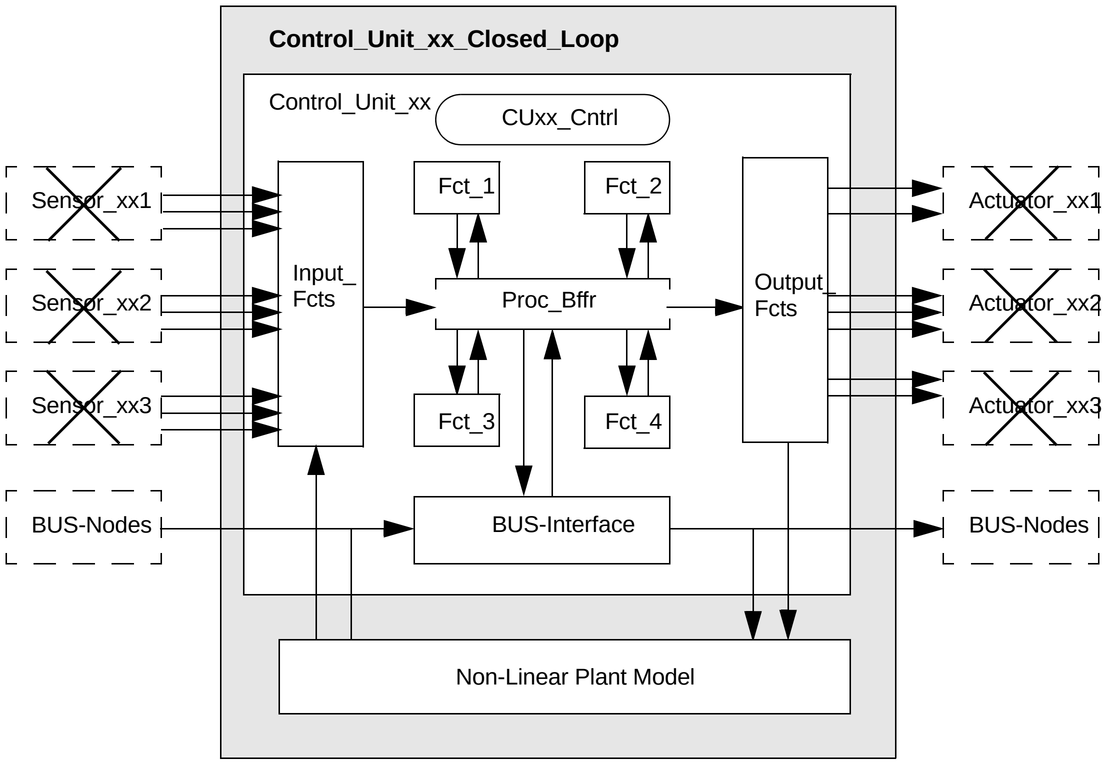
Because the Statemate testbench is not part of the system specification, concurrency and broadcast can be freely used. In fact, the broadcast mechanism gives testbenches the capability to see down into the model and monitor or change any variable. This is a very powerful capability and means testbenches can be used for a variety of different applications such as:

- Failure Analysis (FMEA)
- Statistical Analysis
- Test Generation
- Initializing Variables

Closed-Loop Validation by Linear Testbench-Models



Closed-Loop Validation by a Non-linear Plant-Model (User C-Code)



Summary

We have modeled the system’s:

- Functionality
- Architecture
- Behavior
- Implementing details.

We have verified and validated the design at each level of decomposition. We have also created test scenarios for analysis, and tested our design through hardware-in-the-loop analysis.

At this point, we should have both a formal specification and design document in addition to the executable model from which software engineers can implement the target application.

Style Guidelines for Activity-Charts

In functional specifications, the Activity-chart is the most important view in communicating what the system does. Hence, readability is of paramount importance. The Activity-chart provides the visualization of internal and external functions and their interconnections via information flows.

This section presents guidelines that have been developed through years of experience on many projects and have proved to significantly enhance the readability of specifications developed using Statemate.

Graphical Settings/Drawing Preferences

To enhance readability and get the maximum size working area to allow for later expansion, stretch the window so the Activity-chart fills the entire screen.

- **General Settings**
  - Grid **ON** with 0.25 spacing and 10-pixel Trap Radius
  - Full screen (for later expansions)
- **Flow-Lines**: Straight Lines:

  Standardize the size of Function Blocks, Routers, Data Stores, External Activities, and Control Activities. As a reference use:

  - Function Blocks and External Activities:
    - Box Height 2
    - Box Width 3.5
  - Data Stores should be slightly thinner.
  - Control Activities should be double the width.
- **Standardize on the font used for function names and flow labels**:
  - Names: Black Courier **`12 point bold`**

Note

Labels: Black Courier *`12 point medium italic`*
A preferences file with all of the above settings enforced is supplied as `STYLE_GUIDE.pref`.

Drawing Data-Sources/Sinks (External Activities)

- **Shape of External Activities:**
  - All external activities should have the same size in height and width (refer to section 2.1).
  - If a bigger size is needed, then the relevant size should be a multiple of the pre-defined ‘basic’ height.

    Do not change the width. Long names should be broken over two or more lines.
- **Locating External Activities in a Chart:**

  Data sources are placed on the left side of the Activity- chart; data sinks are placed on the right side of the Activity-chart.

Drawing Routers

- **Shape of Router:**
  - All routers should have the same size height and width.
  - If a bigger size is needed, then the relevant size should be a multiple of the predefined ‘basic’ height.

  Note:

  Do not change the width.

  Long names should be broken over two or more lines.
- **Locating Routers in a Chart:**
  - Data source routers are placed on the left side of the Activity-chart.
  - Data sink routers are placed on the right side of the Activity-chart.

Drawing Activities

- **Number of Function Blocks/Chart**:
    - There is no magic figure for the number of function blocks/chart. The priority is readability and this requires a clear layout of activities and associated connectivity. Therefore, use on-page hierarchy in an Activity-chart only when readability is maintained.
    - Top-Level Chart (System Level 1):

      Due to the generic approach, the number of activities is limited by the main system functions. No further hierarchy decomposition should be done on this chart.
    - System Level 2:

      Refer to **Functional Decomposition - System Levels > 2.**
- **Shape of Function Blocks**:
    - All function blocks should have the same size in height and width (refer to **Graphical Settings/Drawing Preferences**).
    - If a bigger size is needed, then the relevant size should be a multiple of the pre-defined 'basic' height or width.
- **Locating Activities in a Chart**:

  Function blocks should be placed one below the other. Arrange the activities so that they illustrate the dynamics of the system as you read from top-to-bottom and left-to-right.
- **Naming Activities**:

  The activity names should be verb/adverb clauses that tell you what the function does. Long names should be broken into two or more lines.

Drawing Control Activities

- **Shape of Control Activities**:
  - All control activities should have the same size in height and width (refer to Graphical Settings/Drawing Preferences).
  - If a bigger size is needed, then the relevant size should be a multiple of the pre-defined 'basic' height or width.
- **Locating Control Activities in a Chart**:

  Control activities should be located at the top of the respective hierarchy.
- **Naming Control Activities**:

  A control activity should have the name of the chart or the name of the hierarchy with the ending '`_CTRL`' (= Control).

Data/ Control Flows, Information Flows

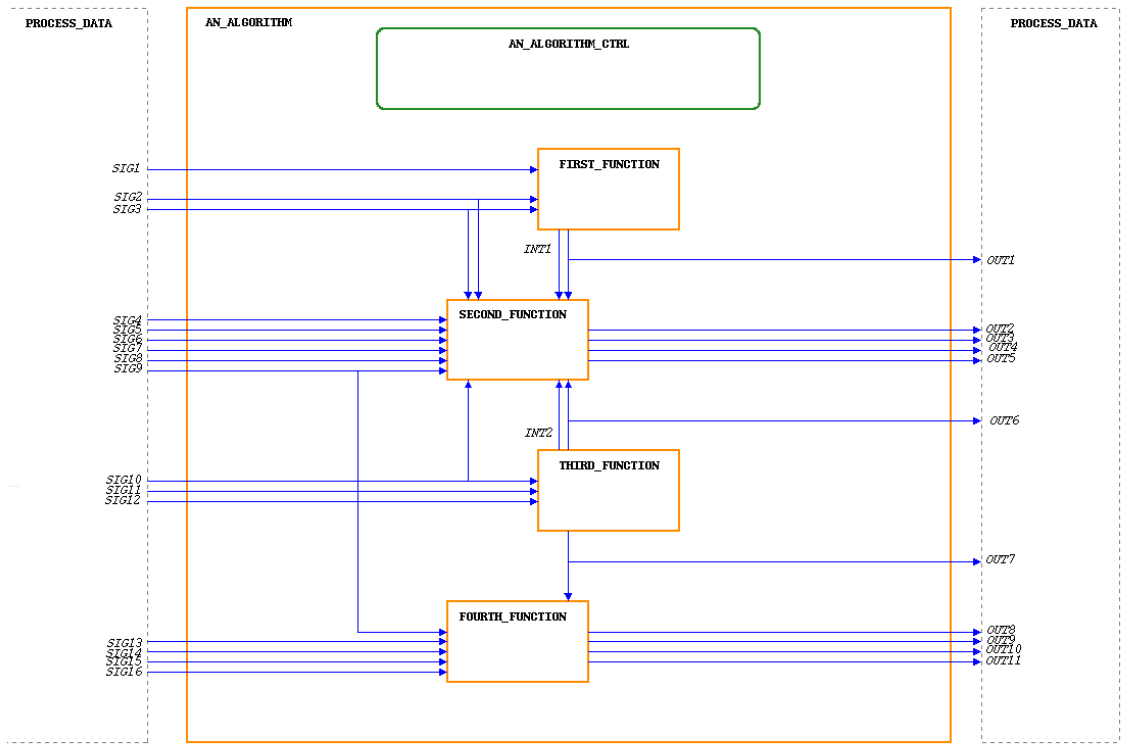
Graphically, there is no difference between data and control flows; solid lines are used for both.

Note

From external data sources or to external data sinks, draw data/control flows explicitly with flow-labels positioned in the external activities ("Pin-Labels"). There should only be one label at the source/sink for each signal.

Internal communication between function blocks is visualized by Information Flows. Thus there are no multiple flows internally between activities. The exception is when you have a flow with a fork (see the Internal and External Flow Lines figure.

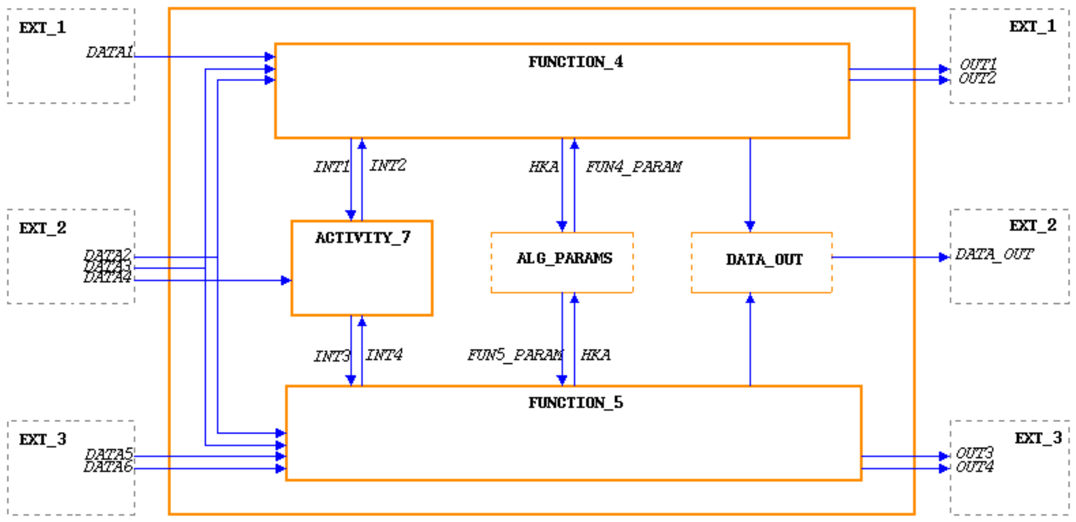
Internal and External Flow Lines



Joints/Forks

When data/control flows or information flows cross within a chart, only a “T” is a joint or fork while a “+” is simply a crossing line with no connection (see the Joints and Forks figure).

Joints and Forks



Data Stores

- **Shape of Data Stores**:
  - All data-stores should have the same size in height and width (refer to Graphical Settings/Drawing Preferences) For readability reasons the shape should be different from the shape of function blocks.
  - If a bigger size is needed, then the relevant size should be a multiple of the pre-defined ‘basic’ height or width.
- **Interpretation of Data Stores**:
  - *Normal data-store* as known in Structured Analysis (see ALG_PARAMS in the Joints and Forks figure).
  - *Junction* of information that is given by different function blocks within different time steps (see DATA_OUT in the Joints and Forks figure).
- **Naming convention for flows to/from Data-Stores**:

  Unlabeled flows to/from data-stores assume the name of the data-store.

In-Page and Off-Page Connectors

- In-Page-Connectors should only be used when the readability of the Activity-chart is disturbed by a direct connection.
- Corresponding In-Page-Connectors should be aligned horizontally or vertically.
- Off-Page Connectors are not recommended, but may be necessary in certain situations.

Mini-Specs

For readability and debugging purposes, it is recommended that you use mini-specs that are no longer than one screen-page. If an existing validated C program can replace the mini-spec, then you can use that program instead.

Style Guidelines for Statecharts

Readability is the most important criteria with respect to all models created in Statemate. This section presents guidelines that enhance the readability of Statecharts.

Graphical Settings/Drawing Preferences

To enhance readability and get the maximum size working area to allow for later expansion, stretch the window so the Statechart fills the entire screen.

- **General Settings:**
    - Grid **ON** with 0.25 spacing and 10-pixel Trap Radius
    - Name font: Black Courier **`12 point bold`**
    - Label font: Black Courier *`12 point medium italic`*
    - Transition color: OrangeRed
    - State color: ForestGreen
- **Standardize the size of states**:

    Box Height 1.0

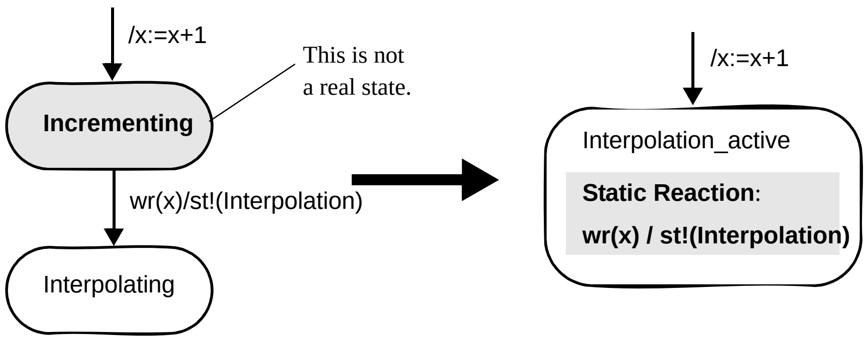
    Box Width 3.5

    A preferences file with all of the above settings enforced is supplied as `STYLE_GUIDE.pref`.

Drawing States

Check carefully that the identified “states” are really states. An error, which is often observed, is that artificial states are introduced to get a simulation-step within Statemate’s execution semantics (see Incrementing in the**True States** figure).

True States



True states are characterized by the fact you spend time in them; either you are waiting for a signal change or there is a function active while in the state.

- **Number of States/Chart:**
  - Statecharts describing the control activity’s hierarchy of System Levels 1 and 2 (refer to **Methodology**) should be structured in a specific way:

    Start with only the major stable states and, if there are any, the transition-states between these states.

    Refine the transition states only when readability is maintained, otherwise declutter them.
- **Shape of States:**
  - All states within a Statechart should have the same size in height and width (Refer to **Graphical Settings/Drawing Preferences**). If a bigger size is needed, then the relevant size should be a multiple of the pre-defined ‘basic’ height or width.

- **Naming States:**
  - Avoid dummy names like **`Idle`** or **`Wait`**.

    Use more meaningful names like **`Wait_for_xxx`**.
  - If a state is connected to a function `entering / st!(Function_xx)`, the name of the state should reflect this: `Function_xx_active` (see the **True States** figure).
  - Break long names into two or more lines.
- **Static Reactions**:
  - Use static reactions only for Statemate-specific actions during a transition (refer to **Labeling Transitions**) or for Statemate-Simulation-specific features as shown in the **True States**figure.
  - Use explicit `entering / st!(Function_xx), exiting / sp!(Function_xx)` rather than `Within / Throughout` syntax for flexibility (see the **True States** figure).

Hierarchy

There are two types of hierarchies:

- Hierarchies due to structure—a state is structured into sub-states
- Hierarchies for visualization of different levels of interrupts

To visualize the different types of hierarchies, only those describing a state should be named.

Concurrent States

- Concurrent states are typically only used in a Functional Model. If the Functional Model is changed to an Implementation Model (refer to **Methodology**), concurrency should not be used.
- For readability, concurrent states should be restricted to **one** hierarchy level (no decluttering).

  If the concurrent state becomes more complex, it should be split into an additional function block.
- Broadcasting should only be used in combination with Testbenches (refer to **System Validation**).

Transitions

Drawing Transitions

- **Rule**:
  - Draw transitions between states as straight lines.
  - Within a Statechart, the transitions should have a common sense of rotation (either clockwise or counter-clockwise).
- Transitions should never cross each other, states, or hierarchies.

Joints / Forks

In principle, do not use forks and joints due to the restricted use of concurrency.

Labeling Transitions

- For readability reasons ("pattern recognition"), standardize the positioning of transition labels.

  In case of a clockwise orientation of the transitions:
  - Transition directing from left to right should have their labels on top.
  - Transitions directing from right to left should have their labels on bottom.
  - Transitions directing vertically down should have their labels to the right.
- Transitions directing vertically up should have their labels to the left.

  Label Syntax:
  - Put actions on the transition so they can be read directly from the chart. If the label is too long, either break it into several lines or use a macro with a meaningful name.
  - Avoid Statemate-specific semantics on transitions!

    Statemate-specific semantics (see the **True States** figure) should be hidden in static reactions.

Default Entries

For readability reasons, minimize the number of default entries. Not all hierarchy levels need a Default Entry and it is often possible to draw a single default through the hierarchy to the lowest level state.

In-Page and Off-Page Connectors

- Use in-page connectors only when the readability of the Statechart is degraded by a direct connection.
- Restrict the number of in-page connectors to a minimum.
- Positioning the In-page Connectors:

  For readability reasons, align corresponding in-page connectors horizontally or vertically.
- When using off-page connectors, pay special attention to the layout so that when the parent and decluttered chart are printed, the connectors are aligned between the charts.
- Off-page connectors should have meaningful names to help establish the connectivity through the hierarchy.

Decluttering

- Use automatic decluttering only when there is only one transition in and one transition out of the state to be decluttered.
- Automatic decluttering of states with multiple in/out-transitions results in an unstructured layout of off-page connectors in the parent chart and decluttered chart. Therefore, it is recommended to declutter by hand to improve the readability.