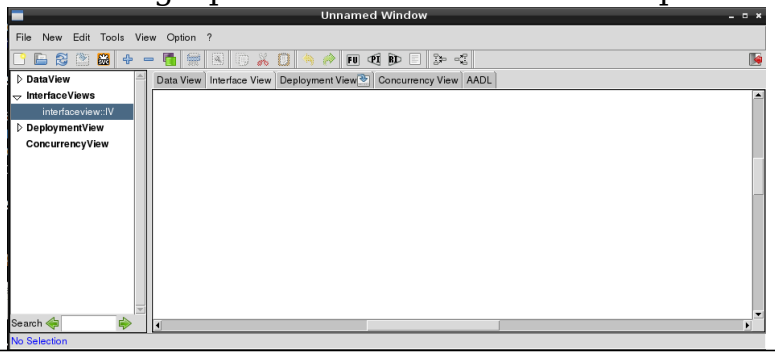
TASTE V2 Reference Card

How to quickly build a system using TASTE
Version 1.6 (03/07/2019)

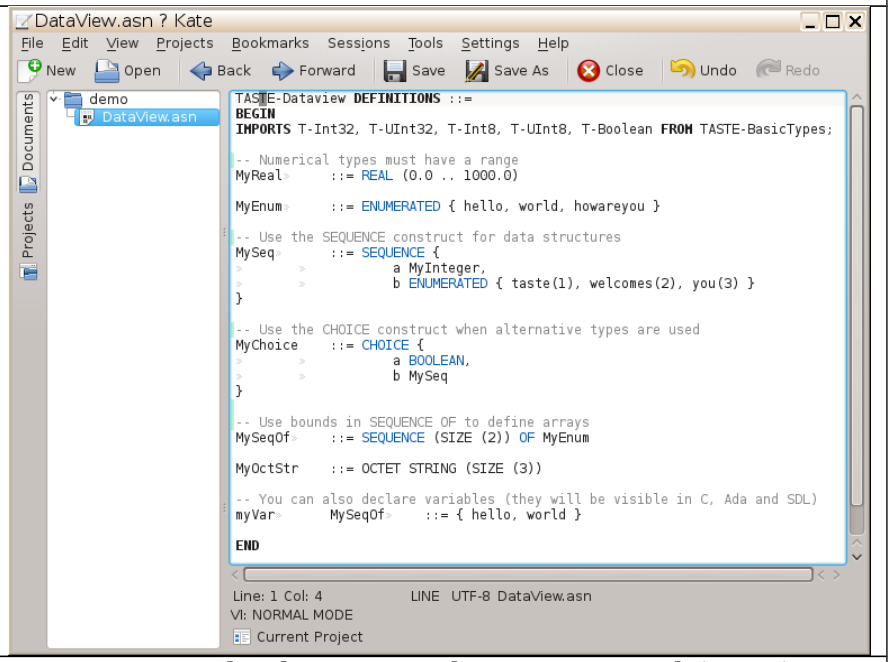
IMPORTANT - *Always make sure you are using the latest version of the TASTE tools. From within the TASTE Virtual machine, you can click on the Update-TASTE icon. From a terminal, you can run the Update-TASTE.sh script from the ~/tool-src folder, and open a new terminal when it is done.*

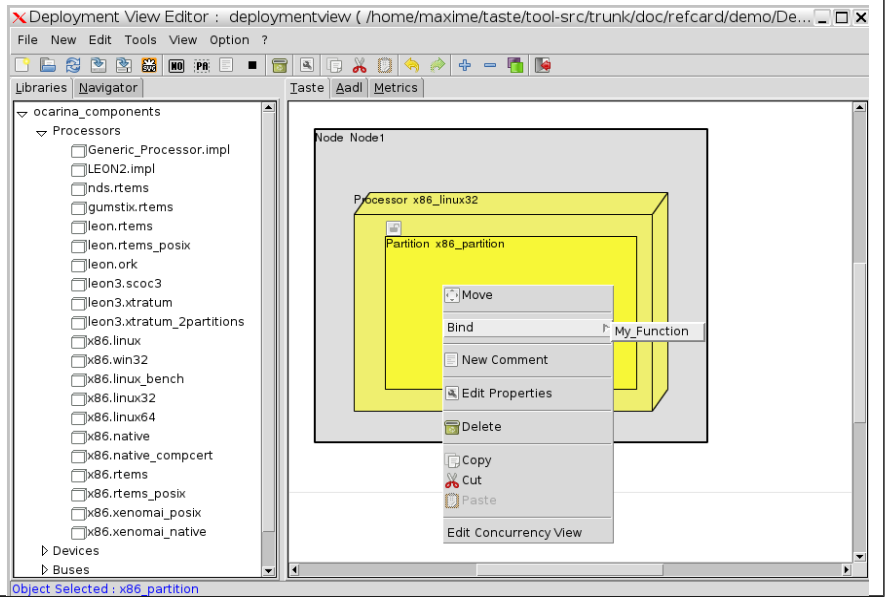
STEP-BY-STEP TUTORIAL

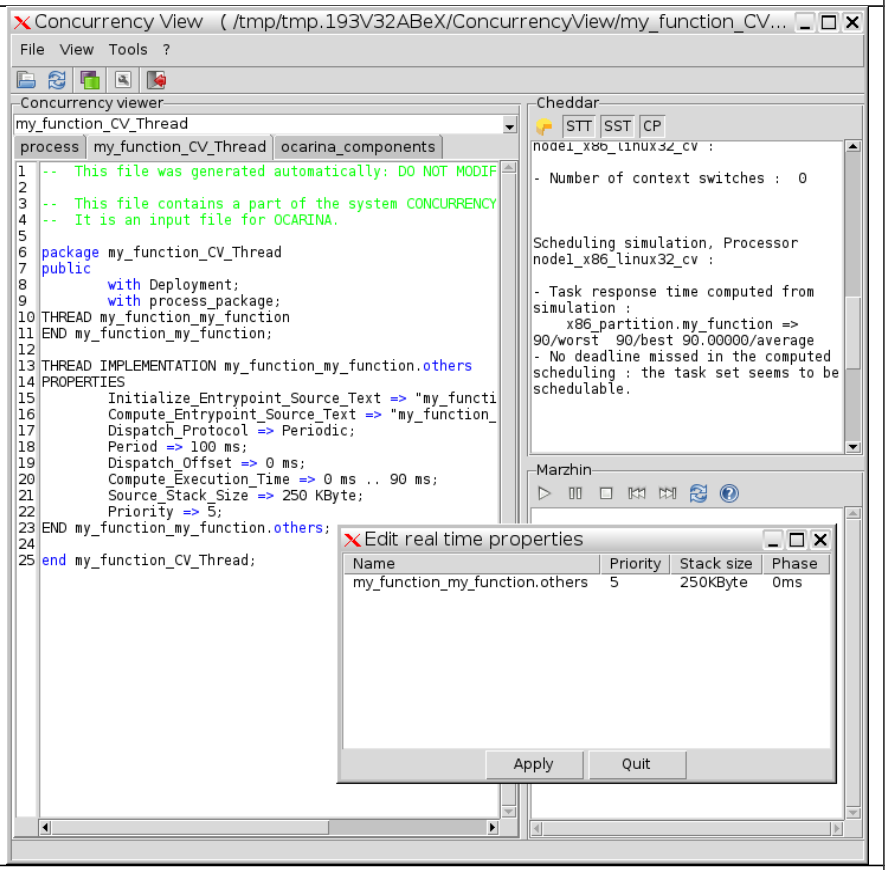
Step	***Actions***	***Comments***

Create a new project	Create a new working directory and start the TASTE editor. Most of the work can be done from within this single tool.
	
	`$ taste-create-project`
	
	`And further edits can be done with:`
	
	`$ taste-edit-project`
		The graphical AADL editor shows up:
		 

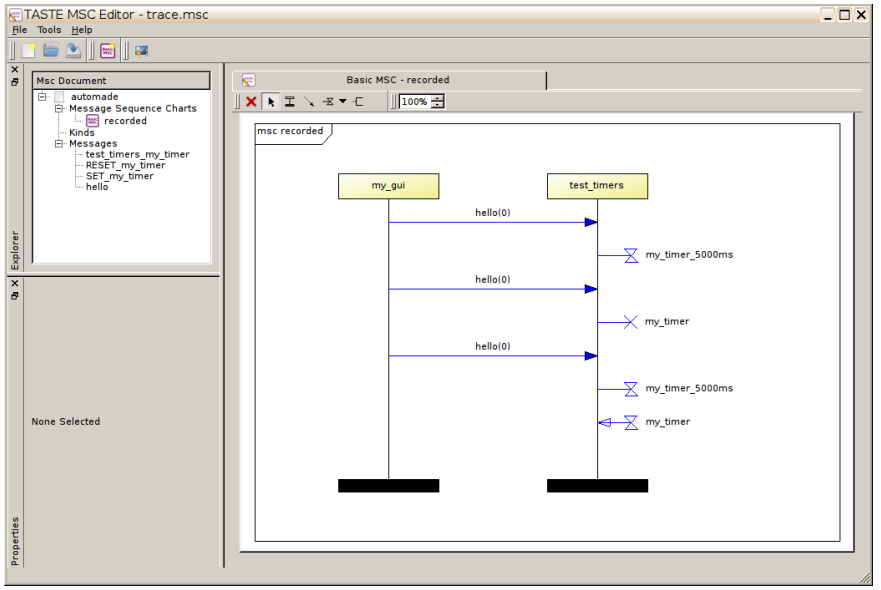
Add functions and containers	In the editor, **right-click** to open the contextual menu Add functions and specify for each of them: - Their **name** - Their **interface** (provided and required) - Their implementation **language** - Their **description** - Their **context parameters** (if any) With the mouse, you can click on a required interface and connect it to the provided interface of another function.	Context parameters allow to specify: - Typed static data (usable in the functional code) - Timers - Compilation flags - Context-dependent data that can be processed during the build, such as reference to some external initialization parameters, etc. Provided interface can carry parameters. You can use the default data types (UInt32, Boolean, etc) or better, create your own types (see step below)

Step	Actions	Comments
Specify data types	The Dataview editor is available from the GUI in the tab named “DataView”. You can create and modify the datatypes used in your system. There can be more than one ASN.1 file used in the project. You can also edit the ACN files, if you need to specify custom memory layouts for the data. If you prefer using the command line, you can edit your dataview with this command: `$ taste-edit-data-view`	
Edit the functional code or models	On the main diagram, right-click on a function to open the contextual menu. Depending on the implementation language you chose for the function, select the relevant editor (“Edit Ada code”,”Open SDL editor”, etc.) If you want to work with your own external tools (e.g. Simulink or Pragmadev Studio) you have to generate the code skeletons first using the menu option *Tools->Generate code skeletons*.	For C and Ada a text editor is opened (Kate). For SDL the OpenGEODE tool allows to create graphical state machines and generate code. For all supported languages a model (or code) skeleton is automatically generated, ensuring consistency of the interfaces in the complete system.

Step	***Actions***	***Comments***
Create deployment view	The tab named “DeploymentView” allows you to specify the nodes of your system and map the software functions from the Interface View onto them.	On the left side of the editor, you can select processor boards, busses, and drivers. Drag and drop what you need to the diagram. On the *partition*, right click and select the functions you want to bind to the chosen processor. The name of a partition is the name of the target application that will be generated. 

Step	***Actions***	***Comments***
(Optional) Tune the real-time attributes of your system	Select the tab named “ConcurrencyView” if you want to know more about the threads that will be created at runtime. You can edit some properties for fine tuning of the application: - Thread priority - Stack size per thread - Phase (or offset) You can also run the Cheddar and the Marzhin analysis tools that are built-in, to check scheduling analysis of your system (*prototype tools).* For these functions to work you must have specified the worst case execution time of each provided interface of your system.	
Build the system	From the GUI you can build your system from the *Tools->Build the system* option. Another window will show you the build progress and report errors if any.	**IMPORTANT** Between two builds you may want to use the option *Tools->Cleanup output (binary) directory*. It can happen that files from a previous build pollute the next build in some situation. If you do not clean-up, a subsequent build will be done much faster as only the modified data will be recompiled.

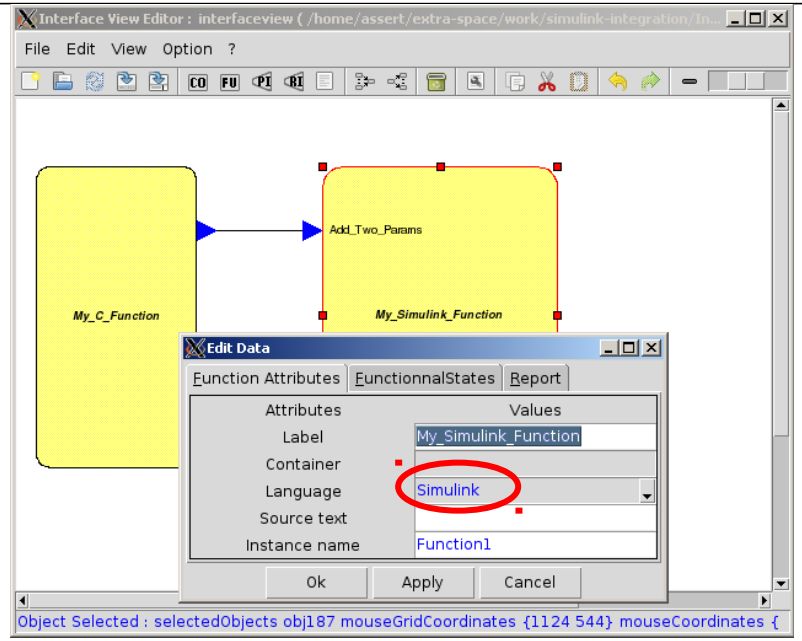
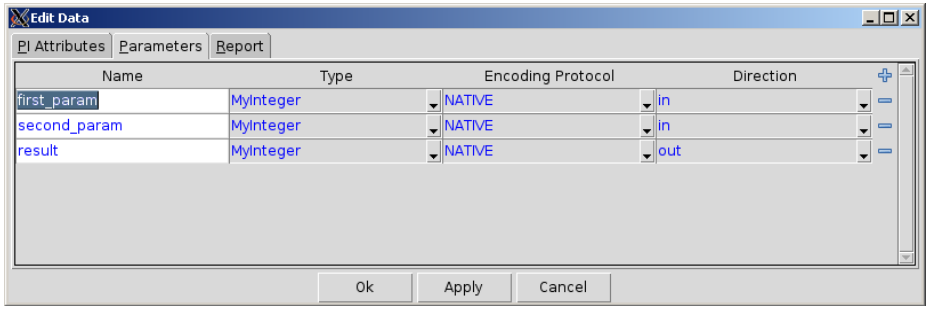
Step	Actions	Comments

Run the system and interact with it	When the build is done, you can quit the editor and explore the directory where the generated application was created.
	
	`$ cd binary.c/binaries`
	
	If your system contains GUI components, a binary per GUI is placed in that same directory.
	
	You can either run your applications directly (on the chosen platform) or activate tracing function:
	
	`$ taste-run-and-trace ./my_demo`
	
	At the end of the execution (stop it with Ctrl-C) a file trace.msc will appear. Open it with the MSC editor:
	
	`$ msce.py –o trace.msc`
		The tracing tool records all the internal communication between your functions, as well as the timers.
		
		 

FOR MORE INFORMATION – *Check the TASTE wiki here: http://taste.tools*
You will learn more about the SDL editor, the use of timers, the use of Python scripts to test your system, and the use of SQL databases in combination with your ASN.1 data model.

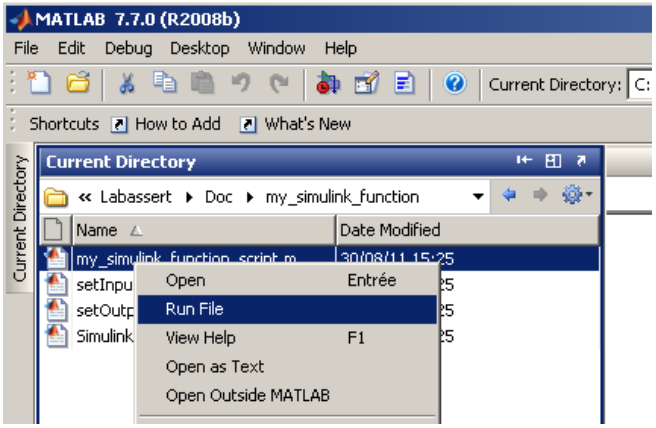
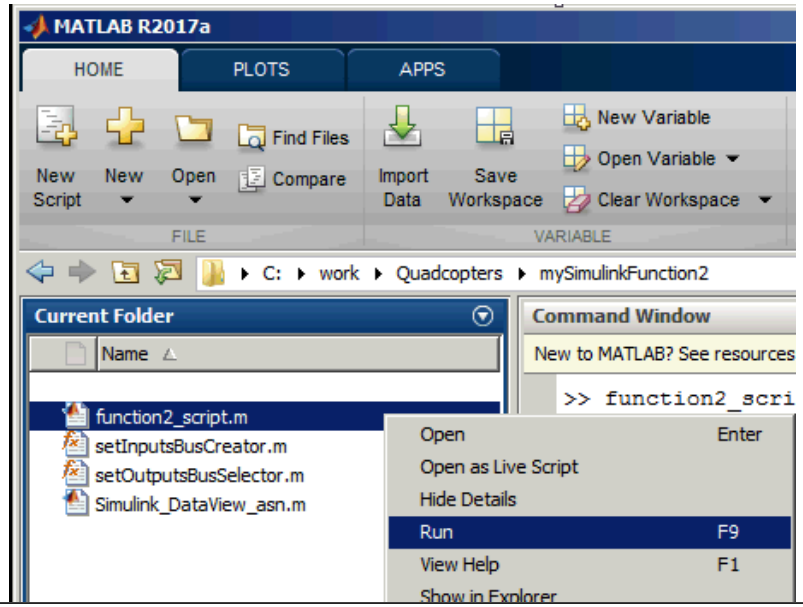
TASTE V2 Quick Reference Card
Integration of a Simulink block as part of a TASTE system

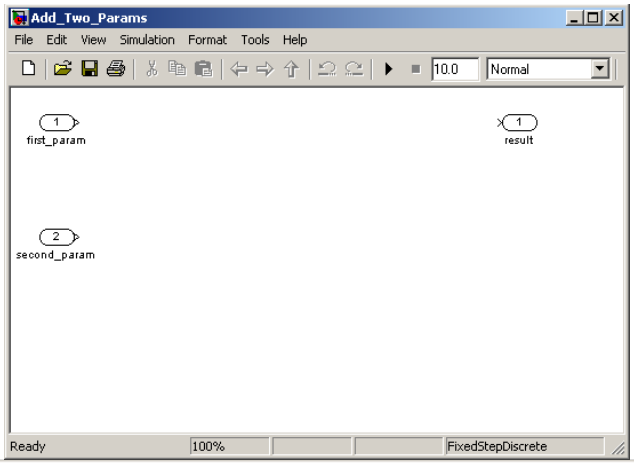
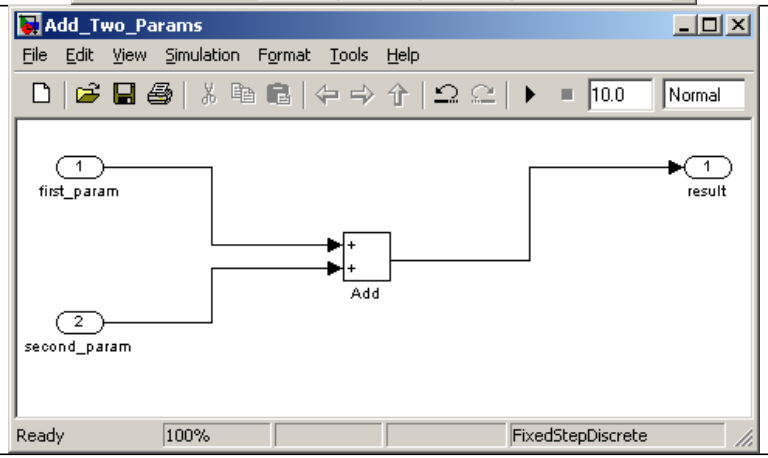
Tested with Simulink R2008B and R2017A

Action	*Illustration*	*Notes*
Create an interface view to capture the Simulink function	 	The *Simulink* block must have only ONE provided interface and NO required interface. The provided interface can contain any number of INPUT and OUTPUT parameters using the ASN.1 types. The provided interface must be set either as PROTECTED or UNPROTECTED.

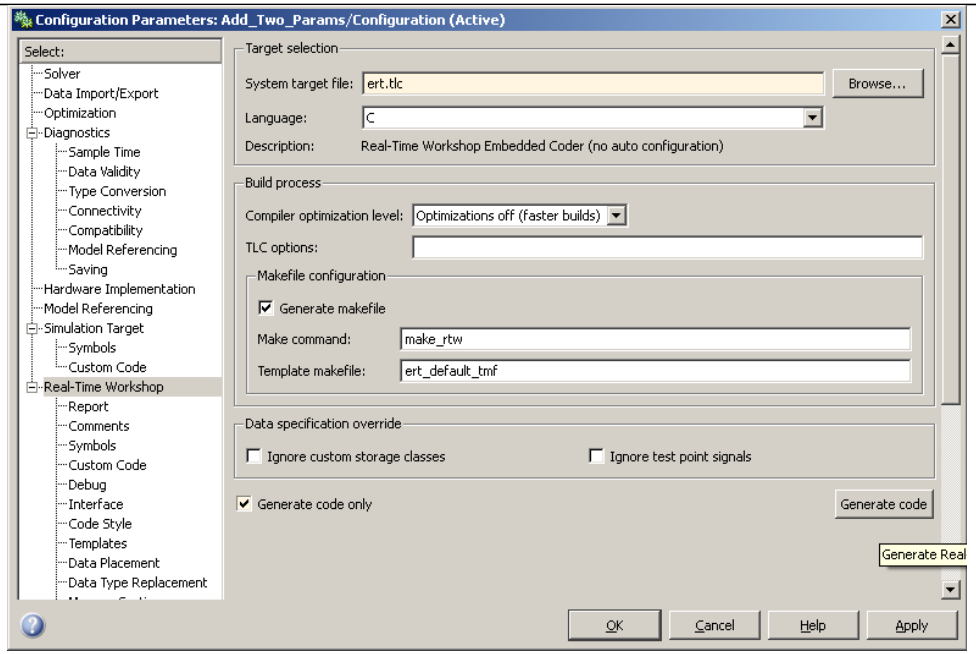
Action	*Illustration*	*Notes*

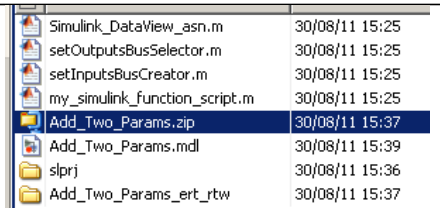
Generate the function skeleton	Right-click on the diagram and select the option *Generate code skeletons.*	A new directory is create with the following files:
		
		`my_simulink_function/`
		`|-- Simulink_DataView_asn.m`
		`|-- my_simulink_function_script.m`
		`|-- setInputsBusCreator.m`
		`` `-- setOutputsBusSelector.m ``

Start Matlab and run the script generated by TASTE	Inside Matlab right-click on "`my_simulink_function_script.m`" and choose *Run File:*  It is similar in recent versions of Simulink: 	This script calls the other ones. The Matlab workspace will be updated with new data types and busses, that result from the ASN.1 to Matlab type conversion. After a short while a new file will appear next to the Matlab scripts: `Add_Two_Params.mdl` (this is the name of the interface we gave as an example). *Important Notes*: (1) On recent versions of Simulink, the file extension will be .slx and not .mdl (2) If you close Simulink and want to reload the model, you must populate again the Matlab workspace by executing the TASTE-generated script named `Simulink_DataView_asn.m`

Action	*Illustration*	*Notes*
Open the mdl-generated file	Double-click on Add_Two_Params.[mdl/slx] to open the Simulink editor: 	What you see is the skeleton of the function you specified in the TASTE interface view.
Fill the function by connecting the input and the output of the block. You can use blocks from the Simulink library.		
Generate the code from the Simulink model	Usually this is straightforward. Go to the menu *Tools->Real-Time Workshop->Options* then tick the *Generate code only* option and click on *Generate code.* (Check below if you are using a version of Simulink more recent than 2012a)	This might take a while, you can follow the progress on the main Matlab console. When it is done, the following file appears in your working directory: *Add_Two_Params.zip*

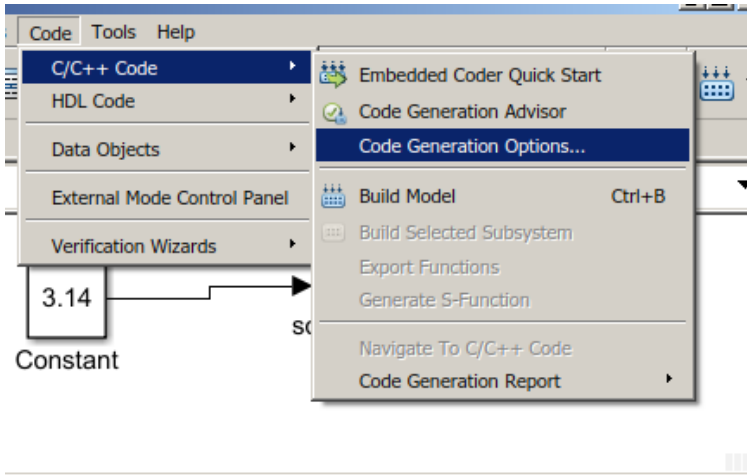
Action	*Illustration*	*Notes*



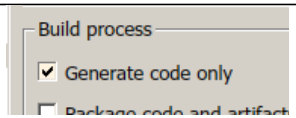
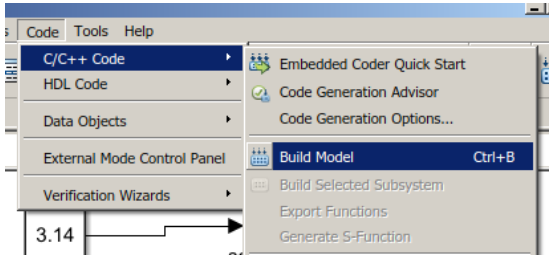


2017 Update:

In recent versions of Simulink, the menu entry is different, and the code generator is not named “Real-Time Workshop” any longer.



Then the same option appears:

Action	*Illustration*	*Notes*
	 To generate the code and get the *zip* file, go back to the menu item: 	

Copy and unpack the generated code back to TASTE working folder	If Matlab was not installed in your TASTE Virtual machine and you had to copy the .m scripts to a different machine, copy back the generated zipfile to your TASTE working folder and unzip it.
	
	`cd my_simulink_function`
	`unzip Add_Two_Params.zip`
		A *lot* of files may appear. The reason is that Simulink copied in the zipfile ALL files required to make an independent compilation of the project (which is what TASTE needs).

Call the Simulink block from another TASTE function	As an example you can add a periodic interface to a function you may call “`My_C_Function`” (implemented in C) Calling the Simulink block is like invoking any other required interface. The call is synchronous, which means you get the result “immediately”.	

<table>
<tr><th>Action</th><th>Illustration</th><th>Notes</th></tr>
<tr><td></td><td>assert@assertvm: ~/extra-space/work/simulink-integration/my_c_function
<pre>
/* Functions to be filled by the user (never overwritten by buildsupport tool) */

#include "my_c_function.h"
#include <stdio.h>

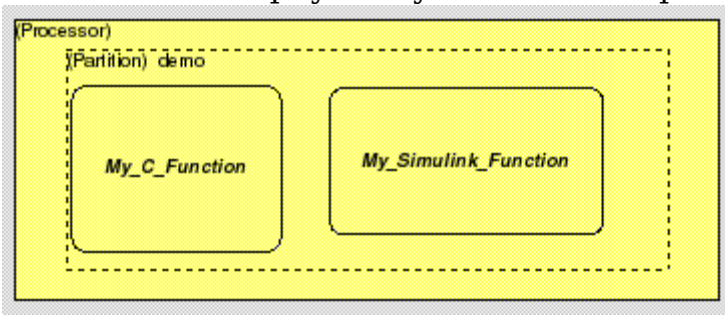
void my_c_function_startup()
{
        /* Write your initialization code here,
           but do not make any call to a required interface!! */
}

void my_c_function_PI_pulse()
{
    /* Write your code here! */

    static asn1SccMyInteger a = 0,
                            b = 0;
    asn1SccMyInteger res = 0;

    my_c_function_RI_Add_Two_Params (&a, &b, &res);

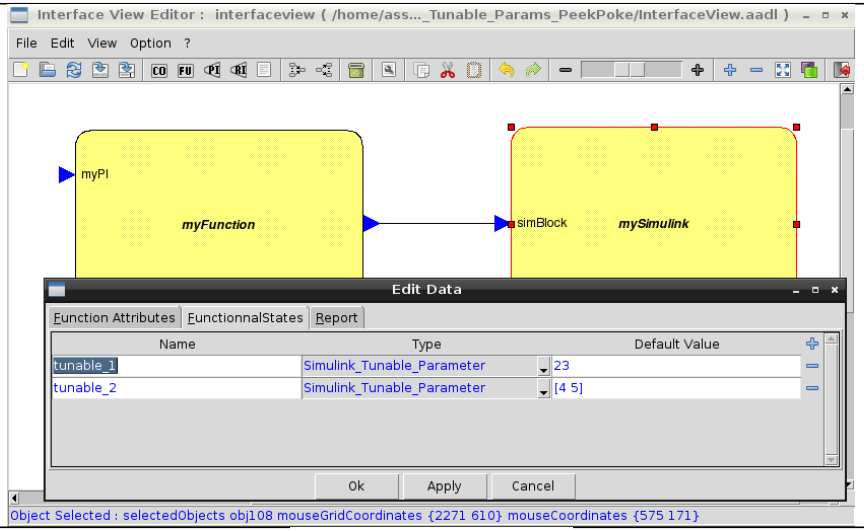
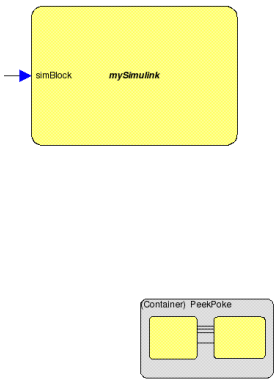
    printf ("Result: %lld + %lld = %lld\n", a, b, res);

    a++;
    b++;
}
                                                    26,1   Top
</pre></td><td></td></tr>
<tr><td>Build the system and run it</td><td>Create a deployment view – do not forget to put both functions in the SAME partition (synchronous functions cannot reside in a physically different computer)
 

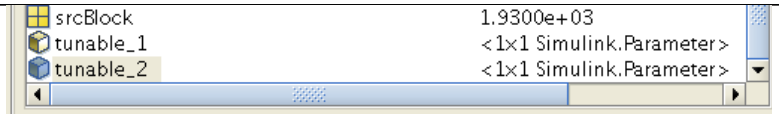
Then run <code>./build-script.sh</code></td><td></td></tr>
</table>

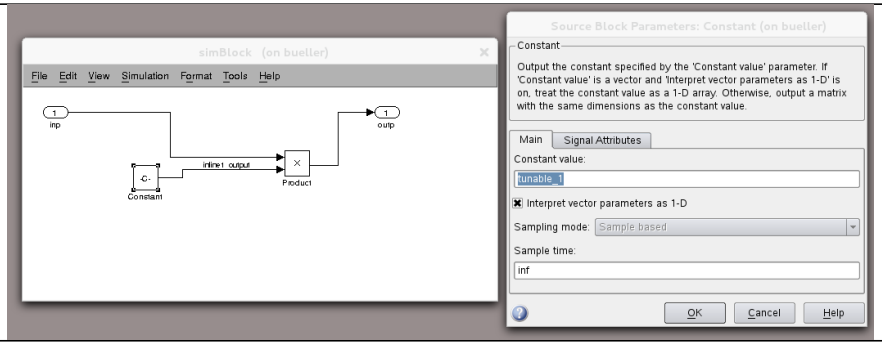
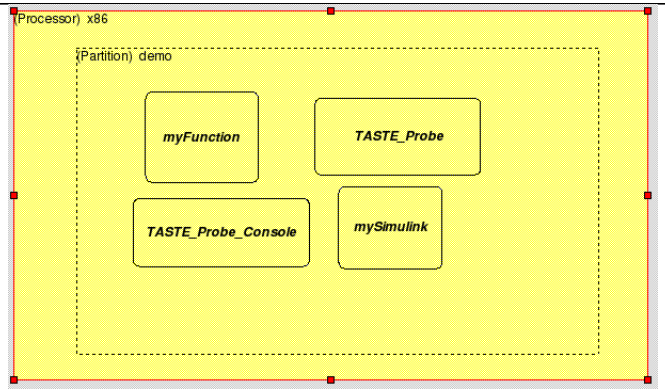
TASTE V2 Quick Reference Card
Using Simulink *Tunable Parameters* and TASTE *PeekPoke* functionality

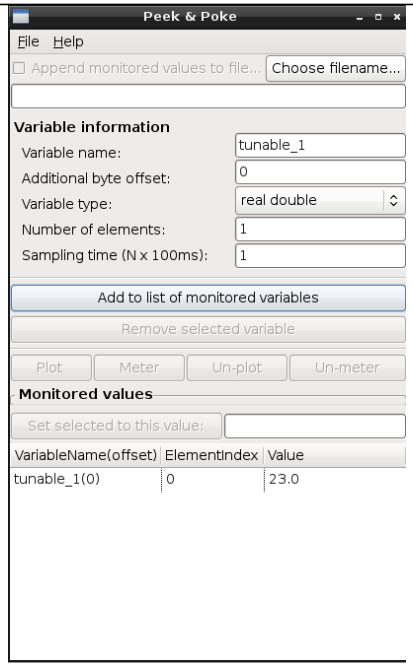
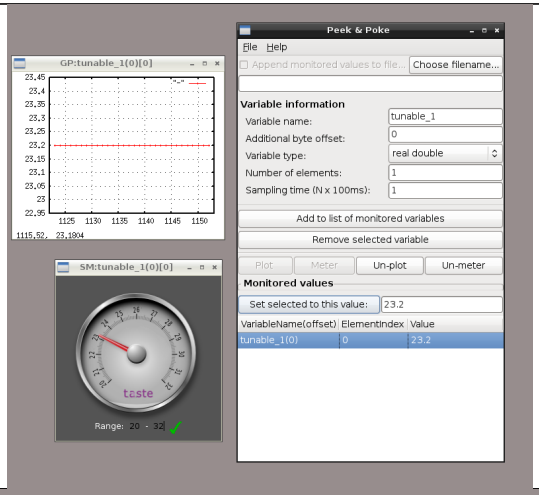
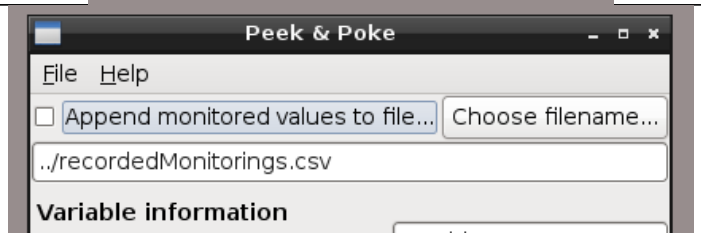
Checkout demo in ~/tool-src/testSuites/Regression_AADLv2/Demo_Tunable_Params_PeekPoke
This tutorial explains how to import the special PeekPoke component to a TASTE system. The PeekPoke component allows to monitor and change parameters of any function of the system without having to add dedicated interfaces. It can be used to tunealgorithms or to check the evolution of any global variable of the system at runtime (it can plot and record data).

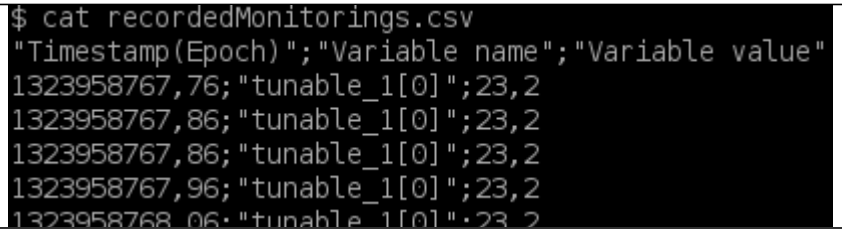
Action	*Illustration*	*Notes*
Create a system that contains a Simulink block, and click on the "Functional States" tab in the Simulink function block Enter variable names, choose "`Simulink_Tunable_Parameter`" type, and set a value. Click on OK when you are done.		Values have to be numerical (integer or real). They can be multi-dimensional as shown in the screenshot.
Right click and select "Import". Choose file "`export_PeekPoke.aadl`" in directory `/home/assert/tool-inst/share/peekpoke/component` and click on Open You can save and close the interface view editor.		A small container with two functions will appear in the lower right hand corner of the interface view.

Generate function skeletons	`taste-generate-skeletons`	Result:
		`mysimulink/`
		`|-- Simulink_DataView_asn.m`
		`|-- mysimulink_script.m`
		`|-- setInputsBusCreator.m`
		`|-- setOutputsBusSelector.m`
		`` `-- tunable_parameters.m``

Open Simulink and run the main script: `mysimulink_script.m`		The two tunable parameters appear in the Matlab

Action	*Illustration*	*Notes*
		workspace.
Fill up the Simulink skeleton *and make use of the tunable parameters* (otherwise the code generator will skip them)		Use tunable parameters wherever you need at runtime to monitor and patch data (e.g. to tune an algorithm).
Generate the code from the Simulink model and unzip the resulting file back in the folder where TASTE generated the .m scripts.		
Create a deployment view and map your functions on hardware.		The TASTE_Probe component must be placed on the same node as the function containing the parameters you want to monitor, while the TASTE_Probe_Console component must reside on a native platform (Linux).
Build the system	From the interface view editor, run the menu option: *Tools->Build the system* (in C or in Ada)	
Run the main system binary	`$ cd binary.c/binaries` `$ ./demo`	
Open a new terminal and run the PeekPoke GUI	`$ cd PeekPoke` `$ ./peekpoke.py ../demo`	A GUI shall appear

Action	*Illustration*	*Notes*
In the GUI, start typing the variable name you want to monitor. The complete name shall appear. Select the variable type (real double) Select the number of elements (1) Select the sampling time (1) Click on: `Add to list of monitored variables`		The tunable parameter value will appear immediately in the bottom table. If the Simulink model execution modifies the value, it will be reflected at the next sample time. You can monitor all the binary's global variables (not only Simulink tunable parameters). Just start typing a variable name and add it to the list.
Select a value, and click on one of the options (`Plot`, `Meter`, `Un-plot`, `Un-meter`, `Set selected to this value`)		Use this feature to patch data at runtime (possibly on target) and see how your system reacts (inject faults, tune algorithms…).
Record monitored values: click on Choose filename and select a (.csv) text file. Tick/untick the "`Append monitored values to file`" checkbox.		The resulting csv file can be open in a spreadsheet for post-processing.

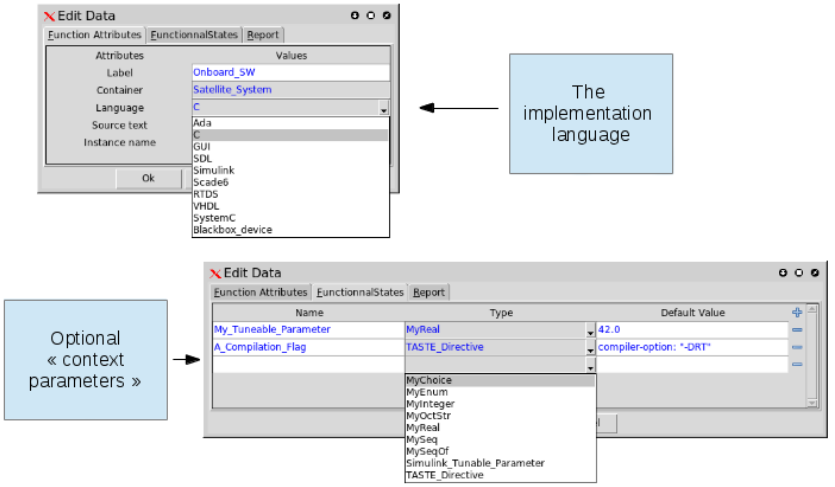
Action	*Illustration*	*Notes*
		
You can save the graphical layout. When you reload it, all plots/meters will appear at the same place and monitored variable values will automatically be updated again. `File -> Save As` Then `File -> Open`		

TASTE V2 Quick Reference Card
Function semantics (from the TASTE Training slides)

Function

- A function is a terminal level entity. It has a behaviour that can be triggered through a set of **provided interfaces**.
- All interfaces of a function have visibility and control access on the function's internal data (static data).
- With one exception, the interfaces of a function are mutually exclusive, and run to completion (it is not possible to execute concurrently two interfaces of a function, as they share state data).

Properties of a function

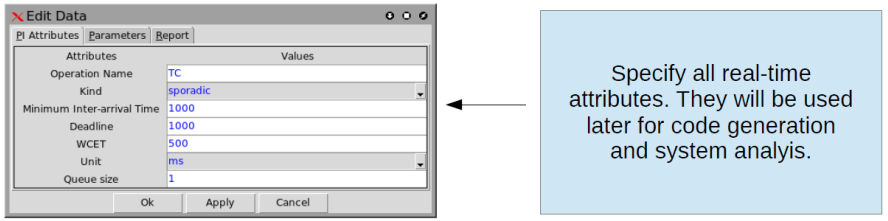


Context Parameters

- The « Functional State » tab offers a space for flexibility :
  - **Context parameters** allow defining constants at model level and make them accessible from user code
    - Support for C, Ada and Simulink (instructs code generator to generate « tuneable parameters », which are global variables)
    - Value can be generated from an external source
  - **TASTE directives** are used to fine-tune the build process with additional properties (e.g. compilation or link flags that are specific to a piece of code)
    - Used to integrate Simulink code when it requires special defines (-DRT, -DUSE_RTMODEL)
    - When a property proves usefulness, it gains a dedicated entry in the GUI

Provided and required interfaces

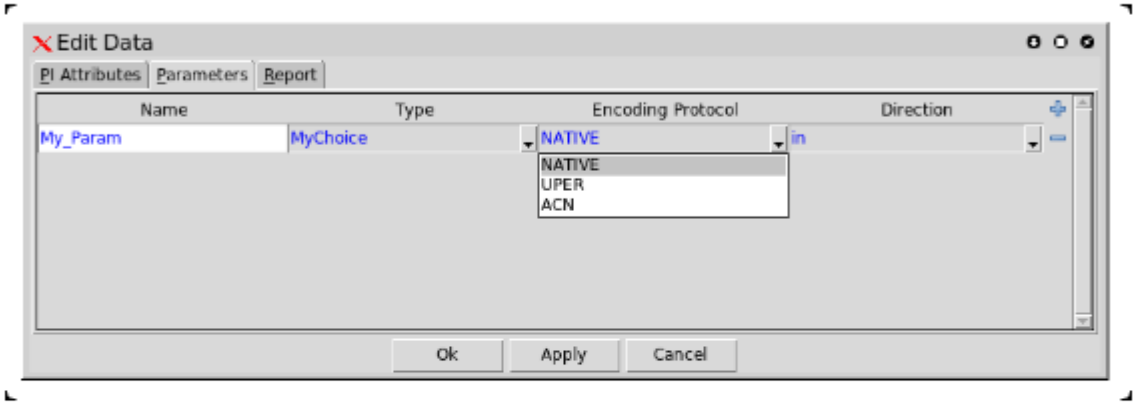
- A provided interface (PI) is a service offered by a function. It can be
  - **Periodic**, in which case it does not take any parameter, and is used to handle cyclic tasks
  - **Sporadic (**or **asynchronous**) and optionally carry a parameter. The actual execution time is decided by the real-time scheduler (call is *deffered*)
  - **Synchronous**, with or without **protection** and optionally carry parameters (in and out)
    - The protection is a semaphore (in C) or a protected object (in Ada) preventing concurrent execution of several interfaces of the same function.
    - Use unprotected interface to implement e.g. « getter » functions
    - Caller blocks on execution (call is *immediate*) – Just like a direct function call.
    - At runtime, synchronous functions execute in the caller's thread space.



TASTE V2 Quick Reference Card
ASN.1 (1)

ASN.1 is used to describe the data type of function parameters

Function parameters



Each parameter has a type (from the ASN.1 model),
a **direction** (in or out), and an **encoding protocol** :

Native : means memory dump – no special treatment
UPER : compact binary encoding
ACN : user-defined encoding

TASTE V2 Quick Reference Card

ASN.1 (2)

ASN.1 – basic types

INTEGER

→ My-int ::= INTEGER (0..7)

value My-int ::= 5

REAL

→ My-real ::= REAL (10.0 .. 42.0)

BOOLEAN

ENUMERATED

→ My-enum ::= ENUMERATED { hello, world }

OCTET STRING

→ My-string ::= OCTET STRING (SIZE (0..255))

value My-string::= 'DEADBEEF'H

BIT STRING

→ My-bitstring ::= BIT STRING (SIZE (10..12))

value My-bitstring ::= '00111000110'B

ASN.1 – complex types

- SEQUENCE

→ My-seq ::= **SEQUENCE** {
x My-int,
y My-enum OPTIONAL
}

value My-seq::= { x 5 }

- CHOICE

→ My-choice ::= CHOICE {
choiceA My-real,
choiceB My-bitstring
}

value My-choice ::= choiceA : 42.0

- SEQUENCE OF

→ My-seq ::= SEQUENCE (SIZE (0..5)) OF BOOLEAN

value My-seq:= { 1, 2 ,3 }

- SET / SET OF

TASTE V2 Quick Reference Card
ACN

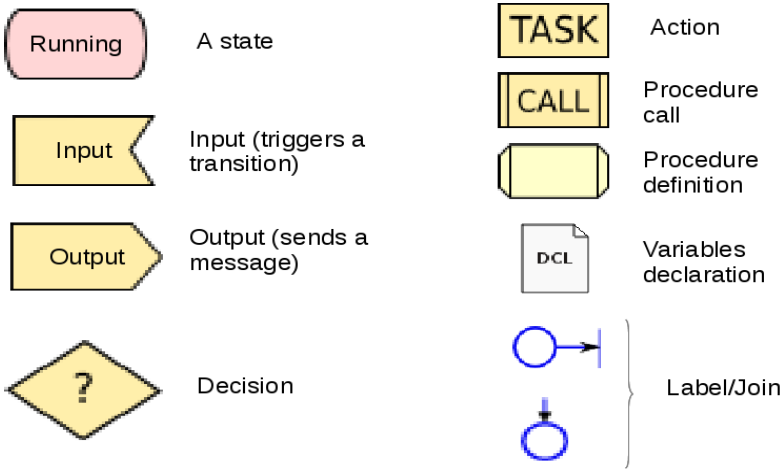
ACN allows to specify legacy encodings – It can be used to describe the format of PUS packets, leaving only the "interesting part" (payload data) in the ASN.1 model
Check the documentation in /home/assert/tool-src/doc/acn

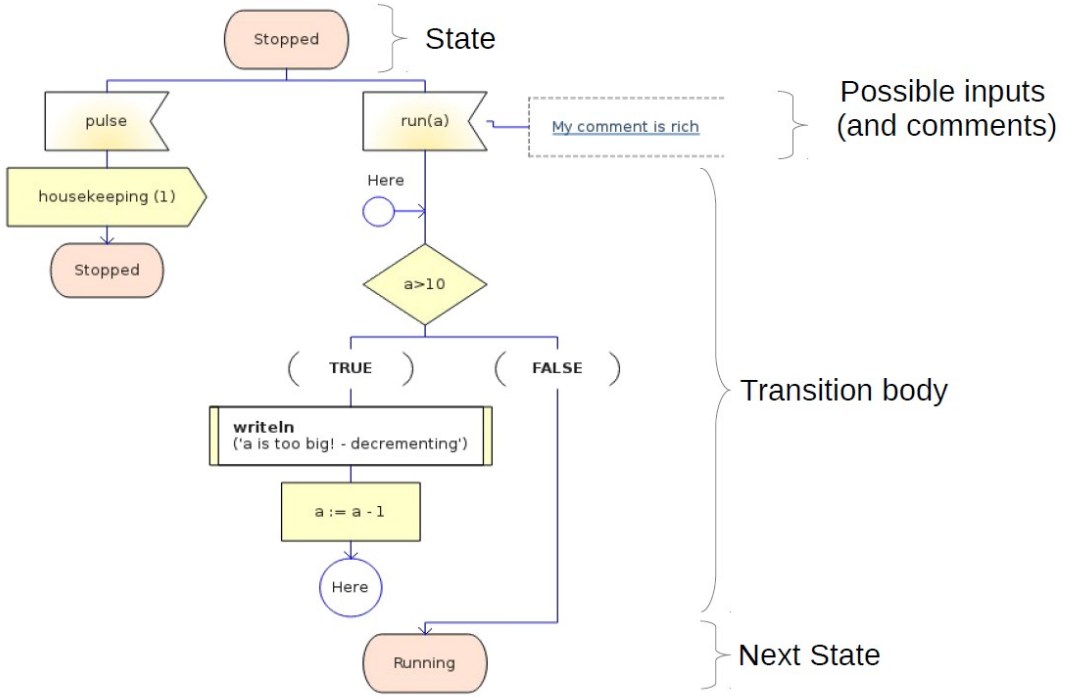
```
MySeq ::= SEQUENCE {
    alpha    INTEGER,
    gamma REAL OPTIONAL
}
```

```
MySeq[] {
    alpha [],
    beta  BOOLEAN  [],
    gamma [present-when beta, encoding IEEE754-1985-64]
}
```

ASTE V2 Quick Reference Card
SDL - OpenGEODE

***SDL** is language that can be used to model state machines, and generate code. TASTE support a commercial tool (RTDS), and hasits own built-in editor (opengeode) for simpler functions.*
Check the training material for description of all symbol. Additional information on www.opengeode.net


Running
A state
Input
Input (triggers a transition)
Output
Output (sends a message)
?
Decision
TASK
Action
CALL
Procedure call
Procedure definition
DCL
Variables declaration
Label/Join


Stopped
State
pulse
run(a)
My comment is rich
Possible inputs
(and comments)
housekeeping (1)
Here
Stopped
a>10
TRUE
FALSE
writeln
('a is too big! - decrementing')
a := a - 1
Here
Transition body
Running
Next State