sinelabore*RT* User Manual

Revision: 6.5.2
April 2025

Disclaimer

The information in this document is subject to change without notice and does not represent a commitment on any part of Peter Mueller. While the information contained herein is assumed to be accurate, Peter Mueller assumes no responsibility for any errors or omissions. In no event shall Peter Mueller be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

Other Copyrights

Contents

Terms of Use, License Agreement

This software and its associated documentation are the intellectual property of Peter Mueller. The full license agreement is available on the web page in German language. If you do not agree to this license do not use the software!

This Software is Licensed, not Bought

The licensee obtains the non-exclusive, non-transferable right, to use the software under the conditions given below. The license remains valid for an unlimited period of time.

Storage and Network Use

Each licensed copy of the software may either be used by a single person who uses the software personally on one or more computers, or installed on a single workstation used non-simultaneously by multiple people, but not both. Each licensed copy may be accessed through a network, provided that you have purchased the rights to use a licensed copy for each workstation that will access the software through the network. You may make copies of the Software for backup purpose. The Software may be installed and used on an unlimited number of computers, provided the software is used in evaluation mode (i.e. without the license file). Evaluation mode of the software may only be used for evaluation or demonstration purposes, not for development or production.

No Derivative Works

The licensee agrees not to translate, disassemble, de-compile or in any other way create derivative works of the software.

No Resale

The licensee agrees not to rent, sell, lend, license or otherwise distribute the software to any third party without the prior written consent of Peter Mueller. (However, the licensee is encouraged to recommend the software.)

Limitation of Liability

The software is provided “as is", without warranty of any kind, express of implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and non-infringement. In no event shall Peter Mueller or any other contributor be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealings in the software. It is the users responsibility to test the generated code as required for the application domain. In no event shall Peter Mueller’s liability arising out of the use or inability to use the software exceed the amount that you have actually paid for the software.

This is only a short form text of the *Terms Of Use. The full text is available on the sinelabore homepage.*

1. Introduction

1.1. Overview

This document serves as a tutorial on how to use the sinelabore code generator. It is intended for software developers who want to generate code from UML state machine or activity diagrams.

A *state machine* shows the dynamic behaviour of an application. It is a graph of states and transitions that describe the response to events depending on the current state. State machines are used for decades in hardware design. And during the last years also more and more in the area of software development. Especially in the embedded real-time domain the use of state machines is popular because the behaviour of devices in this domain can be often very well described with state machines.

An important aspect of state machines is that the design can be directly transformed into executable code. This means that there is no break between the design and the implementation. This is all the more important if the device under development has to be certified (e.g. according to IEC61508).

Activity diagrams show the control flow of an algorithm based on actions which describe the single steps making up the activity. In contrast to state machines an activity maintains no state.

Please note that the code generator is not certified in any way. It is your responsibility to test the generated code as required for your application domain.

The way the code generator works is depicted in figure 1.1. From a state machine or activity model – either designed with an UML tool or the built-in editor[1] – the code generator generates the complete code for the state machine or the activity.

Generating state machine code from a model file called `oven.cdd` the command line to generate C-code would looke like as follows

```
java -cp "path/to/codegen/*" codegen.Main -p CADIFRA -l c -o oven oven.cdd.
```

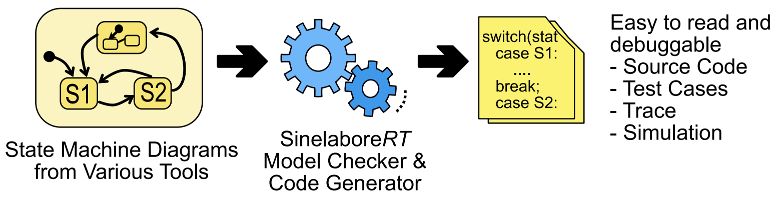


Figure 1.1.: From design to code

- The code-generator was built especially for embedded real-time developers. It focuses on just one task: code generation from state machine or activity diagrams. A command line tool and a configuration file is all what is needed.

- Use the tool only for those parts of your software that benefit from state machine/activity diagram modelling and code generation. Use your existing development environment for all the other code. The code-generator does not dictate an “all or nothing” approach as many other commercial tools.

- The generated code is based on nested `switch/case` and `if/then/else` statements. It is easy to read and understand. The generated code will not create any headache when using static code analysers.

[1] only for state machines

- The code-generator does not dictate how you design your system. Therefore it is no problem to use the generated code in the context of a real-time operating system or within an interrupt service routine or in a foreground / background system.
- The generation process can be influenced to meet specific needs.

1.2. What is new in this version?

The latest major release version is 6.5.2. A list of changes in the previous versions is available in appendix K.

- Refactored core modules for better maintainability and consistency and improved test coverage.
- Improved C++ code generation to eliminate unnecessary dynamic memory allocations.

1.3. Known Limitations

- Regions are fully supported for code generation, the built-in editor and test case generation purposes. But are ignored when generating state tables on Excel basis. This is an open TODO.
- Simulation of a state machine with regions: Presently events with guards are sent to the simulation engine as string. Based on that input the simulator finds the transition to go and changes state accordingly. In machines with regions it might happen that in one region an event (e.g. $ev1$) has guard $[i == 2]$ and in another region $[i >= 0]$. To simulate the event $ev1$ with guard condition $i == 2$ means that both events must trigger a transition but it is only possible to send exactly one event/guard statement – e.g. $ev1[i > 0]$ – to the simulator. This means a state change happens only in one region – which is not correct and does not reflect what actually happens in the generated code! To overcome this problem it would be necessary to *understand* the semantics of the guard condition which would be a major task and is not planned yet.

1.4. How to go on from here?

- *State machines:* If you are not familiar with the state machine notation or need a refresh consult chapter 3. With the help of an application that allows you to interactively send events to a rather complex state machine you can learn how state machines work. Furthermore the elements of a state machine are briefly explained there.
- Section 1.5 describes how to install the code generator on your computer.
- Appendix B and following pages describe briefly how to create state diagrams with the UML tool of your choice in a way that you can generate code from them. Within the samples folder ready made examples for all supported UML tools are available. Start loading such an example and play with it and regenerate the code.
- Section 2 describes the different possibilities to influence the code generator.
- From section 3.15 onwards more advanced features like automated robustness tests, tracing and interactive simulation are described.
- The section 3.2 describes details of the different language backends and the execution model of the generated code.
- In appendix A different options are discussed on how to integrate the generated state machine into your code. Also take a look in the tools section in the appendix. It helps you to get started with a specific UML tool.

- *Activities:* If you want to generate code from activity diagrams read chapter 4. It starts with a general introduction about activity diagrams and gives examples how to use the various node types of activity diagrams.

1.5. Installation

It is necessary to install both the Sinelabore*RT* and the state machine modelling tool of your choice. The order doesn't matter. If you use the internal state machine editor, no UML modelling tool is required.

- *Step 1:* Sinelabore*RT* is bundled as a Java jar file: No installation is required for the Sinelabore*RT* jar file. An installer is also provided for Windows and Mac users. It installs an executable wrapper around the jar file, which can then be used like any other native GUI program installed on your computer.
- *Step 2:* Java Runtime Environment: The editor is completely written in Java. It can therefore run on different operating systems such as Windows, Linux or MacOS. The required Java Runtime Environment (version 11 or higher recommended) is available for download here: `https://jdk.java.net`
- *Step 3:* `Graphviz` layout engine: The Graphviz layout engine is used internally and is only needed if you want to use the built-in visual state machine editor. Install the version for your operating system from here: `www.graphviz.org/`.

See also the getting started articles on the website.

1.6. Limitations in the Demo Version

The demo version is fully functional. It is only limited regarding

- the number of possible states and transitions
- the copyright notice in the generated code can't be changed

The demo version is intended for evaluation purposes only. It is not allowed to use the generated code for product development.

2. Command-Line and Generator Flags

2.1. Overview

The Code Generator is a command line tool and can be fully controlled by a number of command line parameters and a configuration file. The code generator can be invoked as follows

```
%java -cp "path/to/codegen/jar/file/*" codegen.Main
sinelaboreRT codegen version xxx
Usage: java -cp "PATH-TO-JAR/*" codegen.Main [-xls] \\
[-gencfg] [-doxygen] [-Trace] [-verbose] [-U configfile][-t route:to:class] \\
[-l cppx|cx|SSC|java|swift|csharp|lua|python] [-A] [-s|-S|-E] [-c|-c1] \\
[-p EA|MD|CADIFRA|UMODEL|SSC|ASTAH|VP|Modelio|MM|PAPYRUS] [-o outfilename] modelfile
```

A real command line to generate C-code from a model created with the integrated state diagram editor on Windows might look like as follows:

```
D:\>c:\jdk-11\bin\java.exe -cp "c:\sinelaboreRT5.0\bin\*" codegen.Main -p ssc
-o manual  manual.xml
Config path: D:\\codegen.cfg
Starting robustness tests of state machine ...
State names: ..............
Machine hierarchy: ........
Machine height = 1
Transitions: ..............
Default states: ...........
Final states: .............
Choices: ..................
No. of children in composites: ...
Connectivity of states: ...
Can't find the License.txt file or invalid file. Codegen is running in demo mode
Expected license file location: /C:/sinelaboreRT5.0/bin/License.txt
Running in demo mode!
```

The `input` file (here manual.xml) is the state diagram file produced from the modeling tool. The `outfile` (here manual) – defines the name of the generated files and is used as prefix at many places in the generated code.

2.2. Command Line Flags

Option	Options
`-verbose`	Enables the output of information messages during parsing and code-generation
`-t`	In case a XMI file shall be parsed the path to the class that contains the state machine definition must be specified. This option allows you to model more than one state-based class and then generate one after the other by calling the code-generator with the corresponding path. See section G.1 for more info. This option is not necessary if you are using the Cadifra UML editor.

`-c or -c1`	Print transition coverage information and creates an Excel sheet with test routes. See section 3.19 for more information. -c uses a depth-first tree search algorithm which produces fewer but longer test routes. -c1 uses a breadth first algorithm which creates more but shorter test routes.
`-U <file>`	Use the given name as filename for the configuration file. Useful if in one project different config files are needed.
`-L <file>`	Full path to license file e.g. /Users/paul/License.txt
`-Lstates`	List the states in the model on the console.
`-Levents`	List the events in the model on the console.
`-xls`	Creates a file file called 'outfilename.xls' which contains a state table.
`-doxygen`	Creates a dot based description of the state diagram as part of the C/C++ file. This allows Doxygen to add a state machine diagram to the software documentation.
`-s`	Start in interactive simulation mode. Also consider to use the -v flag in addition which enables the printout of the executed C-code per simulation step.
`-S`	Start in graphical interactive mode. Make sure Graphviz [4] is installed on your PC.
`-E`	Start the code generator in editor mode.
`-p`	Defines the editor tool of the input file. Several formats like Enterprise Architect (EA), Magic Draw (MD), UModel, Visual Paradigm (VP), Papyrus, the sinelabore built-in editor (SSC) or the Cadifra UML editor (CADIRFRA) are supported. The webpage informs about newly supported tools.
`-o <file>`	Name of the output file without ending. The generator then adds the endings '*.h', '*.c' automatically to the output file names. If no filename is provided the input filename is used. It is recommended to always specify the output filename! The output name can also contain parts of a path for example to an output folder. Example: `-o generated_files\machine`
`-l`	Define target language. Can be either C, C++, SSC or Java etc. To generate e.g. C++ code type the following '-l cppx'.
`-Trace`	Activates the generation of trace code. Additionally two header files named *_trace.h and *_trace.java are generated that allows to translate the trace id to the event/guard string used in the state chart.
`-gencfg`	Prints out all configuration options for a specific language (use -l to define the language). To quickly generate a config file e.g. for C type in the following: `java -jar codegen.jar -gencfg -l cx > codegen.cfg`

Table 2.1.: Command-Line Flags

2.3. Configuration File

The code generator requires a configuration file named `codegen.cfg` which is normally located in the same directory than the input file. The key / value pairs in this file can be used to adjust the code generator to your needs. Table 2.2 lists all the generator flags and explains their role during code generation. If not otherwise stated all keys are written as one long word (without any spaces and '-' chars).

If no configuration file named `codegen.cfg` is found in the project folder a file named `codegen.cf` is searched. This is sometimes handy due to some IDEs treat cfg files in a special way which is not always wanted.

If no configuration file is found in the project folder the codegen looks for a `codegen.cfg` file in the user home directory.

If no configuration file is present at all the internal default settings are used. To generate a configuration file with the relevant keys for e.g. the C-backend use the following command (example on Windows/Linux):

```
Windows:
c:\jdk-11\bin\javaw.exe -cp "c:/sinelaboreRT5.0/bin/*"
    codegen.Main -gencfg -l cx > codgen.cfg

Linux
$ java -cp "~/sinelaboreRT5.0/bin/*" codegen.Main -gencfg -l cx > codegen.cfg
```

There is a number of keys which are language backend independent. Others are only usable for a specific language backend.

Key	Value	Supported language
Copyright	Defines the text each generated file starts with. Use `'\n'` for multi line comments. Default is `/*\n * (c) Peter Mueller` ...	All
StateMachineFunctionPrefixHeader	Prefix of the state machine function in the C file. Default is void.	C
StateMachineFunctionPrefixCFile	Prefix of the state machine function in the header file. Default is void.	C
ChangeStateFunctionPrefixHeader	Prefix of the state change function in the C file. Default is void.	C
ChangeStateFunctionPrefixCFile	Prefix of the state change function in the header file. Default is void.	C
HsmFunctionWithInstanceParameters	Defines if the state machine function has a point to the instance data as parameter or void. Options are yes or no. Default is yes.	C
UseInstancePointer	If set to 'no' instance data is accessed by value. If set to 'yes' instance data is accessed by reference. If a (user defined) instance is handed over to the state machine this option must be set to 'yes'.	C
HsmFunctionWithEventParameter	If set to 'yes' an event is generated as parameter for the state handler function. HsmFunction WithInstance Parameter must be set to 'yes' also.	C
HsmFunctionUserDefinedEventParameter	Allows to define a user to define an own user defined type as parameter for the state machine handler. Use this parameter in combination with parameter `HsmFunctionWithEventParameter`. This parameter is useful if you want to hand over data in addition to the event itself. E.g. data received from a communication interface or the like.	C

Key	**Value**	**Supported language**
EventFirstValue	Defines if event definitions start from zero or another value. Default is zero. This parameter is specifically useful if some events are defined outside the state machine and it must be ensured that event definitions does not overlap with the generated ones.	C, C++
EventDeclarationType	Defines the C mechanism used for event definition. Options are 'define' or 'enum'. Default is ENUM	C
EventsAreBitCoded	This flag can be used to instruct the code generator to generate bit-coded events. Use this flag whenever events are coded as bits within a variable. E.g. in the context of a realtime operating system where every task has e.g. a 1-byte (8-bit) mask, which means that 8 different events can be signaled to and distinguished by every task. Make sure that the number of events used in the state chart is not larger than the available bits of the message data type. This flag only is used if *EventDeclarationType* is set to *Define* at the same time.	C
EventTypeInCaseOfDefine	In case the above key is set to 'define' the event type can be specified here e.g. to *unsignedchar*.	C
HsmFunctionUserDefinedParameter	A user provided type that is used as parameter for the state handler function. HsmFunction WithInstance Parameter must be set to 'yes' also.	C
InlineChangeToStateCode	If set to yes (= default) the code to change the state variables is inlined. If set to 'no' you have to provide own functions. The required functions are defined in the state machine header file. Provide own functions if you want to run specific code during a state change (e.g. tracing a state change). This switch is only considered from the C code generator back-end.	C

Key	**Value**	**Supported language**
PrefixStateNamesWithMachineName	This parameter allows to prefix state names with the machine name. Use this option if multiple state-machine header files are included into one other file (e.g. `main.c`) to avoid definition conflicts due to double used state names. See also parameter PrefixMsgWithMachineName.	C
PrefixStateNamesWithParentName	This parameter allows to prefix state names with the parent state name. Use this option if you have a hierarchical state machine and want to use the same state names in different child states. Example: There are two parent states called `EngineOn` and `EngineOff`. And in both states you have children `FuelPumpOn` and `FuelPumpOff`. To make the children names unique they can be prefixed with the parent name.	C
Realtab	Options 'yes' and 'no' select if real tabs or spaces are used for indentation.	All
Tabsize	In case of spaces are used for indentation the tabsize can be given here.	All
PrefixMsgWithMachineName	Prefix the message variable (msg) within the state machine with the machine name. This avoids naming conflicts if several state machines are called from within a single file (e.g. from `main.c`) and the message variable is globally defined. Default is 'no'. See also parameter PrefixStateNamesWithMachineName	C
ReturnEventProcessed	If set to yes the machine returns *1U* if a normal event was processed or *0x10h* if a conditional event was processed or *0U* if no event was processed. Use this flag if the application has to know if an event could be processed. This feature is only supported for hierarchical state machines. Note: Erase the 'void' value of the keys state machineFunctionPrefixHeader and state machineFunctionPrefixCFile. Otherwise your C-compiler will create an error. For C++ see `ReturnAndProcessEventTypeOfStateMachine`	All, excl. C++

Key	**Value**	**Supported language**
TypeOfDbgString	User defined string that is used in the debug output file to prefix the state name and event name string array e.g. to place it in a specific segment . . .	C
ValidationCall	Create validation code file and insert a call to the user defined validation handler before each state change. By default it is set to 'no'.	C, C++
UseHammingCodesForEvents	Create events that have a defined hamming distance. By default this is switched off.	C
UseHammingCodesForStates	Create states that have a defined hamming distance. By default this is switched off.	C
HammingDistance	Set the hamming distance for states and events. By default a distance of two is used. Do not use too high values here. We have tested two and three.	C
AdditionalValidate Includes	Allows to define include statements for the validate header file. This is required to define the types used in the validate function. Default is *#include <stdint.h>*	C
AdditionalMachineInclude	One ore more include statements can be provided which will be added to the top of state machine include file. The default is language dependent. Use it for example if you want to provide your own data types. Default is *#include <stdint.h>*	C, C++
UINT8, UINT16, BOOL	Allows to change the used data types to your own definition in the generated state machine code. By default the types from stdint.h are used (uint8_t,uint16_t, uint8_t). If you change these parameters you probably also have to change the parameter above (AdditionalMachineInclude).	C

Key	**Value**	**Supported language**
UMLString = char* UMLInt = int UMLLong = long UMLDouble = double UMLFloat = float UMLShort = short	For attributes and operations in a UML class it is required to provide the used data types. It is recommended to provide a data type definition in the UML model. But it is also possible to use the standard UML data types and then provide a mapping in the configuration file. Generation of attributes and operations is only supported for C and EA right now.	C
INSTANCE_ATTRIBUTE_x INSTANCE_ATTRIBUTE_DEFAULT_VALUE_x INSTANCE_ATTRIBUTE_TYPE_x Example: INSTANCE_ATTRIBUTE_0 = anAttribute INSTANCE_ATTRIBUTE_DEFAULT_VALUE_0 = false INSTANCE_ATTRIBUTE_TYPE_0 = bool	These parameters allow you to create additional attributes that are added to the instance variable of the generated state machine. x has to be 0 for the first attribute, 1 for the second attribute and so on in an increasing order. There must be no gap. This allows to define multiple attributes. If you are using EA it is possible to define the attributes directly in the class diagram. Then these parameters are not needed. Details on EA see in the EA chapter G.	C
AdditionalLocalMachineVars	Allows to define local variables etc. within the state machine. Code defined here is inserted at the very beginning of the state machine function even before any action code. Use `'\n'` for multi line statements.	All
PrefixEvents	Allows to define naming conventions for events	All
PrefixSimpleStates	Allows to define naming conventions for simple states	All
PrefixCompositeStates	Allows to define naming conventions for composite states	All
PrefixChoice	Allows to define naming conventions for choice states	All
UnknownStateHandler	Allows to define code that is executed if a state variable holds an invalid state. Can be used for debug purposes as an example.	All

Key	**Value**	**Supported language**
UnknownEventHandler	Allows to define code that is executed if an event could not processed in a state. Can be used for debug purposes for example.	C
DotPath	Path to 'dot.exe'.	Built-in Editor
ShowOnlyHotTransitions	Options 'yes' and 'no' are possible. If set to 'yes' only hot transitions (i.e. transitions which are accepted from the actually active state) are displayed. If set to 'no' also all other transitions are displayed in gray.	Built-in Editor
NumberOfTransitionChars	It is possible to limit the length of the event text. This keeps the image compact. By default a value of 9 is set.	Built-in Editor
UdpPort	Port the graphical simulator listens for event strings. By default the port is set to 4445.	Built-in Editor
DisplayEntryExitDoCode	If set to yes action code is displayed in the state diagram of the integrated editor.	Built-in Editor
NumberOfEntryExitDoCodeChars	Limit action code in the integrated editor to the given number of chars e.g. 20 to not blow up the diagram to much.	Built-in Editor
SaveCheckedOnly	If set to 'yes' it is always possible to save the model. Otherwise only after a successful check.	Built-in Editor
IncludeDateFileHeaders	If set to 'no' the data and time info is suppressed	All
IncludeTransitionsIntoStatesTheyStartFrom	Export SCXML from the graphical editor in a hierarchical manner. Otherwise exported in a flat way.	Built-in Editor

Key	**Value**	**Supported language**
OptimizeExitCode	If multiple transitions are triggered from the same event leaving the same state the exit code is copied into each transition (I.e. multiple times). This can be a problem if the result of the exit code should be used as guard for the leaving transitions. With this option the exit code is placed in front of the exit evaluation code. But **you** have to make sure that the exit code should really be executed each time an event fires even if no guard is true (i.e. no state change happens). Using this options must be considered quite well! Ensure that at least one guard is true.!!!	All
BaseClassStates, BaseClassMachine	Define an optional base classes for the generated state classes or the machine class.	C++, Java, C#
UseEventObject, EventObjectType	If UseEventObject set to 'yes', EventObjectType allows to set a user defined type as parameter to the state machine handler - e.g. EventObjectType=UserData.	C#
AdditionalTraceInclude	Allows to define additional include code for the trace header file	C, C++
CreateFactoryMethodsVirtual	If set to yes virtual create methods are generated in the factory class. Useful if it the state classes should be specifically initialized after creation.	C++
CreateOneCppStateHeaderFileOnly	If set to yes all state classes are generated into a single cpp/h file.	C++
SeparateStateClasses	Do not create separate state classes. Inline all state code into the state machine.	C++ , C#
StateMachineClassHasDestructor	If set to `yes` a destructor for the state machine class is generated. If set to `virtual` a virtual destructor is generated. If set to `no` no destructor is generated.	C++
Namespace	Namespace the class is generated inside.	C++, Java, C#
UseUnderlineForIncludeProtection	Avoid using __ to protect the state machine include headers. Set *UseUnderlineForIncludeProtection=NO* to remove the underlines.	C,C++

Key	Value	Supported language
UseStdLibrary	Activates the use of several C++ feature introduced in more modern releases of the standard e.g. *std::string* instead of *char** for managing the event/state strings for debugging purposes. Your compiler must support at least *-std=c++11* features. See parameters below for further details.	C++
UseEnumBaseTypes	Activates the scoped and typed definition of the state and event enumerations. Compile with clang requires –*std=c++11* to be set. Also set parameter *UseStdLibrary*	C++
NotUseRedundantVoidArgument	Not use *void* for methods without any parameter.	C++
EnableTrailingReturnType	Switch on trailing-return-type syntax. Requires –*std=c++11* and *UseStdLibrary=yes* to be set.	C++
EnumBaseTypeForEvents	Type of event enumeration e.g. *std::uint16_t*. See also *UseEnumBaseTypes*	C++
EnumBaseTypeForStates	Type of state enumeration e.g. *std::uint16_t*. See also *UseEnumBaseTypes*	C++
ReturnAndProcessEventTypeOfStateMachine	Defines the type of the *evConsumed* variable and thereof the return type of the processEvent() method	C++
ReturnAndProcessEventTypeOfStateMachineIsBool	Sets the *evConsumed* type and thereof the return type of the processEvent() method as bool. Default setting is 'no'	C++
InitializedFlagTypeOfStateMachine	Defines the type of the *m_initialized* variable	C++
VisibilityInitializedVar	Defines the visibility of the variable *m_initialized*. By default it is set as *protected:*	C++
VisibilityStateVars	Defines the visibility of the state variables. By default it is set as *protected:*	C++
CppHsmFunctionWithEventParameter	Defines whether the processEvent() method has an event parameter. In case of your state machine is based the concept of conditional triggered transitions (see 3.1.4) you can avoid warnings from the compiler.	C++
CallInitializeInCtor	Defines whether the state machine initialise method is called within the CTOR or must be called by the user separately. If called in the CTOR don't call it a second time yourself	C++
ProtectHeaderPrefixString	Provide own prefix for include header protection	C++

Key	**Value**	**Supported language**
IncludeNamespaceInHeaderProtection	Includes the namespace in the header protection string	C++
CppxTemplateUse	If set to yes the code generator will generate the state machine function with a template parameter. Also note the two parameters below. Default is 'yes'.	C++
CppxTemplateParameter	Use this parameter to specify the template parameter. Default is const T& userdata. Also note the parameter below and above.	C++
CppxTemplateMsgAccessor	Use this parameter to specify how to access the event embedded in the template parameter. Default is userdata.msg;. Change this according to your needs. Also note the two parameters above.	C++
EnabledSCXMLObfuscation	If set to yes (default is false) the code generator can generate a xml file with obfuscated state machine information. The state names, entry/action/exit code, event names are all modified in a way that the state machine still works, but no internal information is published.	ssc
SplitEventGuardActionInSeparateLines	If set to yes (default) event/guard/action data is displayed on a transition is broken up into different lines. This makes the diagram more compact.	Built-in Editor
ShowStateNotes	Set to 'yes' by default state and region nodes are displayed in the state diagram	Built-in Editor
StateNotesFillColor	Notes fill color in the state diagram. Set to #fffceb by default (light brown)	Built-in Editor
StateNotesFont	Font used for the notes field in the state diagram. Set to 'Arial' by default.	Built-in Editor
StateNotesFontSize	Font size used for the notes field in the state diagram. Set to font size '9' by default.	Built-in Editor
StateNotesFontColor	Font color used for notes in the state diagram. Set to #a18148 by default	Built-in Editor
StateNotesFrameColor	Defines the frame of the note field in the state diagram. Set to #8a6e3d by default.	Built-in Editor
PythonUseEnums	Use enums for event and states. Set to 'yes' by default.	Python

Key	Value	Supported language
PythonUseFunctionAnnotation	Use function annotations. Set to 'yes' by default.	Python

Table 2.2.: Generator flags in file `codegen.cfg` which allow to influence the code generation process. C and C++ refers to the latest CX and CPPX language generator back-ends. To generate a configuration file for the required target language (e.g. C) call the code generator as follows: java −jar codegen.jar −l cx −gencfg > codegen.cfg

3. Statemachine Code Generator

A state machine shows the dynamic behaviour of an application. It is a graph of states and transitions that describe the response to events depending on the current state. State machines are used for decades in hardware design. And during the last years also more and more in the area of software development. UML state machines have some significant improvements compared to classical Moore or Mealy state machines. They support hierarchical designs, it is possible to model parallel execution (and states) and several pseudo-states were added to further improve expressiveness.

Especially in the embedded real-time domain the use of state machines is popular because the behaviour of devices in this domain can be often very well described with state machines.

An important aspect of state machines is that the design can be directly transformed into executable code. This means that there is no break between the design and the implementation. This is all the more important if the device under development has to be certified *(e.g. according to IEC61508)*.

The remainder of this chapter explains the state diagram elements in more detail. But before you go into details play with the interactive example presented in the next section.

3.1. Introduction

3.1.1. Representing Statemachines

Basically a state machine can be represented as a tree of states (see figure 3.1). A transition e.g. $\xrightarrow{e13}$ in figure 3.1 connects the two states *S12* and *S22*. If the transition is triggered you have to walk upwards in the tree starting from *S12* until you reach a common parent of *S12* and *S22*. Then walk downwards in the tree until the target state (in this case *S22*) is reached. On the way all the entry and exit code of the visited states has to be executed.

If the starting state is a composite state, it must also be determined which child state is to be exited. If the target state is a composite state, it must also be determined which child state is to be entered. If history states are used, their history must be taken into account when entering states.

Fortunately, the code generator takes care of all these conditions in the generated code. So you don't have to worry about all the details involved in implementing state machines in a particular language.

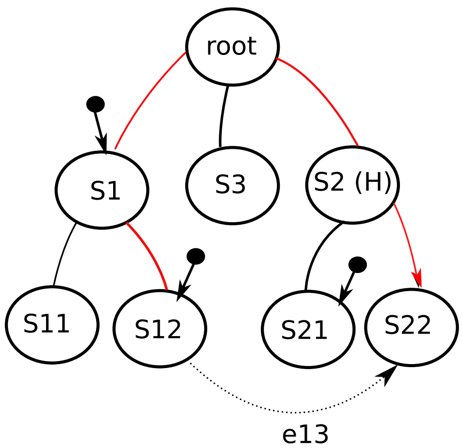


Figure 3.1.: The code generator internally creates a tree from a state diagram. Here the state tree of the state chart from figure 3.2 is shown.

3.1.2. State machines at work

For a quick start into state machines play with an example state machine which is available in the learnings subfolder within both the *Windows* and *LinuxMacBSD* folder on GitHub. Start the code generator in simulation mode[1] and type in events used in the state diagram (e.g. $\xrightarrow{e13}$ or $\xrightarrow{e2}$ followed by a return). Then the simulation performs the state change and calls the related entry / exit and other action code. You can follow what is going on by watching the printouts. Observe especially:

- The order the entry and exit actions are executed in case of event $\xrightarrow{e13}$
- When $\xrightarrow{e1}$ triggers a state change to *S2* and when it triggers a self-transition to *S11*
- Make sure you understand what happens if transitions start or end at composite states
- Make sure you understand the effect of the history marker in *S2*

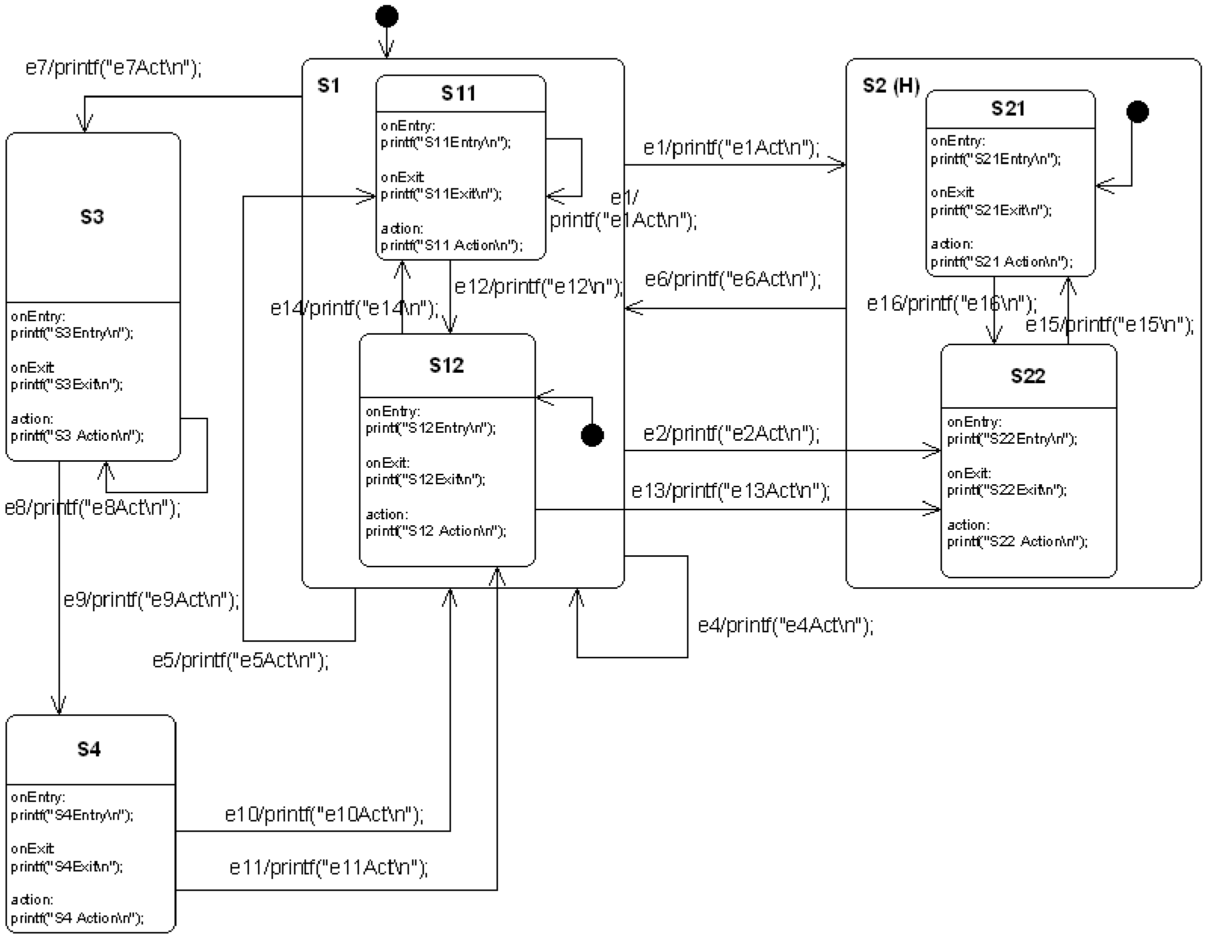


Figure 3.2.: This state diagram example shows many of the features supported from the code generator. It was created using the Cadifra UML editor.

3.1.3. States

State machines can be hierarchical or flat. A state with sub-states is called a hierarchical state machine. States can have entry code that is always executed if a state is entered. Exit code is executed whenever the state is left. Note that the entry and exit code is also executed if a self transition takes place. If events shall be processed without triggering the entry and exit actions

[1] Use `sim.bat` or `sim.sh` depending of your system and set the *CLASSPATH* before. It must include to the parent folder of the codegen.jar. An example how to do this is shown in the *bat/sh* files.

so called inner events[2] can be used. If for a state no entry and exit actions were declared an inner event behaves exactly like a self transition.

A state can also have a *do* activity. The *do* activity code is executed whenever the state is active just before event transitions are evaluated. This means that calculation results from the action code can be used as triggers for state transitions.

Actions within states shall be non-blocking and short regarding their execution time. On every hierarchy level a default state must be specified. A final state is a state that can't be left anymore. I.e. the state-machine must be re-initialised to be reactive again.

It is possible to specify inner events, entry and exit code . . . for a state by linking a note to a state. See figure 3.3 on the right side for an example. The note must start with the text `compartment:`. Sometimes it is useful to use this option despite the UML tool allows to specify entry/action/exit code directly. As the 'constraints:' is not supported by any UML tool it must be always defined within an attached comment note (see section 3.19 for more about using constraints).

In principle states can be nested again and again. The code generator was intensively tested with up to four hierarchy levels. If you use more levels reconsider your design!

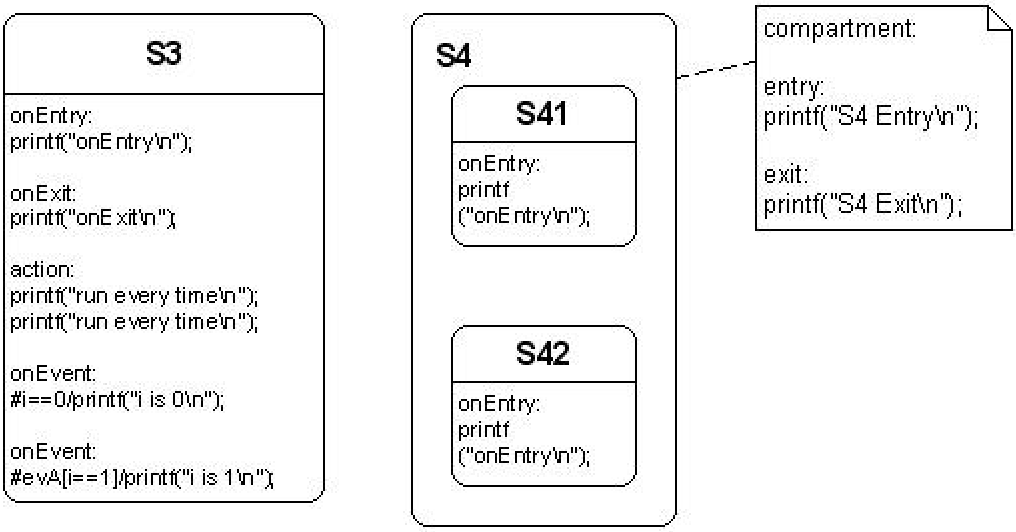


Figure 3.3.: Left: A state with *entry-*, *exit-*, *do-code* and inner events. Right: Complex state with *entry-* and *entry-code* specified in a linked note.

3.1.4. Transitions

There are two types of transitions supported from the code generator. a) event based transitions and b) conditional triggered transitions. An event based transition has the following syntax: `eventName[guardExpression]/action`. The transition is only taken if the guard expression evaluates to *true* and the event was sent to the state machine. Only in this case the action code is executed.

From a transition like

```
evDoorClosed[timer_preset()>0]/timer_start();
```

the code generator generates the following source code:

```
if((msg==(OVEN_EVENT_T)evDoorClosed) && (timer_preset()>0)){
  /*Transition from Idle to Cooking*/
  evConsumed = 1U;

  /*Action code for transition */
  timer_start();
  ...
```

[2] Inner events are presently only supported on the innermost states of hierarchical states and on top level states if they have no children.

A conditional (or guard) transition is not triggered from an outside event but when a guard expression is evaluated to true. It has the syntax: `#condition/action`. Please note that the hash character `#` must be typed in (i.e. prefix your code statement) to indicate this special type of trigger. From a transition like this
`#i==1/printf("i==1\n");` the code generator generates the following code:

```
if((i==1)){
    ...
    /*Action code for transition*/
    printf("i==1\n");
    ...
```

Externally defined events: Normally, for all events used in a state machine diagram, the necessary event definitions are generated in the header file `*_ext.h`. And only these events can be handled by the state machine.

But sometimes it is required to react on events defined externally from the state machine diagram (e.g. existing code is re-used defining own events or the operating system sends predefined events e.g. in case of a timer timeout). The code generator needs to know that a specific event should be or shouldn't be included in the event definition process. Therefore such events must be prefixed with an exclamation mark (!). In all other areas of the generation process (state machine handler code, debug helpers) they are included as any other event. Note: Only the C/C++ backend consider externally marked events.

Action code defined in transitions must be non-blocking! Figure 3.4 shows examples for all types of supported transitions. Action code from an initial pseudostate is only used if the target state of the transition is not in a parent state with history. *In history states the usage of actions on the init transition is often misapplied and therefore ignored!*

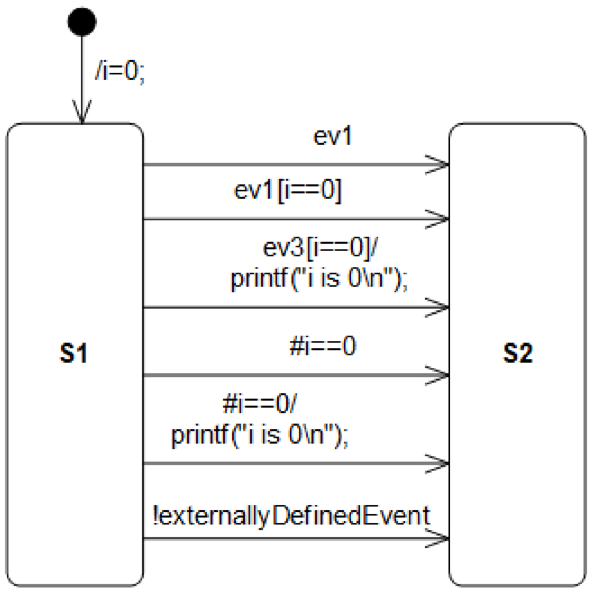


Figure 3.4.: All possible transitions.

From top to bottom:

1. Transition from an init pseudostate. Only an action expression is allowed here.
2. Simple event with no guard and no action
3. Event with guard. The guard expression enclosed in brackets [] is denoting that this expression must be true for the transition to take place.
4. Event with guard and action. If the transition takes place the action code is executed. The action code must be one or more lines of valid C code.

5. Conditional transition. The transition takes place if the variable i is zero.
6. Conditional transition with action. The transition takes place if the variable i is zero. Additionally the action code is executed.
7. Externally defined event

3.1.5. Regions

In state diagrams usually only one state is active at a time. In UML state diagrams regions also allow to model concurrency – i.e. more than one state is active at a time (AND states).

A UML state may be divided into regions. Each region contains sub-states. Regions are executed in parallel. You can think of regions as independent state machines displayed in a diagram. The state machine in figure 3.5 shows the well-known example of a microwave oven designed using regions. Several regions, each running in parallel in the state Active. Dashed lines are used to divide a state into regions.

The power setting, light and microwave are considered as independent (concurrent) parts of the oven, each with its own state. The door and the timer as the main triggers are used in the regions to trigger state transitions. For example, the radiator is switched on when the door is closed and the timer is > zero.

As you can see multiple concurrent regions can be used to explicitly visualize different parts of a device. And all the states in the one diagram.

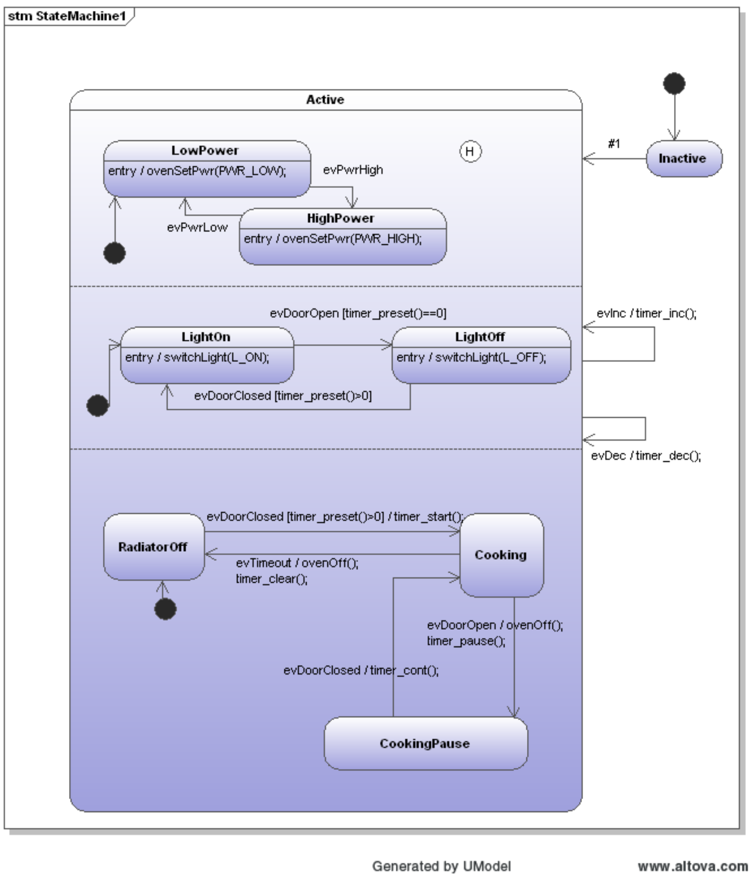


Figure 3.5.: A microwave oven designed with regions.

Points to consider with regions

- Transitions must not cross region boundaries: In the figure 3.5 state transitions do not cross region boundaries and therefore the modelers' intention is clear. But look at the next diagram 3.6. Now it is not clear anymore what the modeler had in mind. And it is also not very obvious what a code generator should generate. For that reason the following constraints were defined.

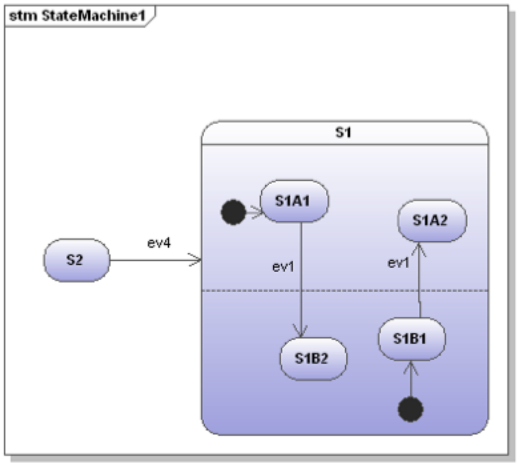


Figure 3.6.: Transitions must not cross region borders. From this diagram no code can be generated.

- Regions must work on the same instance data: State diagrams follow the *run-to-completion* concept. Transitions that fire are fully executed and the state machine reaches a stable state configuration until it returns and can respond to the next event. To ensure this a copy of the instance data is kept and state changes are only performed on that copy. In practice this means that changes in one region does not influence other regions. Look into the following figure below.
  - If the event $\xrightarrow{evClosed}$ is sent – region *ValveA* and *ValveB* change state.
  - But there is no state change in region `Motor` at the same time. The reason is that the transition from *Stop* → *Run* was not triggered at the beginning of the machine execution.
  - This behavior ensures that the result of a machine execution step is 100% predictable and not dependent of the execution order of the regions.
  - But on the other side it means that a second run of the machine is required to reach state *MachineRun*. I.e. the region *Motor* is always one cycle behind the "Valve" regions.

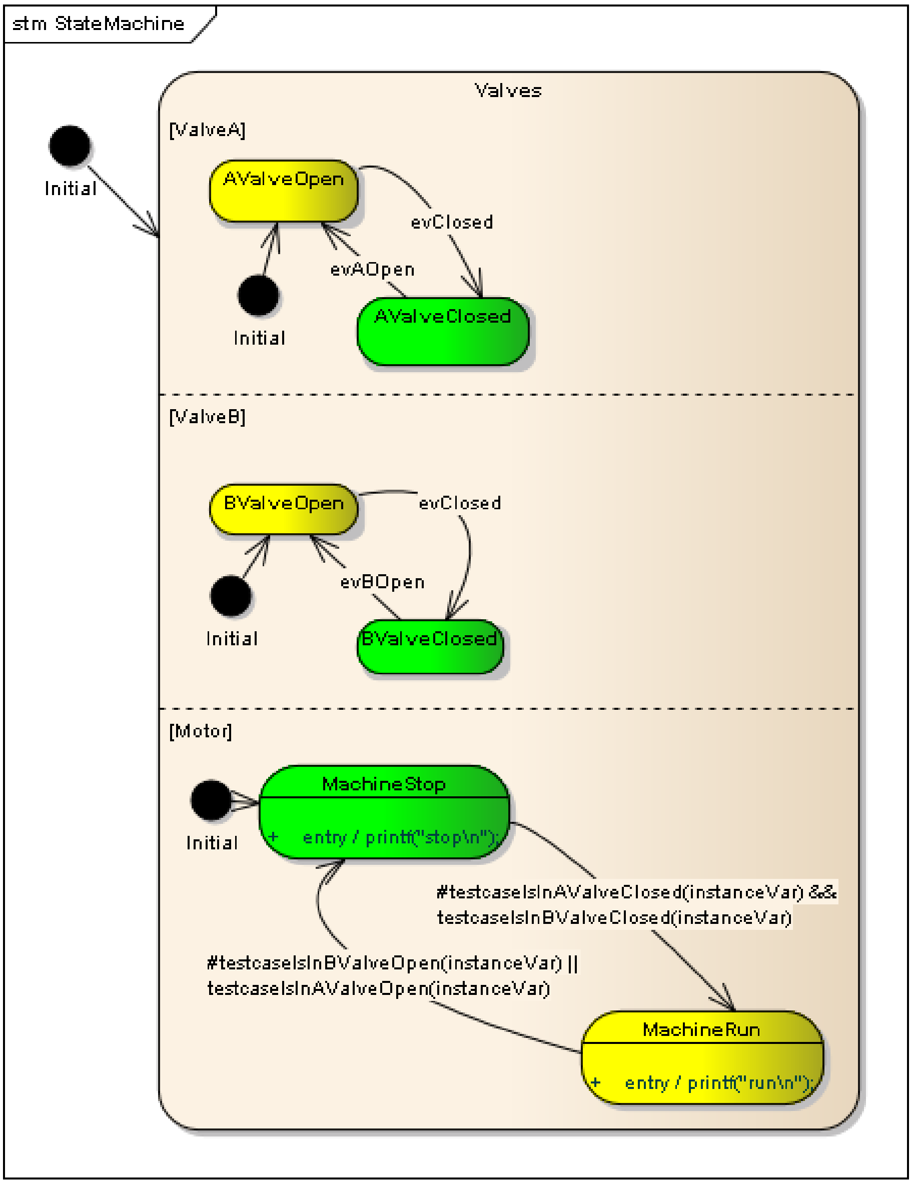


Figure 3.7.: Each region should 'see' the same state set of the whole state machine. Independent of the execution order of the region code. Therefore regions work on copies of the instance variable.

3.1.6. Header, action, postAction and unknownStateHandler Notes

To adapt the generated code to your needs you can add notes to your design that have to start with either '*header:*' or '*postAction:*' or '*action:*' or '*unknownStateHandler:*'.

All code following the '*header:*' keyword is added at the top of the generated statemachine code code file (i.e. the c-file for example). This allows to include required header files or the definition of local variables needed within the statemachine.

See figure 3.8 for an example.

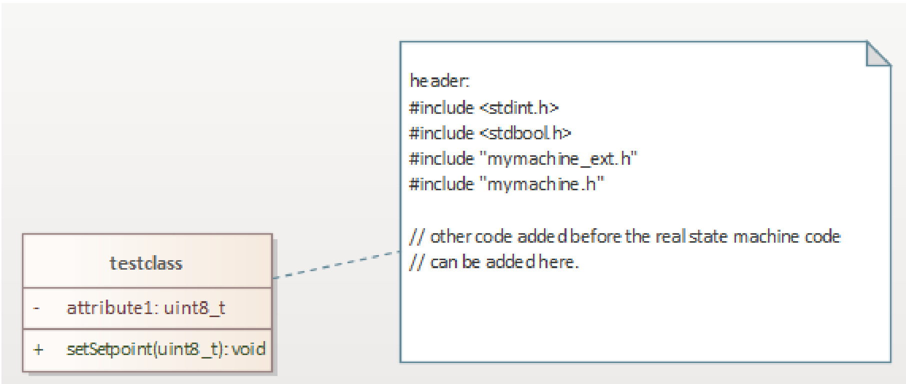


Figure 3.8.: It is possible to define own code inserted on top of the generated file. This allows to specify own include files or other required local code in the state machine file.

Code following the '*action:*' keyword is inserted at the *begin* of the statemachine function. This allows to execute own code whenever the statemachine is called just before event processing starts. In section A.2 this was used to receive events via a message queue.

Code following the '*postAction:*' keyword is inserted at the *end* of the statemachine function. This allows to execute own code after the statemachine code was processed e.g. to enable an interrupt at the end of an interrupt handler function implemented as state machine. *Please note that this generator keyword is only available for the following backends: cx, cppx, java, ssc.*

Code following the '*unknownStateHandler:*' keyword is inserted between each default/break pair of the generated code. The given code will be executed if the state variables do not have a valid state. This should never happen and indicates a serious problem in the system (e.g. memory is corrupted due to a stack overflow). Alternatively, it is possible to include the code in the in the *codegen.cfg* file.

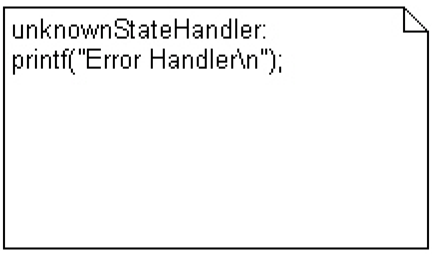


Figure 3.9.: A message is printed whenever an invalid statevar was found.

3.1.7. Choices

The OMG UML specification states: *"...choice vertices which, when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions. This realizes a dynamic conditional branch. It enables splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to-completion step. If more than one of the guards evaluates to true, an arbitrary one is selected. If none of the guards evaluates to true, then the model is considered ill-formed. (To avoid this, it is recommended to define one outgoing transition with the predefined 'else' guard for every choice vertex."*

The simplified state chart in figure 3.10 shows different options how choices can be used. A choice can be placed within a composite state or at root level. A choice must have at least one incoming transition. *Guards specified at the incoming transition(s) are ignored from the code-generator. Actions are simply copied to the outgoing transitions.* Place them at the outgoing transitions.

Usually choices can have only one incoming transition and multiple outgoing transitions (at least two). But the code–generator also allows more than one incoming transition. This is a convenient function to allow the compact specification of complex structures. Internally this construct is handled like two choices with one incoming transition each and the same outgoing transitions.

If more than one incoming transition is found, the choice is doubled for each incoming transition. I.e. it is like modelling two choices with one transition each and having the same outgoing transitions. At least two outgoing transitions must be defined, each with a guard. One of the guards must be an *else* statement as described above. Depending on the target state of each outgoing transition, the corresponding entry/exit code is generated. The creation of choice chains is not supported by the code generation

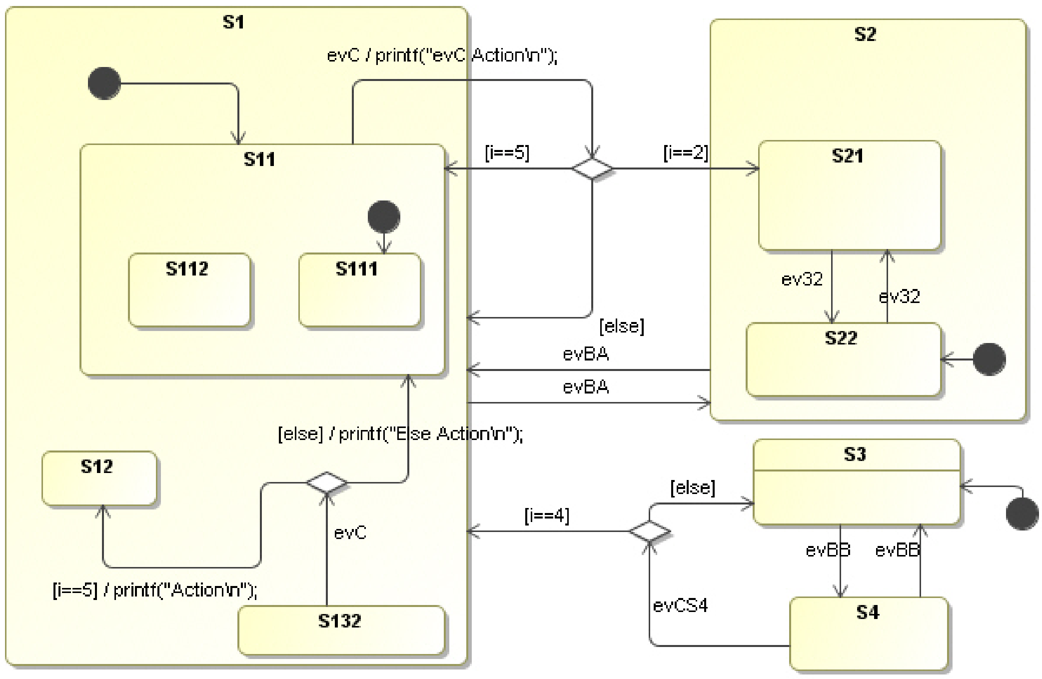


Figure 3.10.: Different options to use choices. A default path marked with *else* must always exist.

3.1.8. Determining the default state dynamically (Init to Choice)

Sometimes the initial state of a state machine or sub–machine shall be determined during run–time and not design–time. Examples:

- If the hardware self-test of a device fails the machine should enter an error state and not a normal operation state
- Depending on a parameter (for example set by a user) a specific state shall be entered

In such cases it is possible to connect the initial pseudo–state with the input side of a choice and connect the outgoing transitions with the target states. **It is important that there exists one else path to ensure that there is always one option that can be taken by default**. Several examples are shown in the following figure 3.11. Note that it is possible to define an action on the incoming transition of a choice that reads a value or performs a check and use the result of that function as guard for the outgoing transitions of a choice.

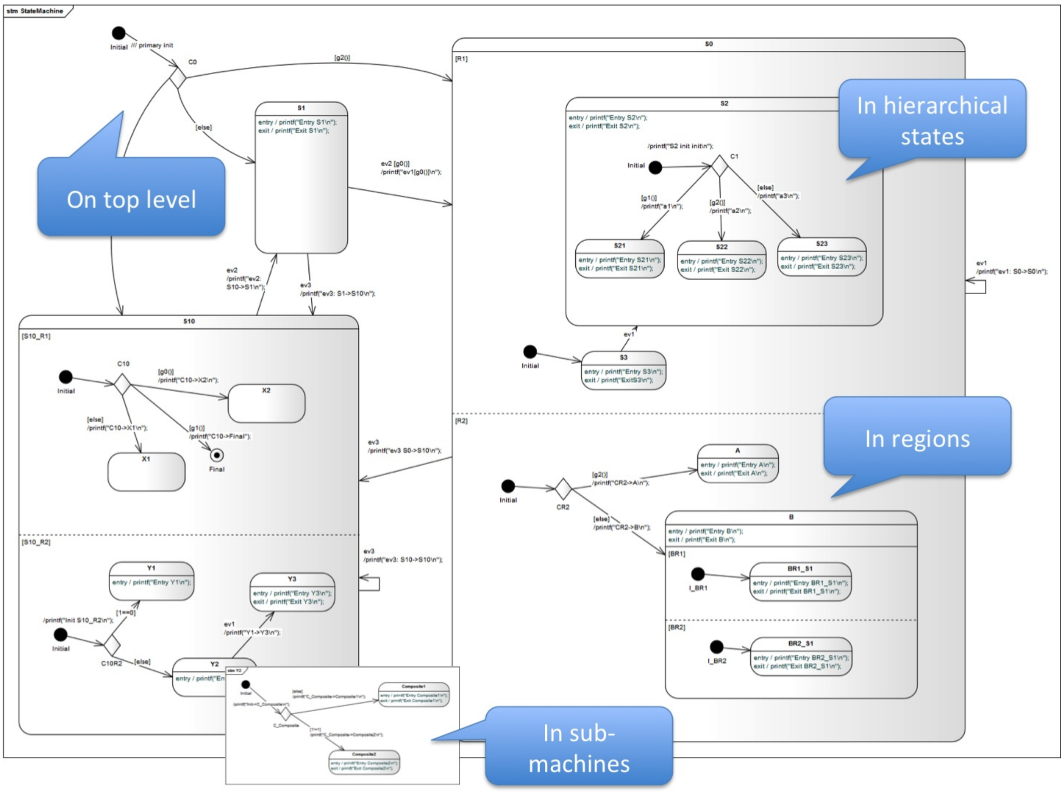


Figure 3.11.: Choices can be used to determine the initial state at run–time. The figure shows several possibilities how to use this feature.

3.1.9. Junctions

The OMG UML specification states: *. . . junction vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path (this is known as a merge).*

The UML specification also discusses the possibility of multiple outgoing transitions. **This is not supported by the codegen!**

The Junctions can be seen as a kind of drawing helper in the case you have several transitions ending all in the same state and all of these transitions share some common action. In such a case place the different triggers, guards and action code to the incoming transitions and the common part the the outgoing transition.

The code-generator creates two separate transitions out of this model. The first one from S1 to S3. The second one from S2 to S3.

Limitations and rules:

- A junction should have at least two incoming transitions
- A junction must have exactly one outgoing transition.
- On the outgoing transition no guard and trigger must be defined.
- The action in the outgoing path is appended to actions defined on the incoming paths.
- Incoming transitions must not start from a pseudo state e.g. another junction, choice, . . .
- The outgoing transition must not end in a pseudo state e.g. another junction, choice, . . .

3.1.10. Final States

Final states have only incoming transitions. Once a state machine enters a final state it will not react on any other event.

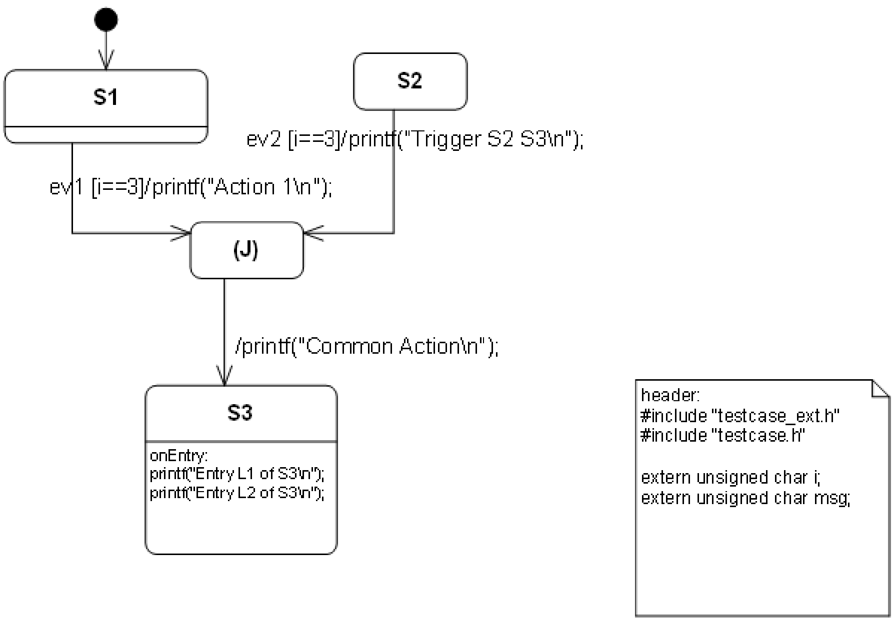


Figure 3.12.: Junction example drawn with Cadifra UML. There is no junction symbol available in Cadifra UML. The shown (J) state is the supported replacement.

Exceptions are final states inside a hierarchical state machine. Transitions leaving the parent state of a final state can still be taken. In figure 3.13 state `Final` can be left via event $\xrightarrow{ev3}$ or $\xrightarrow{evRealEnd}$ while state `Final1` can't be left anymore..

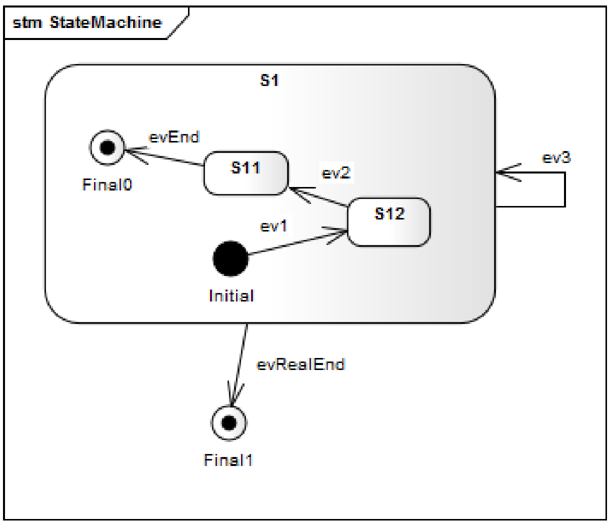


Figure 3.13.: Example usage of final states. Once the machine is in state `Final1` it can't be left anymore. State `Final` can be left via event $\xrightarrow{ev3}$ or $\xrightarrow{evRealEnd}$.

3.1.11. History States

Since version 6.4 the history state handling was changed. It is now possible to end a transition at a history state. Only if a transition ends in a history state will the previous history be considered when entering the state set. Otherwise the default entry chain is used. The new `TransitionsCanEndInHistoryStates` parameter has been added to enable this feature. Consider the example shown in figure 3.14. After reset the state machine is in state `S1`. State `S2` can be entered in two ways. Either via $S1 \xrightarrow{evA} S2$ or $S1 \xrightarrow{evB} S2/H$. There is not yet any difference. Both will enter state `S22` because there is no history for S2 yet. Now let's enter state `S21`, triggered by event $S22 \xrightarrow{evA} S21$. Then let's leave state `S2` to state `S1` with the transition $S21 \xrightarrow{evB} S1$. There is a difference now if we re-enter state `S2` via transitions triggered by event $S1 \xrightarrow{evA} S2$ or event $S1 \xrightarrow{evB} S2/H$. If entering via transition triggered by event $S1 \xrightarrow{evB} S2/H$ the previous history will be used and thus state `S21` is entered. If entering via transition triggered by event $S1 \xrightarrow{evA} S2$ the default state is entered which is state `S22`.

This new possibility makes history state handling more powerful and the designer's intention more clear than before.

Note: It is also possible to reset the history to its default state using the generated reset history function. Consider the situation where a user is interacting with an HMI and has left the configuration menu. But has forgotten to make a setting. The designer wants the user to re-enter the last menu they left before, to make it easy to resume work where they left off. But after a minute, the user should reach the top configuration menu by default. This behaviour is easily achieved by resetting the history on a timed event.

Note: The integrated state machine editor does not allow transitions to end in history states. Also, models with transitions ending in a history state can't be used with the integrated state machine editor.

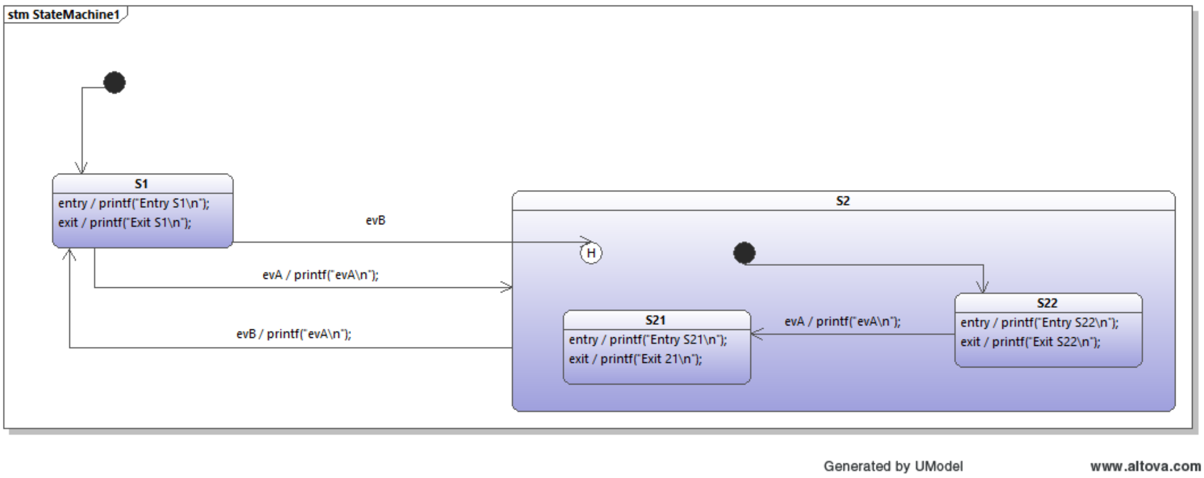


Figure 3.14.: Example that shows the new style of history handling.

3.2. Generating Code

3.2.1. Execution Model of the Generated Code

The execution model of the generated code is as follows:

- Single event processing: The generated state machine function takes the given event and processes it. Transitions are triggered by at most one event. If an event was consumed the machine returns. Queuing mechanisms can be easily added by the user. Take a look into the appendix section A.2.
- Run-to-completion processing: An event stimulates a run-to-completion step. Transitions that fire are fully executed and the state machine reaches a stable state configuration until it returns and can respond to the next event.
- Priority concept for transitions triggered by the same event: A transition has higher priority the deeper its source state is in the state hierarchy.
- Transition types: Transitions can be either triggered by events send from outside to the state machine or by conditions that are checked within the state machine (see section 3.1.4).
- Machine execution: To take a transition the user has to execute the state machine function. There is no principle limitation regarding the context the state machine function can be called. It can be integrated in an operating system task, called from within an interrupt service routine or in the main loop of a foreground-background system.
- Cycle time: If all transitions are triggered from events created outside of the machine it is not necessary to call the state machine if no event is available for processing. If the state machine also contains transitions triggered by internal conditions (e.g. testing if a port pin has changed from low to high) then it should be called as often as the conditions needs be checked. In this case the event variable must be set to 'NO_EVENT'. The cycle time for calling the state machine depends solely on the application needs.
- State machine realisation: The state machine function uses (nested) switch/case statements to select the active state and if/then/else statements to select the triggering event within a state.
- Transitions leaving a state are always triggering the exit code of this state - i.e. they are considered as external transitions. So called local transitions are also handled as external transitions.

The code generator supports the generation of state machines in different programming languages. The language to generate can be defined on the command line. Depending on the language features the generated code differ in its structure. The following sections describe what you need to know per programming language.

The code generator supports up to four state hierarchy levels.

3.2.2. Generate Code from State Machines with Regions

This section briefly describes how regions are implemented.

For each region an own function is generated. Its name is automatically derived from the region name. If a state contains several regions they are called one after the other (in alphabetical order). If the event sent to the state machine was processed in one of the regions no further event handling happens in the parent state. Otherwise the event is processed in the parent state. This is similar to the event handling of normal hierarchical state machines.

To maintain consistency during execution of machine code a copy of the instance data is created at the beginning of the state machine code. All tests are performed on the original instance data.

All changes are done on the copy. This ensures that all regions see the same situation when running. At the end of the machine code the modified instance data is copied back to the original data.

3.3. Generating C Code

Sinelabore creates compact and clearly readable C code from UML state charts. There are a number of different ways in which the generated code can be fine-tuned through the setting of configuration parameters. This allows generation to follow known patterns, which are illustrated with examples below.

- Object pattern: Instance related data is always grouped in a struct. All state machine related functions operate on this data struct. The advantage is that the data flow is easy to follow, the code is re-entrant because there is no global manipulated state, and multiple instances are easy to realise.

- Opaque object pattern: This is similar to the object pattern, but hides the implementation from the outside of the generating state machine code. This further reduces dependencies between the state machine implementation and other code.

- Single instance pattern: Typically there is only one instance and no instance pointer is needed. For example, implementing an IRQ handler as a state machine, or using a hardware module in the state machine that exists only once.

In the generated code, state hierarchies are mapped to nested switch/case code. Event handling code is mapped to if/else-if/else structures. The various configuration parameters allow you to generate code that is optimal for your system. See section 2.2 for a list of all available options.

To generate C code call the code generator with the command line flag `'-l cx'`.

To generate a configuration file with all parameters related to the C code generation call the code generator as follows once:

```
java -cp path_to_bin_folder/* codegen.Main -l cx -gencfg > codegen.cfg
```

3.3.1. Data Types

The datatypes used in the state machine code are all defined in the generated state machine header file.

Simple data types like `unsigned char` or `unsigned int` are not anymore used from the code generator (since version 2.40) but it uses the data types defined in `stdint.h`[3] instead.

If you want to specify your own data types change to following parameters in the codegen configuration file:

- AdditionalMachineInclude - default is: #include <stdint.h>

- UINT8 - default is uint8_t

- UINT16 - default is uint16_t

- BOOL - default is uint8_t

3.3.2. Own include files, attributes and operations

By default the code generator includes a standard set of include files into the generated code. Often it is required to add own includes and other code before the generated code. Attach a note to the class as described in section 3.1.6. See figure 3.15 for an example.

With some UML modelling tools (built-in editor, EA) it is possible to specifiy additional attribtues and functions added to the header and implementation files of the generated state machine code. This makes it possible to add own functionality e.g. an entry function of a state or to set parameters needed in the state machine guards or actions. Figure 3.15 shows an example.

[3] see http://en.wikipedia.org/wiki/Stdint.h

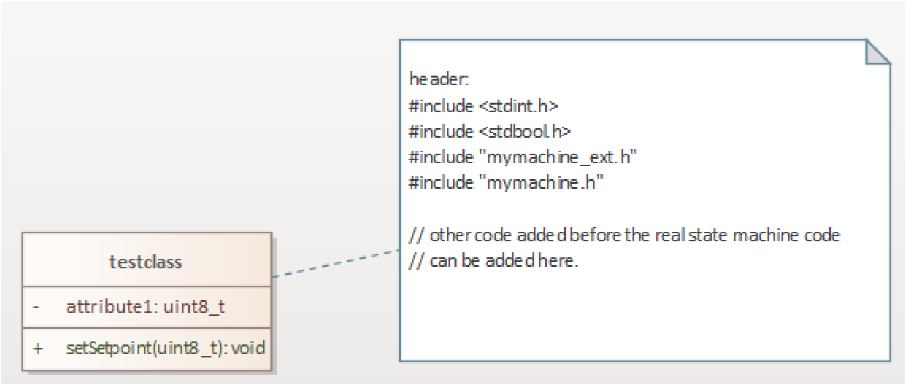


Figure 3.15.: It is possible to define own code inserted on top of the generated file. This allows to specify own include files or other required local code in the state machine file.

3.3.3. Specification of the Statemachine Function return Value

It is possible to specify whether the state machine handler function should return a value or not. If the **ReturnEventProcessed** parameter is set to *no*, no value is returned (void). If set to *yes* a flag is returned indicating whether an event or conditional trigger has been processed.

If the ReturnEventProcessed parameter is set to *yes*, the return type used can be specified with the **StateMachineFunctionPrefixHeader** parameter for the header file and with the **StateMachineFunctionPrefixCFile** parameter for the C file. By default, both parameters are set to *void*. If the handler function needs further compiler specific decorations, use these two parameters (e.g. to place the function in a specific code segment or to specify it as an interrupt service routine...). An example is given below.

If *ReturnEventProcessed* and *StateMachineFunctionPrefixHeader* or *StateMachineFunctionPrefixCFile* is set at the same time a warning is printed to indicate a configuration conflict. In this case the *ReturnEventProcessed* parameter has priority.

Further info e.g. how the state handler function can be defined as interrupt handler is provided in section 3.3.4 further below. Also see section A.7.

3.3.4. Specification of the Statemachine Function Parameters

To optimize the state handler signature you can combine a number of generator parameters for your specific environment. The following examples show typical generator parameter combinations:

- **HsmFunctionWithEventParameter and HsmFunctionWithInstanceParameters**: Set these two flags to *yes* to best follow the object pattern. Use this option also if several instances of the state machine should exist in your system. The generated header file contains all relevant structures and makes them visible to the outside.

```
void testcase(TESTCASE_INSTANCEDATA_T *instanceVar, TESTCASE_EVENT_T msg){

    TESTCASE_EV_CONSUMED_FLAG_T evConsumed = 0U;
    ...

    switch(instanceVar->stateVar){
        case S1:
        ...
            if(msg==(TESTCASE_EVENT_T)evA){
```

- **EnableOpaqueStateMachinePattern, UseTypedefForInstanceData**: Setting the first flag to *yes* and the second to *no* together with the two flags explained above gives the best fit if you want to use the opaque object pattern. In this case, the state machine **_ext.h** header file contains all the relevant declarations but not the instance data struct declaration. A full example is available in the samples folder called *microwave_builtin_editor_c_opaque*.

```
void testcase(struct TESTCASE_INSTANCEDATA *instanceVar, TESTCASE_EVENT_T msg);
```

```
void testcaseInitMachine(struct TESTCASE_INSTANCEDATA * const instanceVar, TESTCASE_INST_ID_T instId);

size_t testcaseSizeOf(void);
```

- **HsmFunctionWithInstanceParameters**: If set to *yes* the state handler expects a pointer to the instance variable. No other parameters can be send to machine with this option. The internally needed message variable (msg) must be provided in the 'header' comment in the UML file e.g. as a global variable. Use this option if the message (i.e. the evens) are available in a global variable e.g. set from an interrupt handler.

  The generated handler looks as follows.

```
void testcase(TESTCASE_INSTANCEDATA_T* instanceVar){

    TESTCASE_EV_CONSUMED_FLAG_T evConsumed = 0U;

    switch(instanceVar->stateVar){
        case S1:
  ...
```

- **HsmFunctionWithEventParameter**: If set to *yes* and the above option is set to *no* the state handler expects only an event as parameter. Use this option if only one instance of the state machine is available in your system and events should be sent from outside to the state handler. The internally needed instance variable must be defined in the 'header' comment in the UML file.

```
void testcase(TESTCASE_EVENT_T msg){

    TESTCASE_EV_CONSUMED_FLAG_T evConsumed = 0U;

     switch(instanceVar->stateVar){
         case S1:
          ...
```

- **HsmFunctionUserDefinedParameter**: With this parameter you can provide the type name of a self defined struct which is then used as parameter to the state handler. This is the most flexible option. Use this option if you want to hand over own parameters, the event, the instance variable and more to the state handler. function. The self defined type must contain at least a field named `instanceVar` which is used from the state handler.

  Let's take the example of a system with multiple serial ports and the state machine should get the parameters like baud rate, parity for each machine ...

  First declare the new instance variable type in a file called e.g. `own_inst_type.h` containing

```
typedef struct InstanceData MY_TESTCASE_INSTANCEDATA_T;

#include <stdint.h>
#include <testcase_ext.h>
#include "testcase.h"

struct InstanceData{
        uint16_t baudrate;
        uint8_t noBits;
        uint8_t parity;
        uint8_t stopBit;
        TESTCASE_INSTANCEDATA_T instanceVar;
};
```

  Then the generated state machine handler will look like the code below. Make sure you include the `own_inst_type.h` in the state machine c-file before the other state machine headers. To do this, use the `header:` attachment to the class to provide the code that should be included at the top of the c-file.

```
#include <stdint.h>
#include <stdio.h>
#include "own_inst_type.h"
#include <testcase_ext.h>
#include <testcase.h>

void testcase(MY_TESTCASE_INSTANCEDATA_T* userInstanceVar, TESTCASE_EVENT_T msg){
        TESTCASE_INSTANCEDATA_T *instanceVar; /* ptr to instance data */

        /* Set instance var once */
        instanceVar=&(userInstanceVar->instanceVar);

        /*execute entry code of default state once to init machine */
        if(instanceVar->s1Entry==1U){

                printf("Baudrate   = %d\n",userInstanceVar->baudrate);
                printf("NumberBits = %d\n",userInstanceVar->noBits);
                printf("Parity     = %c\n",userInstanceVar->parity);
                printf("Stopbit    = %d\n",userInstanceVar->stopBit);

                instanceVar->s1Entry=0U;
        }


        switch (instanceVar->stateVar) {
  ...
```

- **HsmFunctionWithEventParameter and HsmFunctionWithInstanceParameters and HsmFunctionUserDefinedEventParameter**: You can define an own user defined type as parameter for the state machine handler. This is useful if you want to hand over data in addition to the event. E.g. data received from a communication interface or the like.

  Let's make an example and set the following parameters
  `UseInstancePointer=yes`
  `HsmFunctionWithEventParameter=yes`
  `HsmFunctionUserDefinedEventParameter=USER_EVENT_T`

  In this example the definition of the `USER_EVENT_T` is as follows. It is important that the msg parameter is part of the type definition. It is expected from the code generator.

```
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include "testcase_ext.h"


typedef struct {
        TESTCASE_EVENT_T msg;
        uint8_t i;
        uint8_t something_else;
}USER_EVENT_T;
```

  The generated code for the state machine looks like as follows. The generated code accesses the message (msg) inside the user defined type. Any other variables can be used in action or guard code as needed. Take care to include the required header as shown in 3.3.2

```
#include "user_defined_event.h"
#include "testcase.h"
#include "testcase_ext.h"
#include <stdio.h>

void testcase(TESTCASE_INSTANCEDATA_T *instanceVar, USER_EVENT_T *userData){
    ...
   switch (instanceVar->stateVar) {
      case S1:
        if(userData->msg==(TESTCASE_EVENT_T)ev2){
           // e.g. use user define data as guard
           if((userData->i)==1){
```

...

- **Statemachine as Interrupt Handler**: Usually it is necessary to decorate interrupt handlers with compiler specific keywords etc. Furthermore interrupt service handlers have no parameters and no return value. To meet these requirements the keys `StateMachineFunctionPrefixHeader`, `StateMachineFunctionPrefixCFile` and `HsmFunctionWithInstanceParameters` can be adjusted according to your needs.

  The example below shows an interrupt service routine with the compiler specific extensions as required by mspgcc.[4]

```
interrupt (INTERRUPT_VECTOR) IntServiceRoutine(void)
{
    /* Statemachine code goes here */
}
```

  To generate such code set the key/value pairs in your configuration file the following way:

```
StateMachineFunctionPrefixCFile=interrupt (INTERRUPT_VECTOR)
HsmFunctionWithInstanceParameters=no
```

  If the prefix spans more than one line the line break `'\n'` indicator can be inserted as shown below:

```
StateMachineFunctionPrefixCFile=#pragma vector=UART0TX_VECTOR\n__interrupt void
```

  Please note that the prefixes for the header and the C file can be specified separately.

3.3.5. Using events declared outside of the state machine diagram

Usually for all events used in a state machine diagram, the required event definitions are generated in the header file `*_ext.h`. And only these events can be handled from the state machine.

But sometimes it is required to react on events defined externally from the state machine diagram (e.g. existing code is re-used defining own events or the operating system sends predefined events e.g. in case of a timer timeout). The code generator needs to know that a specific event should be or shouldn't be included in the event definition process. Therefore such events must be prefixed with an exclamation mark (!). In all other areas of the generation process (state machine handler code, debug helpers) they are included as any other event.

Note: The header with the definition of the events must be included into the state machine using the method as described in section 3.1.6). An example for such a machine is shown below in figure 3.16.

To avoid collisions between the generated event values and the externally defined event values it is possible to define the start value of the generated event definitions or enums. Use the following parameter `EventFirstValue` as described in table 2.2.

Let's assume some external events are defined as follows:

```
#ifndef __EXTERNAL_EVENTS_H__
#define __EXTERNAL_EVENTS_H__

#define ev1 1U
#define ev2 2U
#define ev3 3U
#define ev4 4U

#endif
```

Then a useful value for the parameter would be for example `EventFirstValue=10`. The generated header for the events looks as follows then. Note that the events start from 10 and not from 0.

[4] See http://mspgcc.sourceforge.net/manual/x918.html for further details.

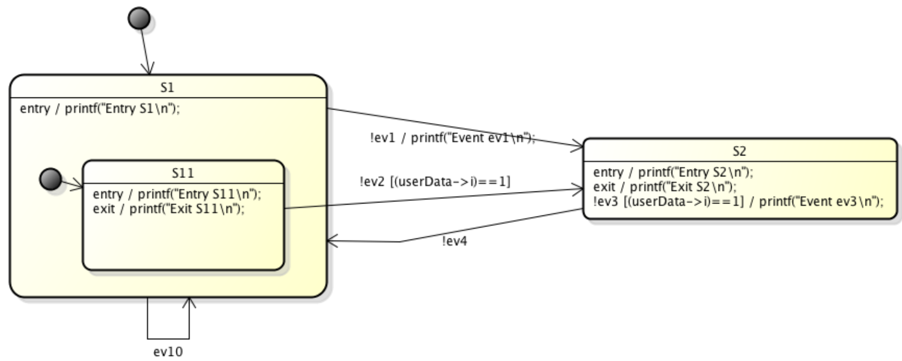


Figure 3.16.: Events that should be excluded from the event definitions can be marked with an explanation mark (!). They must be declare somewhere else instead (e.g. existing code).

```
#ifndef __TESTCASE_EXT_H__
#define __TESTCASE_EXT_H__

/*Events which can be sent to the state-machine */
typedef enum
{
        ev10=10U,
        TESTCASE_NO_MSG
} TESTCASE_EVENT_T;

#endif
```

Initializing the Instance Variable

The state machine instance data type contains the state variables and other flags needed in the state handler function. Before the state machine is called the very first time this variable must be initialized. A macro is provided in the header to do this.

Example:

```
TESTCASE_INSTANCEDATA_T instData = TESTCASE_INSTANCEDATA_INIT;
```

3.3.6. Resetting the State Machine

Sometimes it is necessary to reset the machine to its default state. A reset function is generated for this purpose. Note that the instance id is not changed from this function.

```
testcaseResetMachine(&instData); // testcase is the machine name in this case
```

3.3.7. Features for High-Availability Applications

For high availability applications it is desirable to detect serious errors happening outside the state machine code but effecting the correct execution of the state machine. Such errors might come from a runaway pointer overwriting key variables, power brownout corrupting a bit or a bad ram cell losing a bit to name a few. To detect and handle such situations is an overall system design task. But the code generator can help you to detect inconsistencies of the state machine offering the following two mechanisms:

- You can provide error handler code that is executed in the default path of the switch/case statements. This helps to detect undefined values of the state variables. See configuration key 'UnknownStateHandler'.

  Example:

```
switch (instanceVar->stateVar) {

  case S1:
      ...

  case S2:
      ...

  default:
    error_handler();
  break;
}
```

- The code generator optionally generates a validate function that checks if a transition from a present state to a new target state is allowed (i.e. modeled in the state diagram). To enable the generation of the validate function set parameter '*ValidationCall=yes*' in the config file. This validate function has to be called from a user provided handler function that is automatically called from the state machine code. This indirection allows that you can define the reaction if a transition is not allowed.

  In a single CPU set-up the validation function runs on the same controller that executes the state machine code. But it is also possible to execute the validate function on a second CPU in a redundant CPU setup.

  Example state machine:

```
    case S22:
        if(msg==(TESTCASE_EVENT_T)ev32){
                /* Transition from S22 to S21 */
                testcaseValidationHandler(S22, S21, instanceVar->inst_id);
                evConsumed=1U;
                /* OnExit code of state S22 */
                ...
```

  Example for an user defined handler:

```
// your own handler begins here
void testcaseValidationHandler(uint16_t from, uint16_t to, uint8_t machineId){

  uint8_t retCode;

  retCode = testcaseValidate(from, to);

  if(retCode!=0U){
    // transition not allowed
    reboot(); // or whatever is best in your system
  }else{
    printf("Transition allowed\n");
    return;
  }
}
```

  The validate code uses some predefined types. Define them in a header that fits your system needs and include it by setting the parameter '*AdditionalValidateIncludes*' in the config file.

- It is possible to generate the states and event enumerations (or defines) in an ascending order but using a user defined hamming distance between them. So in case one or more bits swap accidentally the state machine will end up in the default path which can then call an error handler. See parameters *UseHammingCodesForEvents*, *UseHammingCodesForStates* and *HammingDistance*.

3.3.8. Regions

Regions allow to model parallel activities in state diagrams and can be defined in top level states. The microwave oven design is available in the examples folder as reference.

Here is the simplified C-code example of the oven state machine shown in figure 3.5.

```
void oven(OVEN_INSTANCEDATA_T *instanceVar){

  OVEN_EV_CONSUMED_FLAG_T evConsumed = 0U;
  OVEN_INSTANCEDATA_T instanceVarCopy = *instanceVar; // create copy of instance variable

  switch (instanceVar->stateVar) {

    case Active:
      /* calling region code */
      evConsumed |= ovenActiveLight(instanceVar, &instanceVarCopy, msg);
      evConsumed |= ovenActivePower(instanceVar, &instanceVarCopy, msg);
      evConsumed |= ovenActiveRadioator(instanceVar, &instanceVarCopy, msg);

      /* Check if event was already processed */
      if(evConsumed==0U){
        .... /* handle event on parent level */


    break;
   } /* end switch stateVar_root */

   /* Save the modified instance data */
   *instanceVar = instanceVarCopy;
}

OVEN_EV_CONSUMED_FLAG_T ovenActiveLight(OVEN_INSTANCEDATA_T *instanceVar,
      OVEN_INSTANCEDATA_T *instanceVarCopy, OVEN_EVENT_T msg){
  ...
}

OVEN_EV_CONSUMED_FLAG_T ovenActivePower(OVEN_INSTANCEDATA_T *instanceVar,
      OVEN_INSTANCEDATA_T *instanceVarCopy, OVEN_EVENT_T msg){
  ...
}

OVEN_EV_CONSUMED_FLAG_T ovenActiveRadioator(OVEN_INSTANCEDATA_T *instanceVar,
      OVEN_INSTANCEDATA_T *instanceVarCopy, OVEN_EVENT_T msg){
  ...
}
```

3.3.9. Running Multiple Instances of the State Machine

Sometimes you want to run multiple instances of the same state machine (e.g. processing tree interfaces with the same state machine). Let's extend the example from above where we already defined an own type of instance type. Follow these simple steps to prepare the generated code for multiple instances:

1. Set parameter `HsmFunctionWithInstanceParameters` to `yes` (as above)
2. Set parameter `HsmFunctionWithEventParameter` to `yes` (as above)
3. Declare as many instances of the instance variable as you need. To quickly initialize it use the predefined macro. Here is an example:

```
// init three different serial ports
MY_TESTCASE_INSTANCEDATA_T instDataA = {9600,8,'N',1,TESTCASE_INSTANCEDATA_INIT};
MY_TESTCASE_INSTANCEDATA_T instDataB = {19200,8,'E',1,TESTCASE_INSTANCEDATA_INIT};
MY_TESTCASE_INSTANCEDATA_T instDataC = {9600,8,'N',1,TESTCASE_INSTANCEDATA_INIT};


int main(int argc, char* argv[]){

  TESTCASE_EVENT_T msg=TESTCASE_NO_MSG;

  // Set object ID if the machine needs to know which object it is
  // E.g. which serial port to open ...
  instDataA.instanceVar.inst_id=0;
  instDataB.instanceVar.inst_id=1;
  instDataC.instanceVar.inst_id=2;
```

```
        // call "object A"
        testcase(&instDataA, msg);

    // call "object B"
    testcase(&instDataB, msg);

    // call "object C"
    testcase(&instDataC, msg);

    return 0;
}
```

4. If it is necessary to distinguish inside the state machine between the different instances (e.g. to decide to initialize the serial interface PORT0 or PORT1) you can use code as shown in the following figure 3.17 on page 48.

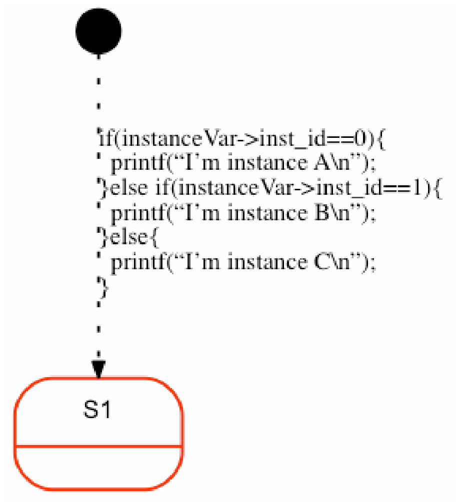


Figure 3.17.: Use the instance id to distinguish between multiple instances inside the state machine.

The output of this state machine is the following (as you might expect):

```
I'm instance A
I'm instance B
I'm instance C
```

Sometimes it might be necessary that you need additional local variables per instance. In this case you have to define your own instance data type which you then hand over to the state machine handler. This is described in section 3.3.4 on page 41.

3.3.10. Important Types and Helper Functions

This section describes some important type definitions that you should understand and the type definitions and helper functions you have to provide to be able to compile the state machine code.

3.3.11. User Defined Typedefs

Within the generated state machine code different flags and variables are used. By default the generated state machine header file contains all required type definitions to compile the generated code (Note: Before version 1.7 the typedefs were not automatically generated. It was always up to you to provide them). If you want to use your own types set the define as shown below.

For a state machine called 'testcase' the following definitions are generated:

```
// Predefined types required by the codegen.
// You can provide your own
// definitions by setting the following define somewhere
// in your built env.
```

```
#ifndef __PROVIDE_OWN_TESTCASE_STATEMACHINE_TYPES__
  typedef uint8_t TESTCASE_ENTRY_FLAG_T;
  typedef TESTCASE_STATES_T TESTCASE_STATEVAR_T;
  typedef uint8_t TESTCASE_INST_ID_T;
  typedef uint8_t TESTCASE_EV_CONSUMED_FLAG_T;
#endif
```

The following table 3.1 explains these defines.

Typedef	**Meaning**
`xxx_ENTRY_FLAG_T`	Used as a flag that the state machine code runs the very first time. If true the onEntry code of the default states is executed. Afterward the flag is reset.
`xxx_STATEVAR_T`	Type of the variable the state machine uses to store the present state into.
`xxx_INST_ID_T`	Type of the variable that can be used to differentiate between several instances of the same machine (see multiple instances). You can set this variable to a different value per state machine instance and use this figure within the machine to distinguish between the different instances.
`xxx_EV_CONSUMED_FLAG_T`	Internal flag used to find out if an event was already handled within an inner state or if it must be handled in the outer state.

Table 3.1.: The 'xxx' is replaced from the code generator with the name of the state machine.

3.3.12. Important (Type) Definitions

This subsection lists the typedefs the code generator creates in the state machine's header file.

(Type) Definition	Meaning
`xxx_INSTANCEDATA_T`	Structure that contains all instance specific variables of the state machine.
`xxx_STATES_T`	Enumeration that contains all possible states.
`xxx_INSTANCEDATA_INIT`	Macro that can be used to initialise the instance variable. Especially all state variables are set to their default states.
`xxx_IS_IN_yyy`	For each state a macro gets generated that returns 1 if the machine is in the state or 0 if not. The machine returns 1 for all the parent states of a child state the machine is in at the moment. E.g. S111 is a child of S11 which is a child of S1. If the machine is in S111 then it is also in S11 and S1 which means `xxx_IS_IN_S1` and `xxx_IS_IN_S11` returns both 1. This macro is useful if two machines shall cooperate and a transition in one machine shall be triggered if the other one is in a certain state.
`xxx__RESET_HISTORY_yyy`	For each history state this macro is generated which allows to reset the history if needed.

Table 3.2.: Important (type) definitions the code generator creates. The 'xxx' is replaced from the code generator with the name of the state machine. The 'yyy' is replaced from the code generator with a state name.

3.4. Generating C++ Code

3.4.1. Introduction

C++ is used more and more also in smaller embedded systems. Therefore the code generator can generate code using only a very small subset of C++. Actually only classes, public/protected/private member variables and methods. are used. Features such as virtual functions, templates etc. are not used by default, but can be enabled if required. Since version 4.0 several new configuration parameters have been added. With the new parameters there is much more flexibility to customise the generated code to suit your needs. Some of the new parameters require a compiler that supports at least the C++ 2011 standard (i.e. set --std=c++11 for g++ or clang++). Generated code was checked with clang−tidy and −checks='modernize−∗' switched on.

For your convenience there is a codgen command line parameter that activates all C++11 features by default. If -std c++11 is used on the command line the following parameters are used for code generation:

```
UseEnumBaseTypes = YES
EnumBaseTypeForEvents = std::uint16_t
EnumBaseTypeForStates = std::uint32_t
ReturnAndProcessEventTypeOfStateMachine = std::uint16_t
InitializedFlagTypeOfStateMachine = std::uint16_t
VisibilityInitializedVar = private:
VisibilityStateVars = private:
CallInitializeInCtor = yes
NotUseRedundantVoidArgument=YES
EnableTrailingReturnType=YES
UseStdLibrary = YES
```

If -std c++14 is used on the codegen command line the code generator uses either `make_unique` or `make_shared`. You as user must define which type you want to use. Set the following parameter:

```
StatePointerStyle = SHARED | UNIQUE
```

To generate C++ code call the code generator with the command line flag `'-l cppx'` and optionally with the standard option `'-std c++11'` or `'-std c++14'`.

To generate a configuration file with all parameters related to the C++ code generation call the code generator as follows once:

```
java −cp "path to bin folder" codegen.Main −l cppx −gencfg > codegen.cfg
```

The generated code does not follow the state pattern as you might expect in case you are familiar with common design patterns. This is because the machine code is completely generated and no hand coding is involved. The following figure 3.18 on page 52 shows the structure of the generated code. The classes marked as `<<generated>>` are generated from the code generator. Classes marked with `<<optional>>` are optional and must be provided by you if needed.

- The `StateMachine` class realises the state machine as modelled in the state diagram. The name of the class (here StateMachine) can be defined with the command line flag `-o`. The `initialize()` method must be called once to init the machine (i.e. set default states ...). After initialisation the `processEvent()` method can be called with the actual event as parameter. Methods to reset the history of a composite state and to check in which state the machine is are available too. It is possible to specify a base class the machine should be derived from (here shown as `MachineBaseClass`). To do so specify the base class name in the `codgen.cfg` file.

- For each state defined in the state chart diagram a class is created with the methods `onEntry()`, `onExit()` and `action()` if needed (here shown as `StateClass`). The state classes are named like the states in the state chart diagram. It is possible to specify a base class (here shown as `BaseStateClass`) the states should be derived from. To do so specify the base class name in the `codgen.cfg` file. If the key '`CreateOneCppStateHeaderFileOnly`' is set to 'Yes' all state classes are generated in one cpp/h file. if the key '`SeparateStateClasses` is set to no (= default for the new cppx backend) no state classes and no factory class is generated. Using the keyword '`stateheader`' in an UML comment (like header or action) allows to define additional include files in the generated state cpp files.

- A `StateMachineFactory` can be optionally provided in the constructor of the `StateMachine` class. The factory separates the construction of the state classes from the machine class but let the factory decide which state class to instantiate. See *Design Patterns, Elements of Reusable Object- Oriented Software; Addison-Wesley 1997* for background information about the factory design pattern.

- The generated machine class can optionally have a base class provided by you. Set the base class name in the configuration file as follows: `BaseClassMachine=YourMachineBaseClassName`. By default no base class is expected.

- The generated state classes can optionally have a base class provided by you. Set the base class name in the configuration file as follows: `BaseClassStates=YourStateBaseClassName`. By default no base class is expected.

- The generated state handler method can optionally have a template parameter. This makes it easily possible to hand over all kind of data togehter with the event to process.

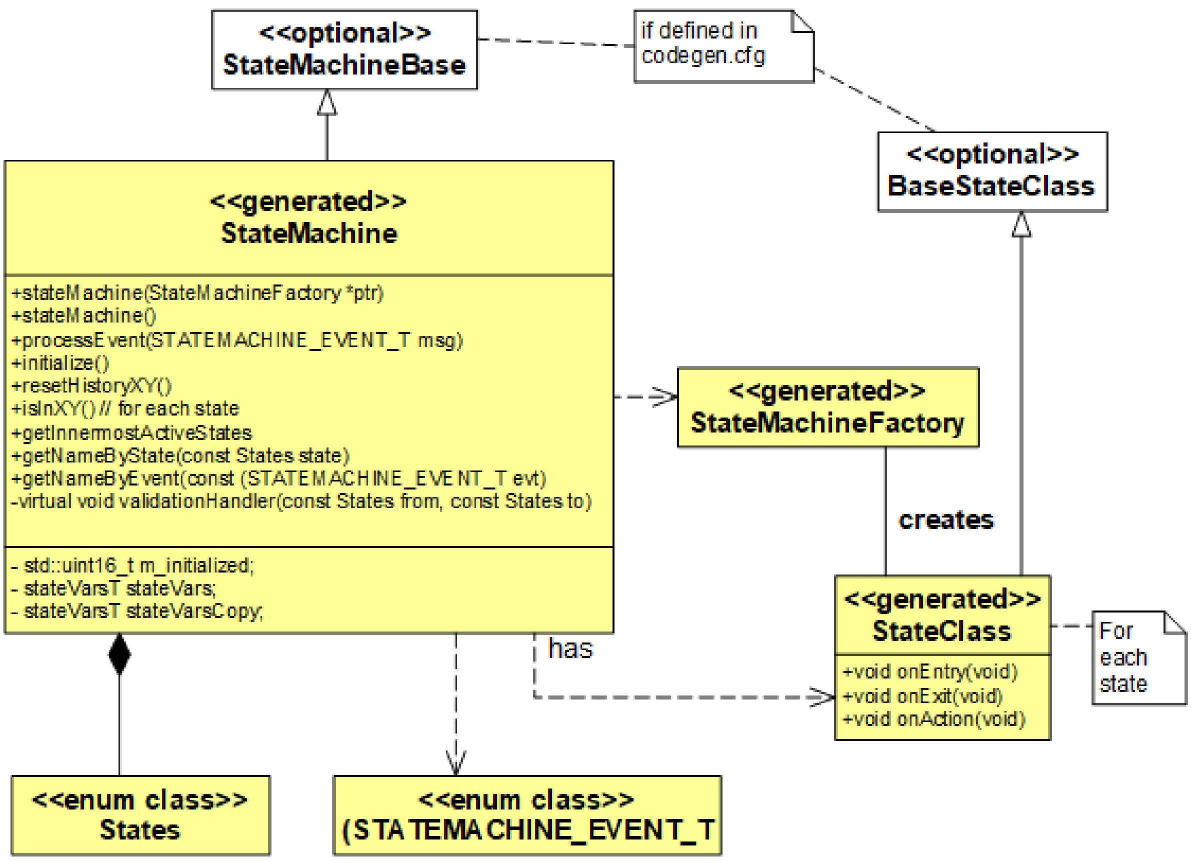


Figure 3.18.: Class diagram of the generated state machine classes and optional base classes. CPPX backend: The Factory and State classes are only generated if you have set the related configuration key. Otherwise the state classes action code is simply integrated in the generated state machine code.

Sometimes it is necessary to access the state machine object from the state classes. To make this possible the code generator can automatically set a reference to the state machine object in each generated state class.

To enable this feature set parameter `BackrefToMachineInStateClasses` = `yes`. The class diagram on page 53 shows the classes and methods generated for this configuration. During the generation of the state objects a back reference to the state machine object is set into the state objects. The `entry()/do()/exit()` code of the state objects can then access the state machine object. Typically methods or members of the state machine that should be accessed from state classes are implemented in a base class of the generated state machine. The base class must be written by hand.

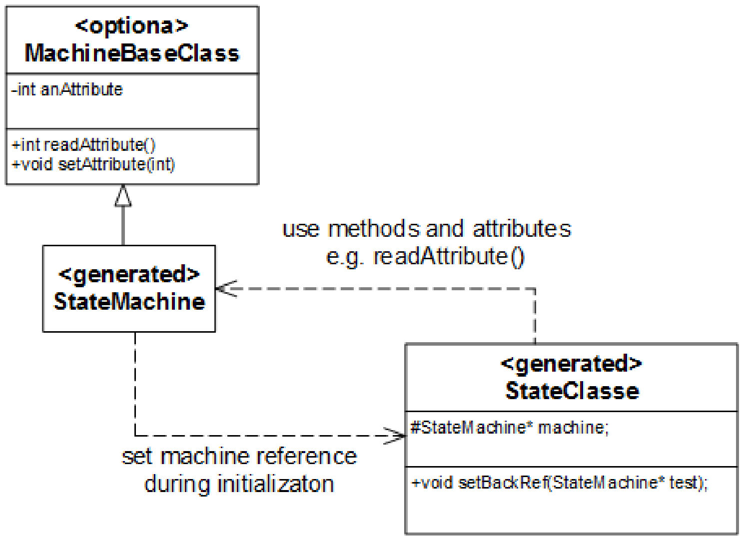


Figure 3.19.: Class diagram of the generated state machine classes if parameter `BackrefToMachineInStateClasses=yes`. Factory class not shown here
.

3.4.2. Regions

Regions are supported since version 3. See 3.2.2 for more information.

3.4.3. State Machine Destruction

In embedded systems, objects are often created at startup and not deleted (at power-down). Therefore, no destructor is generated by default. If your system is more dynamic and objects are created and deleted at runtime, it is necessary to delte the state classes. There are several ways to do this:

- Write a destructor for the factory class and delete the created state classes there
- Set the keyword `StateMachineClassHasDestructor` to `yes` to create a non virtual destructor for the state machine class.
- Set the keyword `StateMachineClassHasDestructor` to `virtual` to create a virtual destructor for the state machine class. Create a virtual destructor if a base or subclass of the state machine class exists and *delete* is called on the parent and not the subclass of the two classes. See the example code below:

```
class testcase: public MachineBase
{
public:
        testcase(void);
        testcase(testcaseFactory* ptr);
        virtual ~testcase();
        ...
}

int main(int argc, char* argv[]){
  testcase* machine=new testcase(new ownTestcaseFactory());
  ...

  // creates a memory leak if the dtor of the state
  // machine and the base class are not virtual.
  delete (MachineBase*)machine;
}
```

3.4.4. How to define the processEvent method's signature

It is possible to flexibly configure the signature of the event handler method. The following list shows the different options for a machine name 'testcase':

- Method only with the event parameter

Signature:	`int processEvent(const TESTCASE_EVENT_T msg);`
Set the following parameter:	`CppHsmFunctionWithEventParameter=yes` `CppxTemplateUse=no`

- Method with event and template parameter

Signature:	`template <typename T>` `int processEvent(const TESTCASE_EVENT_T msg, const T& userdata)`
Set the following parameters:	`CppHsmFunctionWithEventParameter=yes` `CppxTemplateUse=yes`

- Method template parameter that contains also the event

Signature:	`template <typename T>` `int processEvent(const T& userdata)`

Set the following parameters:	`CppHsmFunctionWithEventParameter=no`
	`CppxTemplateUse=yes`
	`CppxTemplateParameter=const T& userdata`
	`CppxTemplateMsgAccessor=TESTCASE_EVENT_T msg=userdata.msg;`

- Method without any parameter

Signature:	`int processEvent(void);`
Set the following parameters:	`CppxTemplateUse=no` `CppHsmFunctionWithEventParameter=no`

With the two parameters `ReturnAndProcessEventTypeOfStateMachine` and `ReturnAndProcessEventTypeOfStateMachineIsBool` the return type of the `processEvent()` method can be influenced.
If `ReturnAndProcessEventTypeOfStateMachineIsBool=yes`, `bool` is choosen as return type. In this case the information whether a normal event or a conditional event was processed is not available anymore to the caller.

3.4.5. Validaten Handler, Unknown State Handler and Tracing the Event Flow

With the parameter *ValidationCall=yes* it is possible to automatically insert method calls that allow to check if a transtion is allowed or simply to trace state changes. The validation method has to be implemented by the user e.g. in a base class.

With the parameter *UnknownStateHandler=unknownStateHandler();* it is possible to insert a method that is called in case no fitting state could be found (e.g. due to memory corruption).

And finally the code generator can be called with the *-Trace* command line parameter which then adds methods that allows to trace the event flow e.g. for debugging, test or validation purposes.

The following first code snippet exemplarily shows generated code for validate handler.

```
void StateMachineBlinkBase::validationHandler(const States from, const States to) const{
  cout << "Change from: " << getNameByState(from) << " to " << getNameByState(to) << endl;
}
```

And then how the generated code looks like.

```
...
switch (stateVars.stateVar) {
  case State_LedBlink:
    if(msg==EVENT_CHANGE_MODE){
      /* Transition from State_LedBlink to State_LedOn */
      validationHandler(State_LedBlink, State_LedOn);
      /* adjust state variables */
      stateVarsCopy.stateVar = State_LedOn;
      StateMachineBlinkTraceEvent(0U);
```

```
    }else if(msg==EVENT_UPDATE){
      /* Action code for inner transition */
      led_blink();
    }
  break; /* end of case State_LedBlink */

  case State_LedOff:
    if(msg==EVENT_CHANGE_MODE){
      /* Transition from State_LedOff to State_LedBlink */
      validationHandler(State_LedOff, State_LedBlink);
      /* adjust state variables */
      stateVarsCopy.stateVar = State_LedBlink;
      StateMachineBlinkTraceEvent(0U);
    }

  default:
    unknownStateHandler();
  break;
}
...
```

3.4.6. Separate generated from non-generated Code

Even if the state machine is fully generated this is usually only a smaller part of your application whereas the lager part is coded manually. For several reasons it is important to clearly separate generated code from non-generated code. Using the features of C++ the code generator offers several possibilities to achieve this.

- The most basic method is to put hand written code into libraries and call the library from within the state machine.
- Generated classes can also subclass non-generated classes (base class of StateMachine or StateClass). Such base classes can contain useful methods that can be called from within the generated subclasses.
- Using the StateMachineFactory user provided state classes can be created and used from the StateMachine class.
- Hand written code is located in a child class of the state machine. I.e. the state machine classes are parts of other classes.

3.4.7. Realising Active Objects in C++

An active object is an object with its own thread of control. A common design pattern is to design an application as a number of active objects. Active objects usually interact through an asynchronous event exchange. The received events are then processed in a state machine (here in `processEvent()`). The machine might react with sending events back or to another active object.

A very good introduction on ActiveObjects can be found in *Pattern-Oriented Software Architecture, Vol. 2 published by WILEY*. This book describes also a number of other interesting patterns that are usually used to design concurrent (realtime) applications.

As nowadays realtime operating systems are still mostly written in C-code and do not provide a C++ interface for tasks, queues, timers etc. a typical problem is how to wrap a task (or thread) into a C++ class so that the thread body can access methods of the C++ class (e.g. the processEvent method). The commonly used approach is as follows:

- Define a static member function with a signature of underlying RTOS task (e.g. `void tasks(void* thisPtr)`)
- Usually it is possible to handover a parameter to the task creation function which is then available in the task body. Provide the `this` pointer as parameter.
- Cast the `this` pointer in the task body back to the object which owns the task.

The following listing shows an example based on the Posix pthreads library. The static method `doWork()` has the signature as needed for a Posix thread. The same procedure works also for other realtime operating systems (e.g. RTEMS).

```
// file ActiveMachine.h

class ActiveMachine : public SomeStateMachine
{
public:
  int start(void);
  void stop(void);

private:
  static void* doWork(void *thisPtr); // thread body
  TUSIGN8 shouldRun(void);
};



// file ActiveMachine.cpp

void* ActiveMachine::doWork(void *ptr){
   ActiveMachine* mePtr = (ActiveMachine*)ptr;
   printf("Hello World! It's me\n");

   while(mePtr->shouldRun()){
    // wait for event
    // execute state machine's processEvent(msg)
   }

   pthread_exit(NULL);
}


int ActiveMachine::start(void){
  pthread_t threads;

  int rc;

  rc = pthread_create(&threads, NULL, &ActiveMachine::doWork, (void *)this);
  if (rc){
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    return -1;
  }
  return 0;
}
```

3.4.8. Virtual Create Methods

Sometimes it might be required to initialise the state objects from the factory. In this case the create functions in the factory can be generated as virtual functions. So they can be overloaded in a subclass of the factory. Set the `CreateFactoryMethodsVirtual` flag to `yes` to instruct the code generator to generate virtual methods.

3.5. Generating SCC

Since version 2.0 SinelaboreRT supports the generation of SCC as output format. SCC is a simple XML format used from the built-in visual editor to store its model information. To use a model file given in SCC format use the following parser option: `-p SCC`. To generate SCC from a file exported by any of the supported UML modeling tools use the following language option: `-l SCC`.

Example to create a ssc xml file from a Cadifra input file:

```
java -jar codegen.jar -p CADIFRA -l ssc -o model inputModel.ccd
```

In the next step we can edit the generated ssc file in the codegen's editor.

```
java -jar codegen.jar -p ssc -E -o model.xml model.xml
```

3.6. Generating C# Code

3.6.1. Introduction

Since version 3.3 sinelabore*RT* supports the generation of C# code again. If you used the C# generator before (version 1.7.1) upgrade to the latest version.

To generate C# code call the code generator with the new command line flag `'-l csharp'`. All states are created into one source file. The file name is determined by the `'-o'` command line switch. An optional namespace can be provided in the `codgen.cfg` file as well as the following C# specific parameters:

- SeparateStateClasses: If set to yes separate state classes are generated. The entry/do/exit code from the state diagram is copied over into methods and called from the generated code.
- BaseClassStates: Allows to define an own base class for the generated state classes. Generated code example:

```
namespace MyNamespace
{

  public class S3 : StateBase{
    public S3()
    {
    }
    public virtual void Entry()
    {
      Console.Write("Enter S3\n");
    }
    public virtual void Exit()
    {
      Console.Write("Exit S3\n");
    }
    public virtual void Action()
    {
      Console.Write("Action S3\n");
    }
     ...
```

- BaseClassMachine: Allows to define an own state machine base class. This is useful if you have longer action code that you want to place into own methods and call them from within the generated code.
- AdditionalLocalMachineVars: Allows to provide own code inserted at the beginning of the state machine handler method.

```
public class testcase : BaseMachine
 {
    ...

    public int ProcessEvent(Events msg){

      ...

      //AdditionalLocalMachineVars goes here

      ...
```

- Possibility to hand over an object with user defined type

```
  public int ProcessEvent(UserData userData){

    int evConsumed = 0;

    // event handler with user data
    // Set the following parameters:
    // UseEventObject = yes
    // EventObjectType=UserData
    // AdditionalLocalMachineVars=Events msg = userData.msg; // access event parameter
    Events msg = userData.msg; // access event parameter
```

The generated code does not follow the state pattern as you might expect (if you are familiar with common design patterns). The reason is that the machine code is completely generated and no hand-coding is involved. The following figure 3.20 shows the structure of the generated code. The classes marked with <generated>are generated from the code generator. Classes marked with <optional>are optional and must be provided by yourself if needed.

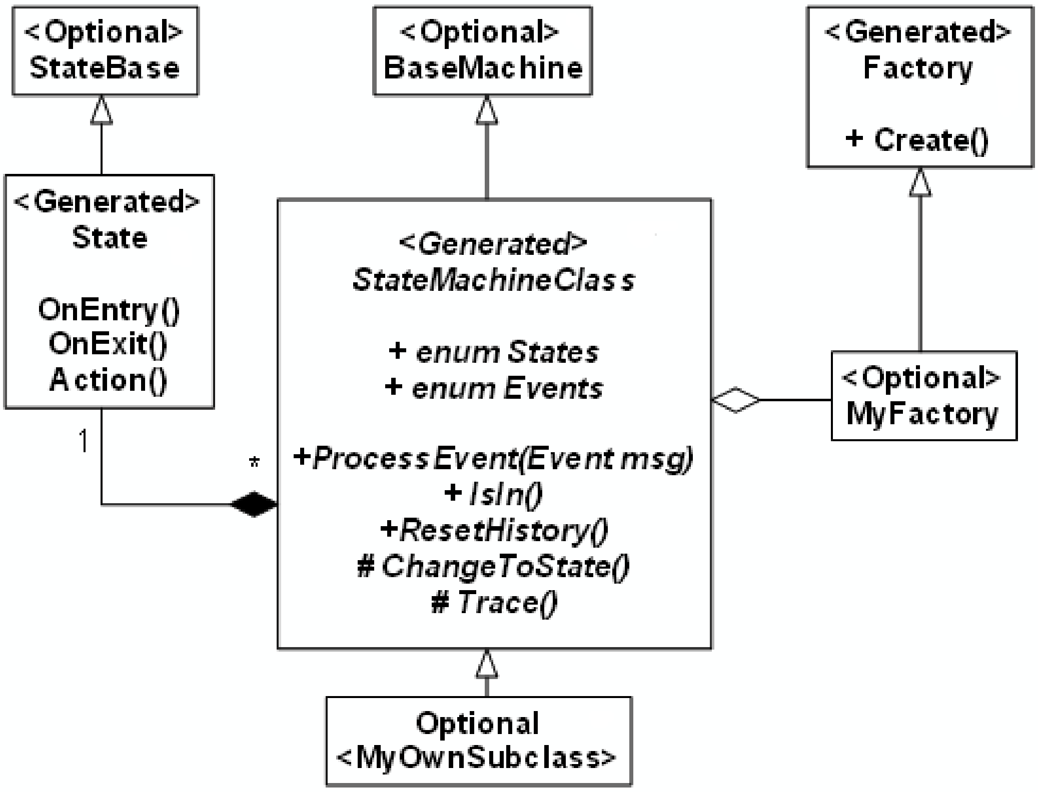


Figure 3.20.: Class diagram of the generated state machine classes and optional base classes.

- The `StateMachine` class realises the state machine as modelled in the state diagram. The name of the class (here StateMachineClass) can be defined with the command line flag `-o`. The `Initialise()` method must be called once to initialise the machine (i.e. to set default states ...). After initialisation the `ProcessEvent()` method can be called with the actual event as parameter. Methods to reset the history of a composite state and to check in which state the machine is are available too. It is possible to specify a base class of the machine (here shown as `BaseMachine`).

- Beside some helper methods the code generator generates `ChangeToState` and the `Trace` methods for you too if the appropriate configuration options are set. To provide own code create a derived class from the state machine class and overwrite the methods according to your needs.

- For each state defined in the state chart diagram optionally a class is created in the state machine class file (same namespace) with the methods `OnEntry()`, `OnExit()` and `Action()`. The state classes are named like the states in the state chart diagram. It is possible to specify a base class (here shown as `StateBase`) the states should be derived from. You can also create subclasses of these state classes. Overwrite the needed methods in the factory in this case.

- A `Factory` is generated in the `StateMachine` class file (same namespace). The factory separates the construction of the state classes from the machine class but let the factory decide which state class to instantiate. An example for an own simple factory is shown below. It creates S1 and S2 on its own but lets the state machine factory generate all the other state classes.

```
public class myFactory : testcaseFactory
{
    public override S2 CreateS2()
    {
        return new myHandwrittenS2();
    }

    public override S1 CreateS1()
    {
        return new myHandwrittenS1();
    }
}
```

3.6.2. Supported / Unsupported

The C# backend of the code generator has the following features and limitations.

The supported elements are as follows:

- Hierarchical states
- Entry/Exit/Do Activities of states
- (Signal-)Events with event name, guard and action
- Initial and final pseudo-states
- History states
- Choices

The unsupported elements are:

- Constraints
- Sync-states and junctions
- Entry and exit points (not to compare with entry/exit actions within states)
- Regions
- Submachines
- Terminate and Fork/Join

3.7. Generating Java Code

3.7.1. Introduction

To generate Java code call the code generator with the command line flag `'-l java'`.

To generate a configuration file with all parameters related to the Java code generation call the code generator as follows once: `java -cp "path to bin folder" codegen.Main -l java -gencfg > codegen.cfg`

The generator generates just one Java class which implements the complete state machine. This has the benefit that your Java project does not become bloated with all kinds of helper classes. If required an optional base class can be specified in the config file. Also the package (Namespace=...) can be defined there. The events that can be sent to the machine are defined in a public enumeration.

3.7.2. Regions

Not yet supported in this back-end.

3.7.3. Typically needed Configuration Parameters

Use the configuration parameter `Namespace` to set the package the generated Java code is in. Example: `Namespace=package oven.Model;` Use the configuration parameter `BaseClassMachine` to set the base class. Example: `BaseClassMachine=ReactorBase`

3.7.4. Example Usage

The Java back-end was used to define the logic behind the check button of the visual editor. This logic was designed as state machine and controls if the 'save' and 'save as' buttons are enabled or disabled. This depends on a configuration flag and several events sent from other parts of the code. The configuration flag `SaveCheckedOnly` determines if the machine is in the state '`SaveAllowedAlways`' or in '`SaveOnlyIfErrorFree`'. Within the latter state two sub-states define when the save buttons are enabled or disabled. Several transitions are going back and forth between these two states. In a base class the methods to actually access the buttons were implemented. The code generated from this design is directly used in the code generator.

The state machine and parts of the generated code are shown in the following figures.

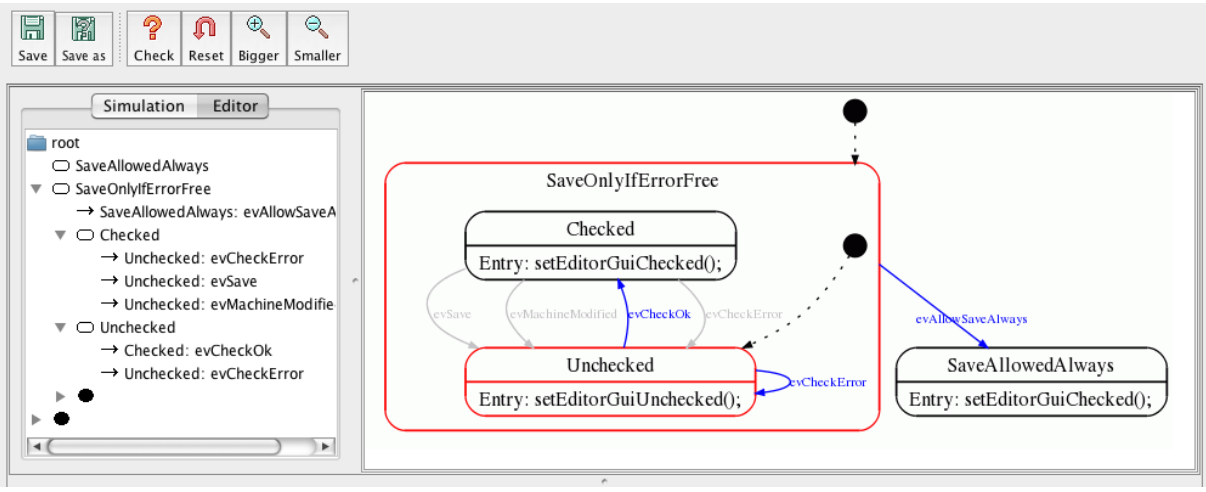


Figure 3.21.: State machine controlling the activation / deactivation of the two save buttons depending on several conditions.

```
public class CheckLogic extends CheckLogicBase
{

  public enum States {
    Unchecked,
     ...
```

```
  }

  public enum Events {
    evSave,
    evCheckOk,
     ...
  }

   ...

  States stateVar;
  States stateVarSaveOnlyIfErrorFree;

   ...

  public void initialize(){
       ...
    }
  }


  public int processEvent(Events msg){

    int evConsumed = 0;

    if(m_initialized==false) return 0;


    switch (stateVar) {

      case SaveAllowedAlways:
      break; /* end of case SaveAllowedAlways */

      case SaveOnlyIfErrorFree:

        switch (stateVarSaveOnlyIfErrorFree) {

          case Checked:
            if(msg==Events.evCheckError){
              /* Transition from Checked to Unchecked */
              evConsumed=1;

              /* OnEntry code of state Unchecked */
              setEditorGuiUnchecked();

               ...

      default:
        /* Intentionally left blank */
      break;
    } /* end switch stateVar_root */
    return evConsumed;
  }
}
```

Listing 3.1: "Automatically generated Java class from the state diagram shown above. "

The state machine can be used the following way:

```
CheckLogic checkLogic=new CheckLogic();
checkLogic.initialize(); // init state machine
checkLogic.processEvent(CheckLogic.Events.evAllowSaveAlways);
```

3.8. Generating Swift Code

Please note that the Swift back-end is work-in-progress. Your feedback is highly welcome!

Swift is a new object-oriented programming language for iOS and OS X development. To generate Swift code call the code generator with the command line flag `−l swift` . To generate a configuration file with all parameters related to the Swift code generation call the code generator as follows once: `java −jar codegen.jar −l swift −gencfg > codegen.cfg` .

The events that can be sent to the machine are defined in a public enumeration.

The generator generates just one Swift class which implements the complete state machine. This has the benefit that your Swift project does not become bloated with all kinds of helper classes. This means that the generated code does not follow the usual state pattern as you might expect (if you are familiar with common design patterns). The reason is that the machine code is completely generated and no hand-coding is involved.

3.8.1. Separate generated from non-generated Code

Even if the state machine is fully generated this is usually only a smaller part of your application whereas the lager part is coded manually. For several reasons it is important to clearly separate generated code from non-generated code. Use one of the following possibilities to achieve this.

- The most basic method is to put hand written code into libraries and call the library from within the state machine.
- Generated classes can also subclass non-generated classes (base class of StateMachine). Such base classes can contain useful methods that can be called from within the generated subclasses.
- Hand written code is located in a child class of the state machine. I.e. the state machine classes are parts of other classes.

3.8.2. Supported state machine features

- States and sub-states
- Deep – and flat hierarchy
- Entry, Exit and Action code of states
- Regions are supported and implemented as sub-functions called from the main state machine handler.
- Option to define state machine signature
- Choice pseudo-states

3.9. Generating Python Code

3.9.1. Introduction

To generate Python code call the code generator with the command line flag `'-l python'`.

To generate a configuration file with all parameters related to Python call the code generator as follows once: java −cp "*" codegen.Main −l python −gencfg > codegen.cfg

The generator produces a Python class that implements the entire state machine. This approach offers the advantage of keeping your Python project clean without cluttering it with various helper classes. Additionally, you have the option to specify an optional base class in the configuration file if needed.

To get started, you can find the microwave oven project introduced in the earlier section. It's available at this GitHub link. This project is fully functional and includes a graphical user interface (GUI) implemented with Tkinter. The GUI enables you to send events to the state machine and observe its reactions.

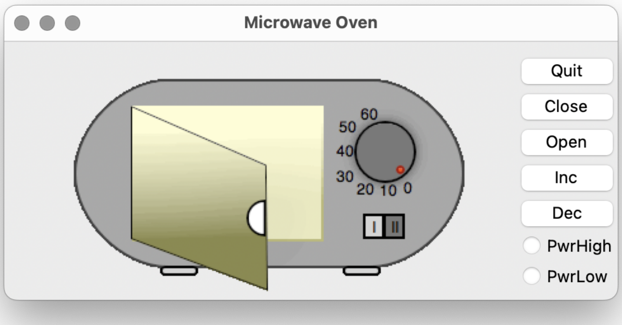


Figure 3.22.: Tkinker oven frontend. Use the buttons to send events to the state machine

3.9.2. Supported state machine features

- States and sub-states
- Deep – and flat hierarchy
- Entry, Exit and Action code of states
- Choice pseudo-states
- Transitions with event, guard and action

3.9.3. Unsupported state machine features

- Regions
- Submachines

3.10. Generating Lua Code

Please note that the Lua back-end is work-in-progress. Your feedback is highly welcome!

Lua is a powerful, efficient, lightweight, embeddable scripting language (http://www.lua.org/). To generate Lua code call the code generator with the command line flag `-l lua` .

The generator generates one Lua module which implements the complete state machine. For a simple state machine with 4 states and 2 events the generated module looks like this:

```
function testcase:new()
        local new_inst = {}
        setmetatable( new_inst, testcase)

        -- machine states
        new_inst.states = {
                S1="S1",
                S11="S11",
                S12="S12",
                S3="S3",
                __UNKNOWN_STATE__="__UNKNOWN_STATE__"
        }

        -- machine events
        new_inst.events = {
                evB="evB",
                evA="evA",
                TESTCASE_NO_MSG="TESTCASE_NO_MSG"
        }

        -- Set state vars to default states
        new_inst.stateVar = new_inst.states.S1
        new_inst.stateVarS1 = new_inst.states.S11 -- set init state of S1

        new_inst.init=false
        return new_inst
end

function testcase:processEvent(Event)
 ...
return evConsumed;
end

return testcase
```

The statemachine can be used as follows:

```
testcase = require "testcase"

local testcase1 = testcase:new()
Event = {event = testcase1.events.TESTCASE_NO_MSG, condition=false};
testcase1:processEvent(Event)

Event.event=testcase1.events.evA;
testcase1:processEvent(Event)
```

3.10.1. Supported state machine features

- States and sub-states
- Deep – and flat hierarchy
- Entry, Exit and Action code of states
- Regions are supported and implemented as sub-functions called from the main state machine handler.
- Choice pseudo-states

3.11. Generating Rust Code

Please note that the Rust back-end is work-in-progress. Your feedback is highly welcome!

Rust a is a relatively new programming language aiming to guarantee memory-safety and thread-safety. Rust is fast and memory-efficient with no runtime or garbage collector. More information about Rust is available at https://www.rust-lang.org

To generate Rust code call the code generator with the following command line flag `–l Rust –o test`.

The generated code is using basic Rust features like `match`, `if/then/else` and can easily be understood.

The output name defines the generated state machine file name (Note the lower case 't' to follow Rust convention), the name of the state machine type and the prefix for the event and state enums.

The simplified code looks as follows:

```rust
// states of the state machine
pub enum TestStates{
    S1,
    ...
}

// Events that can be sent to the state machine
pub enum TestEvents {
    Ev1,
    ...
    TestNoMsg,
}

    #[macro_export]
    macro_rules! def_fsm{
        ($data_t:ty , $timer_service_t:ty $(,$element:ident: $ty:ty = $ex:expr)*) => {

            // Test struct definitions ... comes here
            pub struct Test {
                pub is_init: bool,
                pub state_var: TestStates,
                ...
                /* Start of user defined attributes */
                $($element: $ty),*
                /* End of user defined attributes */
            }

            impl Default for Test {
                fn default() -> Self{
                    Test {
                        is_init : false,
                        state_var : TestStates::S1, /* Set init for top level state */
                        ...
                        /* Start of user defined attributes */
                        $($element: $ex),*
                        /* End of user defined attributes */
                    }
                }
            }

            impl Test{
                pub fn handle_event(&mut self, ev: TestEvents,
                    data:$data_t, timer_service:$timer_service_t
                    ) -> usize {
                        // generated state machine handler code
                        ...
                    }
                }
        };
}
```

The generator uses the macro features of Rust. A concrete type of a machine must be defined by using a `def_fsm` macro call before the machine can be used.

```
//def_fsm!(UserData, &mut crate::Timers<TestEvents>, x:i32=99, y:i32=99);
def_fsm!(UserData, &mut crate::Timers<TestEvents>);

let mut timers = Timers::new();
let some_data = UserData {guard1 : true} ;

let mut myfsm : Test = Default::default();

myfsm. initialize (some_data, &mut timers);

// sending events to the machine
myfsm.handle_event(TestEvents::Ev1, some_data, &mut timers);
myfsm.handle_event(TestEvents::Ev2, some_data, &mut timers);
myfsm.handle_event(TestEvents::Ev3, some_data, &mut timers);
```

The first macro parameter is a user defined type used as a parameter of the state machine handler. It is available inside the event handler with variable name `data` and type `UserData`.

The intended purpose is to provide a way for the user to hand over any kind of data to the state machine handler function which can then be used in the state machine. A simple use case is to hand over variables that acts as guards in state transitions.

The other intended purpose is that the `UserData` type must provide a log function of a predefined signature in case the state machine was generated with the trace command line parameter `-Trace`.

If tracing is enabled log calls are automatically added to the code such as

```
data.log("S1".to_string(), "S3".to_string(), "Ev1".to_string());
```

The UserData type has to implement the log function e.g as follows:

```
pub fn log(&self, from:String, to:String, event:String){
    println!("Trace: from:{}, to:{}, event:{}", from, to, event);
}
```

The second macro parameter allows to hand over another user defined type intended to realise a timer service. Whether the state machine needs and uses a timer service and how it is implemented is totally in the user's hand. A microwave oven example is provided in the samples folder. It implements a basic timer service for realising the timeout for the cooking time.

Note: presently there is no possibly to avoid the handover of the `UserData` and `Timer` struct. This might change in later versions.

All further parameters are optional and simply added to the state machine struct. In the code above an example with parameters `x` and `y` is shown.

3.11.1. Regions

Regions are fully supported by the code generator. See 3.2.2 for more information how regions are implemented in general. In Rust an internal clone is crated at runtime which is then becoming the new state at the end of the state handler.

3.11.2. Supported state machine features

- Flat or hierarchical state machines
- Deep – and flat history
- Entry, Exit and Action code of states
- Choice pseudo-states
- Final states
- Transitions with event, guard and action
- Regions

3.11.3. Unsupported state machine features

- Submachines

3.12. Generating GO Code

3.12.1. Introduction

To generate GO code, call the code generator with the command line flag `'-l go'`.

To generate a configuration file with all parameters related to the GO code generation, call the code generator as follows once: `java -cp "path to bin folder\*" codegen.Main -l go -gencfg > codegen.cfg`

The generated code does not follow any state pattern as you might expect (if you are familiar with common design patterns). This is because the machine code is completely generated and no hand coding is involved.

In the generated GO code, state hierarchies are mapped to nested switch/case code. Event handling code is mapped to if/else-if/else structures. There are several configuration/command line options to adjust the generated code. Before calling the event handler function , call the init function to initialise the state machine and run all the state's entry code once.

Example:

```
events := []MachineEvent{Ev1, EvFinal}

sm := Machine{}
sm.Init()

for _, event := range events {
    _, err := sm.ProcessEvent(event)
```

Configuration file options:

- `BaseClassMachine`: GO supports the language concept of embedding of structs to express a more seamless composition of types. With the `BaseClassMachine` parameter the parent's structure Type (embedding) can be specified. The nice thing in GO is the possibility to access structure members or functions of the embedding structure in the generated state machine code. This feature is used for example in the oven tutorial example to provide access to a Timer service and to store some data (timer IDs).
- `Namespace`: The specified value is used as package name of the generated code.
- `ValidationCall`: With the parameter *ValidationCall=yes* it is possible to automatically insert method calls that allow to check if a transition is allowed or simply to trace state changes. The validation method has to be implemented by the user usually as function of the structure that embeds the state machine (see oven example in GO).

3.12.2. Regions

Regions are supported. See 3.2.2 for more information.

3.12.3. Error Handling

GO has a built-in error type. Several generated functions uses error values to indicate an abnormal state.

3.12.4. Tracing the Event Flow

The code generator can be called with the *-Trace* command line parameter which then adds methods that allows to trace the event flow e.g. for debugging, test or validation purposes.

The following first code snippet exemplarily shows generated code for validate handler.

```
// called by the state machine if tracing is enabled
func (smBase *Reactor) TraceEvent(traceIdx int) {
  var traceString = smBase.Statemachine.TraceLookup(traceIdx)
  debugPrint(fmt.Sprintf("Trace msg: %s\n", traceString))
}

// called by the state machine if validation hander is enabled
func (smBase *Reactor) ValidationHandler(from OvenState, to OvenState) {
  toString, _ := smBase.Statemachine.StateToString(to)
  fromString, _ := smBase.Statemachine.StateToString(from)
  debugPrint(fmt.Sprintf("Change state from %s to %s\n", fromString, toString))
}
```

And then what the generated code looks like. Line 4 shows how the validation function is called. And line 21 shows the trace call. There are utility functions to convert both the state and event numbers to text strings. into text strings.

```
1  ...
2  if msg == EvError {
3    /* Transition from Super to Error*/
4    smBase.ValidationHandler(Super, Error)
5    eventConsumed=1
6
7    /* Exit code for regions in state Super*/
8
9    if(sm.stateVarMainRegion==Cooking){
10     smBase.Timers.Stop(smBase.idBlink)
11     fmt.Println("LED Off")
12
13   }else {
14     /* Intentionally left blank */
15   };
16
17
18   smCopy.stateVar = Error;/* Default in entry chain */
19   smCopy.stateVarError = Error1;/* Default in entry chain */
20
21   smBase.TraceEvent(3)
22 }
23 ...
```

3.12.5. Realising Active Objects in GO

An active object is an object with its own thread of control. A common design pattern is to design an application as a number of active objects. Active objects usually interact through an asynchronous event exchange. The received events are then processed in a state machine (here in `processEvent()`). The machine might react with sending events back or to another active object.

A very good introduction on ActiveObjects can be found in *Pattern-Oriented Software Architecture, Vol. 2 published by WILEY*. This book describes also a number of other interesting patterns that are usually used to design concurrent (realtime) applications.

GO provides excellent out-of-the-box functionality to implement active objects using only the built-in language features and standard library types. No additional classes or libraries are required. GO routines with the `select` and `channel` concepts are the basis for creating active objects.

3.12.6. Supported state machine features

- States and sub-states
- Deep – and flat hierarchy
- Entry, Exit and Action code of states
- Choice pseudo-states
- Transitions with event, guard and action
- Regions
- Final states

3.12.7. Unsupported state machine features

- Submachines

3.13. Generating JavaScript Code

The JavaScript documentation is available online here: https://www.sinelabore.de/doku.php/wiki/backends/javascript

3.14. Generating Doxygen Documentation

Doxygen (http://www.stack.nl) is a popular tool to generate different kinds of documentation from annotated source code. Doxygen also allows to embed dot based descriptions of graphs in the source code. The graph visualisation is then integrated in the generated html documentation.

With the command line option `-doxygen` the code generator embeds a dot based diagram of the UML state machine into the generated state machine source file. An example of the documentation for the microwave machine is shown in figure 3.23.

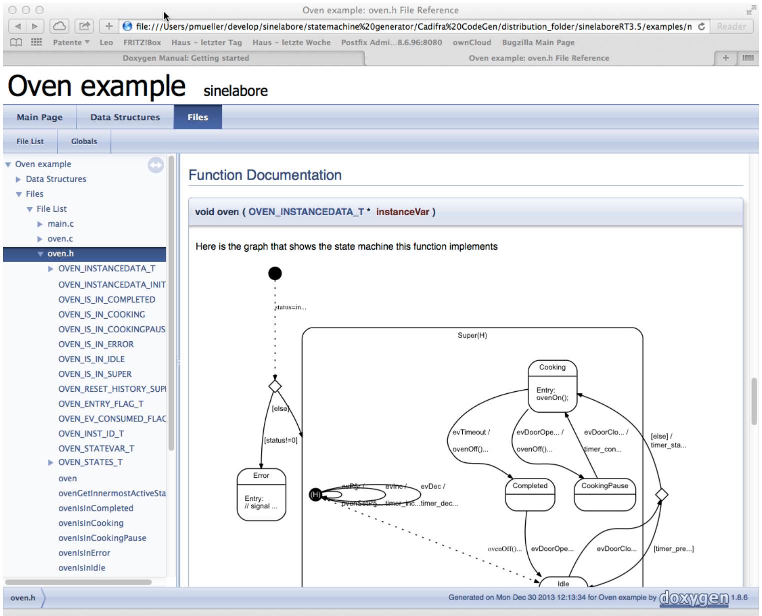


Figure 3.23.: Doxygen generated source code documentation of the microwave-oven example.

3.15. Robustness of UML State Machines

3.15.1. Introduction

Quality assurance is one of the most essential aspects in the development of embedded real-time systems. One possibility to ensure quality is to perform code reviews but they are time-consuming and very dependent on the reviewer. Therefore it is very attractive to find possibilities for automated error prevention and detection i.e. a software performing robustness checks. For UML state machines the OMG has specified a set of wellformedness rules within the UML specification [2]. These rules as well as a number of additional rules are performed by the codegen before starting the real code generation.

3.15.2. Syntactical Robustness Rules

The following set of rules test the syntactical correctness of the state machine model.

- State names must be unique
- State names must not contain spaces
- Composite states should have more than one child state. If only one child state is defined the composition is superfluous and just creates unnecessary complexity in the generated code.
- A state follows the naming conventions
- Final states must not have outgoing transitions
- Transitions must have a source and a target state
- Transitions must have an associated event (excluding transitions from initial states and transitions starting from choice states)
- A transition follows the naming conventions
- If there are two or more transitions starting from the same state and triggered from the same event, then their guards must not be true at the same time (see figure 3.25b and c. The code generator just checks that guards are present. It can't check if the guards are ambiguous (e.g. guard one is `i<4` and guard two is `i>0`);
- Transitions triggered by the same event leave a child and its parent. This is not a problem because a transition has higher priority than another one if its source state is a sub-state of the source of the other one. Make sure that this is what you want (unambiguous specification). See figure 3.25a for an example.
- Inter level transitions from states at level three (state is child of two parents) are not allowed (high complexity)
- A choice must have only one incoming transition
- A choice state should have at least two outgoing transitions otherwise it is useless and should be replaced with a normal transition.
- Every outgoing transition from a choice must have a guard defined
- A junction must have only one outgoing transition
- Outgoing transitions of junctions must not have triggers or guards defined.
- A junction must have a trigger for each incoming transition
- One default transition must be specified for a choice state - i.e. the guard is defined as 'else' (default from choice).
- States should have not only incoming or even no transitions at all (isolated states). Those states are superfluous and just waste space. See figure 3.24 left side. State `S12` has no incoming transition and is therefore not reachable.

- States must be connected by a sequence of transitions outgoing from an initial state (connectivity). Else states are superfluous as they will never be entered. See figure 3.24 right side. In this case every state has at least one incoming transition but still `S12` and `S23` are not reachable.
- Initial states must be defined on every state hierarchy and must have exactly one outgoing transition.

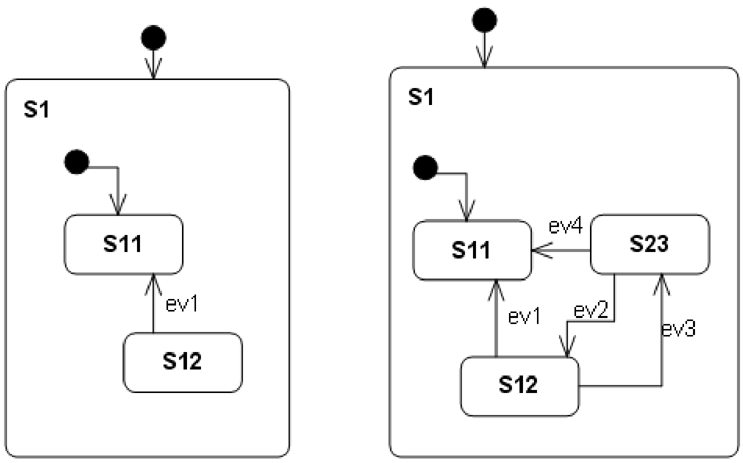


Figure 3.24.: States are not reachable. The code-generator prints a warning during code generation.

Running the code-generator with the startchart from figure 3.24 (right machine) creates the warnings as shown below. The states `S23` and `S11` does violate the rules for state names as defined in 'codegen.cfg'. The non reachable states `s12` and `S23` are reported twice because different rules detect the problem in this case.

```
Starting robustness tests of state machine ...
State names: ..............
Simple state S23 violates naming conventions as defined in codegen.cfg
Simple state S11 violates naming conventions as defined in codegen.cfg
Machine hierarchy: ........
Default states: ...........
Final states: .............
Connectivity of states: ...
State s12 is not reachable -> check your design.
State S23 is not reachable -> check your design.
State s12 is not reachable -> check your design.
State S23 is not reachable -> check your design.
Transitions: ..............
Choices: ..................
Children in composites: ...
```

In codegen.cfg the prefixes for state names were defined as follows:

```
PrefixSimpleStates=s
PrefixCompositeStates=S
```

3.15.3. Semantical Robustness Rules

It would be highly beneficial to also check semantic rules. For example that there is no overlap in the guard specifications of transitions starting from a choice state. But such checks are not performed yet.

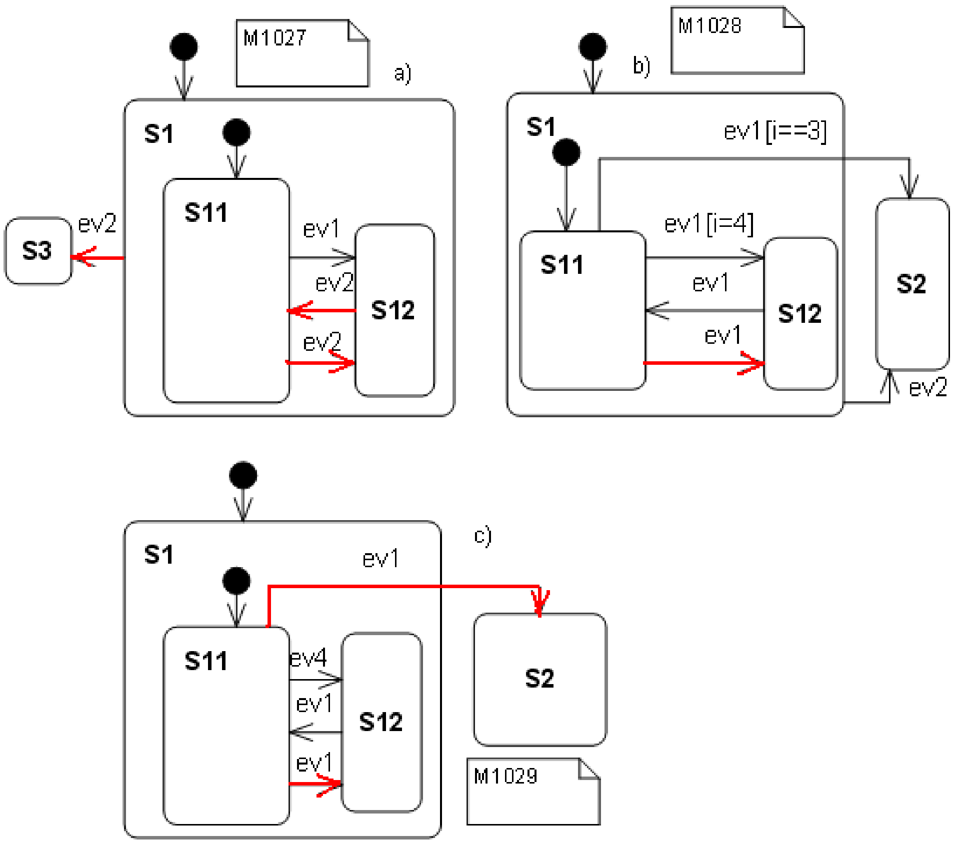


Figure 3.25.: This figure shows several examples where transitions and their guards are not unambiguous. In figures b) and c) transitions triggered by event 'ev1' and starting from state S11 cause the problem. In figure a) transitions triggered by event 'ev2' all have no guard but this is a valid specification because transitions starting from inner states have higher priority.

3.16. Command Line Simulation Mode

If the command line flag `'-s'` (for simulation) is used, the code generator turns into interactive mode after parsing the input file. No code is produced then. After parsing the input file you can type in events and check what the state machine's reaction is[5].

During a simulation step all code that is executed as the reaction to an event is printed out (e.g. the OnEntry code) if the `'-v'` command line flag was set. The simulator does not evaluate guards which you might have defined. It just assumes that a guard would evaluate to true and takes the transition.

In the case of a transition ends in a choice the behaviour is different because the guard information is needed to select the right outgoing transition of the choice. In this case you must also specify the guard of the outgoing transition to take.

Example: One guard of an outgoing transition is `'[i==5]'` and the other one is `'[else]'`. The event to enter the choice is named `'evC'`. To take one or the other outgoing transition type in either `'evC[i==5]'` or `'evC[else]'`.

If you are interested to see the presently active state you can type in `'dump'` instead of an event name. Then the innermost active state is printed.

The following example was done using the 'complex' state diagram from figure 3.2. After startup the simulator initializes the state machine. As result the entry code of the default states is executed.

```
$ java -cp "../bin/" codegen.Main -v -p CADIFRA -s complex.cdd
Command line simulation mode enabled-->No C-code will be generated!
No output file name specified. Using complex.c/.h
Used license file: ....
printf("Outer test action\n");
printf("S12Entry\n");
Enter event name and press return
```

Now you can type in event names followed by a `'return'`. The following is the output for the event `e1`.

```
e1
msg: e1
printf("Outer test action\n");
printf("S12 Action\n");
printf("S12Exit\n");
printf("e1Act\n");
printf("S21Entry\n");
----------------------
```

To find out the present state type in `'dump'` followed by a return.

```
dump
Innermost active state is: S21
```

It is also possible to create a file which contains one event per line and use it als input for the simulator. The command line (in cygwin) for a test file with name `'tst_input.txt'` looks like this:

```
$ java -cp "../bin/codegen.jar" codegen.Main -v -p CADIFRA -s complex.cdd < tst_input.txt
```

The simulator reads event after event and prints out the reaction. If the last line was processed it exits.

3.17. Visual State Chart Simulation and Editor

The description of the graphical editor built into SinelaboreIT is available online here: https://www.sinelabore.de/doku.php/wiki/manual/editor

[5] Please consider the limitations if the model contains regions as described in section 1.3

3.18. State Tables

State tables are just a different representation of state machines. A state table is a table showing what state the machine will move to, based on the current state and the given event and guard. The code-generator automatically generates a state table for your state machine if the '-xls' command line switch is used. The generated output is an Excel table.

The horizontal dimension indicates current states, the vertical dimension indicates next states, and the row/column intersections contain the event which will lead to a particular next state.

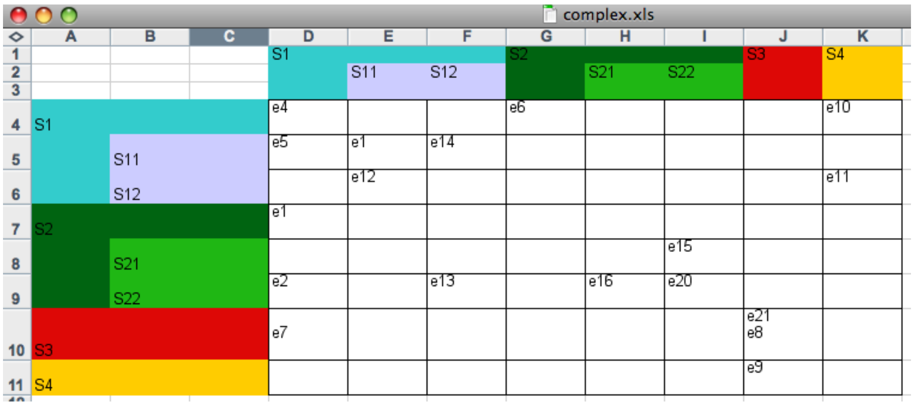
A state table and the state diagram show the same information and can be transformed into each other.

A state table has the great benefit to allow a more systematic check if all transitions were considered in the design. This can be beneficially used in a design review. State tables can also be of great help to create test specification.

> Regions are not yet considered during state table generation.

The following figure shows the state table of the statemachine from figure 3.2. Compare the two representations and build up your own mind which one to use for which purpose.

Examples:
From state S3 the events e21 and e8 lead to state S3. From state S21 the event e16 leads to state S22.



	A	B	C	D	E	F	G	H	I	J	K
1				S1			S2			S3	S4
2					S11	S12		S21	S22		
3											
4	S1			e4			e6				e10
5		S11		e5	e1	e14					
6		S12			e12						e11
7	S2			e1							
8		S21							e15		
9		S22		e2		e13		e16	e20		
10	S3			e7						e21 e8	
11	S4									e9	

Figure 3.26.: State tables are a different representation of a state machine. Example: From S22 event e15 leads to state S21. Even for an example which looks quite complex the state table shows that the only 16 fields are filled out of 64 possible.

3.19. Model–Based Testing of Statemachines

3.19.1. Introduction

Testing is an important but also time consuming part of a project. In general it covers the following steps: (1) building a model, (2) generating test cases, (3) generating expected test results, (4) running the tests, (5) comparison of actual outputs with expected outputs, and (6) decision on further actions (e.g. whether to modify the model, generate more tests, or stop testing).

You can find several articles about testing state machines, but the one written by Martin Gomez [1] nicely summarises the usually used approach even if model based testing gets more and more interest nowadays.

The beauty of coding even simple algorithms as state machines is that the test plan almost writes itself. All you have to do is to go through every state transition. I usually do it with a highlighter in hand, crossing off the arrows on the state transition diagram as they successfully pass their tests. This is a good reason to avoid 'hidden states' - they're more likely to escape testing than explicit states. Until you can use the 'real' hardware to induce state changes, either do it with a source-level debugger, or build an 'input poker' utility that lets you write the values of the inputs into your application.

Written in 2000 this still describes the common practice. The following sections show how the sinelabore code generator supports you in testing state machines and makes testing state machines more efficient. Sinelabore helps you to make a step towards model based testing.

3.19.2. Model

There is some discussion whether the test model should be different from the implementation model. We don't want to discuss this here. Sinelabore can be used in both cases. For the following discussion we assume that the implementation model (i.e. the model you generate code from) is also also used to derive test cases from.

3.19.3. Defining test cases

Defining test cases is another important step that usually consumes a lot of time if it must be done manually. The code generator can save you a lot of time by automatically suggesting test routes through a given state machine. The used algorithm ensures that all transitions are taken at least once (100% coverage)[6]. So it is not anymore necessary to go through the diagram with a highlighter in hand. The following output shows the coverage information from the microwave machine (see folder example 3). *The command line option '-c' or '-c1' switches on the coverage data generation. The first algorithm uses a depth-first tree search algorithm and tries to follow one route as long as possible. This results usually in long but fewer test routes than using -c1. When using -c a breadth-first tree search algorithm is used that produces more but shorter test routes compared to -c.*

```
make
java -jar codegen.jar -c -p CADIFRA -o oven first_example_step3.cdd
Used license file: ...

Transition Coverage:
0:         |From Idle taking event evPwr ending in Idle
1:         |From Idle taking event evInc ending in Idle
2:         |From Idle taking event evDec ending in Idle
3:         |From Idle taking event evDoorClosed ending in Cooking
4:         | |From Cooking taking event evDoorOpen ending in CookingPause
5:         | | |From CookingPause taking event evDoorClosed ending in Cooking
6:         | | | |From Cooking taking event evTimeout ending in Completed
7:         | | | | |From Completed taking event evDoorOpen ending in Idle
Transition Coverage for suggested path(s) 100%
gcc -Wall -g oven.c -c -o oven.o
gcc -Wall -g main.c -c -o main.o
gcc -Wall -g oven_hlp.c -c -o oven_hlp.o
gcc  -o oven oven.o main.o oven_hlp.o
```

[6] Under some circumstances the generator will not be able to generate routes for a 100% coverage

You can see that the first test route starts from the initial state at line 0 and ends at line 7 where no untaken transitions are left to go. The output moves to the right at every new transition on the route that ends in a different state. In this simple example it was possible to trigger all transitions on one test route. For more complex state machines this is usually not possible.

The output for the more complex PLCopen `equivalent function block` (see example one on our webside https://www.sinelabore.de/doku.php/wiki/examples/plcopen_function_block) is given below.

```
Transition Coverage:
0:      |From Idle taking event enable==1 ending in Init
1:      | |From Init taking event a && !b && enable ending in WaitChannelB
2:      | | |From WaitChannelB taking event !a && !b && enable ending in Init
3:      | | | |From Init taking event a && b && enable ending in SafetyOutEnabled
4:      | | | | |From SafetyOutEnabled taking event !a && !b && enable ending in Init
5:      | | | | | |From Init taking event !a && b && enable ending in WaitChannelA
6:      | | | | | | |From WaitChannelA taking event evTimeout ending in Error12
7:      | | | | | | | |From Error12 taking event !a && !b && enable ending in Init
8:      | | | | | | | | |From Init taking event enable==0 ending in Idle
9:      | | | | | | |From WaitChannelA taking event !a && !b && enable ending in Init
10:     | | | | | | |From WaitChannelA taking event a && b && enable ending in SafetyOutEnabled
11:     | | | | | | | |From SafetyOutEnabled taking event a && !b && enable ending in FromActiv
12:     | | | | | | | | |From FromActiveWait taking event evTimeout ending in Error3
13:     | | | | | | | | | |From Error3 taking event !a && !b && enable ending in Init
14:     | | | | | | | | |From FromActiveWait taking event !a && !b && enable ending in Init
15:     | | | | | | | |From SafetyOutEnabled taking event !a && b && enable ending in FromActiv
16:     | | |From WaitChannelB taking event a && b && enable ending in SafetyOutEnabled
17:     | | |From WaitChannelB taking event evTimeout ending in Error12
Transition Coverage for suggested path(s) 100%
```

In this case not all transitions can be tested on one route. There are several routes necessary each starting from the init state. In the output new routes have the same indentation level. E.g. from `Init` three routes start (line 2, 16 and 17). Altogether seven test routes are necessary to achieve 100% transition coverage.

Beside the console output which gives an overview about the test effort an Excel file is created. It contains one route per sheet. Each line is a single test step on the test route starting from the init state. There are as many sheets as routes are necessary to achieve the 100% transition coverage. A sheet lists the present state and the trigger to reach the listed next state. In addition the constraints of the source and the target states are listed. The constraint information is taken from the state diagram if specified (see next section). The following figure `equivalent function block` shows the Excel sheet created for the equivalent machine.

Usually it is necessary to model just the transition that leads to a state change. Transitions that do not lead to a state change can be omitted. But from a testing point of view the latter can be very useful too. Because usually you want to test also if the machine reacts correctly on events which shall not trigger a state change.

3.19.4. Specify constraints (test oracle data) in state diagrams

Constraints can be used to specify the expected output of a state (step 3). This is a valuable input for the tester. The format can be a more formal specification as shown in figure 3.27 which might be used directly in a testbed. Or it can be just informal text providing instructions for a tester. 'Test oracle' is another name sometimes used for this data.

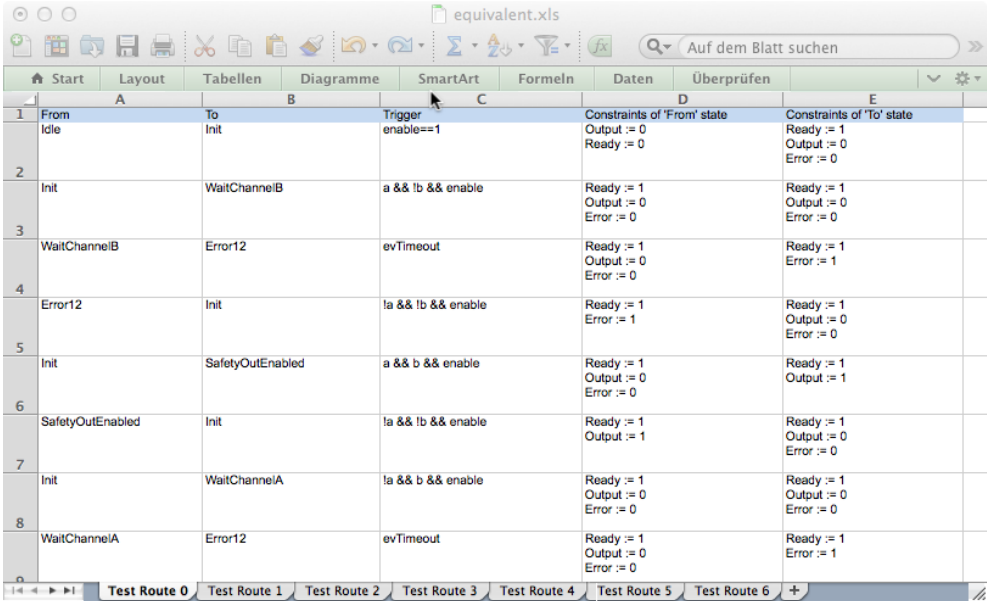
State constraints must be specified within the state diagram. The code generator uses UML comments for that purpose. The comment must be attached to a state (either parent or child state or both). If constraints are specified for both parent and child they are merged together.

The comment follows the same syntax as used to define a state compartment e.g. to define entry/exit ... actions. Use the keyword **'Constraints:'** to start the 'constraints' part. See section 3.28 for more information and the following figure for an example.

For the codegen internal state diagram editor you can directly specify the constraints as the following figure 3.29 shows for the state 'SafetyOutputEnabled'.

3.19.5. Writing the testbed and executing tests

The next step (4) is to write a test-bed that allows you to execute the state machine and to send events to it. Usually test-beds are very hardware specific and different for most embedded



From	To	Trigger	Constraints of 'From' state	Constraints of 'To' state
Idle	Init	enable==1	Output := 0 Ready := 0	Ready := 1 Output := 0 Error := 0
Init	WaitChannelB	a && !b && enable	Ready := 1 Output := 0 Error := 0	Ready := 1 Output := 0 Error := 0
WaitChannelB	Error12	evTimeout	Ready := 1 Output := 0 Error := 0	Ready := 1 Error := 1
Error12	Init	!a && !b && enable	Ready := 1 Error := 1	Ready := 1 Output := 0 Error := 0
Init	SafetyOutEnabled	a && b && enable	Ready := 1 Output := 0 Error := 0	Ready := 1 Output := 1
SafetyOutEnabled	Init	!a && !b && enable	Ready := 1 Output := 1	Ready := 1 Output := 0 Error := 0
Init	WaitChannelA	!a && b && enable	Ready := 1 Output := 0 Error := 0	Ready := 1 Output := 0 Error := 0
WaitChannelA	Error12	evTimeout	Ready := 1 Output := 0 Error := 0	Ready := 1 Error := 1

Figure 3.27.: Test routes to achieve 100% transition coverage are specified in an Excel sheet. Constraints define the expected behaviour in each state. The constraints can be directly specified in the state diagram and are taken over from there.

systems. Therefore the code-generator does not generate test-bed code for you. This is a step that requires manual work.

3.19.6. Analysing the test results

To finally anaylse (steps 5 and 6) the test results the execution must be traced and stored. Partly the trace feature of the code-generator can help here. See the next chapter 3.20 for more information. But mostly test evaluation is also a manual step.

3.19.7. Summary

Gomez [1] says that testing state machines *'require a fair amount of patience and coffee, because even a mid-size state machine can have 100 different transitions. However, the number of transitions is an excellent measure of the system's complexity. The complexity is driven by the user's requirements: the state machine makes it blindingly obvious how much you have to test.*

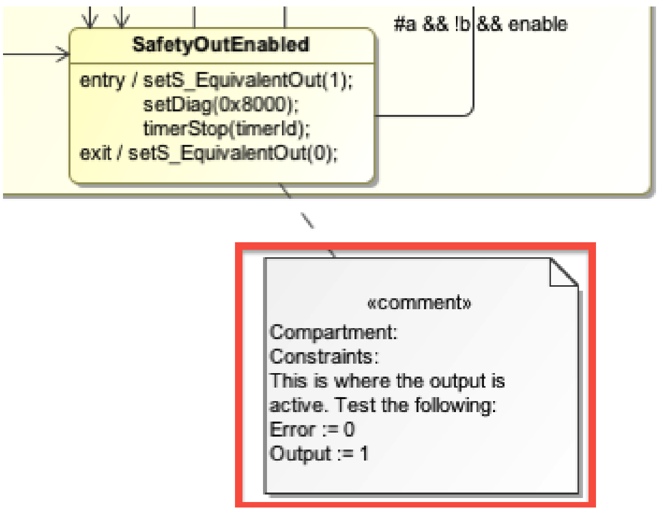


Figure 3.28.: Defining constraints by attaching a note to a state.

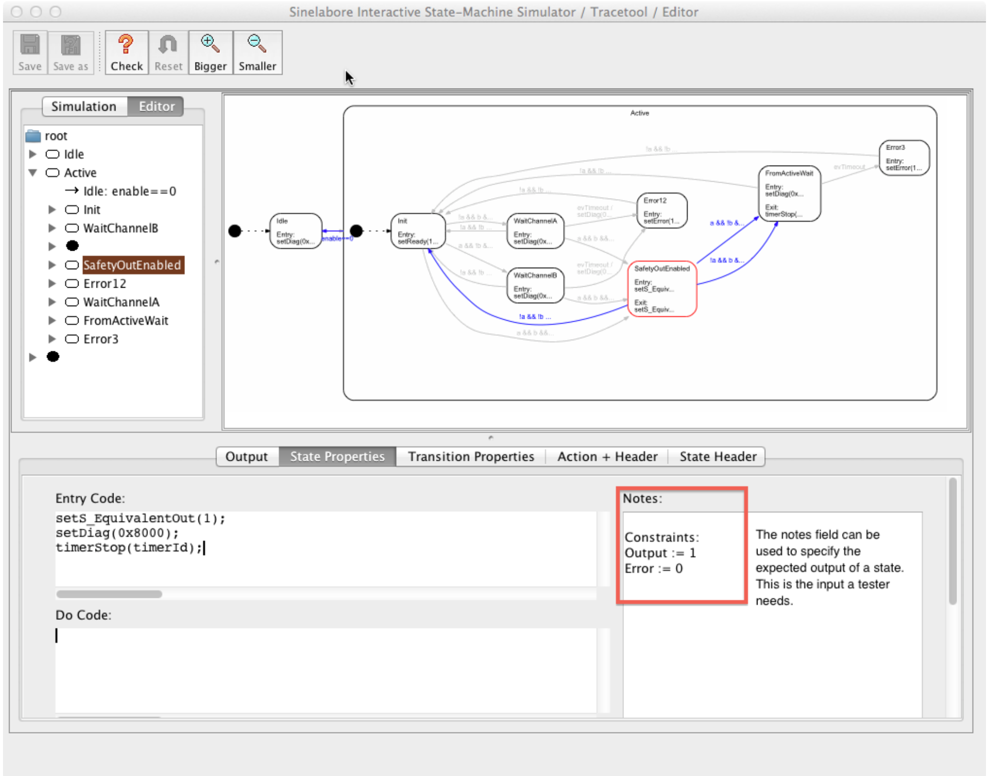


Figure 3.29.: The notes field can be used to specify expected outputs of a state. This information is valuable input for a tester.

With a less-organised approach, the amount of testing required might be equally large-you just won't know it.'

The suggested test route(s) ensure that every transition was taken at least once. Consider this as the minimum number of test steps that needs to be performed to test a state machine. Often this is not sufficient. In our example other possible test cases are:

- Running the transition coverage tests multiple times to test longterm behaviour (e.g. is the timer correctly restarted ...)
- Testing with different cooking time settings (i.e. testing the timer)
- Decrementing the timer to zero after the cooking has started

3.20. Tracing the Event flow of a State Machine

3.20.1. Introduction

Sometimes it is useful to monitor a system in real-time and trace the sequence of events. For that purpose events are usually written into a buffer for post mortem analysis or sent via a serial link to a monitoring device e.g. a PC.

In any case it is necessary to instrument the state machine code. This can be done automatically by the code-generator. The command line switch `-Trace` enables the generation of trace code. The following code snippet shows the generated trace code for a machine called 'complex'. Lines 2 and 4 are not generated if the trace flag is absent.

```
1 if((msg==(COMPLEX_EVENT_T)e1)){
2  complexTraceEvent(8U);
 ...
3 }else if((msg==(COMPLEX_EVENT_T)e12)){
4  complexTraceEvent(12U);
 ...
```

The trace code uses unique ids to represent an event. The type of the ids is the same as you have defined for the events itself. The translation back into the event name etc. can be done outside of the embedded system (e.g. on the PC) where memory/performance is not limited.

Therefore additionally to the trace code a C/C++ header file (see below) is generated which allows you to easily translate the trace id (`8U` and `12U` in the shown case) into the event/guard text string as used in the state machine diagram. This string can be used e.g. to print out the event on the PC side[7]. Separate trace ids are generated if different guards for the same event are used. This allows to follow the event flow in all cases. Line 8 and 9 in the header file below shows such a case.

The trace code is prefixed with the machine name (`complex` in this case) as the other generated code.

The header file for the shown example looks the following:

```
1 #define complexTraceEventLen 17 // number of text strings
2 // prototype of trace fct.
3 void complexTraceEvent(COMPLEX_EVENT_T evt);
4
5 const char* const complexTraceEvents[] = {
6   "e2",
7   "e6",
8   "evC[i==5]",
9   "evC",
10  ...
11  "e16"
12 };
```

The trace calls are generated automatically but you must implement the function `xxxTraceEvent()` which allows you to use the trace data in the way you want. In one case you might want to fill a circular buffer. In another case you might want to send the trace data to a PC. In a third case you want to just toggle a port pin if a certain event occurs. It is up to you how to use the trace data. The following example just prints out the event string.

```
void testcaseTraceEvent(TESTCASE_EVENT_T evt){
  printf("Trace: %s\n",testcaseTraceEvents[evt]);
}
```

3.20.2. Using the Visual Simulator to Trace the Event Flow

The visual simulator can be stimulated by selecting events from the event tree as discussed in section 3.17. But it can also be stimulated by event strings sent via UDP to port 4445[8].

[7] Please note that also a Java class is generated. This class provides the same definitions than the header file but allows you to write a Java application to evaluate the trace data.

[8] You can change to UDP port in the codegen config file.

Under examples a folder named `guisim_client_server` shows how this can be used to send trace data via UDP to the visual simulator. If the target executing the state machine has an Ethernet interface (as the PC in the example) the events can be directly sent from the state machine code to the visual simulator.

In this case the function `complexTraceEvent()` might look like this:

```
// called from the state machine
void complexTraceEvent(COMPLEX_EVENT_T evt){
  rc=sendto(s,complexTraceEvents[evt],
            strlen(complexTraceEvents[evt]),
            0,(SOCKADDR*)&addr,sizeof(SOCKADDR_IN));
  if(rc==SOCKET_ERROR)
  {
    // handle send error
  }
}
```

In any other case e.g. if there is a RS232 or a CAN interface a small program must be written which receives the event id sent from the embedded target, translates it into the event string using the generated translation table and forwards it via UDP. The following figure 3.30 shows the principal setup in this case. The upper part shows the hardware point of view. The lower part the software side.

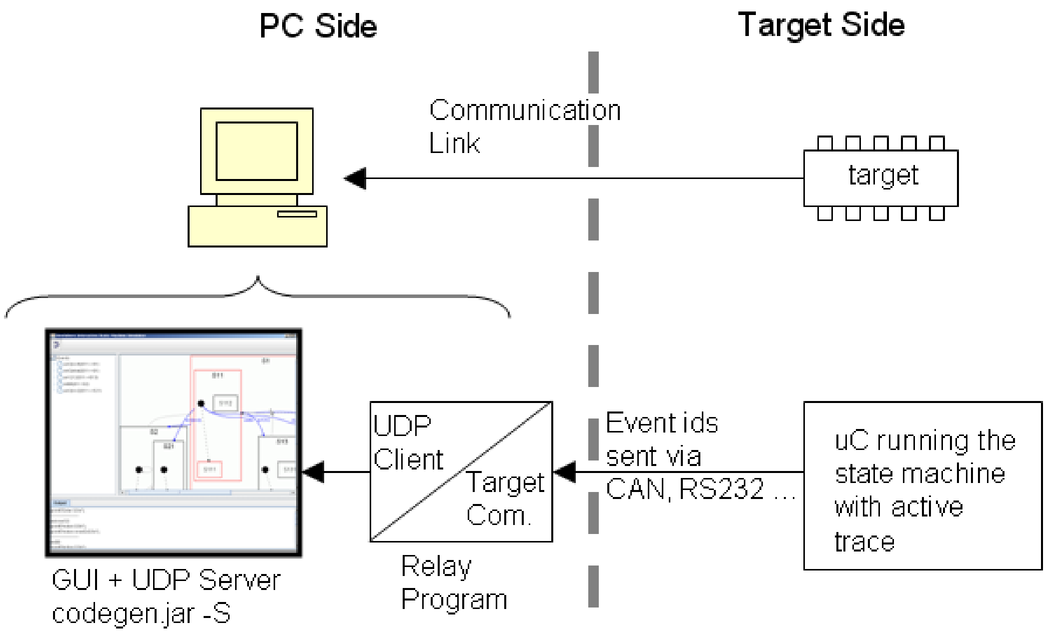


Figure 3.30.: Sending trace data to the visual simulation.

The visual simulator was separated with intent from the required relay application. The reason is that the relay application is probably very special because of the event transport mechanism and protocol which might be different for every embedded system.

3.20.3. Reset the simulation

To reset the visualisation e.g. after rebooting the embedded target you can send the following command to the visualisation. This command must not be used as normal event within the state machine.

Reset command: ___RESET___

4. Activity Code Generator

4.1. Introduction

There are two diagrams that can be used to express dynamic behaviour in the UML. State machine diagrams express the states of an object and the events and transitions to change the state (see chapter **??**). In contrast activity diagrams (aka flow diagrams) describe the control flow of an algorithm. How to generate code from activity diagrams is described in this chapter. For more information about activity diagrams see the UML Superstructure Specification [3].

4.2. Basic Node Types

To describe algorithms UML tools provide activity diagrams. Activity diagrams describes an algorithm (e.g. a function in C or a method in C++) that runs from the begin (defined by the *Initial Node* represented by a filled black circle) to the end (defined by the *Final Node* represented as circle with a filled black circle inside). Between the initial node and the final node *Action Nodes* define the single steps of an activity (shown as rectangles with rounded edges). Inside an action the action name can be shown and more important the code executed during this step. Actions can also contain activities again (nesting). Action names are used in the generated code to enumerate the actions. The name must be a valid code name of the generated language. If the name is missing a unique name is generated automatically (not recommended). If the name contains spaces they are replaced by underline characters.

Nodes are connected by arrows that depict the control flow between nodes.

Use a *Decision Node* to represent alternatives in the control flow. A decision node must have at least one incoming arrow and at least two outgoing arrows. Each outgoing arrow must have a guard. One of the guard must be the *else* statement to indicate the default path if no other guard evaluates to true (same with choices in state diagrams). Please note that also arrows between normal actions can be guarded. This is not recommended because it might lead to dead locks if there is no alternative path modeled!

Merge Nodes allow to merge alternative control flows back into one single flow.

The following figure 4.1 shows a basic activity diagram using the elements described so far.

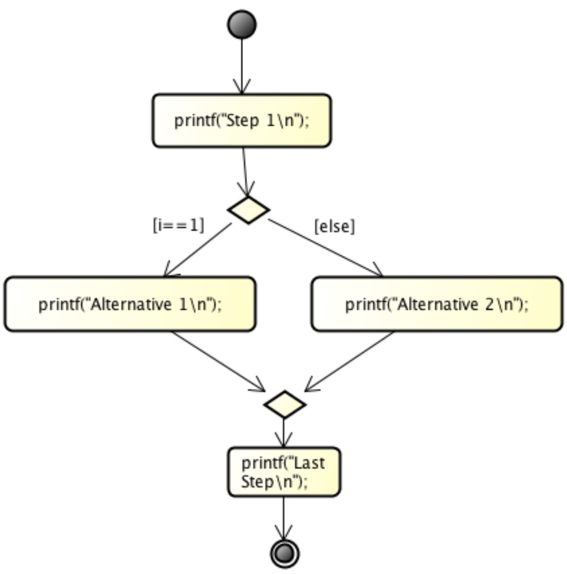


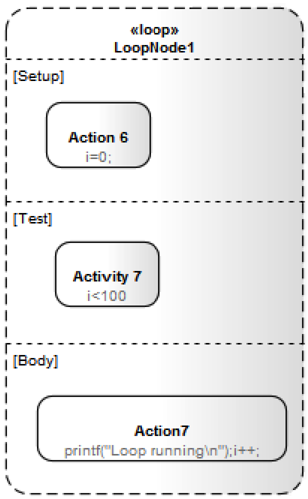
Figure 4.1.: Very simple activity diagram showing the main node types.

	An initial node is a control node at which flow starts when the activity is invoked (UML spec.). There must only be one initial state in an activity. If the activity contains other activities each level requires an initial state.
	Actions define single steps of an activity.
	Decision and merge nodes allows to split or merge the control flow.
	Indicates the end of the activity (i.e. the return of the function). It is allowed to have more than once final node in a diagram.

Table 4.1.: Basic activity elements

4.3. Complex Node Types

The UML defines also *Loop Nodes* and *Conditional Nodes*. The definition from the UML specification is. “A loop node is a structured activity node that represents a loop with setup, test, and body sections.“ Loop nodes consist of a setup part, a test part and a body part. It is possible to specify if the test is at the begin or the end of the loop. Other attributes that tools allow to define are not used from the code generator.

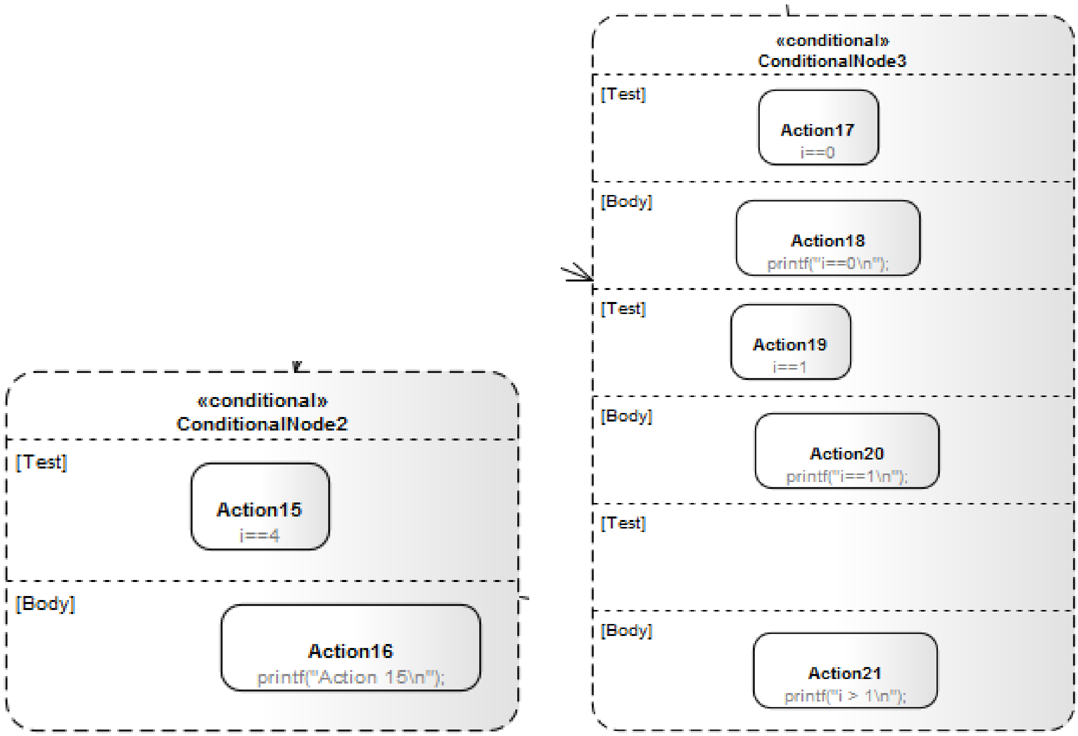



```
/* Loop setup code */
i=0;

while(i<100){
  /* Loop body code */
  printf("Loop running\n");
  i++;
}
```

```
/* Loop setup code */
i=0;

do{
  /* Loop body code */
  printf("Loop running\n");
  i++;
}while(i<100);
```

Table 4.2.: Loop generation depends on UML attribute *isTestedFirst*. The left code is produced if *isTestedFirst* is set to *true*, the right code if it is set to *false*.

A conditional node is a structured activity node that represents an exclusive choice among some number of alternatives [3]. A conditional node consists of one or more clauses. Each clause consists of a test section and a body section. The sinelabore code generator generates *if/else if/else* code from conditional nodes. It is possible to leave the last test section empty. The body section is then automatically used in the *else* part of the generated code.




```
if(i==4){
  printf("Action 15\n");
}
```

```
if(i==0){
  printf("i==0\n");
}else if(i==1){
  printf("i==1\n");
}else{
  printf("i > 1\n");
}
```

Table 4.3.: The code generator generates *if/else if/else* structures from conditional nodes.

4.3.1. Generator Flags

The code generator supports a number of generator flags to influence the generation process according to your needs.

The following table shows the available generator flags, their purpose and and example for the generated code.

ActivityFileNamePostfix	The postfix of the activity c-file name. The first part is defined from the -o command line parameter. The second part from this parameter	-

ActivitySubfunction ParameterDefintion	In case a diagram has subactivities this parameter defines the function parameters used for sub-function declaration. E.g. set this parameter to $uint8\_ti$ generates	`// used in header file`
		`void subactivity(unit8_t i);`
		`...`

ActivitySubfunction CallingParameter	In case a diagram has subactivities this parameter defines the function parameters used for function calling the sub-activity. E.g. set this parameter to 43 generates	`...`
		`subactivity(32);`
		`...`

ActivityFunction ParameterDefinition	Defines the function parameters used for main activity function declaration. E.g. set this parameter to $uint8\_tparA, uint8\_tparB$. Then you must call the activity function as	`// call activity`
		`//   form your code`
		`activity(3,4);`
		`...`
		`// declation in header`
		`void activity(uint8_t parA,`
		`        uint8_t parB);`

ActivityFunctionPrefixHeader	Return type definition of the main activity function e.g. $unit8\_t$ defines that the activity function has to return that type. If the function requires additional prefixes it is also possible to define e.g. specific linker hints where to place the function ...	-
ActivityFunctionPrefixCFile	Prefix of the main activity function e.g. $unit8\_t$ used in the c-file. Similar to the previous parameter but sometimes it is necessary to have different prefixes in the header and c file.	-

ActivityFunctionReturnCode	Defines the return statement of the main activity function. If empty the generator will not return anything. This must be in line with the definition of parameter ActivityFunctionPrefixCFile. If you defined that the activity function should return a value which is stored locally in variable *retVal* define that parameter to *retVal*.	`...`
		`return retVal;`
		`}`

Table 4.4.: Generator parameters for the `codegen.cfg` file to influence the generation process. The spaces in the parameters are only here for layout reasons. In reality the names have to be written without any spaces.

4.3.2. Optimizations

After model check the code generator performs the following optimizations:

- Actions without names receive a unique default name
- For better readability action names can contain spaces in the diagram. The code generator transforms spaces to underline characters during the generation process.
- Successive actions with only have one incoming transition are automatically merged into one action containing all the action code. This makes the generated code shorter and cleaner.

4.3.3. Generating C-Code

Sinelabore creates compact and clearly readable C code from UML activity diagrams. There are various configuration parameters to adjust the generation process. This allows to generate code that is optimal for your system. See table 4.4 for a list of all available options. To generate C code call the code generator with the command line flag `-l cx`.

For each activity in the selected UML class the generator generates an own C-code and header file. The header file contains only the declaration of the main activity – nothing else.

The code is generated as `while loop` that ends if a final node is reached. This kind of generation allows all kind of loops and back links in the model. An alternative generation is to use *goto* statements. But this was considered as not acceptable for most embedded developers and is not used for that reason. The following listing shows the principle structure of the generated C-code for a simple example without sub-activities.

```
/* User defined header text */
#include <stdint.h>
#include <stdio.h>

extern uint8_t n;
extern uint8_t m;

uint8_t product;
/*Enumeration of all node names */
typedef enum {
  TEST_OPERATION_ACTION2,
  TEST_OPERATION_ACTION3,
  TEST_OPERATION_ACTIVITYFINAL,
  TEST_OPERATION_ACTIVITYINITIAL,
  TEST_OPERATION,__END__,
} BRANCHES;


uint8_t test_operation(void ){

  BRANCHES id;

  id = TEST_OPERATION_ACTION2;
```

```
  while(id != __END__) {
    switch(id) {
      case TEST_OPERATION_ACTION2:
        printf("action 2\n");
        id=TEST_OPERATION_ACTION3;
        break;

      case TEST_OPERATION_ACTION3:
        printf("action 3\n");
        product=n*m;
        id=TEST_OPERATION_ACTIVITYFINAL;
        break;

      case TEST_OPERATION_ACTIVITYFINAL:
        id= __END__;
        break;

      default:
        break;

    }
  }
  return product;
}
```

4.4. Activity Diagrams with Enterprise Architect

4.4.1. Introduction

It is possible to create one or more activity diagrams for a class. Right click on the class and insert a new Activity with Activity Diagram. Each activity must have a unique name.

Also like for state diagrams it is necessary to specify the path in the XMI file to the class to generate code for. Use the *-t* command line parameter for that purpose. It is important that you export the model using version 2.4.1 of the XMI standard. The version can be selected in the export dialog and then pressing *Publish*. Don't simply export the model. Then an old XMI version is used not fully supporting activity diagrams.

To generate code from activity diagram diagrams use the *-A* command line switch. For the shown EA diagram the following command line is used for code generation. Note that for all activity diagrams of a class code is generated!

```
java -cp "path to codegen bin folder" codegen.Main -A -p EA -o testcase -t
"Model:test:class_with_activities"  testcase.xml
```

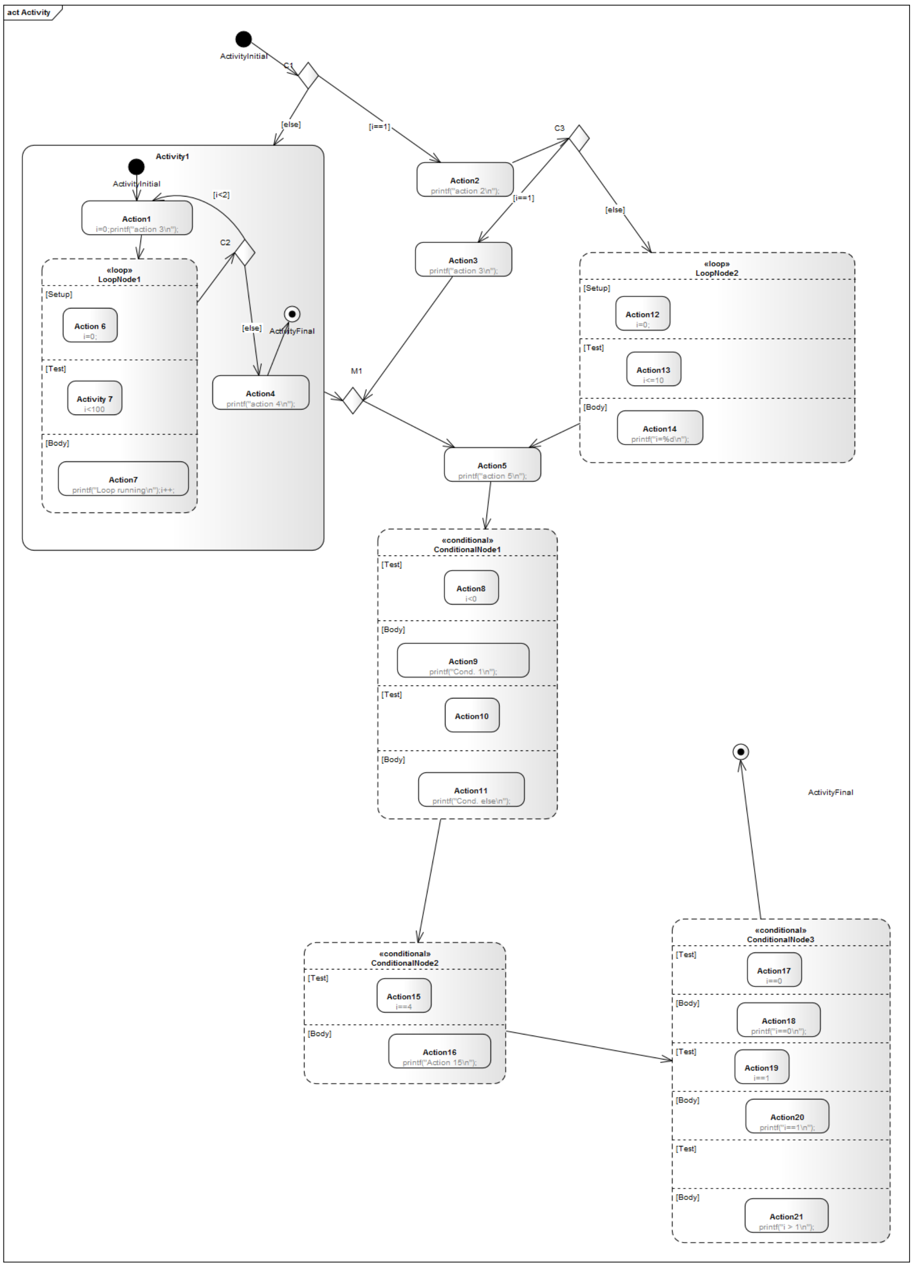


Figure 4.2.: An EA activity diagram with all node types currently supported by the code generator.

4.4.2. Actions

To add code to actions use the *Effect* field. To show the code in the diagram check the option *Show Effect in Diagram* in the same dialog. Unfortunately EA does display all your code in one long line in an action even separate lines were used in the effect field. The code generator tries to format your code best possible.

To add an action or guard into the body field of a loop or conditional node just place an action into the field. Make sure the place for the action is large enough. Even if it looks like the action is correctly placed in the body area check in the model browser whether the action is shown in the correct hierarchy level. Otherwise the generation will not work!

4.4.3. Defining own Include Statements

To include your own set of headers, define local variables or make other definitions attach a note to the class. The first line of the node must contain the text *activity_header:* . An example is shown in the next figure 4.3.

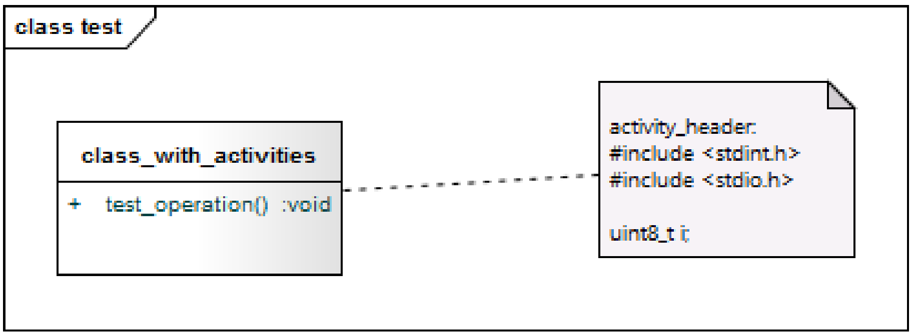


Figure 4.3.: To define own code at the beginning of the generated code use a note with the text *activity_header:* as first line.

4.4.4. Supported / Unsupported

– Action
– Decision
– Merge
– Loop
– Condition

The unsupported elements are:

– Object related data flow
– Fork/Join
– Using activities or multiple actions in the body part of loop – or conditional nodes
– Activities in activities (might work, but untested)
– Choice behind choice. Use conditional node for that purpose
– Crossing border with connections between nodes when using nested activities.

4.5. Activity Diagrams with UModel

4.5.1. Introduction

UModel provides basic activity diagrams modeling features. Add add an activity diagram right click on the class in the class diagram and select *New Diagram → Activiy Diagram*

Tt is necessary to specify the path to the class in the XMI file on the command line. Use the *-t* command line parameter for that purpose. It is important that you export the model using the latest version of the XMI standard supported by UModel (e.g. 2.4.1). The version can be selected in the export dialog.

To generate code from an activity diagram use the *-A* command line switch. For the shown EA diagram the following command line is used for code generation. See figures 4.4 and 4.5

```
java -jar -ea codegen.jar -A -p UModel -o testcase -t
"Component View:Class1"  testcase.xml
```

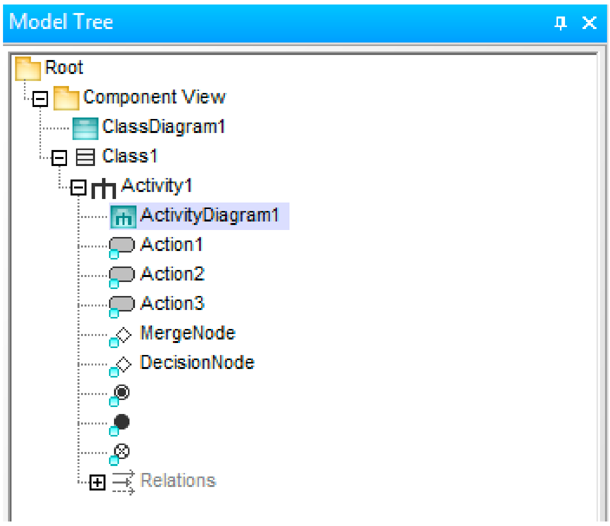


Figure 4.4.: Use the '-t' command line option to specify the path to the class containing the activity diagram. For the shown example use -t “Component View:Class1”

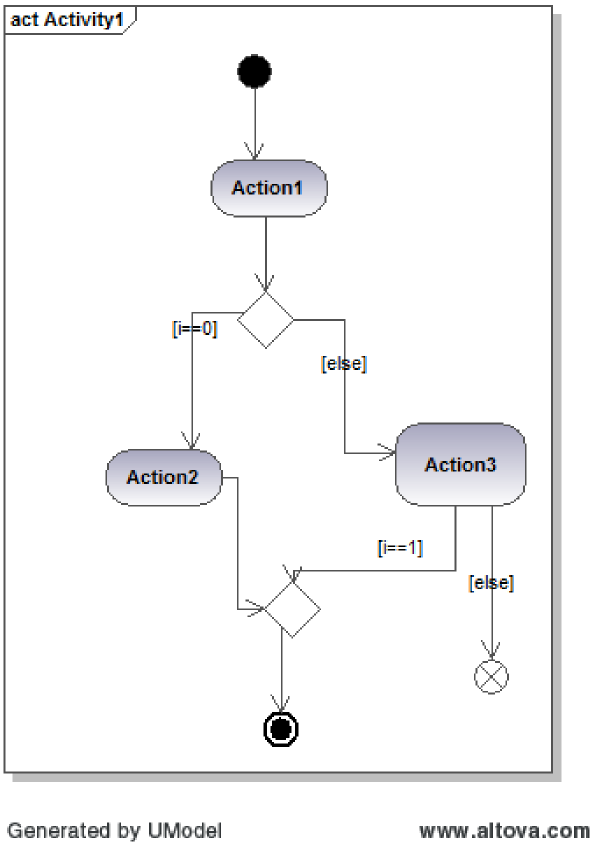


Figure 4.5.: An UModel activity diagram with all node types currently supported by the code generator and UModel.

4.5.2. Actions

Actions must have valid C names. When adding an action use the *Opaque Action* type indicated by a question mark (?) in the icon bar. Only this type of action allows you to define code in the *body* attribute available in the properties of an action.

UModel does not export multi-line code as expected. For multi-line code statements add a paragraph sign § at the end of each line. The code generator replaces each paragraph character with a *CRLF* symbol.

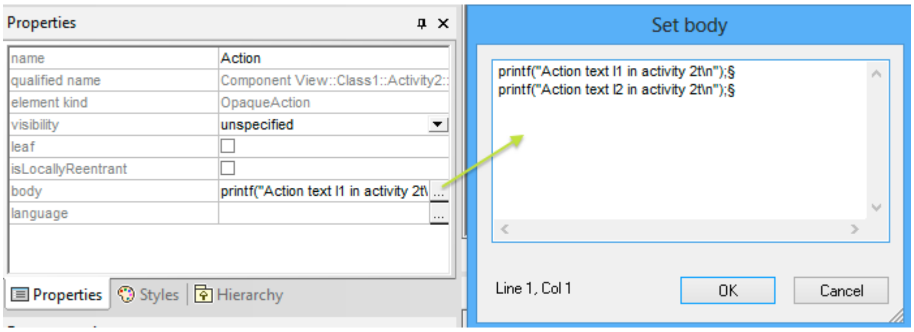


Figure 4.6.: To define action code use the body attribute available in the action property dialog. For multi-line comments add a paragraph symbol as line end.

4.5.3. Defining own Include Statements

It is usually necessary to provide own includes, define local variables or make other definitions at the begin of the generated code. Define this code by attaching a *Comment* to the class. See example in figure 4.7. The first line of the node must contain the text *activity_header:*. It is important to use the *Comment* icon and not the *Note* icon. An example is shown below.

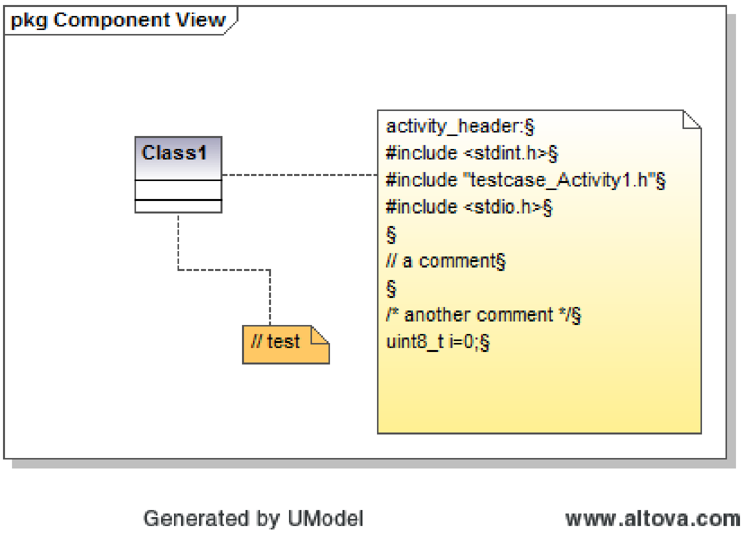


Figure 4.7.: Provide a user defined include statement by adding a comment to the class. Text is copied to the begin of the generated code. Use also paragraph symbols as line ends.

4.5.4. Supported / Unsupported

Supported:

– Action
– Decision

- Merge

The unsupported elements are:

- Loop
- Condition
- Object related data flow
- Fork/Join
- Using activities or multiple actions in the body part of loop – or conditional nodes
- Activities in activities (might work, but untested)
- Choice behind choice. Use conditional node for that purpose
- Crossing border with connections between nodes when using nested activities.

4.6. Activity Diagrams with Astah

4.6.1. Introduction

Astah provides basic activity diagrams modelling features. To add an activity diagram right click on a class in the class diagram and select *Create Diagram → Activity Diagram*. The *microwave_handbook_astah* example (part of the download) contains a full example.

It is necessary to specify the path to the activity diagram in the model on the command line. Use the *-t* command line parameter for that purpose.

To switch on activity code generation from a model file use the *-A* command line switch. Use *-D* to add the astah jar files to the class path (see installation section for more information).

The following example shows how the command line looks on *nix (e.g. Linux or OS X) like operating systems. The parts of the Java class path are separated by colon (Windows uses spaces). The Astah path depends on your installation. For the shown model tree the following command line is used for code generation.

See figure 4.8 for the related model tree, figure 4.10 for a simple activity diagram and figure 4.9 for an example how to use a generated activity in a state diagram.

```
java java -Djava.ext.dirs="../../bin/":"/Applications/astah community/astah
community.app/Contents/Java/" -Djava.awt.headless=true -jar -ea codegen.jar
-A -p ASTAH -o oven -t "final:oven:selftest"  oven_model.asta
```

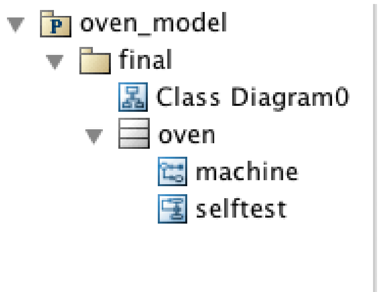


Figure 4.8.: Use the '-t' command line option to specify the path to the activity diagram. For the shown example use -t "final:oven:selftest".

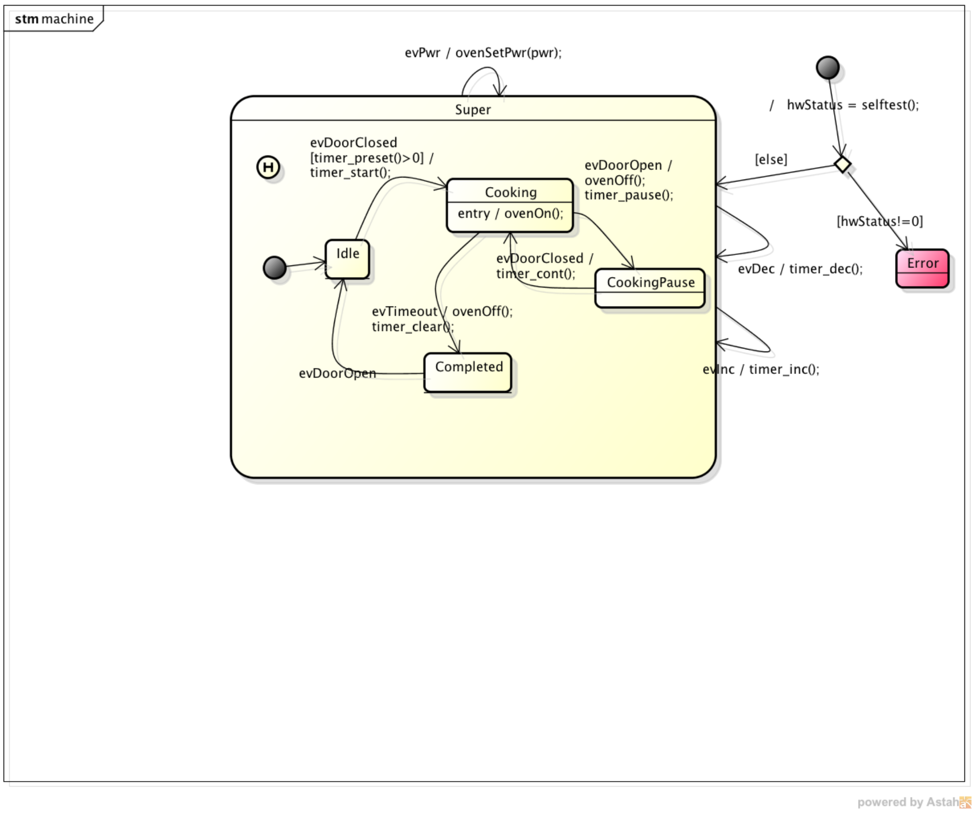


Figure 4.9.: This machine is an extension of the microwave oven from the introduction section. It shows how to use generated activity code in a state diagram. Here the self-test function (from init to choice) is fully generated. Depending of the result the *Error* state or the *Super* state is entered.

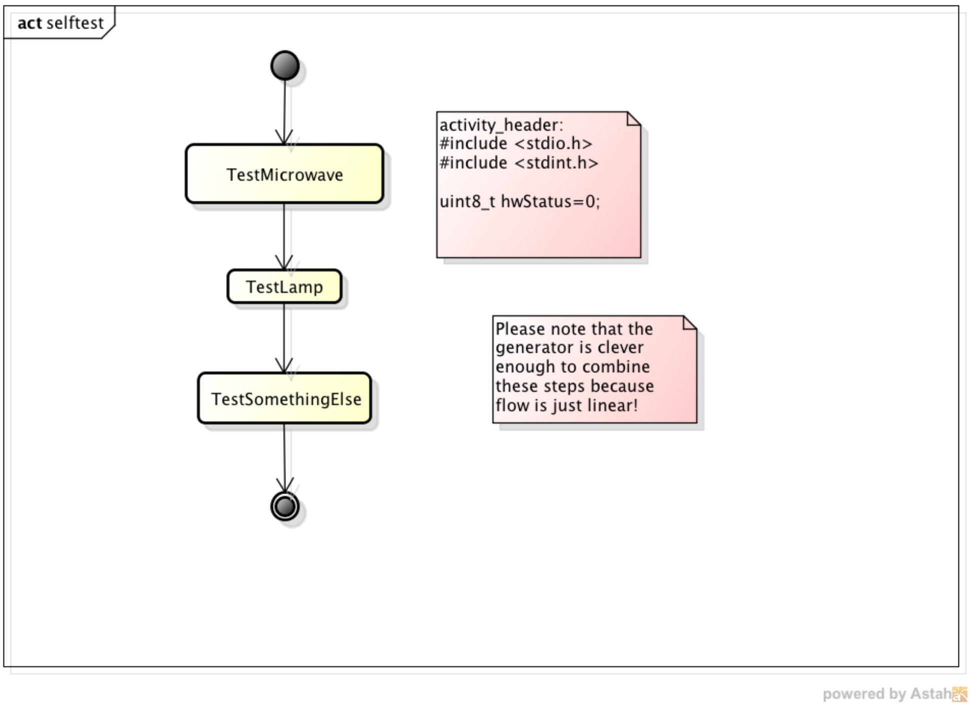


Figure 4.10.: The self-test function is simple but shows the principles. Please note that the generator combines linear actions to generate more compact code!

4.6.2. Actions

Actions must have valid C names. When adding an action use the *Action* type. Specify the action name in the field *Entry* and the code for the action in the field *Definition*.

Figure 4.11.: To define action code use the entry attribute available in the action property dialog.

4.6.3. Defining own Include Statements

It is usually necessary to provide own includes, define local variables or make other definitions at the begin of the generated code. Define this code by placing a *Comment* to the activity diagram. See example in figure 4.12. The first line of the node must contain the text *activity_header:*. It is important to use the *Comment* icon and not the *Note* icon. An example is shown below.

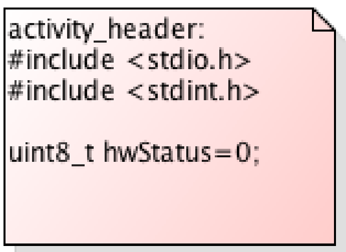


Figure 4.12.: Provide a user defined include statement by adding a comment to the activity diagram. Text is copied to the begin of the generated code. Use also paragraph symbols as line ends.

4.6.4. Supported / Unsupported

Supported:

- Action
- Decision
- Merge
- Final
- Initial

The unsupported elements are:

- Object related data flow
- Fork/Join
- Choice after choice.

4.7. Activity Diagrams with Modelio

4.7.1. Introduction

It is possible to model more or less complex algorithms and functions used in the class and generate code from that model. Right click on the class and select *Create Diagram / Create an Activity Diagram.* Each activity must have a unique name. Before you start read section *Drawing State-Charts with Modelio* on page 149. Also import the Modelio oven example and play with the sample.

4.7.2. Exporting the model as XMI file

Like for state diagrams it is necessary to specify the path of the class in the XMI file. Use the *-t* command line parameter for that purpose. It is important that you export the model using XMI version OMG UML2.4.1. Select the correct settings in the export XMI dialog.

4.7.3. Generating activity code

To generate code from activity diagrams use the *-A* command line switch. For the shown Modelio diagram the following command line has to be used for code generation. Note that you can use more than one activity diagram to define multiple algorithms. Presently all algorithms share the same include part.

```
java -jar -ea codegen.jar -A -p Modelio -o testcase -t
"OvenClass"  oven.xmi
```

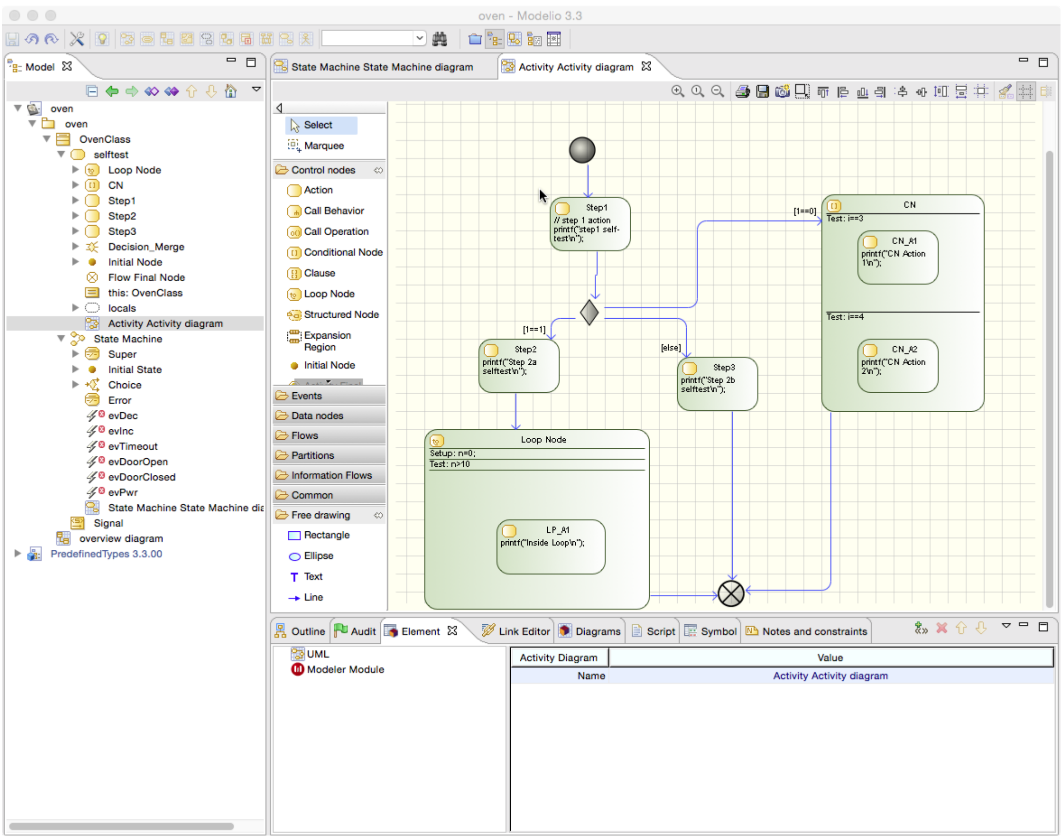


Figure 4.13.: A Modelio activity diagram with all node types currently supported by the code generator.

4.7.4. Actions

To add an action drag and drop an element from the *Control Nodes* palette to the action diagram. Show the action element properties and give the action a name and action code (*Body* field). Action code can span multiple lines.

4.7.5. Loop Node

A loop none requires a test and setup part. To define the code executed inside the loop place an action in the loop and define the code there. See figure H for an example.

4.7.6. Conditional Node

A conditional node allows to execute code depending on one or multiple conditions. For each condition a test must be defined. To add code executed if the condition is true add an action and define the code in the action. See figure H for an example.

4.7.7. Defining own include statements

To define includes or other code at the beginning of the generated activity file, attach a note to the class. The first line of the node must contain the text *activity_header:* . An example is shown in the next figure 4.14.

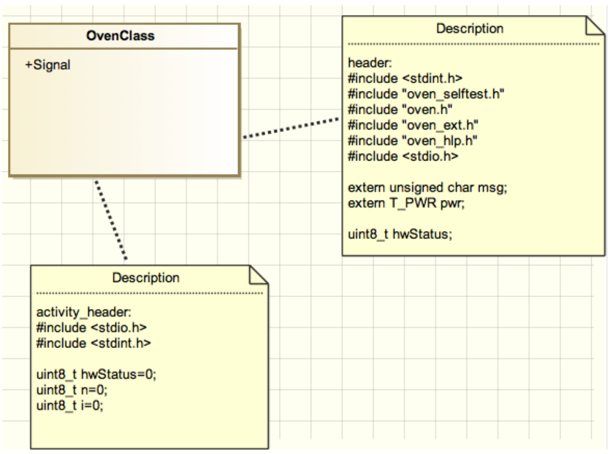


Figure 4.14.: To define own code at the beginning of the generated code use a note with the text *activity_header:* as first line.

4.7.8. Supported / Unsupported

- Action
- Decision
- Merge
- Loop
- Condition

The unsupported elements are:

- Object related data flow
- Fork/Join
- Using activities or multiple actions in the body part of loop – or conditional nodes
- Activities in activities (might work, but untested)
- Choice behind choice. Use conditional node for that purpose
- Crossing border with connections between nodes when using nested activities.

5. Appendix

A. Design Questions

A.1. Defining the state processing order

For UML state charts it is defined that events are handled from inside out. I.e. if there are two transitions ready to fire the transition starting from the innermost state will be taken (see also 3.2.1).

Sometimes it can be useful to influence the processing order of transitions starting from the same state. Consider the following case: There is an emergency stop and if it is pressed, only a specific transition should be taken even if other transitions are ready too (e.g. start pressed at the same time).

Event based machine: If your machine receives its events via a queue it is usually possible to enqueue events to the front of the queue. So the sender of the event has simply to put emergency events to the front of the queue.

Condition based machine: In the case where transitions are triggered by boolean expressions (conditions) it becomes more tricky. Because then it can happen that several transitions might be ready at the same time (the emergency button and the stop button is pressed at the same time!). One solution is to include the emergency stop input into the boolean expression triggering all the transitions. Unfortunately this often creates lengthily condition expressions. Example:

```
// transitions
emStopInput==1/action
inputA==1 && emStopInput==0/action
inputB==1&& emStopInput==0/action
```

A more elegant solution is based on the fact that the generator generates the transition handling code in alphabetical order of the event name. This makes the following possible:

```
// define this in mydef.h or where appropriate
// you can also define other names or more prios
#typedef unsigned char PRIO_A
#typedef unsigned char PRIO_B

// transitions
(PRIO_A)emStopInput==1/action
(PRIO_B)inputA==1/...
(PRIO_B)inputB==1/...
(PRIO_B)inputC==1/...
...
```

Outgoing transitions of choices: Transitions leaving a choice are ordered based on the guard whereas the 'else' guard is used always as last option. Example:

From the figure A.1 the following code is produced. The `if/else if/else` structure triggered by `evB` is ordered according the guard specification.

```
...
case S1:
  if(msg==(CHOICETEST_EVENT_T)evA){
    /* Transition from S1 to S2 */
    evConsumed=1U;


    /* adjust state variables */
    instanceVar->stateVar = S2;
  }else if(msg==(CHOICETEST_EVENT_T)evB){
```

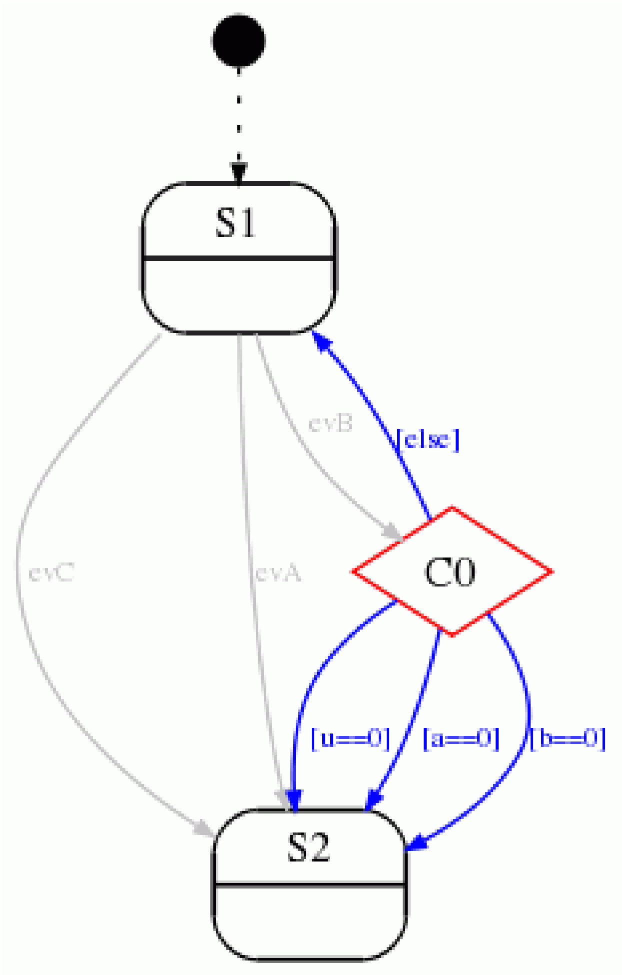


Figure A.1.: A choice pseudo state with evB as input transition and four output transitions.

```
    if(a==0){
      /* Transition from S1 to S2 */
      evConsumed=1U;


      /* adjust state variables */
      instanceVar->stateVar = S2;
    }else if(b==0){
      /* Transition from S1 to S2 */
      evConsumed=1U;


      /* adjust state variables */
      instanceVar->stateVar = S2;
    }else if(u==0){
      /* Transition from S1 to S2 */
      evConsumed=1U;


      /* adjust state variables */
      instanceVar->stateVar = S2;
    }else{
      /* Transition from S1 to S1 */
      evConsumed=1U;


      /* adjust state variables */
      instanceVar->stateVar = S1;
    } /*end of event selection */
  }else if(msg==(CHOICETEST_EVENT_T)evC){
```

```
/* Transition from S1 to S2 */
...
```

A.2. Running the state machine in context of a RTOS

A frequently used design pattern with real-time operating systems is shown in the following figure A.2.

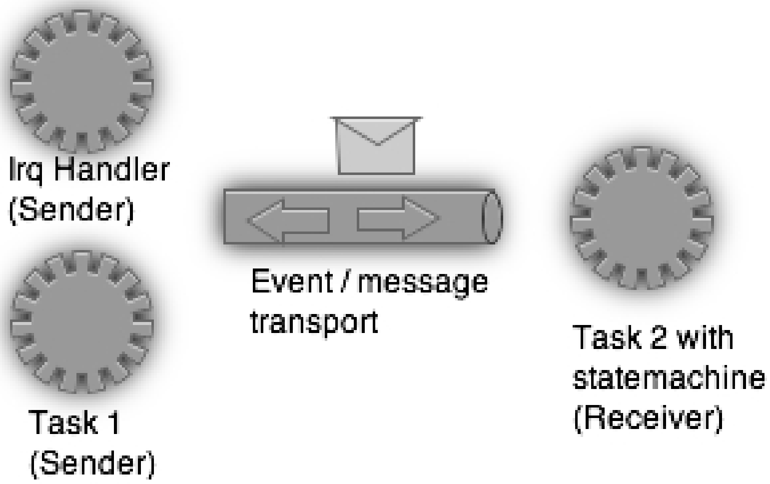


Figure A.2.: Communicating tasks exchanging messages between each other (Asynchronous Event Handling). At least one of the tasks executes a state machine that reacts on the received events. As reaction new events to other tasks might be sent out.

1. A task executes a state machine (often called active object).
2. It waits for events by calling a blocking operating system function that returns whenever a new event is available for processing.
3. The used system mechanism for event signalling can be different but often a message queue is used.
4. Events might be fired from within another task or inside an interrupt service routine
5. If an event was received the state machine reacts on the new event
6. Jump to step 2

This pattern can be realised with every real-time operating system. The generated state machine code can be easily integrated in such a design.

In folder `example2` the microwave oven state machine is embedded into a real-time operating system. In this example RTEMS was used. RTEMS is the Real-Time Operating System for Multiprocessor Systems[1]. To compile the example you have to install a full RTEMS build environment. The example was created for the PC386 target. In `init.c` two tasks were created. One task (`init`) scans the keyboard and creates events according to the input. Then the events are sent via message queue to a second task named `oven_task`. This task calls the state machine code which waits blocking until a new event is available. Figure A.3 shows the slightly modified microwave oven example. In the state machine design some code was added to read events from a queue. This was done with the help of an action text note. As the action code is executed just before the state machine itself the machine reacts to the latest keyboard event.

In the context of RTOS the configuration flag 'EventsAreBitCoded' can be of interest. Some RTOSs provide a mechanism for communication between tasks (e.g. called Task Events). Every task has e.g. a one-byte (eight bit) mask, which means that eight different events

[1] For more info goto http://www.rtems.org

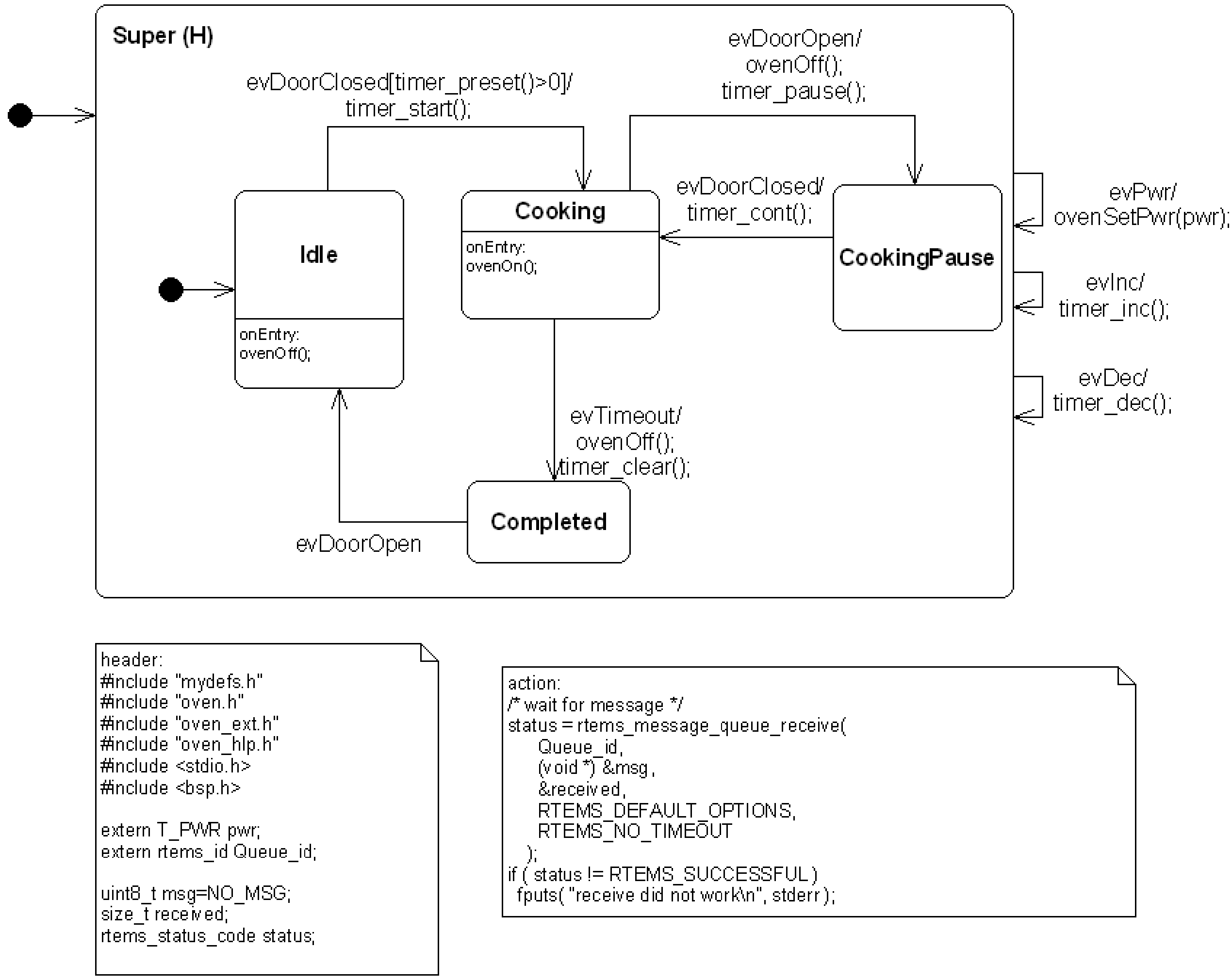


Figure A.3.: Event handling code added as text note to the oven state machine.

can be signalled to and distinguished by every task. Then no translation between bit coded events received from the task and the events accepted from the state machine is necessary. It is your responsibility to ensure that the number of events used in the state machine is not larger than the number of bits offered by the task mask.

A.3. Multiple Instances of a State Machine

Sometimes it is necessary to execute the same state machine multiple times. This is no problem in object oriented languages because you can simply instantiate the same class multiple times (e.g. in C++, Java or C#).

If you want to execute the same state machine in C multiple times (e.g. processing the serial protocol of 2 serial interfaces) please go on reading on section 3.3.9 on page 47.

The other option is to generate the state machine more than once using a different command line parameter for the machine name. This is probably not needs frequently but available for special cases.

A.4. Synchronous Event Handling

Sometimes it is required to process an event synchronously i.e. executing the state machine in the context of the caller task. The caller is blocked as long as the state machine executes.

Is can be realised by simply calling the state machine handler or implementing a proxy calling the state machine handler.

This method is usually used in the context of a real-time operating system. See section A.2 for more info.

A.5. Cooperating State Machines

Sometimes it is useful to create a state transition in one state machine in dependency of the present status of a second state machine. Consider a device which shall be stopped as soon as an emergency stop button gets pressed. A possible design is to implement supervision of the emergency stop button and all other functions of the device like controlling a motor or driving a display in one big state chart. But this is usually not what you want and creates complex and not maintainable designs.

It is much more useful to create separate state charts and let them work together. For each state the code generator generates a macro which returns `1U/0U` to indicate if the state machine is in that state or not (e.g. `OVEN_IS_IN_COOKING`). See table 3.2 for more information.

A.6. Optimisations for Lowest Memory Consumption

In deeply embedded systems code/ram size is usually limited and every optimisation is welcome to reduce the resource usage. If only one instance of a state machine is used in a design the memory consumption can be reduced by avoiding the usage of pointers. Therefore the following two configuration switches can be used:

– HsmFunctionWithInstanceParameters=no

– UseInstancePointer=no

If both switches are set to 'no' the instance variable is accessed by value within the state machine code. Also other helps (like changeToState) does not use a pointer anymore.

If your machine is flat (i.e. no hierarchical states) you can also avoid that the eventProcessed flag is generated. Set parameter *ReturnEventProcessed* to 'no' in this case.

A.7. Was an Event processed?

Sometimes the code calling the state machine needs to know if an event was processed in the presently active state. It is possible to instruct the code generator to generate code that returns *1U* if the event was handled and *0U* if it was not handled in the current state.

To enable this feature set the configuration option `ReturnEventProcessed` to `yes` in the codegen.cfg file.

B. Drawing State-Charts with Cadifra UML editor

The Cadifra UML editor is a very easy to use and lightweight UML modelling tool. It was not directly designed to generate code from its diagrams. Because of this it does not provide special means such as dialogs to enter events, guards, entry or exit actions and so forth. This section describes how to draw diagrams with all needed information using the available editor features.

B.1. Events

To add an event to a transition right click to the transition line and select `'New Text'` as shown in figure B.1. For the event definition you must follow the syntax as described in section 3.1.4. Only text associated with the transition (indicated with a dashed line) is detected by the code generator. A free text element will be ignored and the generator will complain about the missing event. Even if it might look ok for you as the free text is located close to the transition. For layout reasons it is allowed to put the event and guard in different lines.

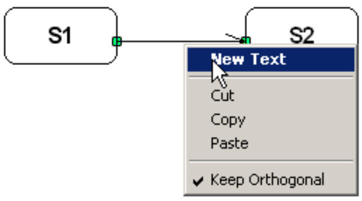


Figure B.1.: To enter events right click to the transition and use a text field to enter the event definition.

B.2. Hierarchical States

To draw hierarchical states (aka composite states) increase the size of the parent state (e.g. S1) until there is enough space to carry the new child state (e.g. S11) . Then move the child state into the parent state's border. The `'Large'` flag is automatically set on the parent state. With the this flag set the state name is shown in the upper left corner. To add another child (e.g S111 or S12) repeat the steps from above. In total 3 levels of nesting are supported from the code-generator. For flags on the third level the `'Large'` flag shall not be set (see figure B.2).

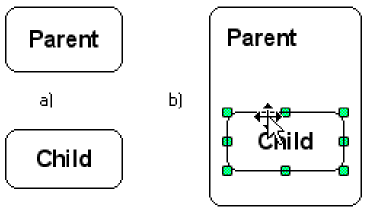


Figure B.2.: Composition of states. Move the child into the parent create a composition.

The following figure B.3 shows a fictious example with three levels of nesting.

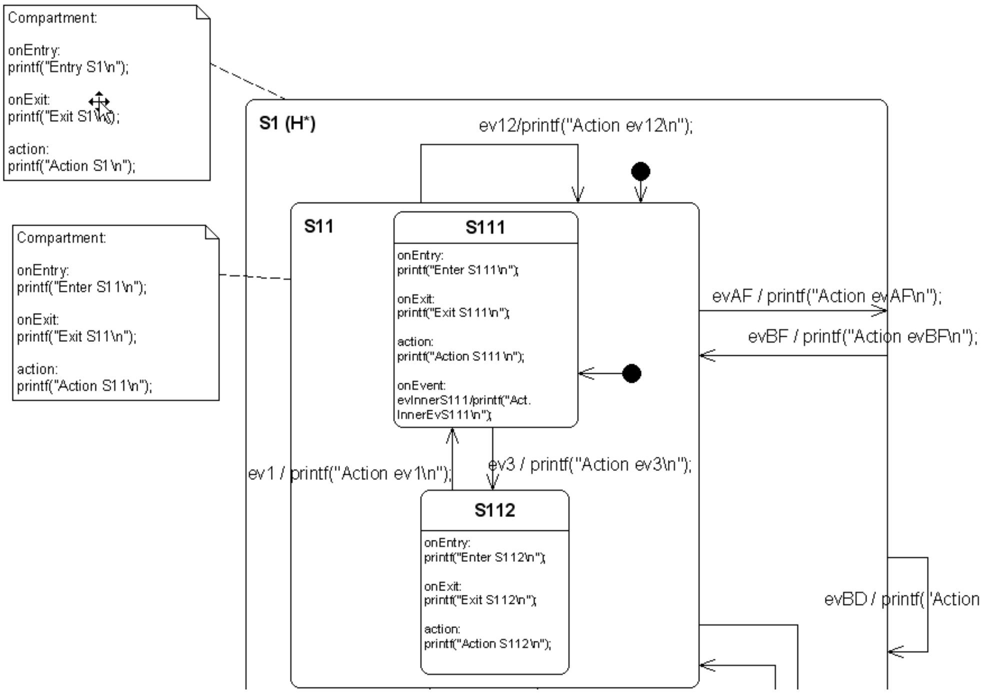


Figure B.3.: Three levels of states drawn with the Cadifra UML Editor. For states on level one and two the `'Large'` flag shall be set.

B.3. Adding State Details

To add entry, exit, action or inner events a compartment must be added to the state. To do so right click to the state and select `'Add compartment'` as shown in figure B.4. To edit the compartment double click on it and enter the definitions as needed. The definitions must follow the syntax as described in section 3.1.3. Compartments can only be used for states without children. For states with children attach a note to the state and put the state details into the note. The note must start with the text 'Compartment'. See figure B.3 for an example.

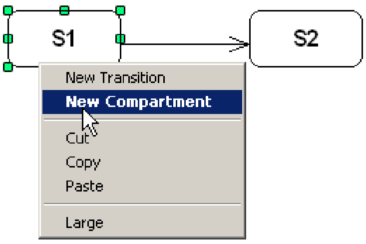


Figure B.4.: To enter state details right click to the state and add a compartment to it.

B.4. History States

Append the text `'␣(H)'` to the state state name if you want to make a state a history state (shallow history) (e.g. `'S1␣(H)'`).

Append the text `'␣(H*)'` to the state name if you want to make all child states history states (deep history). This has the same effect as adding a Shallow History marker to all composite child states (e.g. `'S1␣(H*)'`).

Since version 6.4 transitions can also end in history states. See section 3.1.11 for further explanations. Cadifra does not support history states. Therefore a special syntax for the state name must be used.

– A shallow history pseudostate must begin with *(H* and end with *)*. Examples: *(H)*, *(Hist1)*
– A deed history pseudostate must begin with *(H* and end with **)*. Examples: *(H*)*, *(Hist1*)*

An example for a diagram with history state usage is shown in figure B.5.

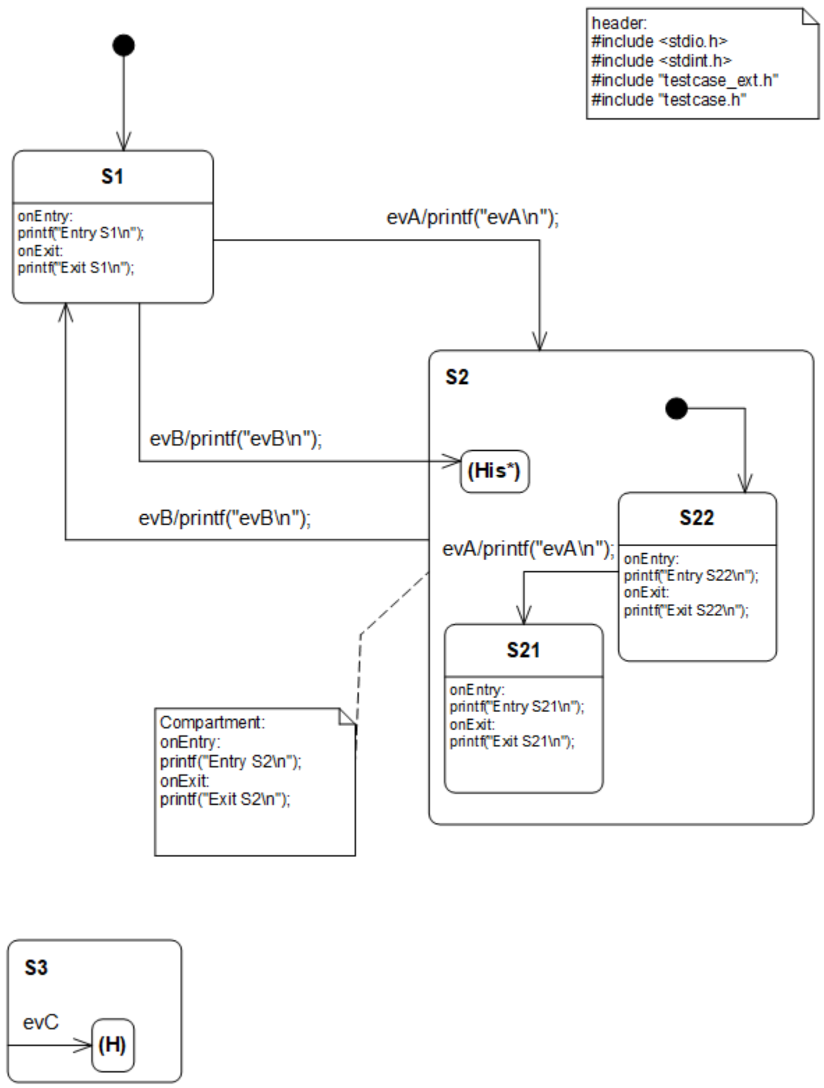


Figure B.5.: History states can have one or more incoming transitions but must have no outgoind transition.

B.5. Choices

Choices are not directly supported from the UML Editor. However with the help of a naming convention it is possible to use choices. If a state name begins and ends with angle brackets it is assumed by the codegen that it is a choice state. Valid names are e.g. <>, < >, <C1> or <AnyText>. The following figure shows an example state chart with three choice states.

B.6. Junctions

Junctions are not directly supported from the UML Editor. However with the help of a naming convention it is possible to use junctions. If a state name begins and ends with

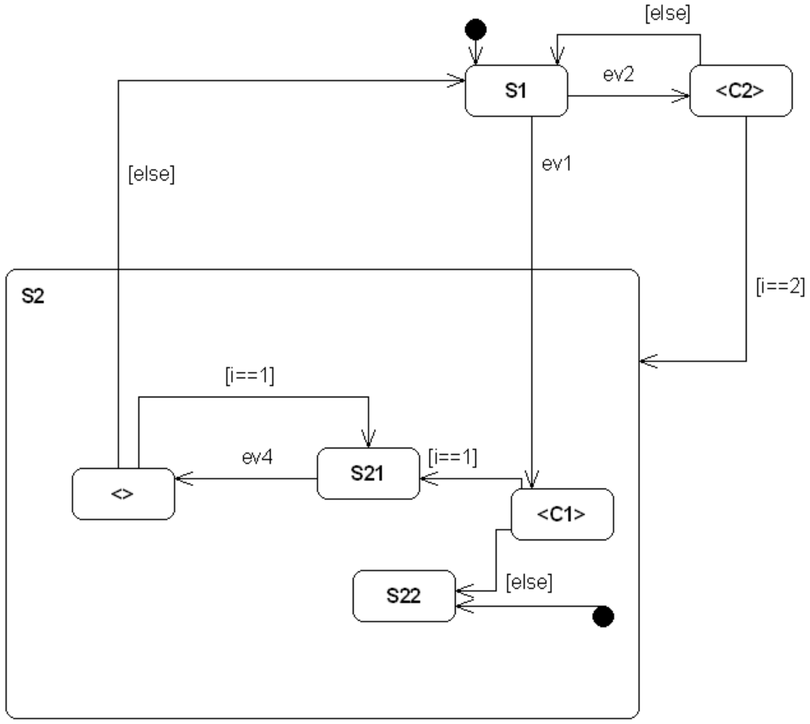


Figure B.6.: Choice states must have angled brackets in its name.

round brackets it is assumed by the codegen that it is a junction state. Valid junction names are e.g. `(J)`, `(J1)` or `(JAnyText)`. The following figure shows an example state chart with a junction.

For more background information about junctions see section 3.1.9.

B.7. Connection Points

This is a feature only available for the Cadifra editor and also not available in UML. The concept of connection points is well known from circuit diagram drawing tools. Because Cadifra has no direct support the realization is simular to Junctions. Exactly two states must have the same name surrounded by square brackets. Internally the code generator removes these states and creates a single transition. One of these states must have an incoming transition. The other an outgoing transition. Event name, guard and action are only taken from the entering transition. An example is shown below in figure B.8.

B.8. Supported / Unsupported

The codegenerator supports the following features for Cadifra UML:

- Hierarchical states
- (Signal-)Events with eventname, guard and action
- Initial and final pseudostates
- History states, Extended history handling (see section 3.1.11)
- Choices
- Junctions
- Constraints

The unsupported elements are:

- Multiple state machines/regions in a diagram

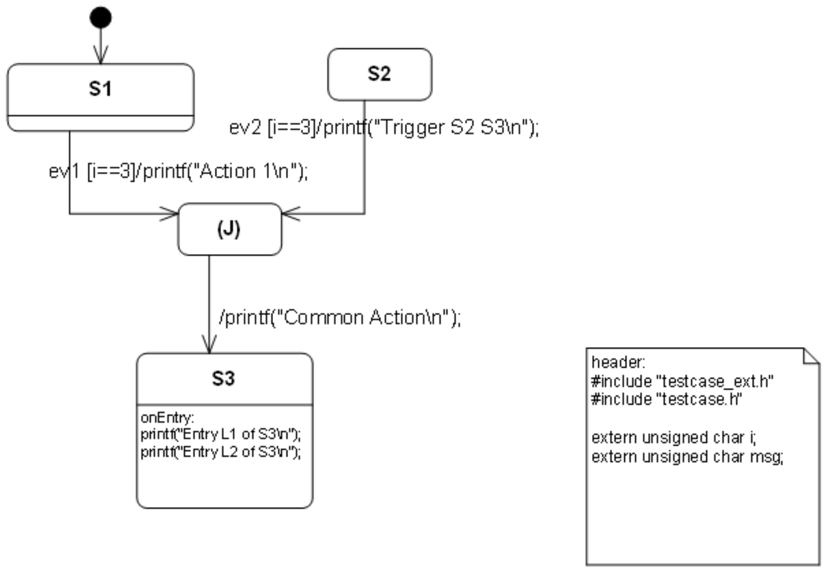


Figure B.7.: Junction states must have round brackets in its name.

- Syncstates
- Entry and exit points (not to compare with entry/exit actions within states)
- Terminate and Fork/Join

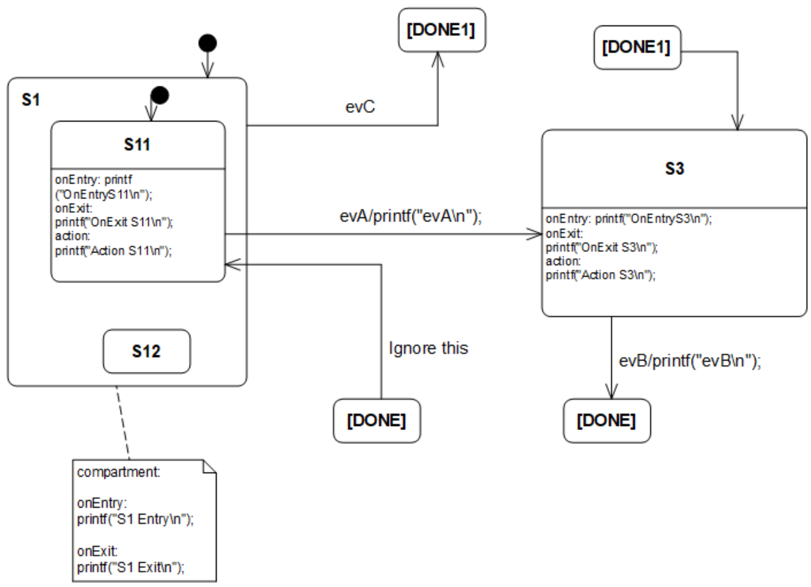


Figure B.8.: Connection point states must have square brackets around their name. In this example transition $S1 \xrightarrow{evC} S3$ and transition $S3 \xrightarrow{evB} S11$ were realized with connection points .

C. Drawing State-Charts with Magic Draw

When using Magic Draw some tool specific things must be taken care of. The following section discusses these points. First take a look how a rather complex state chart looks like in Magic Draw.

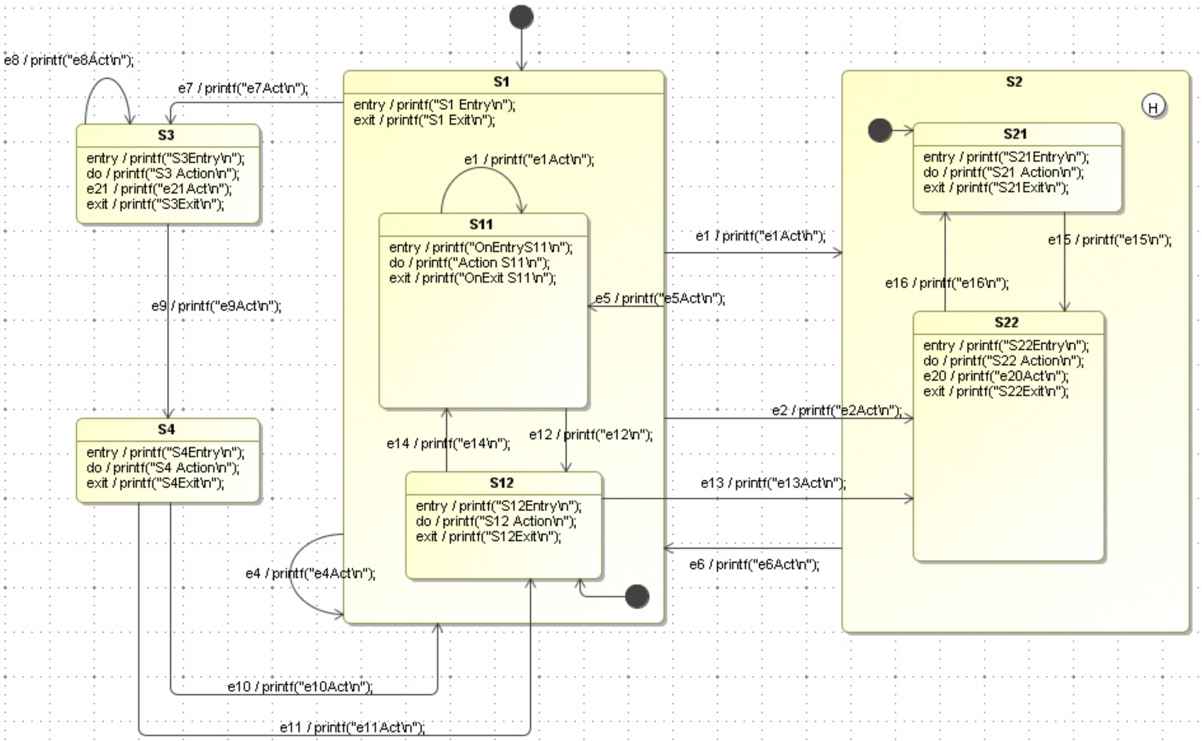


Figure C.1.: A rather complex state chart designed in Magic Draw

C.1. Organizing your project

The code-generator needs to know how to find your state-based class in the XMI file. The path to your class in your project tree must be specified on the command line using the -t flag.

The following figure shows the project browser window for the state diagram from above. From figure C.2 you can directly derive the path which is `-t 'MyModel:Class Model:complex_class'`. The different parts of the path must be separated with colons.

C.2. Attaching action and include comments

Remember that you can specify code that is just copied at the beginning of the state machine c-file. Also you can add action code that is called every time the state machine is called. This can be done by using comments starting with the keywords `action:` and `include:` (see section 3.1.6). It is important that you use a *comment and not a note*! In Magic Draw these two comments must be linked to the class owning the state machine.

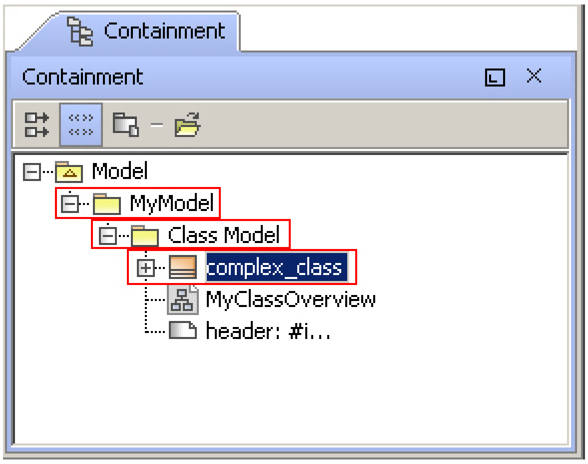


Figure C.2.: Magic Draw's project browser. The package structure defines the path to the class you want to generate code from.

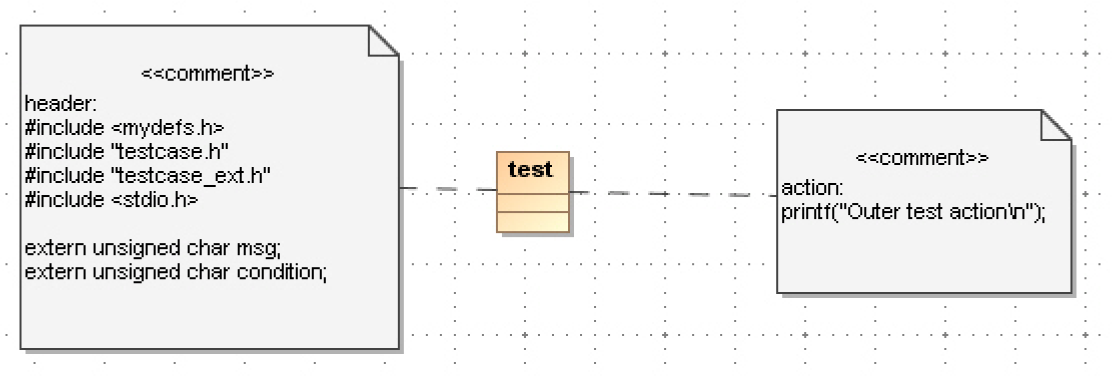


Figure C.3.: Action and inlcude comments must be linked to the class which owns the state machine.

C.3. Saving your project to XMI

Magic Draw allows you to save a project directly in XMI format. All the project information is stored in the XMI file. No other project file is needed. This is very handy as no export of the XMI file is required. Please remember to *tick the XMI 2.1 (rich XMI) check box in the save dialog*. If not checked the generated XML export can't be processed from the code generator.

C.4. State Details

Magic Draw allows you to define entry, do and exit actions as well as inner transitions for a state. Actions are usually operations (i.e. functions) which are triggered in case of the event. This means that for each of these actions a C-function (operation) must be defined. Sometimes this is not what you want especially if the action code is very short and a function would add a lot of overhead. There are two options to overcome this.

Option A: Just misuse the operation name as field where you can type in the C-code you want to execute. It is important that the `Behavior Type` is defined as `Opaque Behavior` as shown in C.5.

Option B: You can attach a comment (not a note!) to a state (either a child state or a composite state) and specify entry/exit or action code in there. To edit the comment click on it and enter the definitions as needed. The definitions must follow the syntax as described in section 3.1.3. Make sure you use plain text. The code-generator is not able handle html text.

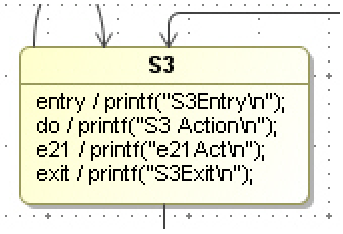


Figure C.4.: A state with entry/exit/do code plus an inner event following option A

To define an inner transition double click on a state and select `Internal Transitions`. The definition of an internal transition is done the same way as a normal transition is defined. See the next section for more info.

C.5. Transitions

Double clicking on a transition brings up the transition properties dialog (see figure C.6. This dialog allows you to define the event that triggers the state transition. As usual you can specify a guard (a valid C-statement evaluating to true or false), the effect and the trigger itself. The trigger type **must be** set to type `SignalEvent`. The effect behaviour type must be set to `Opaque Behaviour`.

To add a guard click on the guard field and then on the tree appearing dots. Now select `Constraint` as guard type. Now type in the guard in the `Specification` field (not in the name field!). The guard should appear in bracket braces after the event name if you click `close`.

C.6. History State

Use the `Shallow History` if you want to make a state a history state.

C.7. Deep History

Use the `Deep History` if you want to make all child states history states. This has the same effect as adding a Shallow History marker to all composite child states.

C.8. Sub-Machines

Sub-machine states allow to "hide" the child states of a state with hierarchy. You only see the child states etc. if you double click on the state which opens the internal view. From the code generator point of view a sub-machine is a normal state with children. There is absolutely no difference compared to a normal hierarchical state.

But using a sub-machine state instead of a state has some other consequences:

1. MD does not allow to provide entry/do/exit actions for a sub-state. You have to attach a comment field to specify state actions to overcome this MD limitation.
2. It is not possible to connect a transition starting from a state outside the sub-machine state to a state in the sub-machine. And vice versa. Use entry and exit pseudo states for this purpose (see next section).
3. MD does not allow to "transform" a sub-machine state into a state with children. You have to do this manually by moving states etc. in the Model Browser.

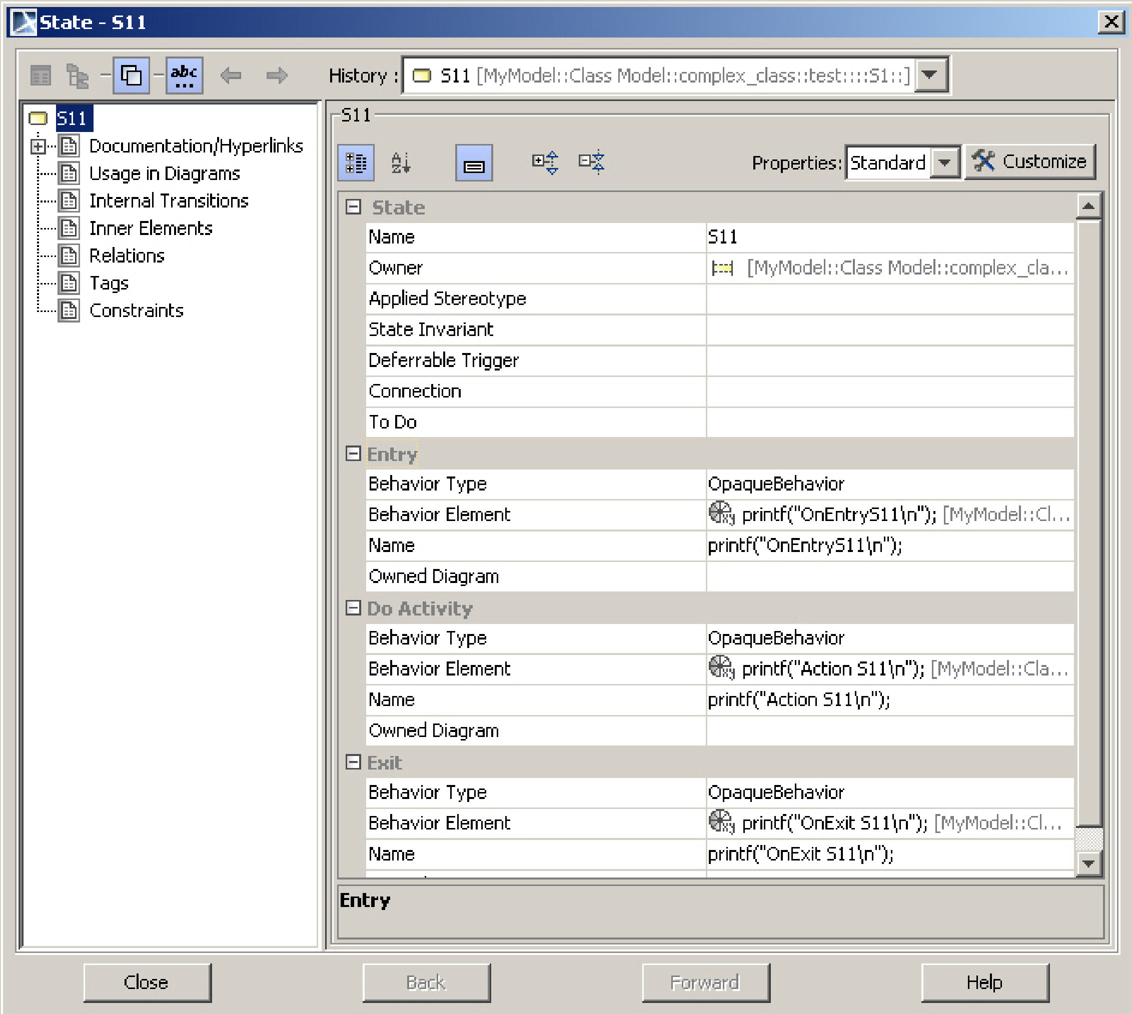


Figure C.5.: Actions must have the behaviour type set to `Opaque Behaviour`.

C.9. Entry and Exit Points

Note: Entry and exit points are only useful together with a sub-machine. Read the previous section before if you are not familiar with the usage of sub-machines.

When entering a sub-machine usually the initial state is entered. I.e. the transition ends at the border of the sub-machine state If this should not be the case for a specific transition it is possible to place an entry point inside the sub-machine state. This entry point serves as glue between the sub-machine state and the internal of the sub-machine.

Exit states provide a similar function. By default only transition can be modelled starting from the sub-machine. If a transition should start from a specific state inside the sub-machine and enter a state outside the sub-machine an exit state can be used. Again this exit state serves as glue between the transition starting inside the sub-machine and ending one level up at another state.

To use entry and exit points in your model use the following recipe:

– Double click on the sub-machine state to open the sub-machine diagram
– Drag and drop an entry and/or exit point inside your sum-machine diagram
– Give the points a meaningful name.
– Go up to the outside diagram
– Place references of the entry / exit points on the boarder of the sub-machine state.
– Connect transitions

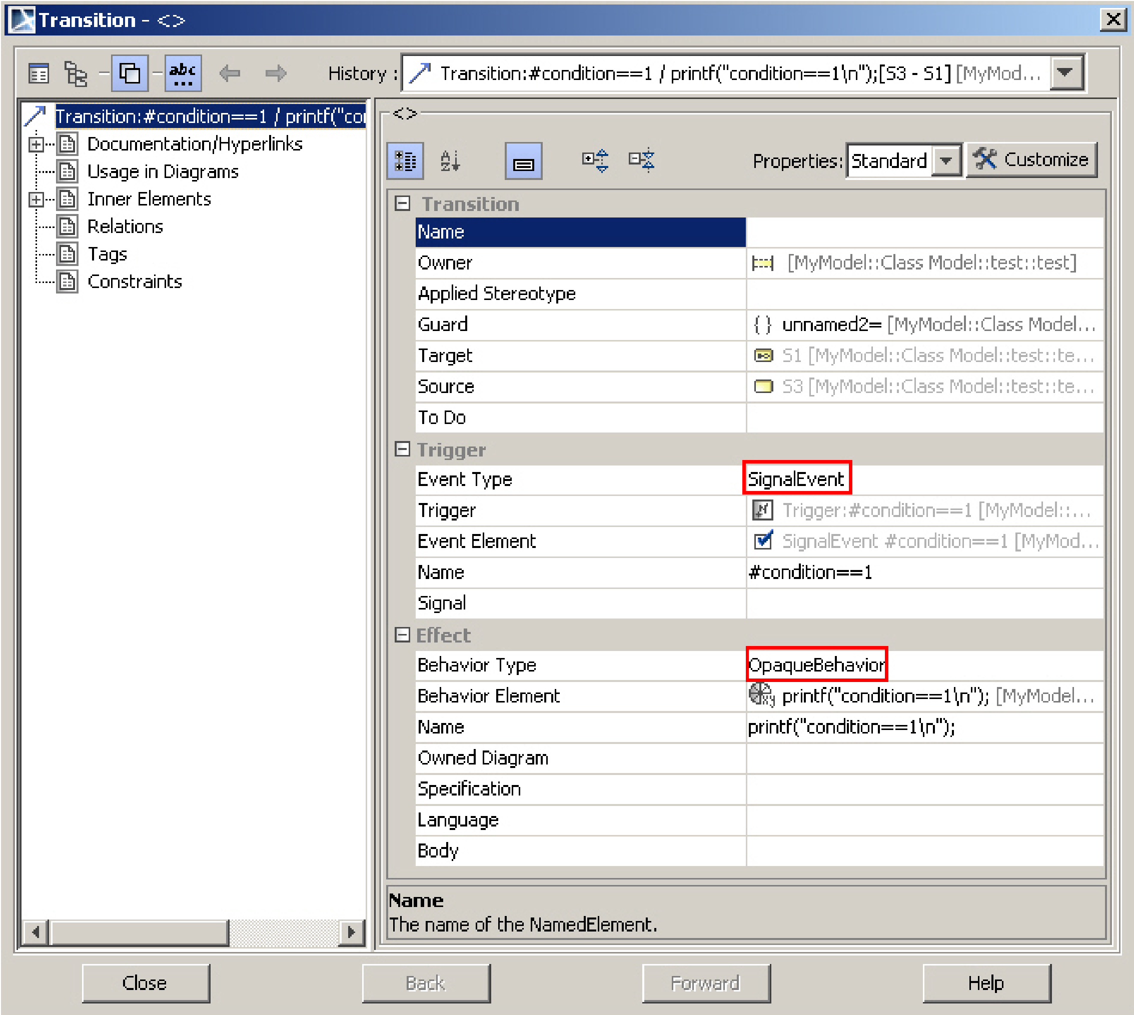


Figure C.6.: Magic Draw's transition properties dialog.

- The sub-machine must be placed below the main state machine and its name must start with for underlines ('____'). This is needed that the code-generator can identify the machine as sub-machine. See figure C.7.

Limitations when using entry and exit points:

- An exit point can have more than one incoming transitions inside the sub-machine. But only one outgoing transition from the sub-machine state.
- An entry state can have more than one incoming transitions on the sub-machine state. But only one outgoing transition inside the sub-machine.
- The transition leaving the exit or entry point most end in a normal state. Chaining of entry / exit states or connecting these transitions to choices or junctions is not allowed.

The following figure C.9 shows an example for a diagram with a sub-machine and the internals of this sub-machine.

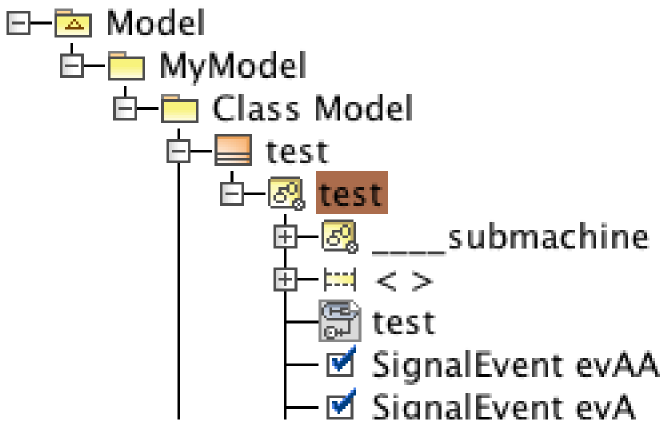
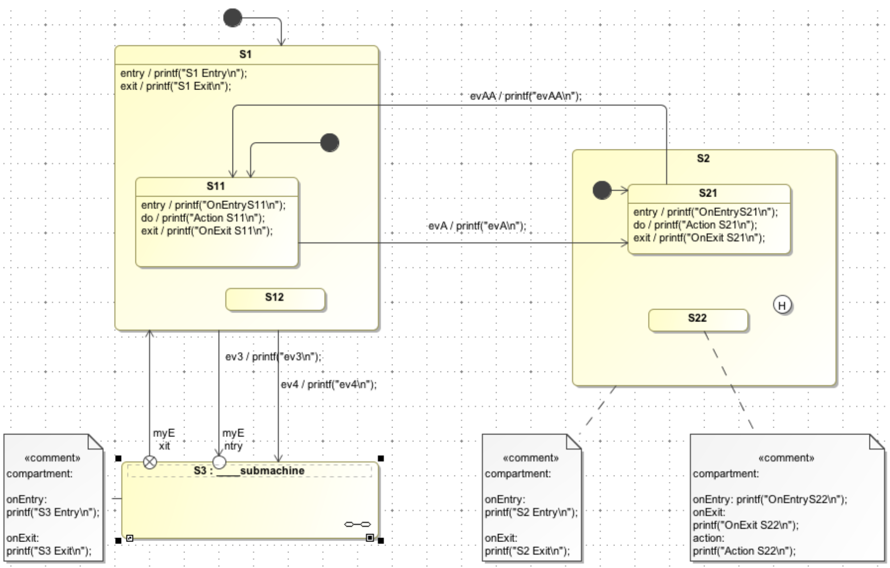
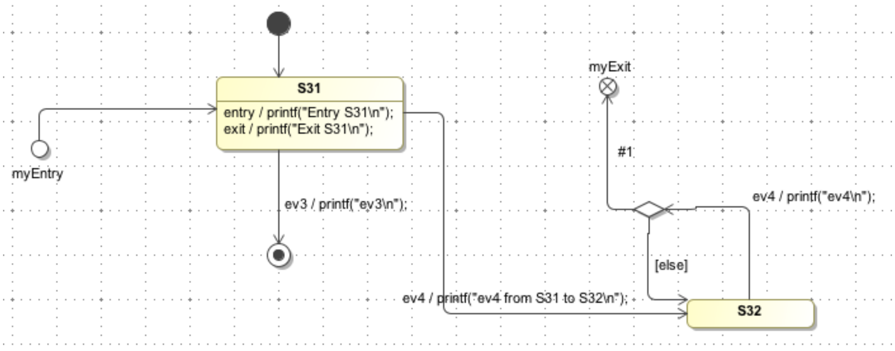


Figure C.7.: Place your sub-machine below the main state as shown in this figure.



(a) Top level diagram with the sub-machine state S3



(b) Internals of the sub-machine state S3

Figure C.8.: This figure shows the top level diagram with a sub-machine state and the internals of the sub-machine state. The transitions between these two diagrams are glued together using entry and exit points.

C.10. Supported / Unsupported

The code generator supports a subset of the design elements provided by Magic Draw. The supported elements are:

- Hierarchical states
- (Signal-)Events with eventname, guard and action
- Initial and final pseudostates
- Sub-machines and entry / exit points
- History states (deep, flat), Extended history handling (see section 3.1.11)
- Choices
- Junctions
- Constraints

The unsupported elements are:

- Multiple state machines/regions in a diagram
- Syncstates
- Terminate and Fork/Join

D. Drawing State-Charts with UModel

When using UModel some tool specific things must be taken care of. The following section discusses these points. First take a look on figure D.1 that shows how a rather complex state chart looks like in UModel.

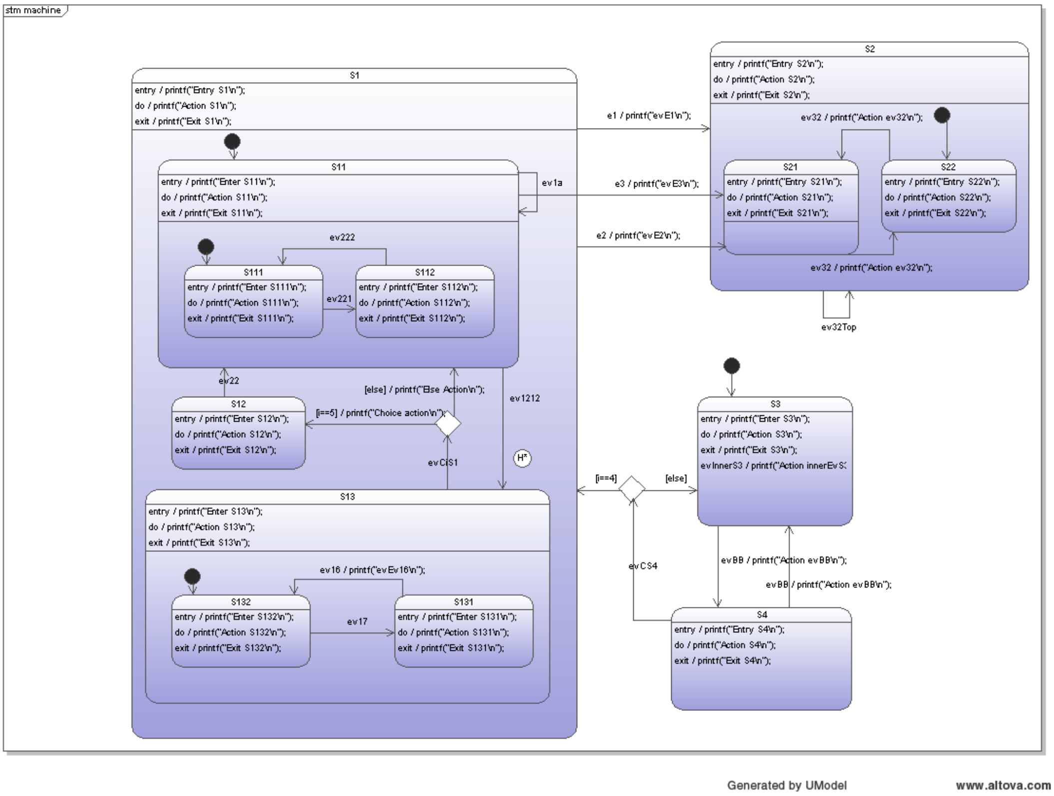


Figure D.1.: A rather complex state chart designed in UModel. Most of the supported elements are used in this diagram.

D.1. Organizing your project

The code-generator needs to know how to find your state-based class in the XMI file. The path to the class containing the state machine in your project tree must be specified on the command line using the `-t` flag.

Figure D.2 shows the project browser window for the state diagram from above. From there you can directly derive the path which is `-t 'Model:class'` for the shown example. The different parts of the path must be separated with colons.

D.2. Attaching action and include comments

Remember that you can specify code (so called header code) that is simply copied at the beginning of the state machine c-file. Also you can specify action code that is called every time the state machine gets executed. Add the `action-` and `header code` in the documentation window of the class containing the state machine. The text must start with either `action:` or `header:` (see section 3.1.6). It is not possible to use a `Note` which is

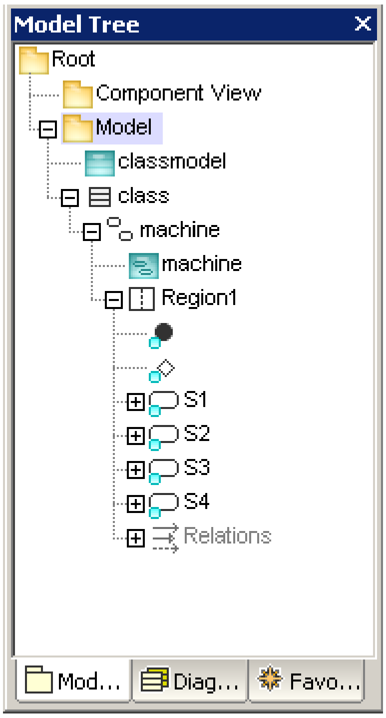


Figure D.2.: UModel's project browser. The package structure defines the path to the class you want to generate code from.

attached to the class as UModel does unfortunately not export notes in the XMI file. The documentation window can contain `action` - and/or `include code`. The order does not matter.

UModel does not export line end information correctly from the documentation window. Therefore it was necessary to define a specific line break character. Use the paragraph §character for that purpose. It is usually not used in normal C-code. An example documentation window with `action` - and `include code` is shown in figure D.3.

D.3. Saving your project to XMI

UModel allows you to export your project in XMI format. All the project information is stored in the XMI file. Check the tick boxes as shown in figure D.4 to export a XMI file which can be processed from the code generator.

D.4. States

UModel allows you to define entry, do and exit activities as well as inner transitions for a state. Do not use the `interactions` for that purpose but the `activities`. Only one single line can be specified for the entry, do and exit activities. If you want to execute more code define a function which contains the code.

There are different type of states in the toolbar. Simple states can't contain children. A composite state can contain further children. If you are unsure if a state will have substates later on draw a composite state right from the beginning. There seems to be no way to change a simple state into a composite one later on. The use of orthogonal states is not supported from the code generator so far. See appendix A.5 for a possible solution.

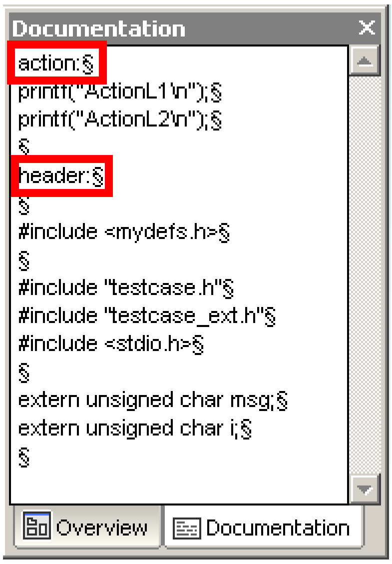


Figure D.3.: Action and include comments can be specified in the documentation window of the class which owns the state machine.

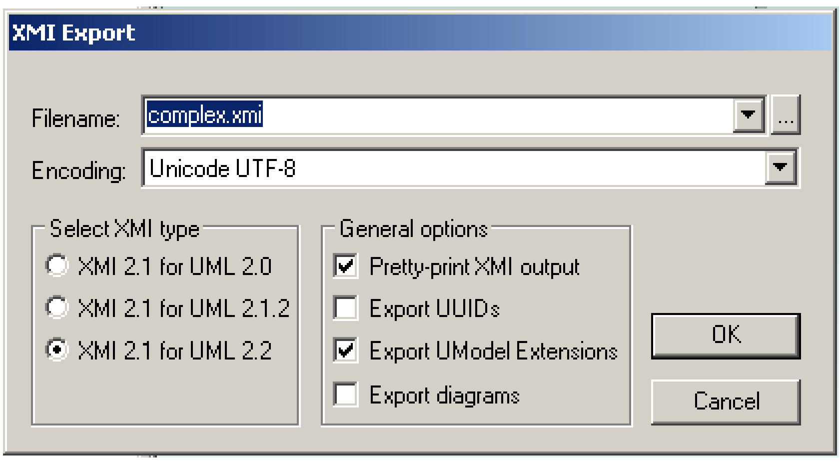


Figure D.4.: Exporting the project to XMI.

D.5. Transitions

To create a transition click the transition handle of the source state (on the right of the element). Then drag-and-drop the transition arrow onto the target state. A text field is shown. Type in the event name and optionally the guard and the action. This is quite convenient as it allows fast editing. The event, guard and actions can also be specified later on by adding the element in the model tree below the transition.

D.6. History State

Use the `Shallow History` if you want to make a state a history state.

D.7. Deep History

Use the `Deep History` if you want to make all child states history states. This has the same effect as adding a Shallow History marker to all composite child states.

D.8. Adding Operations and Attributes to Classes

It is possible to add attributes and operations to the class containing the state machine. Attributes are added to the instance data and always require a default value. Operations are mapped to C-functions and added to the state machine header file. Hint: Directly use data types that you want to use in your code later on.

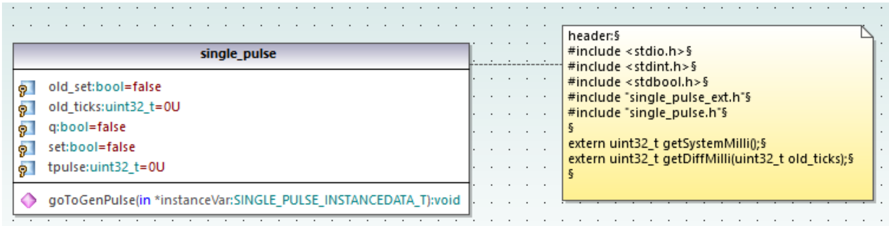


Figure D.5.: Class with user defined attributes and operations. The code generator includes them into the generated code.

D.9. Supported / Unsupported

The code generator supports a subset of the design elements provided by UModel. The supported elements are:

- Hierarchical states
- Events with event name, guard and action
- Initial and final pseudo states
- History states (deep, flat), Extended history handling (see section 3.1.11)
- Choices
- Junctions

The unsupported elements are:

- Constraints
- Multiple state machines/regions in a diagram
- Sync states
- Entry and exit points (not to compare with entry/exit actions within states)
- Terminate and Fork/Join

E. Drawing State-Charts with astah* and astah SysML

astah* formerly known as JUDE is a UML modeling tool created by Japanese company ChangeVision. It is written in Java and can therefore run on different operating systems. This text considers both astah and astah SysML. Differences are explained where necessary.

In opposite to some other tools astah provides a Java API for direct access to the model file. Therefore it is not necessary to export the model (e.g. in XMI format). The Sinelabore*RT* code generator can directly access the model file. This makes the development cycle very fast.

To make it possible for the java runtime to access the astah jar file there are to options:

a: Copy the astah jar files to your Java installation: To make this possible it is necessary to copy the astah* jar interface jar file `astah-community.jar` (or e.g. `astah-pro.jar` if you use astah* professional) from the astah* installation folder into the Java CLASSPATH. This can be done in the same way as for the jdom.jar file. The easiest way is to copy it into the folder where the `codegen.jar` is located. See section 1.5 Installation for the different options. Since version 6.7 also additional jar files are required. For latest information check the Astah* Howto Page on the Sinelabore web site.

Example: You have copied the astah* jar file into the `bin` folder of sinelabore: Let's assume you have two folderes. A `bin` and a `prj` folder. In the `bin` folder all the jar files were located. The model file is located in the `prj` folder. To call the codegen from the `prj` folder use the following command line:

```
user$ ls ../../bin/
JDOM\ LICENSE.txt POI\ LICENSE.txt astah-community.jar codegen.jar jdom.jar
log4j-over-slf4j-1.6.6.jar logback-loader-1.0.9b.jar slf4j-api-1.6.6.jar

user$java -cp "path to codegen bin folder" codegen.Main -t
"oven_pkg:machine_class:oven" -l cx -p ASTAH -o oven oven.asta
```

b: Add the path to the jar files in to the classpath when calling the code generator: This method was used in the example model for astah SysML which can be found in the examples folder. Check out the Makefile how this is done. The class path on your system might be different. On Windows make sure to use the right separator characters in the class path.

```
JAVA=java
JFLAGS= -cp="../../bin/":"/Applications/astah sysml/astah
 sysml.app/Contents/Java/" -Djava.awt.headless=true codegen.Main
 ...
$(JAVA) $(JFLAGS) -l cx -v -p ASTAH -o oven -t "final:oven:machine"
oven_model.asml
```

In case you see a Java exception like shown below the astah* jar files were not found. Carefully check your class path or the path provided in `-Djava.ext.dirs=`

```
Exception in thread "main" java.lang.NoClassDefFoundError:\\
com/change\_vision/jude/api/inf/model/INamedElement at codegen \ldots
```

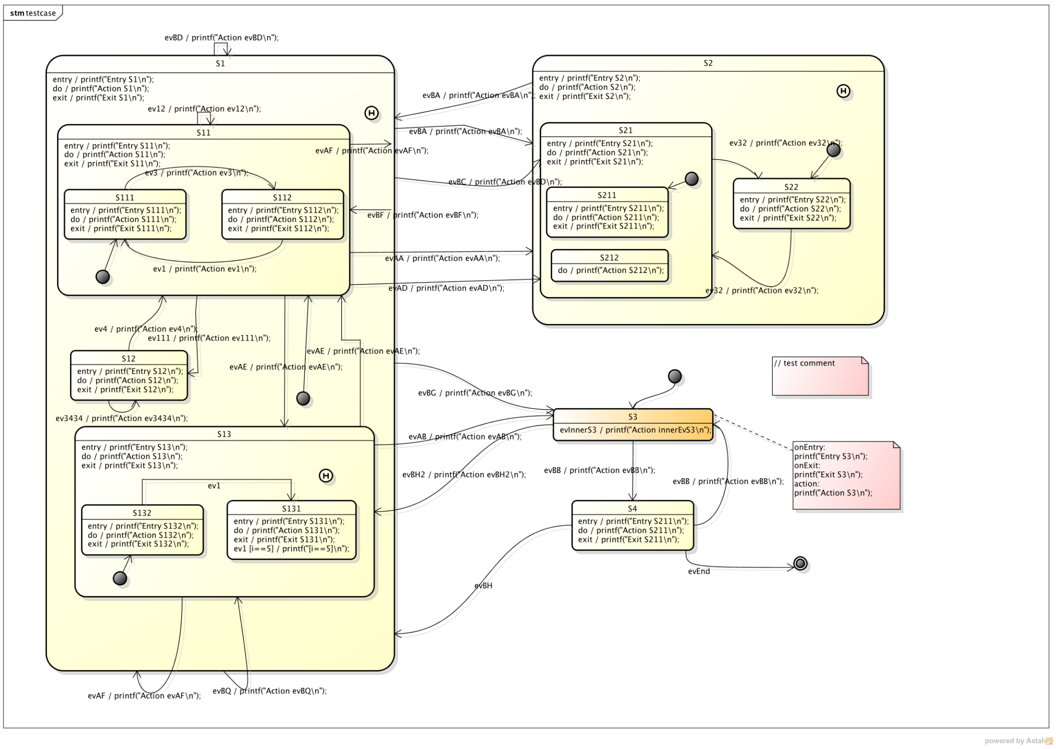


Figure E.1.: A rather complex state chart designed in astah*. Most of the supported elements are used in this diagram.

E.1. Attaching action and include comments

Remember that you can specify code (so called header code) that is simply copied to the beginning of the state machine implementation file. This mechanism allows to add own include files to the generated code. Also you can specify action code that is called every time the state machine gets executed. Put the action- and header code in a `Note` and place it in the class diagram / block definition diagram where it must be linked to the class / block you want to generate code from. The text must start with either `action:` or `header:` as shown in figure E.2. In section 3.1.6 you can find more information about the used syntax.

E.2. Specify the Path to a State Diagram

With the command line flag '-t' you have to specify the path to the state diagram in your model file where you want to generate code from. Start from the root node and then just go down the tree until you reach the state diagram of your choice. The root node itself must be left out as it is the name of the model. Separate each level with a colon. Figure E.3 gives an example.

E.3. States

Astah* allows you to define *entry*, *do* and *exit* activities as well as inner transitions for a state. Unfortunately only one line of text is possible per action. If you want to specify multiple action lines (i.e. code lines) link a comment to a state and provide the code there. An example is shown in figure E.1 for state *S3*. Take care of the required keywords to begin an action in a linked comment. The keywords are *onEntry:* or *onExit* and *action:*.

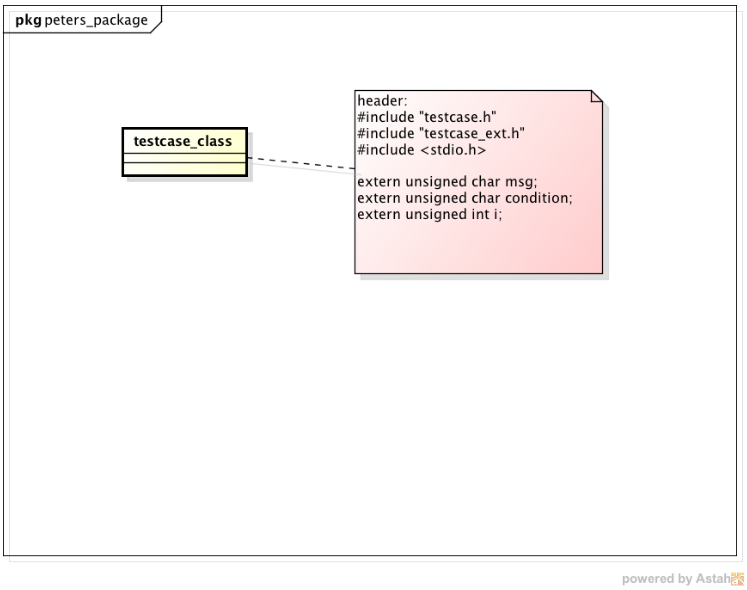


Figure E.2.: Place header and action code in a comment linked to the correct class for astah. When using asta SysML use blocks instead of classes.

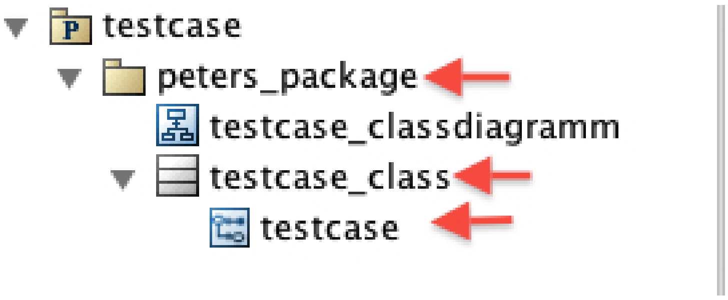


Figure E.3.: To generate code from testcase set the path to `"-t peters_package:testcase_class:testcase"`

E.4. Regions

A region is an orthogonal part of a state. It allows to express parallelism within a state. A state can have two or more regions. Region contains states and transitions. To add a region in astah* right click to a state and select `Add region` from the context menu. See figure E.4 for an example.

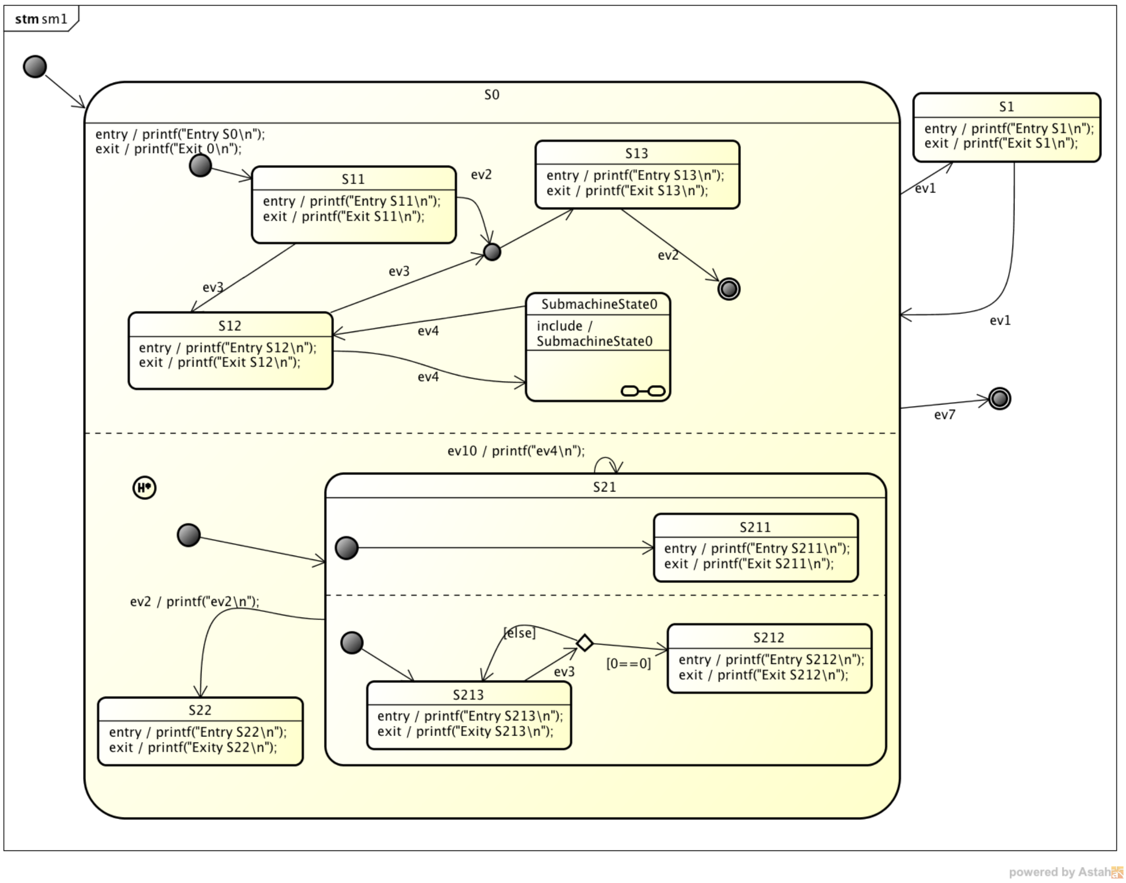


Figure E.4.: Example diagram with regions

E.5. Transitions

Transitions can be drawn from and to states on all levels in the diagram. Click on a transition to specify its details like the triggering event the guard and the action. Transitions can't cross region borders.

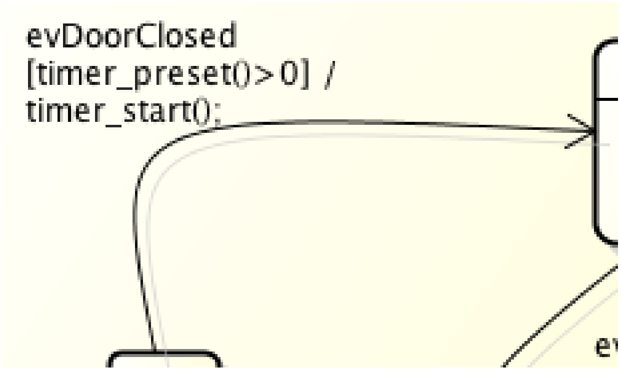


Figure E.5.: Definition of an event, guard an action for a transition between states

Alternatively it is possible to attach a comment to a transition and specify the event[guard]/action in the comment. Usually you don't require to do this (e.g. #rxbuf[i]==0x0d). This type of definition makes most sense together with conditional triggers.

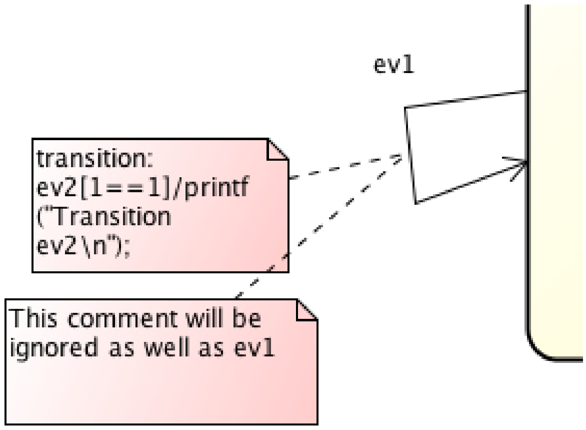


Figure E.6.: Definition of an event using a comment attached to a transition

E.6. History State

Use the `Shallow History` symbol and put it in a state to make it a history state.

E.7. Deep History

Use the `Deep History` icon and put it in a state to make it a deep history state. This has the same effect as adding a Shallow History marker to all composite child states.

E.8. Supported / Unsupported

The code generator supports a subset of the design elements provided by astah*. The supported elements are:

- Hierarchical states
- Events with event name, guard and action
- Initial and final pseudo states
- History states (deep, flat), Extended history handling (see section 3.1.11)
- Choices
- Junctions
- Regions

The unsupported elements are:

- Constraints
- Multiple state machines in a diagram
- Sync states
- Entry and exit points (not to compare with entry/exit actions within states)
- Terminate and Fork/Join

F. Drawing State-Charts with Visual Paradigm

When using Visual Paradigm some tool specific things must be taken care of. The following section discusses these points. First take a look how a rather complex state chart looks like in Visual Paradigm.

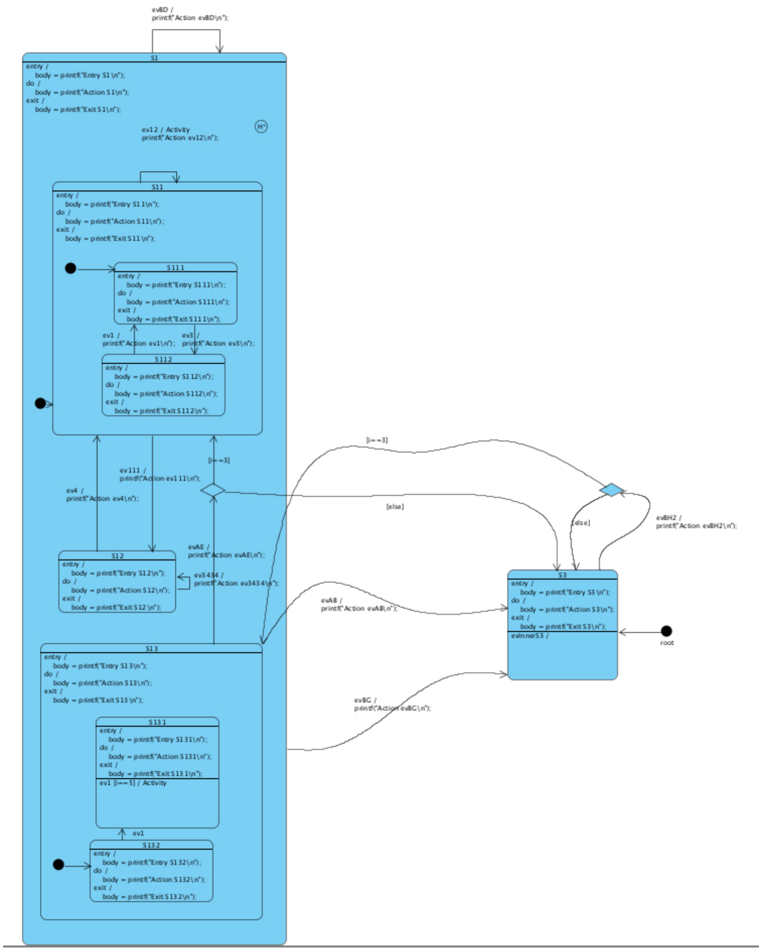


Figure F.1.: A rather complex state chart designed in Visual Paradigm

F.1. Organising your project

The code-generator needs to know how to find your state-based class in the exported XMI diagram. The path to your class in your project tree must be specified on the command line using the -t flag.

The following figure shows the project browser window for the 'oven' example state diagram from the tutorial section. From figure F.2 you can directly derive the path which is `-t 'ModelModel:OvenClass'`. The different parts of the path must be separated with colons. The class diagram must not be specified.

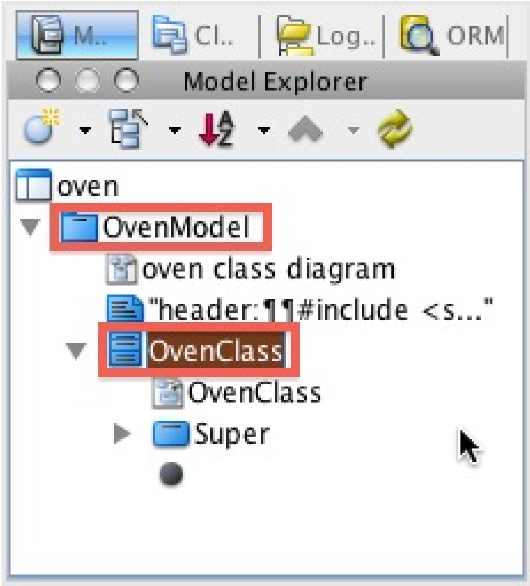


Figure F.2.: Visual Paradigm's project browser. From the used structure the path can be directly derived.

F.2. Exporting your project to XMI

The main purpose of XMI is to enable easy interchange of metadata between modelling tools. Several UML modelling tools support the import and export of XMI file. The exported XMI file is then used as as input for the code-generator. During the development of the XMI standard different versions were created. The latest are based on a XML schema. The code-generator expects version 2.1 for VP generated XMI files. Therefore select `UML 2.1 (XMI 2.1)` in the export dialog before exporting.

The code-generator expects your class at a certain hierarchy in the model (model, class model, class). Tools allow to export a XMI downwards from a selected node e.g. the class model. Then the code-generator will not be able to find the right class to generate code from. Therefore make sure that you select the top level node in the project browser before opening the export dialog. The following figure shows the export dialog. You can open it by selecting `Project -> Import/Export -> Export Package to XMI`.

F.3. Attaching action and include comments

You can specify code that is just copied at the beginning of the state machine c-file. Also you can add action code that is called every time the state machine is called. This can be done by using comments starting with the keywords `action:` and `include:` (see section 3.1.6). Link these two comments to the class owning the state machine. See figure F.4. You must use plain text in the comment. It is not possible to use html comments. See figure F.4 on page 134.

F.4. States

VP allows you to define entry, do and exit activities as well as inner transitions for a state. Also multiline code statements are possible per action. To specify child states you have to add a region to a state first. If you have doubts cross-check in the model view that a state is a child of the parent. See figures F.1 and F.2 for an example. If you edit actions the action name should be deleted for clarity. See figure F.5 on page 135.

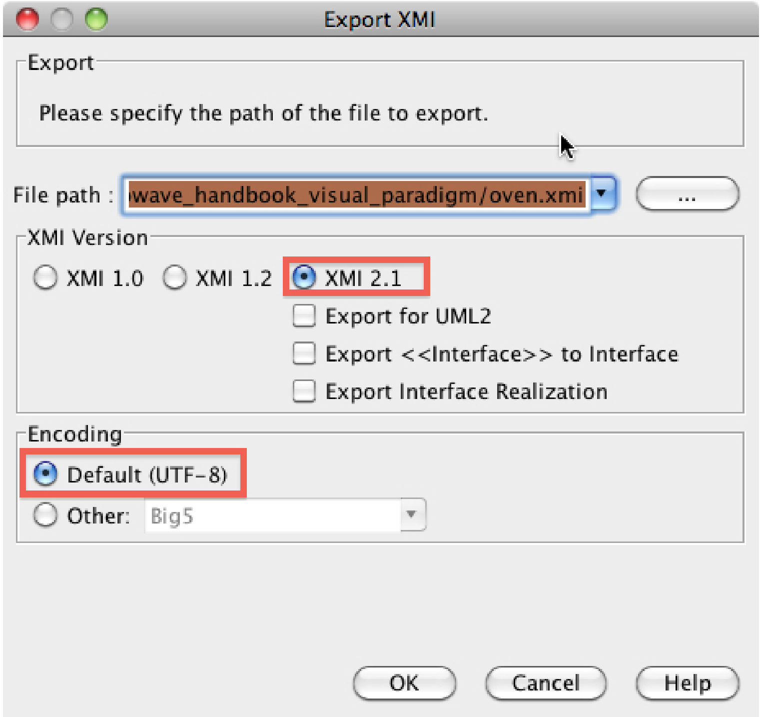


Figure F.3.: VP's export dialog. Ensure that XMI Type is set to `UML 2.1`

F.5. Transitions

Open the property dialog of the transition and add a trigger (i.e. event) that triggers the transition. You must select 'Signal Trigger' as trigger type. Then close the dialog. Open the properties dialog again. Now you can add an optional action code and guard. It is possible to provide multiple code lines for the transition's action which is a nice feature. Delete the transition name ('action' by default) for clarity. To display the action code right click on the diagram and select 'Presentation Options → Transition Display Options → Show Transition Effect Body'. Otherwise the action code is not shown. See figure F.6 on page 135.

F.6. History State and other Pseudo States

Just add final states or history states from the tool bar into your diagram.

F.7. Supported / Unsupported

The code generator supports a subset of the design elements provided by VP. The supported elements are:

- Hierarchical states
- (Signal-)Events with event name, guard and action
- Initial and final pseudo states
- History states (deep, flat)
- Choices
- Regions in a diagram

The unsupported elements are:

- Constraints

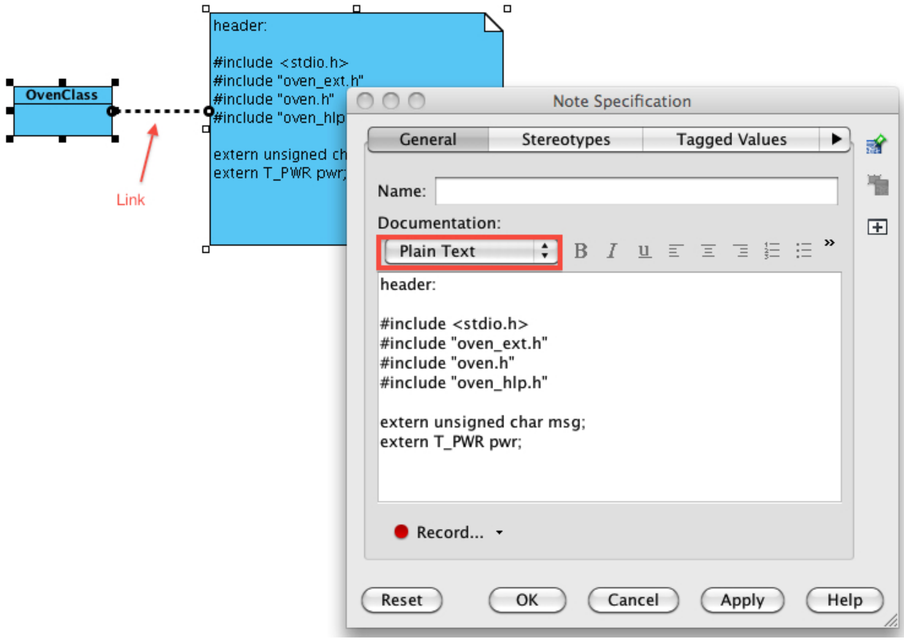


Figure F.4.: Specify header and action code using comments. Make sure the format of the comment is 'plain text'.

- Syncstates and junctions
- Entry and exit points (not to compare with entry/exit actions within states)
- Terminate and Fork/Join
- Extended history handling

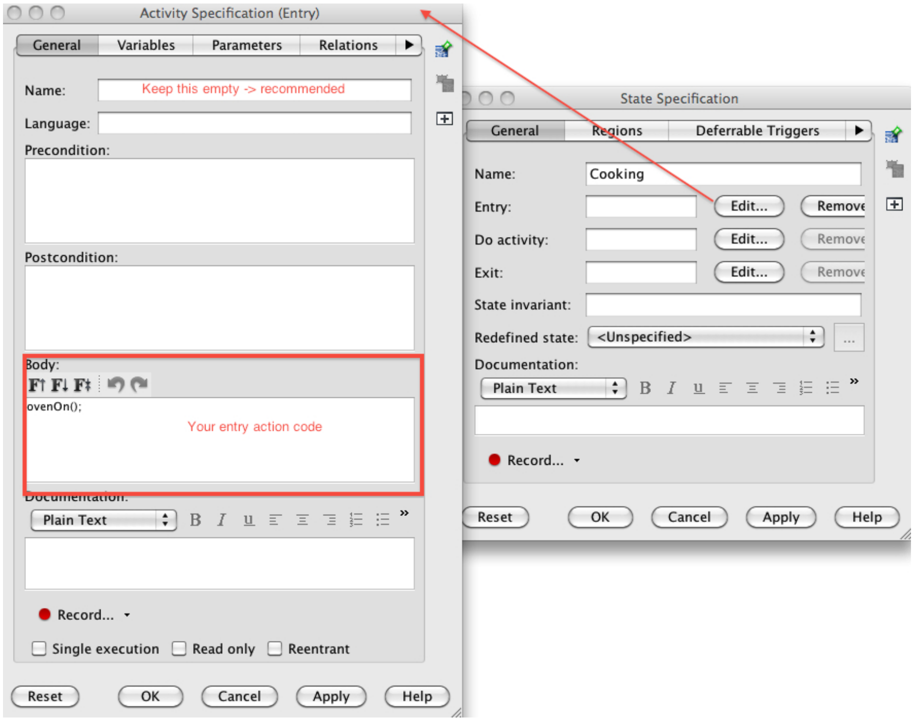


Figure F.5.: State definition. Define the action code and eventually an internal event of a state.

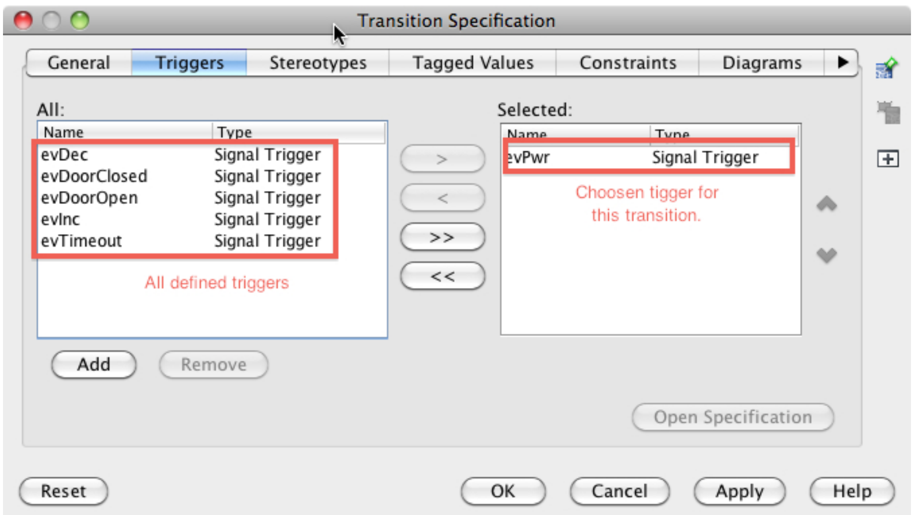


Figure F.6.: Trigger definition. Define signal triggers only and select the one which should trigger the transition. On the tab card 'General' you can set the guard and action code.

G. Drawing State-Charts with Enterprise Architect

When using Enterprise Architect, there are a number of tool specific issues that need to be considered. These are discussed in the following section. Let us first look at how a rather complex state machine looks in Enterprise Architect.

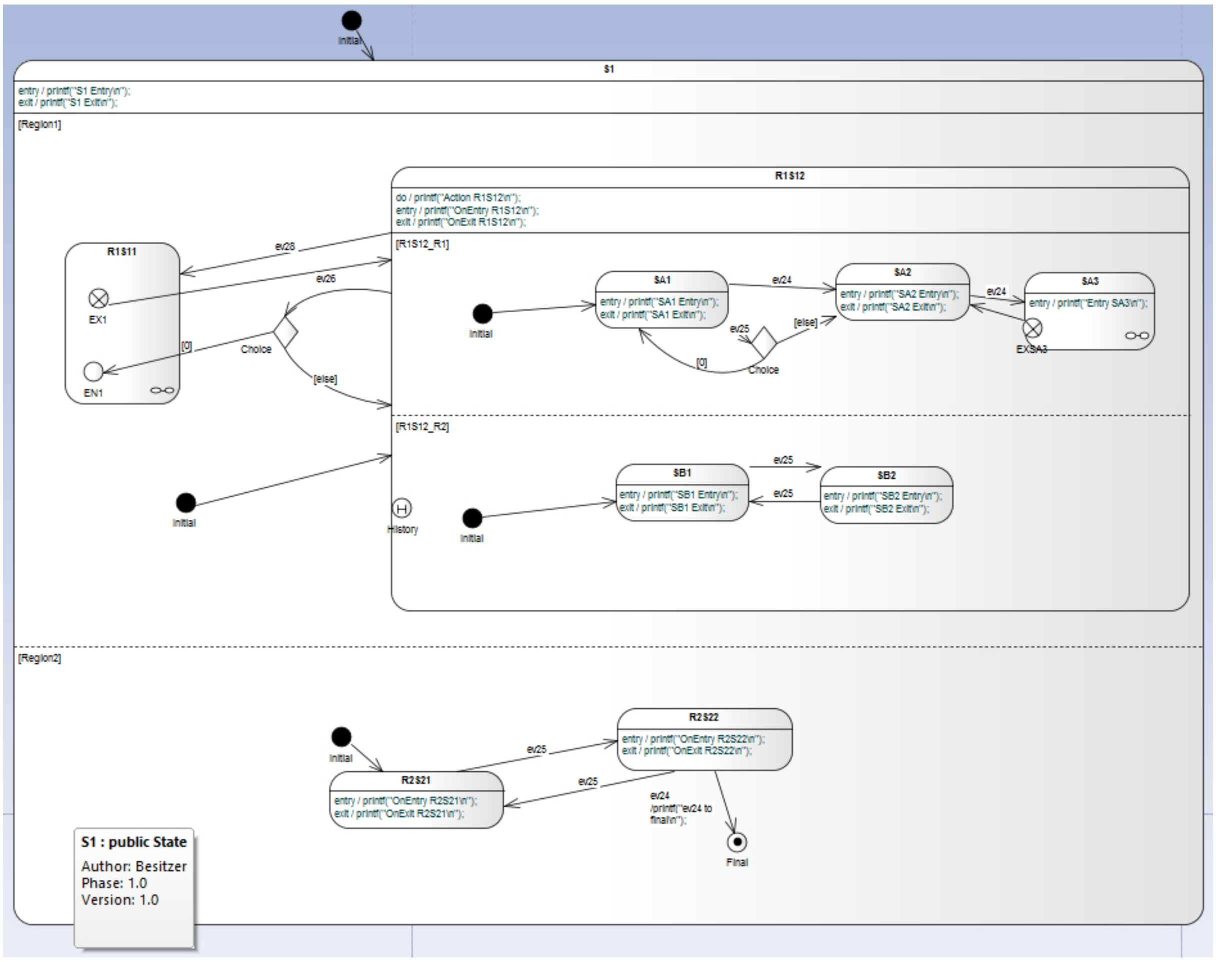


Figure G.1.: A rather complex state chart designed in Enterprise Architect. It shows regions, sub-machines and several pseudo-states like a choice state.

G.1. Organizing your project

Typically you have organized your project into packages. These packages contains classes which again can contain a state machine. The code-generator needs to know how to find your state-based class in the exported XMI diagram. The path to your class in your project tree must be specified on the command line using the *-t* flag. If you want to create several state diagrams it is necessary to put them into separate packages!

The following figure G.2 shows the project browser window for the state diagram from above. In this project not just the *complex_class* is present but also two other classes *Class1* and *Class2*. From the EA *Project Browser* window you can directly derive the path to the state machine which is `-t "Model:Class Model:complex_class"` in this case. The different parts of the path must be separated by colons.

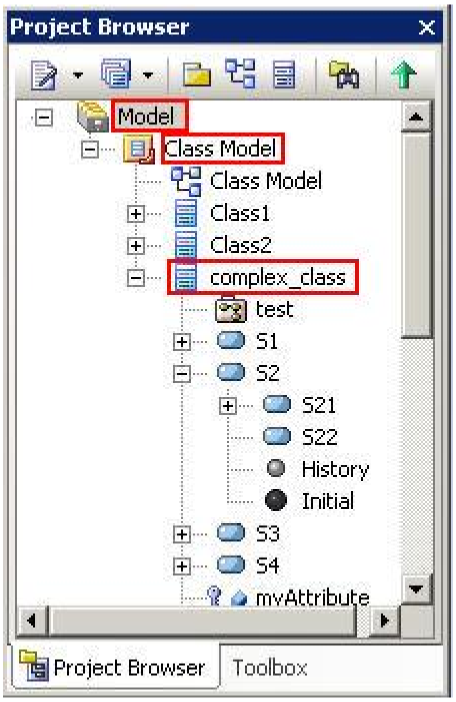


Figure G.2.: Enterprise Architect's project browser. From the used structure the path can be directly derived.

G.2. Exporting your project to XMI

The main purpose of XMI is to enable the exchange of models between between modelling tools from different vendors. Therefore, most UML modelling tools support the import and export of XMI files. The exported XMI file is the basis for the code generator. During the development of the XMI standard, several versions have been have been created. Use the latest available version of UML/XMI. At the time of writing, it is version *version 2.5.1*.

The code-generator expects your class at a certain hierarchy in the model (model, class-model, class). EA always exports from the selected node – e.g. the class model – downwards. Then the code-generator is not able to find the right class. Therefore *always select the top level node in the project browser before opening the publish dialog*. E.g. *Model* in the example above.

G.3. State Details

Enterprise Architect allows you to define *entry*, *do* and *exit* actions of a state. There are currenty three options to specifiy the enty/do/exit action code.

(a) Simply select the state and use the Features/Behaviour field to directly type in the action. See figure G.3 as example.

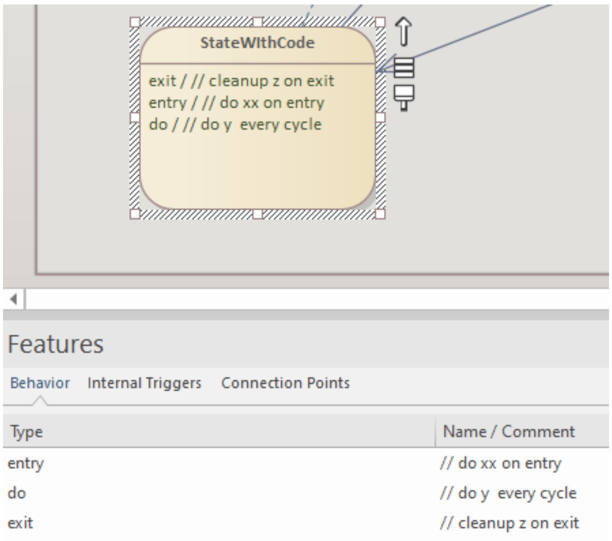


Figure G.3.: Action code specified in the name field of the action function. This is ok if the action code fits in one line.

(b) It is also possible to specify code for the entry/exit/do action code using the *Behaviour* text field. See figure G.4 as example. This makes it easier to specify longer action codes. If the code is found in the behaviour property field, the text in the action name field is no longer used. If you have specified more than one entry/exit/do action, the code from the different entry/exit/do behaviour fields will be merged. Note that you must always enter some text in the action name field. As example type in a short description of what the action code does, as shown in the diagram. *Note: For some reason there is no syntax highlighting for this field.*

Figure G.4.: A state with entry/exit/do code specified following option b) in the state function's behaviour field.

(c) It is also possible to specify code for the entry/exit/do action code using the *Code* text field. This entry field has syntax highlighting. If code is available there all other fields from above are ignored. Please note that you always have to put in some text in the operation name field. As example type in a short description of what the action code does, as shown in the diagram. *Note: The code field is not in the UML part of the XMI export but in the EA specific part.*

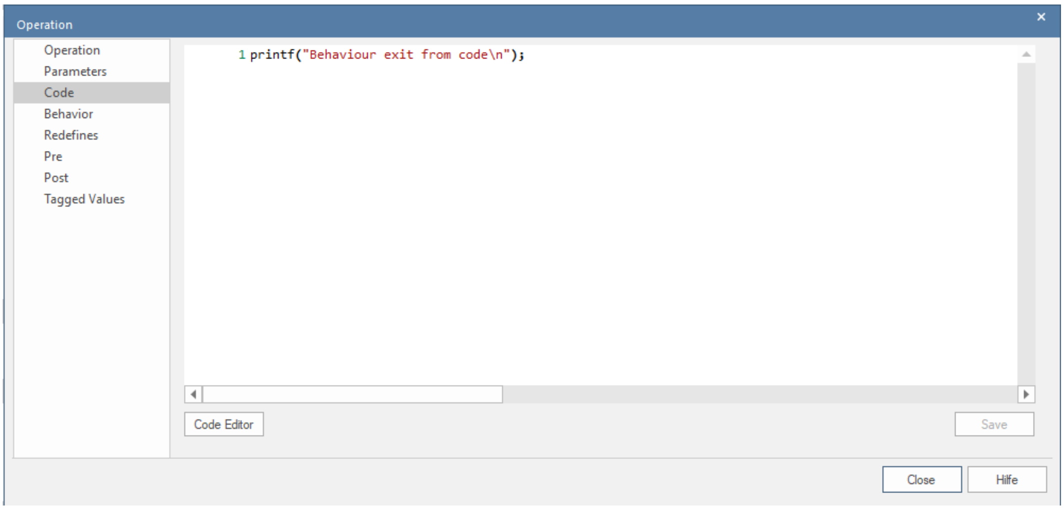


Figure G.5.: A state with entry/exit/do code specified following option b) in the state function's behaviour field.

(d) You can use a comment and specify entry/do/exit action code as plain text. The comment text must follow the syntax as described in section 3.1.3. The comment must be *linked to the appropriate state*. To edit the text double click on the comment and enter the definitions as needed. You can attach such comment to a normal state, a child state of a composite state or a sub-state machine state. Make sure to use *plain text* (i.e. no special font or color etc.) the code-generator is not able to extract the text correctly otherwise. Use this option if you want to specify *inner events* in a state. There seems to be no other way to specify such events in EA.

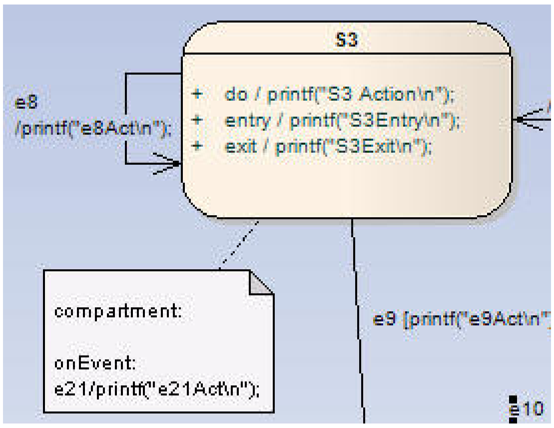


Figure G.6.: A state with entry/exit/do code specified following option a). And an inner-event following option c)

G.4. Regions

A region is an orthogonal part of a state. It allows to express parallelism within a state. A state can have two or more regions. Region contains states and transitions. To add a region in EA

1. Right click to a state and select `Advanced`
2. Then select `Define Concurrent Substates`.

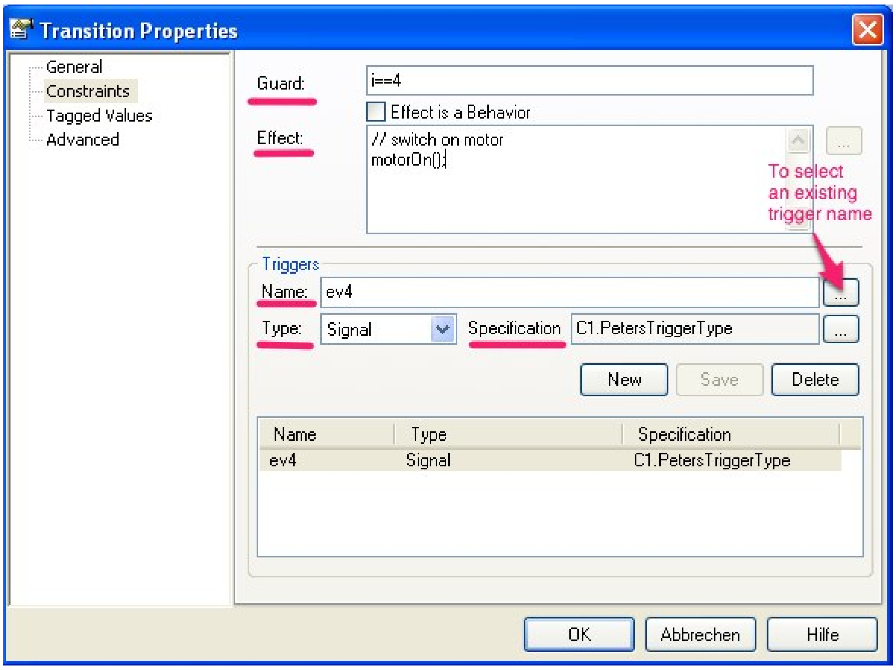
3. Then add as much concurrent regions as you need

G.5. Transitions

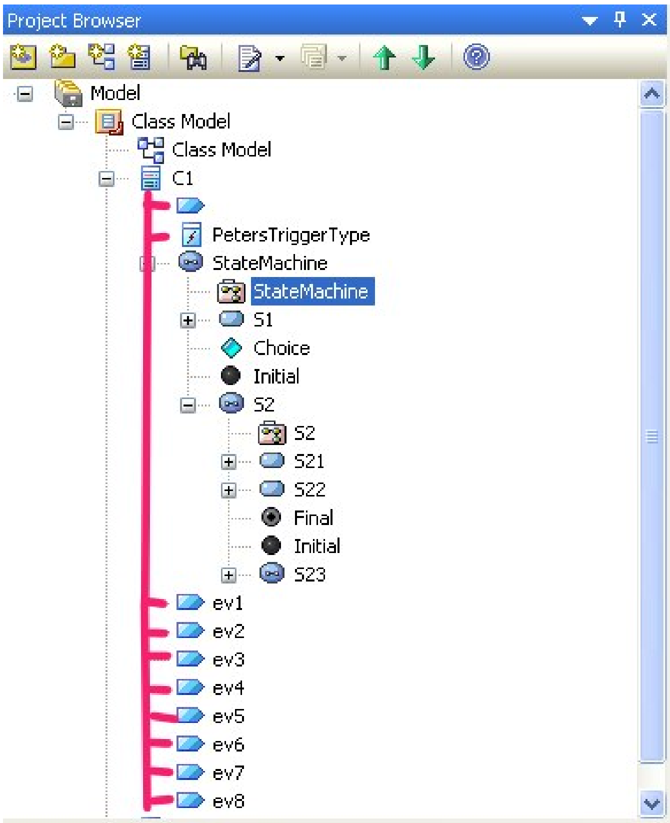
Double-clicking on a transition brings up the *transition properties* dialog (see figure G.7 a). This dialog allows you to define the event that triggers the state transition. You can specify a guard (i.e. a statement that evaluates to true or false), an effect of the trigger and the trigger itself. The trigger type *must be* set to type *Signal*. EA enforces the event type specification. This type is not needed by the code-generator and it is recommended to use only one type for all the triggers. To re-use an existing trigger text use the selection field (three dots). Otherwise a new event is created every time even if you use the same trigger text.

A transition can have more than one trigger. This is the same as two transitions with one trigger each. But can lead to clearer diagrams in case of many transitions between two states sharing the same action code.

The code-generator expects triggers on class level in your model. Move the triggers to that level in the Project Browser. See figure G.7 b) for an example.



(a) EA9 transition properties dialog. The dialog of older versions of EA looks a bit different but offered basically the same fields.



(b) Triggers must be stored directly under the class. Otherwise the code-generator does not find them. Move them below the class in the Model Browser as shown in this figure.

Figure G.7.: Relevant aspects when using transitions.

G.6. History State

Use the *Shallow History* pseudo-state if you want to make a state a history state. Place the pseudo-state in the state that should have history.

G.7. Deep History

Use the *Deep History* pseudo state if you want to make all child states history states. Place the pseudo-state in the state that should have history. Using a Deep History pseudo state has the same effect as adding a Shallow History marker to all composite child states.

G.8. Choices

With EA 7.5 it seems to be not possible anymore to specify transitions without a trigger name (which is in general ok). But for transitions starting from a choice state only guards should be specified (no trigger!). To ensure a clear design only type in one or more spaces as trigger name. The generator detects this and ignores the trigger name. It is recommended to use the same "empty" event for all the transitions starting from choices.

G.9. Constraints

The code-generator can automatically derive testcases from the EA state-machine model. See section 3.19 Testing Statemachines for more details.

G.10. Adding Operations and Attributes to Classes

It is possible to add attributes and operations to the class containing the state machine. Clicking on the class shows the class features where attributes and operations can be easily added. For operations it is also possible to add code in the behaviour field (not the code field). Attributes are added to the instance data and always require a default value. Operations are mapped to C-functions and added to the header and implementation files of the state machine (only if an implementation is given)

The following figure G.8 shows an attribute and operation added to a class. For operations and attributes it is recommended to use own data types defined as primitive types. Create them in the same package where the class is in. If you use standard EA data types there is an automatic mapping of EA types to own data types provided in the configuration file.

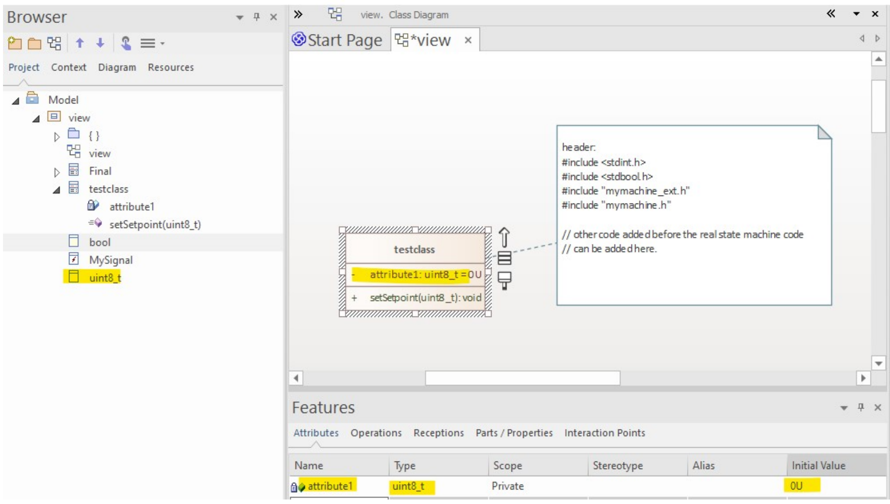


Figure G.8.: Adding a new attribute to a class with a state machine. Use own data types defined as primitive types in the same package where the class is in.

G.11. Sub-Machine States

Sub-machine states allow to “hide” the children states of a parent state with hierarchy. You only see the content of the sub-machine state if you double click on the state which opens the internal view. From the code generator point of view a sub-machine is a normal state with children. There is absolutely no difference compared to a normal hierarchical state.

To create a sub-machine do the following:

1. Place a normal state
2. Right click on the state and select `New Diagram`
3. Select `Composite Structure Diagram`. Now an ∞ symbol indicates the transformation.
4. Double click to the state and you are able to place children into the sub-machine.

Note: Do not mix up a **sub-machine** with a **state-machine**. In EA both have the same symbol i.e. a state with ∞ sign on the lower right. But a sub-machine is a normal state transformed into a sub-machine. Placing state-machines in another state is possible in EA but has no meaning and is not supported by the code generator!

Using a sub-machine state instead of a state has some other consequences:

1. It is not possible to connect a transition starting from a state outside the sub-machine to a state inside the sub-machine. Or vice versa. Use entry and exit pseudo states for this purpose (see next section).
2. EA does not allow to "transform" a sub-machine state into a state with children. You have to do this manually by moving states etc. in the Model Browser.

The following figure shows two state machines which produce exactly the same code – one using a sub-machine, one using a normal hierarchical design.

In principle sub-machines can be located in normal states (or states) or regions.

G.12. Entry and Exit Points

Note: Entry and exit points are only useful together with a sub-machine. Read the previous section before if you are not familiar with the usage of sub-machines.

When entering a sub-machine usually the initial state is entered. I.e. the transition ends at the border of the sub-machine state. If this should not be the case for a specific transition it is possible to place an entry point inside the sub-machine state. This entry point serves as glue between the sub-machine state and the internal of the sub-machine.

Exit states provide a similar function. By default only transitions can be modelled starting from the sub-machine. If a transition should start from a specific state inside the sub-machine and enter a state outside the sub-machine an exit state can be used. Again this exit state serves as glue between the transition starting inside the sub-machine and ending one level up at another state.

To use entry and exit points in your model use the following recipe:

1. Drag and drop an entry and/or exit point in a sub-machine state
2. Give the points a meaningful name.
3. Double click on the sub-machine state to open the sub-machine diagram
4. Place references of the points here. **Do not create new entry/exit points with the same name!**
5. Connect transitions

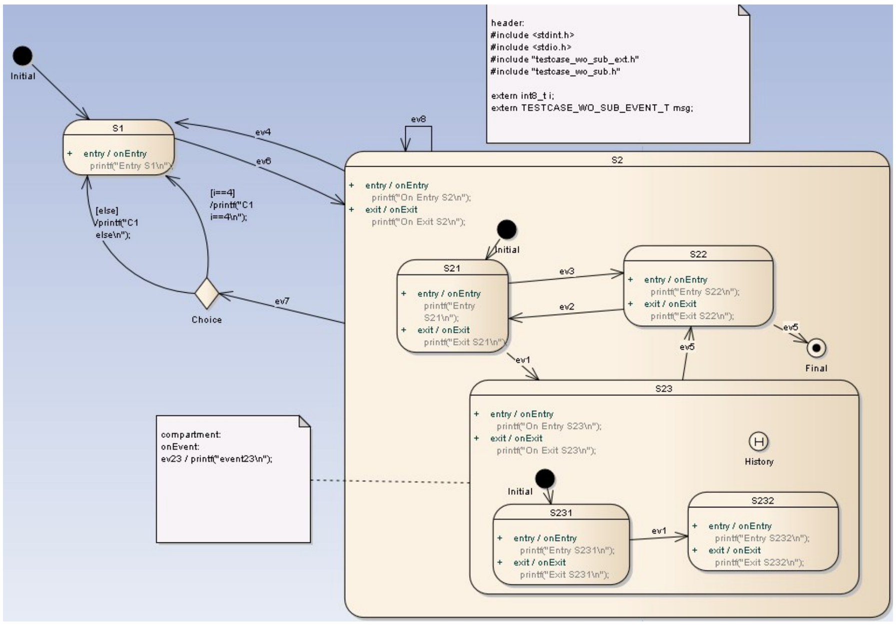
Alternatively (since version 12 of EA) follow the recipe below. In this case the entry and exit points are displayed on the border of the state with sub-machine.

1. Select the sub-machine state that should have and entry or exit point
2. Right click on the sub-machine state to bring up the context menu and then select 'Add'.
3. Add either and entry or exit point. See figure G.10.
4. Inside the sub-machine state drag and drop the existing entry/exit point into the diagram (as link). **Do not create new entry/exit points with the same name here!**
5. Connect transitions

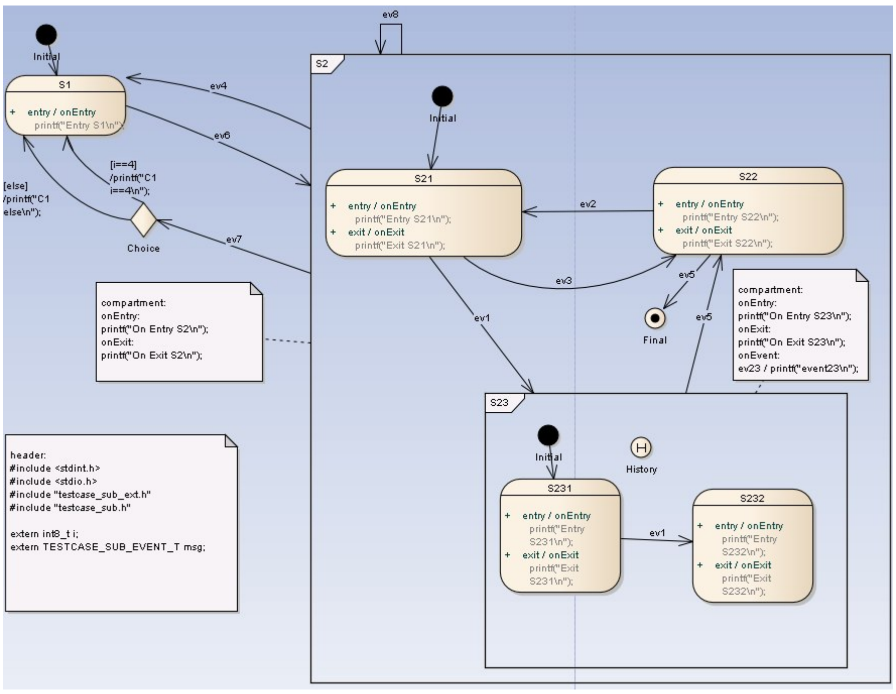
Limitations when using entry and exit points:

- An exit point can have more than one incoming transitions inside the sub-machine. But only one outgoing transition from the sub-machine state.
- An entry state can have more than one incoming transitions on the sub-machine state. But only one outgoing transition inside the sub-machine.
- The transition leaving the exit or entry point most end in a normal state. Chaining of entry / exit states or connecting these transitions to choices or junctions is not allowed.

The following figure G.12 shows an example for a diagram with a sub-machine and the internals of this sub-machine.

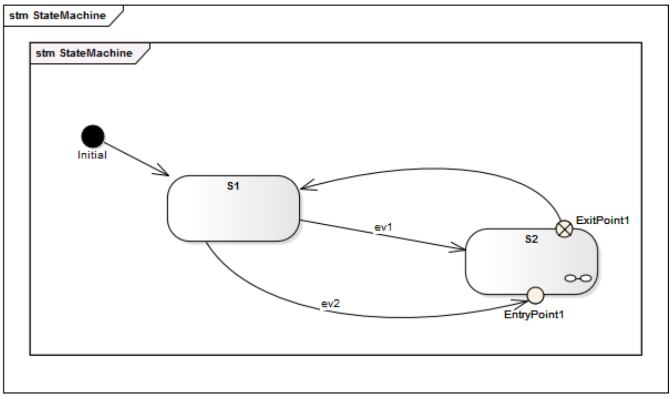


(a) Machine m1 using nested states

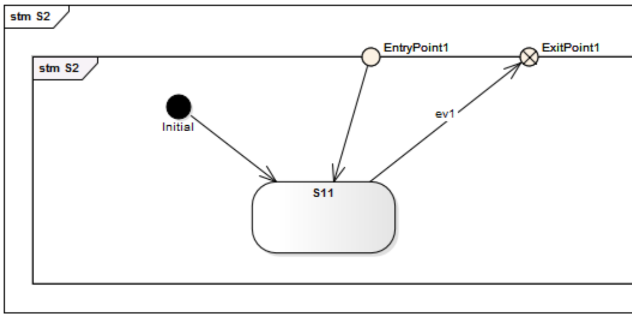


(b) Machine m2 using sub-states

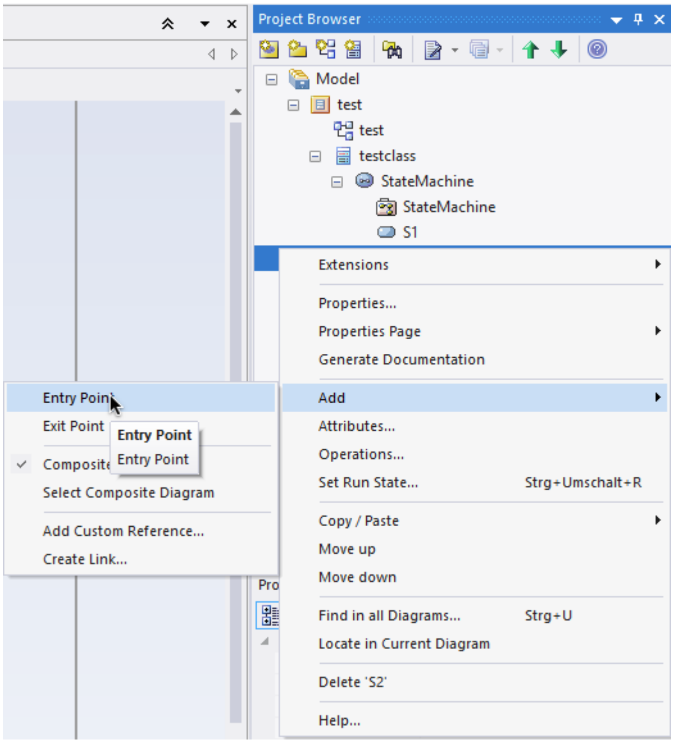
Figure G.9.: This figure shows two state machines producing exactly the same code but one is using a sub-machine state (internals set visible) and one uses a normal hierarchical design. I.e. m1 ⇔ m2.



(a) Top level state diagram with sub-machine

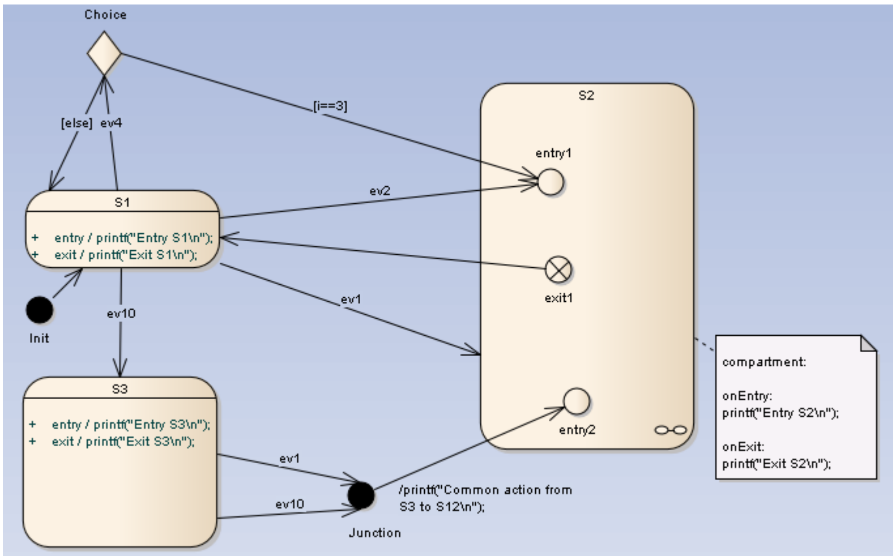


(b) Sub-machine state

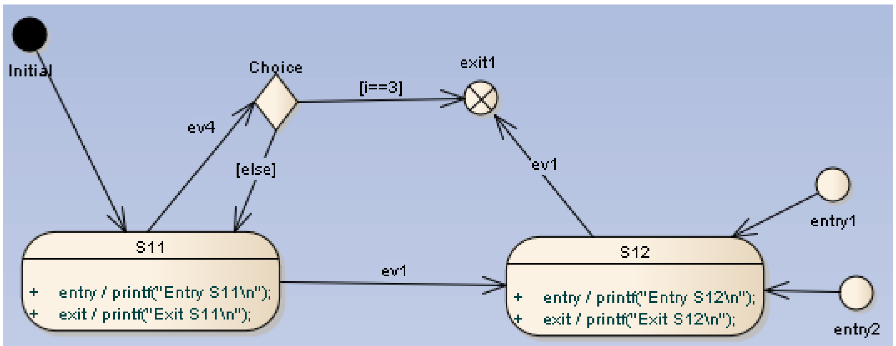


(c) Context Diagram

Figure G.10.: Create an entry or exit point using the context menu of a sub-machine state.



(a) Top level diagram with the sub-machine state S2



(b) Internals of the sub-machine state S2

Figure G.11.: This figure shows the top level diagram with a sub-machine state and the internals of the sub-machine state. The transitions between these two diagrams are glued together using entry and exit points.

G.13. Supported / Unsupported

The code-generator supports a subset of the design elements provided by Enterprise Architect. The supported elements are:

- Hierarchical states
- (Signal-)Events with event name, guard and action
- Initial and final pseudo-states
- History states (deep, flat), Extended history handling (see section 3.1.11)
- Choices
- Junctions
- Constraints
- Sub-machine states on various levels
- Entry and exit pseudo-states
- Regions on various levels

The unsupported elements are:

- Syncstates
- Terminate and Fork/Join

H. Drawing State-Charts with Modelio

When using Modelio some tool specific things must be taken care of. The following section discusses these points. First take a look how the microwave oven example from the introduction section looks like in Modelio.

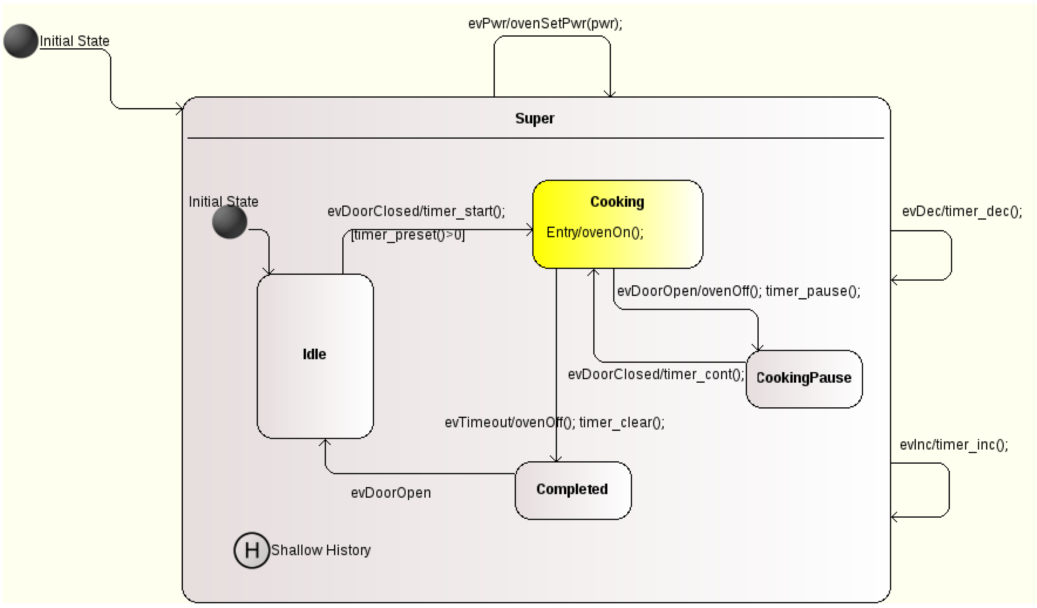


Figure H.1.: The microwave oven state chart designed in Modelio

H.1. Organizing your project

The code-generator needs to know how to find your state-based class in the exported XMI file. The path to your class in your project tree must be specified on the command line using the -t flag.

The following figure H.2 shows the project browser window for the 'oven' example state diagram from the tutorial section. You can easily derive the path which is `-t 'OvenClass'` in this case. In case of multiple packages the different parts of the path must be separated with colons.

H.2. Exporting your project to XMI

The exported XMI file is used as input for the code-generator. Right click on the oven package in the model tree and select `'XMI'` → `'Export to XMI'`. Select `'OMG UML 2.4.1'` and deselect `'Added Modelio annotations'` in the export dialog before exporting.

H.3. States

Modelio allows you to define *entry*, *do* and *exit* activities for a state. Right click to the state and select 'Create an element' → 'Internal transition'. Activities created this way directly

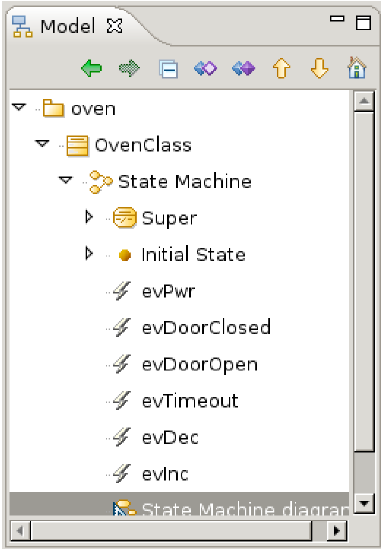


Figure H.2.: Modelio's project browser. From the used structure the path can be directly derived.

in the graphics are immediately visible. If created in the model tree (right click on the state in the model tree → Create an element → Internal transition) it is necessary to drop the activity into the state before they are displayed.

Internal transitions are presently not supported by Modelio. To add an internal transition to an state attach a *note* to a state and specify the transition following the syntax shown in figure 3.1.3.

To add child states to a parent state just drop a new state inside its parent.

H.4. Regions

A region is an orthogonal part of a state. It allows to express parallelism within a state. A state can have two or more regions. Region contains states and transitions. States in regions can contain regions again.

The following figure H.3 shows a state machine with a state containing two regions.

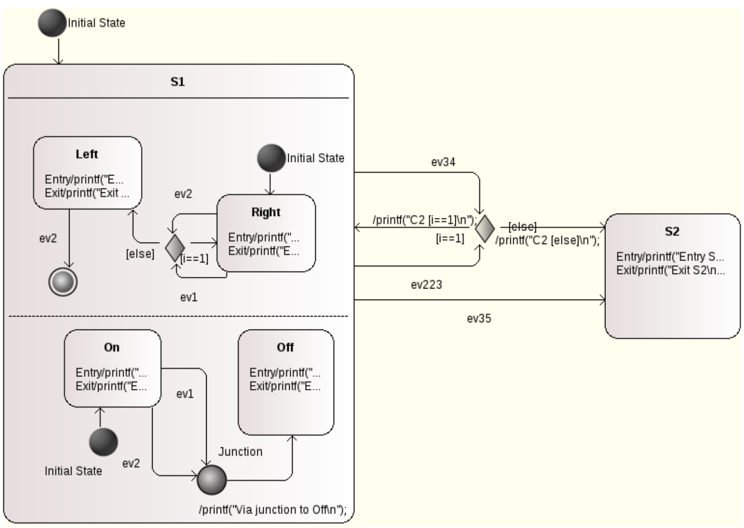


Figure H.3.: Specify the linked sub-machine in the state properties.

Rules when using regions:

– Transitions must not cross region borders. I.e. it is not allowed to add a transition from `Right` to `Off`.
– A region might contain a history state, choices, final states and normal states
– To express that a transition should fire if the machine is in `Right` and `Off` use a conditional transition like follows: `#machineIsInRight(...) && machineIsInOff(...)`
– Regions can contain states which contain regions again

H.5. Sub-Machines

A sub-machine state specifies the insertion of the specification of a sub-machine state machine. The state machine that contains the sub-machine state is called the containing state machine. A sub-machine state is semantically equivalent to a composite state but you only see the top level state. Sub-machines are usually used to "hide" complexity in the containing state machine.

To connect a sub-machine with the containing state you have to link the state containing the sub-machine to the sub-machine diagram.

To link a state to a sub-machine specify the sub-machine in the state properties as shown in figure H.4.

State	Value
Name	S2
Sub-Machine	subS2 (from test)

Figure H.4.: Specify the linked sub-machine in the state properties.

To link transitions between states of the containing state machine with states in the sub-machine diagram you have to use entry and exit points. In Modelio add *connection points* to the parent state of the sub-machine (e.g. S2). And add *entry - and exit points* to the sub-machine diagram (e.g. EX1).

Defining the *connection point reference* in the *connection point properties* finally links the connection point on the top level state to an entry - or exit point in the sub-machine as shown in figure H.5.

ConnectionPointReference	Value
Name	EX1
Entry/Exit	EX1 (from subS2)

Figure H.5.: Link the connection point to an entry – or exit point in the sub-machine.

After linking the connection point to the entry - or exit point the connection point changes its icon and shows either an exit icon or an entry icon. The following figure H.6 show the top level diagram with a state containing a sub-machine (S2) and the sub-machine in figure H.7.

H.6. Transitions

To set transition properties select the transition. Now you define the trigger, guard and action statements. To make sure Modelio knows the Signal it must be defined beforehand in the model tree (add a signal and set its *kind property* to *Signal*). See figure H.1 how the model tree looks like with some signals defined.

To display the transition text enable the `Show label` property. Otherwise the transition trigger and action code is not shown.

H.7. History State and other Pseudo States

Just add final states or history states from the tool bar into your diagram.

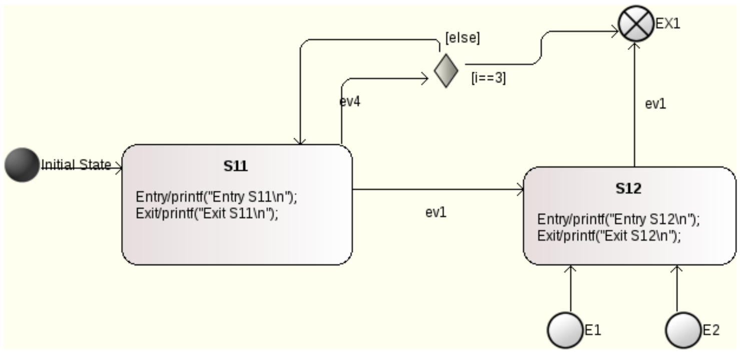


Figure H.6.: Diagram with state S2 referencing a sub-machine.

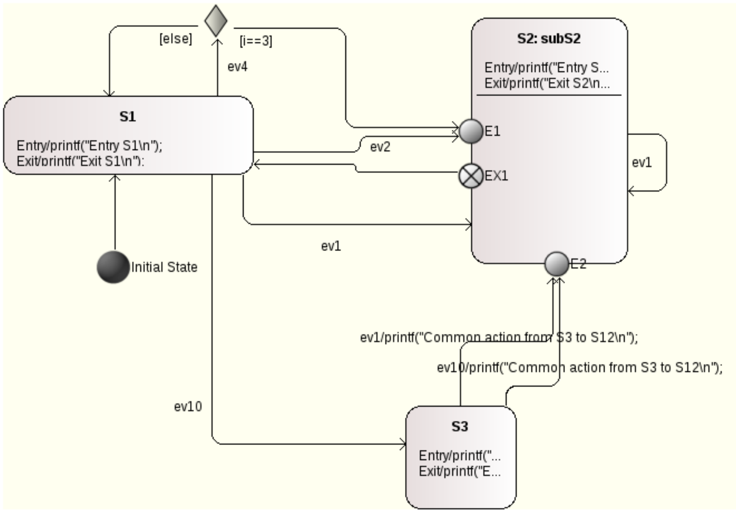


Figure H.7.: Sub-machine diagram.

H.8. Attaching action and include comments

By default the code generator includes the minimally needed header files into the generated code. But often it is necessary to add further includes or define local variables etc. In this case you can specify code that is just copied at the beginning of the state machine implementation file.

You can also add code that is added to the beginning of the state machine handler function. The code can be used to perform some actions each time the state machine is executed.

Add this code to your model by using comments starting with the keywords `action:` and `include:` (see section 3.1.6). Link those comments to the class owning the state machine. An example is shown in figure H.9. You must use plain text in the comment. It is not possible to use html comments.

H.9. Supported / Unsupported

The code generator supports a subset of the design elements provided by Modelio. The supported elements are as follows:

- Hierarchical states
- Regions and “regions in regions”
- Sub-machines in a top level state
- (Signal-)Events with event name, guard and action

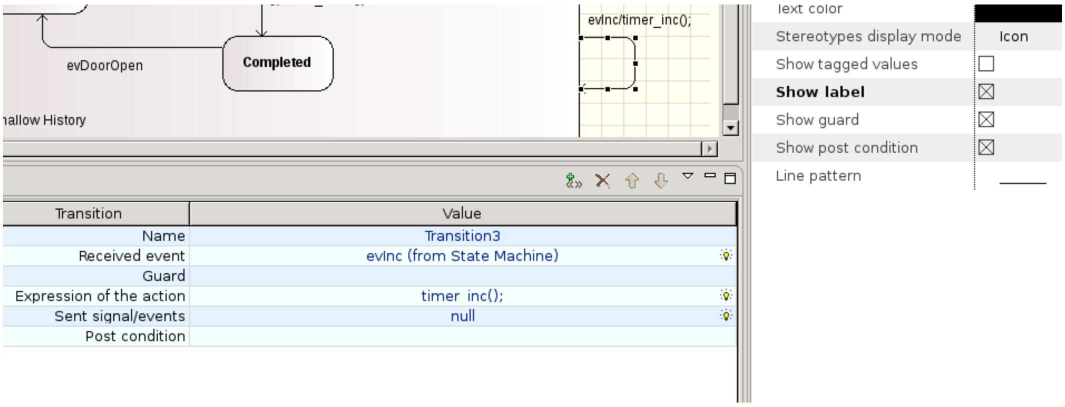


Figure H.8.: Transition properties. Ensure that the 'received event' was previously defined and the (from . . .) is shown. Otherwise the code generator generates an error later on.

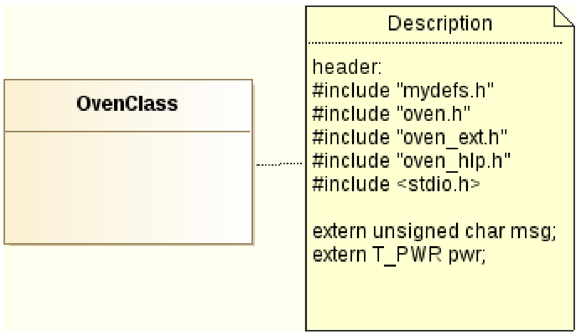


Figure H.9.: Specify header and action code using comments. Make sure the format of the comment is 'plain text'.

- Initial and final pseudo-states
- History states (deep, flat), Extended history handling (see section 3.1.11)
- Choices
- Junctions

The unsupported elements are:

- Constraints
- Sync-states and junctions
- Entry and exit points (not to compare with entry/exit actions within states)
- Terminate and Fork/Join

I. Drawing State-Charts with Eclipse Papyrus™

Code generation is supported for Eclipse Papyrus version 2022-12. When using Papyrus, some tool-specific things need to be considered. The following sections cover these points.

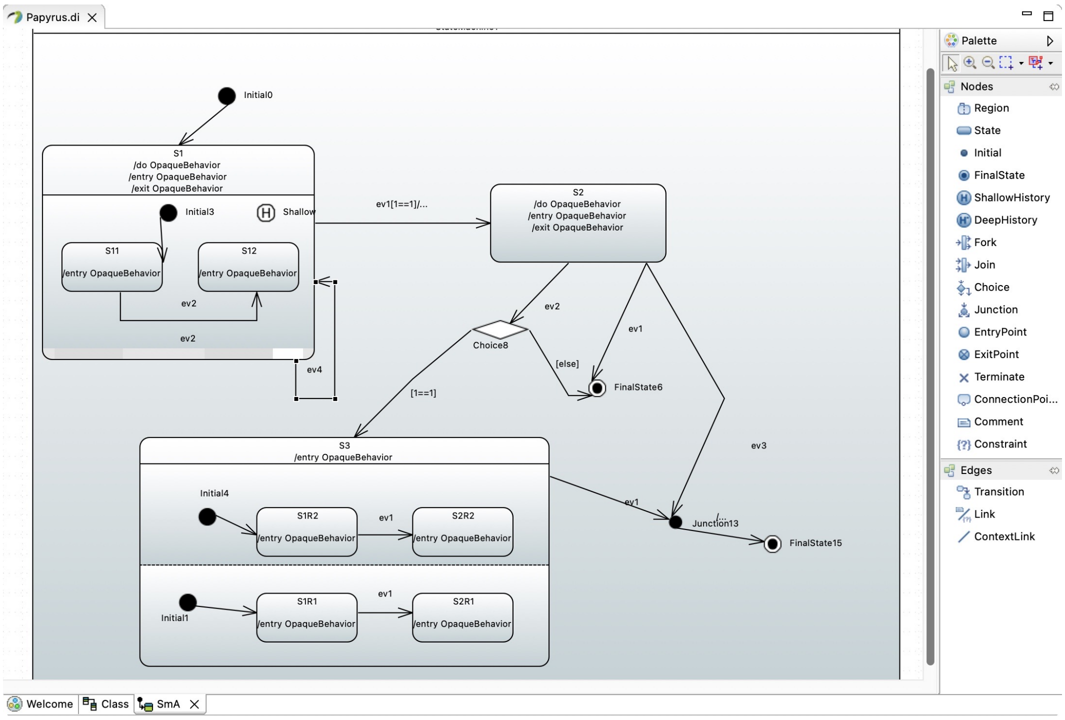


Figure I.1.: An example state machine diagram created with Papyrus

I.1. Organizing your project

The code-generator needs to know how to find your state-based class in the UML file. The path to your class in your project tree must be specified on the command line using the `-t` flag. For the model tree shown in figure I.2 an example command line to generate C-code from a file called would look like as follows:

```
...  -t "MyModel:MyPackage:A" -l cx -p Papyrus -o testcase Papyrus.uml
```

I.2. States

Papyrus allows you to define *entry*, *do* and *exit* activities for a state. Right click to the state and select 'show properties' to bring up the properties view. In the view you can add the activities. It is important that actions are always only created as OpaqueBehaviour. Otherwise code generation cannot generate activity code.

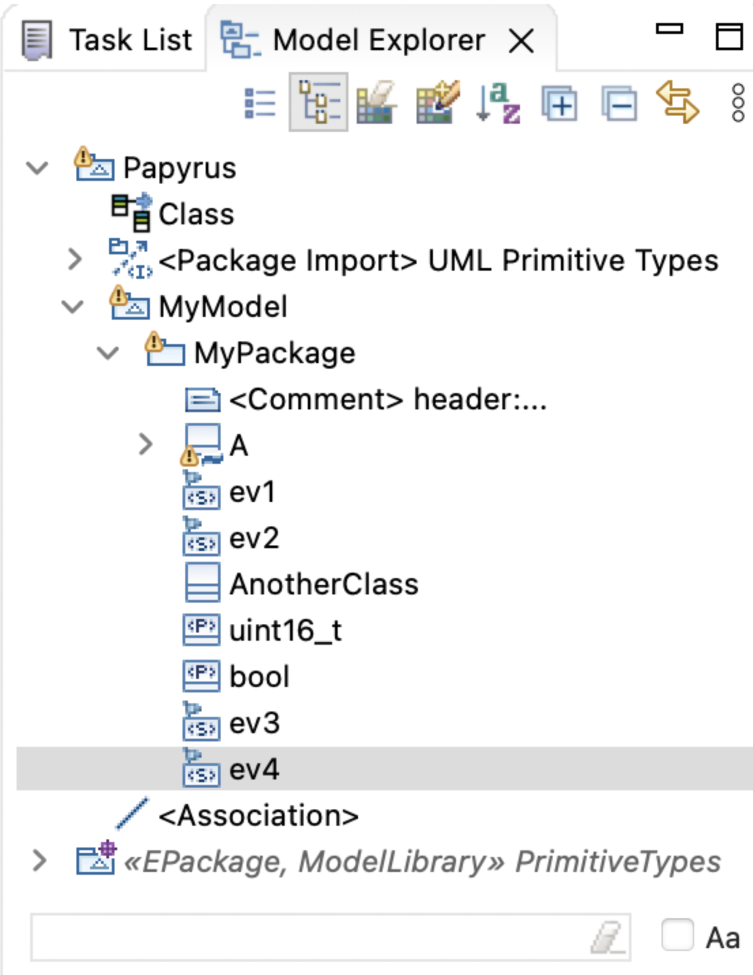


Figure I.2.: Papyrus' project browser. From the used structure the path can be directly derived.

I.3. Regions

A region is an orthogonal part of a state. It allows to express parallelism within a state. A state can have two or more regions. Region contains states and transitions. States in regions can contain regions again. See figure I.1 for an example state diagram with regions.

General rules when using regions:

- Transitions must not cross region borders. I.e. it is not allowed to add a transition from `Right` to `Off`.
- A region might contain a history state, choices, final states and normal states
- To express that a transition should fire if the machine is in both regions `Right` and `Off` use a conditional transition like follows: `#machineIsInRight(...) && machineIsInOff(...)`
- Regions can contain states which contain regions again

I.4. Transitions

Select the transition to set any transition properties. In the properties window the trigger, guard and effect statements can be defined. A transition must usually have a triggering event. To add a new signal event right click on the package that contains the class with the state-machine. Select menu `New Child` and add a `SignalEvent`. It is important to add the signal events at the correct hierarchical level. Otherwise the code generator does not find them. For the code generation only signal events are supported!

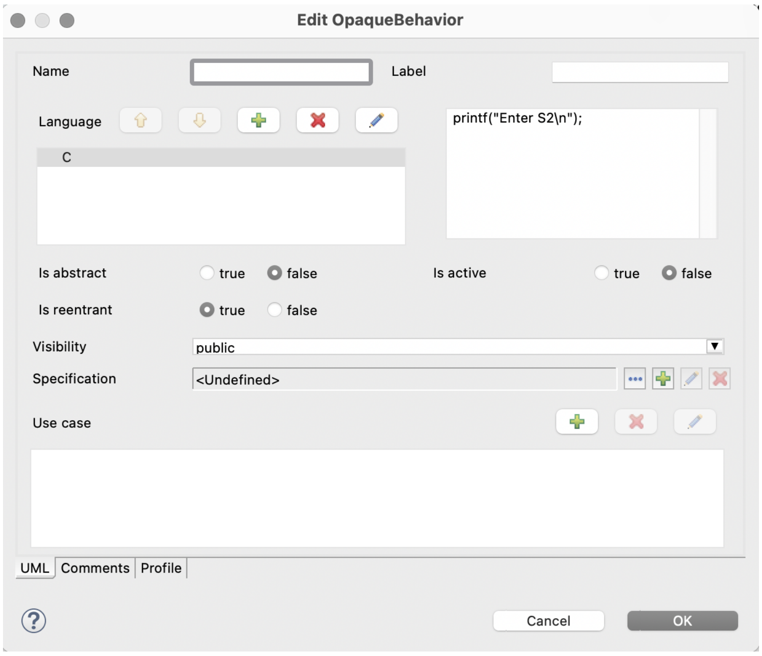


Figure I.3.: Activities must always be created as OpaqueBehaviour. Inside the dialog click on the plus symbol to define the implementation language - e.g. C. Then add the activity code in the shown text box.

The effect of a transition must be defined as `OpaqueBehaviour`.

To add a guard first add a `Constraint` and then add a `OpaqueExpression` to the Specification.

I.5. History State and other Pseudo States

Just add final states or history states from the tool bar into your diagram.

I.6. Attaching action and include comments

By default the code generator includes the minimally needed header files into the generated code. But often it is necessary to add further includes or define local variables etc. In this case you can specify code that is just copied at the beginning of the state machine implementation file.

You can also add code that is added to the beginning of the state machine handler function. The code can be used to perform some actions each time the state machine is executed.

Add this code to your model by using comments starting with the keywords `action:` and `include:`. Link those comments to the class owning the state machine. An example is shown in figure I.4. You must use plain text in the comment. It is not possible to use html comments.

I.7. Adding Attributes and Operations to Classes

It is possible to add attributes and operations to the class containing the state machine. To add a property drag and drop a `Property` symbol from the tool bar to the class. Then specify the name, data type and default value in the properties dialog. The initial value must be specified as `OpaqueExpression`. The data type can be a built-in data type, an own defined `primitive data types` as shown in I.4 or a class. In case of a class, specify the `Aggregation` type.

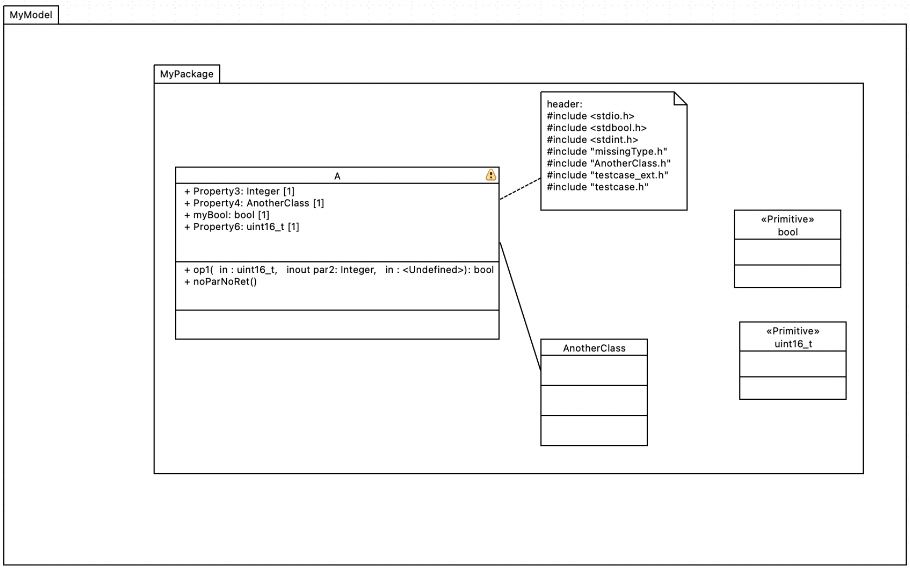


Figure I.4.: Connect comments with the class that contains the state machine to define generation details. Classes can contain properties of various types.

To add operations drag an `operation` from the toolbar to the class. Add details using the operation's property dialog. If no return type is specified `void` is assumed. Using the `Owned Parameter` feature you can add parameters to the operation. For each parameter specify the name, type and default value. Parameters marked as `out` or `inoout` are generated as pointers.

There is a default mapping of UML data types to C/C++ types. If you want to use own data types (e.g. uint16_t) define own primitive data types. The default mapping of the UML data types can be changed in the configuration file.

I.8. Supported / Unsupported

The code generator supports a subset of the design elements provided by Papyrus. The supported elements are as follows:

- Hierarchical states
- Regions and "regions in regions"
- (Signal-)Events with event name, guard and action
- Initial and final pseudo-states
- History states (deep, flat), Extended history handling
- Choices
- Junctions
- Attributes / Operations

The unsupported elements are:

- Constraints
- Sync-states
- Entry and exit points (not to compare with entry/exit actions within states)
- Terminate and Fork/Join
- Submachines

J. Error, Warning and Info Messages

Message number	Explanation	Type
1000	More than one default state on root level	Error
1001	No default state on root level	Error
1002	More than one default state in composite state <state name>	Error
1003	An initial vertex can have at most one outgoing transition and no incoming transitions	Error
1004	Hint: State "state name" has only one substate. This does not make much sense. Reconsider your design!	Warning
1005	Found two incoming transitions going into a choice state. Only one incoming transition is supported!	Error
1006	There is a choice with just one outgoing transition. Check your design!	Error
1007	There is a choice with no outgoing transition. This is not allowed.	Error
1008	There is a transition leaving a choice without a guard. This is not allowed.	Error
1009	A choice must have exactly one outgoing transition with an 'else' guard. Check your design!	Error
1010	Found a transition starting from unknown state: "event name"	Error
1011	Outgoing transition from a choice requires a guard definition!	Error
1012	Transition ending in a choice misses the event definition!	Error
1013	At least one transition has no event definition.	Error
1014	Event name contains one or more spaces	Error
1015	Transition must not cross state borders!	Error
1016	State name is empty ⇒ check state names	Error
1017	State name contains one or more spaces	Error
1018	Inner transitions are presently not possible if state has children	Error
1019	No default state on root level	Error
1020	More than one default state on root level	Error
1021	Child state has children. This is not supported.	Error
1022	A final state cannot have any outgoing transitions	Error
1023	State name already used. State names must be unique.	Error
1024	State is not reachable - check your design.	Warning
1025	Transition must not cross state borders!	Error
1026	Transition must not cross state borders!	Error
1027	Transitions triggered by same event leave a child and its parent. This is not a problem because transitions have higher priority than another one if its source state is a substate of the source of the other one. Make sure that this is what you want and the definition is unambiguous!	Info
1028	Transitions triggered from event leave state but some have no guard defined - check your design!	Error
1029	Several transitions triggered from event leave state but have no guard defined ⇒ check your design!	Error
1030	Choice state violates naming conventions as defined in codegen.cfg ⇒ no name defined	Info
1031	Choice state name violates naming conventions as defined in codegen.cfg	Info
1032	State is not reachable ⇒ check your design.	Info

Message number	Explanation	Type
1033	Simple state violates naming conventions as defined in codegen.cfg	Info
1034	Composite state violates naming conventions as defined in codegen.cfg	Info
1035	Event violates naming conventions as defined in codegen.cfg	Info
1036	Actions and guards of transitions entering a choice are ignored. Place actions and guards on the outgoing transitions.	Warning
1037	There is a 'header' comment which is not linked to the class containing the state machine. It is recommended to link the comment to the correct class.	Info
1038	The 'header' comment is linked to another class than the one that statemachine is in. This note will be ignored.	Info
1039	You have set the return type of the state machine function and also want to return if events were processed. This might be ok but usually it is not. Check your codegen.cfg settings.	Warning
1040	A choice must have at least one incoming transition. Error!	Error
1042	A junction must not have more than one outgoing transition.	Error
1043	The outgoing transition of a junction must not have an event or guard defined.	Error
1044	A junction should have more than one incoming transitions. Otherwise the junction does not really makes sense.	Info
1045	The transition leaving a junction pseudostate must not end in another pseudostate (e.g. a choice).	Error
1046	The transitions entering a junction must not start at another pseudostate (e.g. a choice).	Error
1055	Found a transition with multiple triggers. Create multiple transitions from it.	Info
1058	A history state must have at least one incoming transition if extended history state handling is enabled.	Error
1054	State has no outgoing transitions. This indicates a dead end in the state model and might be a design flaw.	Warning
2008	A transition starts or ends in an unknown state. Possible cause: Maybe you started a transition in a history state?	Error
2009	Both EventFirstValue and ValidationCall is set. The validation code requires that events start from zero.	Error
2010	A transition from an initial pseudo-state to the initial state crosses state borders.	Error
2011	In MD a transtion the action code was defined by name and code attribute. The latter overwrites the first.	Warning
2012	Parsing incomplete of a method or a method parameter of a class. Check that all operations, operation parameters and attributes have a name, type etc.	Warning

Table J.1.: Message overview. Info and warning messages give hints how to improve the design. Errors must be fixed before the generator generates code.

K. Version History

Version 1.02 now supports the specification of entry and exit actions for outer states. As the modeling tool does not support this directly at the moment a linked note with a special keyword at the beginning is used instead. See section 3.1.3 for more info.

Version 1.2: This version supports the code generation from XMI files. Presently only XMI files generated with Enterprise Architect version 7.1 or Magic Draw version 15.5 are tested. Also the command line options have changed. See section 2 for details.

Version 1.3: Test support is now included. A transition coverage algorithm can print out test routes which ensure that each transition was taken at least once. This greatly simplified test specification.

Version 1.4: In this version several features were added that allow to create more complex state machines. Also the support for testing state machines was increased significantly.

- Allowed state hierarchy was increased to three. Transitions between states of the third level must start and end at the third level within the same parent state. This looks like a limitation but in practice it is usually not.
- An interactive simulation was added which is activated with the commandline switch '-s'. You can type in events and get back the executed code as well as the state the machine is in. See section 3.16 for more details. No coding on your side is required!
- Choice states are now supported for XMI generating tools. See section 3.1.7 for details.
- On larger state charts the size of the generated source code size can be reduced by factoring out the entry and exit code of composite states into separate functions. New config file options were added for this purpose (see section 2.2).
- The event definition in *ext.h was changed. Only the event type you specified in 'mydefs.h' is used now. In several cases variables and functions are now prefixed with the machine name to avoid naming conflicts. This requires small changes in your code using the machine if you regenerate an existing state machine with the new codegen.
- Macros were added to support you during debugging. E.g. there are now functions generated that returns the event name and state name.
- Macros were added that allows to reset history within a state.
- Macros were added that return 0/1 to indicate if the machine is in a certain state or not. This can be used if transitions in one state machine shall be triggered depending on the state a second machine is in (see table 3.2

Version 1.4.1: This version supports the code generation from XMI files exported from UModel 2009. See appendix D for more information.

Version 1.5: Generation of C++ code is now supported. See section 3.4.

Version 1.5.1: Option added to either access the instance data by value or by reference. Access by value can be useful if your compiler does not produce optimal code when pointers are used (see section A.6). Furthermore the entry/exit code sequences for composite states were optimized.

Version 1.5.2:

- When running the statemachine multiple times in different threads (tasks) in the context of a real-time operating system no global variables must be used to store thread local data. Therefore it is now possible to specify code that is placed directly at the start of the state machine function body. This allows to create local data in a thread safe manner.
- With EA 7.5 it seems to be not possible anymore to specify transitions without a trigger name (which is in general ok). But for transitions starting from a choice state only guards should to be specified. To ensure a clear design only type in one or more spaces as trigger name. The generator detects this and ignores the trigger name.
- Many robustness tests for state machines were added. Due to automated rule checking the effort required for manual code reviews can be reduced. See section 3.15 for more details.

Version 1.6:

- More robustness tests were added

- Section 3.2.1 explains the execution model of the generated code.
- A third level of states and the deep history pseudo-state is now also supported when using the Cadifra UML editor.
- Possibility to add code that is executed if the statevars are invalid. See section 3.1.6 for more info.
- An experimental graphical simulator is now available. See section 3.17 for more details.

Version 1.6.1:

- Optional tracing of the event flow added. See section 3.20 for more information.
- The graphical simulator can be remote controlled by sending events via to a UDP port.
- Statemachine optionally returns a flag if an event was processed or not
- Creation of a state table in xls format

Fixes and Improvements:

- C++ code generation optionally generates virtual 'create-functions' in the factory class. This allows to provide own factory methods e.g. to initialize state objects.
- Improved error handling and error messages
- Final states must not anymore be placed on top level only.

Version 1.7:

- Support for Objective-C added (experimental)
- Support for nesC/TinyOS added (experimental)
- Separate return codes are now used for normal events and conditional events if the 'Return-EventProcessed' flag is set.
- Bug fix in choice state code. In choices starting from a root state the wrong action code was taken
- In the INSTANCEDATA_INIT macro type casts are used now to avoid problems with static code checkers.
- New configuration flag 'EventsAreBitCoded'. This flag can be used to instruct the code generator to generate bit-coded events.
- Guard evaluation is now separate from event selection code to avoid problems with MISRA rule checkers.
- Code to adjust the state variables is now inlined by default. If needed you can still provide your own function.

Version 1.8:

- Ada code generation added.
- C# code generation added.
- New command line option '-U' to provide config file name
- New parameters for better control of the state handler parameters for the C backend
- Action code from the init pseudostate is used by the codegen for the C/C++/C# backend (see section 3.1.4).
- Support for the modeling tool ArgoUML added
- Possibility to generate a description of a dot state machine graph in the C/C++ file for the doxygen tool
- If no 'header comment' is defined in the UML diagram the minimally required header files and instance/msg variable definitions are automatically generated (for C/C++ only). This makes the start for beginners easier.

Version 1.8.1:

- Improved parsing of transitions defined in Cadifra
- Zoom function added in the simulator
- Updated C++ section

Version 2.0:

- A tree based state diagram editor is available now. This editor follows a different approach compared to the most available UML tools. Instead of drawing diagrams a tree based approach is used. This makes the creation of state machines very efficient (see section 3.17).

- Support for choice states in the Cadifra UML editor (see appendix B).
- Changes in the C++ backend. Some internal vars are now marked as protected to make them accessible in derived classes. The new key 'CreateOneCppStateHeaderFileOnly' allows to put all state class code into one cpp/h file. And other improvements ...
- Several improvements in different parts of the code

Version 2.0.2:

- Improvements in the visualization of the editor. E.g. use of choice symbol . . .
- `HsmFunctionWithEventParameter=yes` and `HsmFunctionWithInstanceParameters=yes` is now possible at the same time (see section 3.3)
- Wrong missing license message in editor mode fixed.
- Wrong handling of flag `PrefixMsgWithMachineName` if set to `YES` in combination with `HsmFunctionWithEventPara`
- New keyword to define the type of the generated destructor of the C++ state machine class. Useful for more dynamic systems.
- Events leaving a state are now generated in alphabetical order. See section A.1 "Defining the state processing order".

Version 2.1.4

- Stabilization of the integrated state-chart editor
- Display of entry/do/exit code in the states
- New configuration keywords added (DisplayEntryExitDoCode, NumberOf EntryExitDoCodeChars)
- Introduction of a new C-backend (-l cx) which supports up to four levels of state nesting without any restrictions. This backend is now the default one.

Version 2.21

- Stabilization of the integrated state-chart editor
- New Java backend (see section 3.7)
- New C++ backend (see section 3.4)
- New configuration key to allow to save the model also if the check was not successful
- New command line switch `'-gencfg'` to output the supported configuration keys for the selected language. This allows to quickly generate a config file with all the supported keys and their default values. Example usage:
  `java -jar codegen.jar -gencfg -l cx > codegen.cfg`

Version 2.24

- Fixing a problem with drag and drop introduced in the last version
- Transitions leaving a choice are now sorted based on their guard definition
- Fixing minor issues in the cppx backend
- Update of the 'installation' and 'getting started sections'

Version 2.25

- Further improvements in the integrated state chart editor. E.g. visualization of modified model properties, changes in properties are not silently discarded anymore if the `Apply` button is not pressed ...
- By default -lcx is selected as language backend if nothing else is selected on the command line.

Version 2.26

- Support for the UML modeling tool astah* from ChangeVision added. See section E for details.
- Fix a problem with the -doxygen option (only occurred in hierarchical diagrams).
- Fix inconsistencies in the manual around the Java VM option -Djava.ext.dirs

Version 2.37 *New features:*

- Also for flat state machines the handler function can now return if an *event/conditional trigger* was processed or not (see ReturnEventProcessed). If no return value is required also the internally used flag is not generated anymore. This is useful for processers with very little RAM.

– It is now possible to add post action code which is executed after the state machine code (in opposite to the action code which is executed before the machine). See section 3.1.6 for details.

Bug fixes:

– A problem with the latest release of ArgoUML was fixed ($\geq$ 0.32) which has changed the XMI export format slightly.

– Conditional triggers do now work for astah* input files.

– In case the `-Djava.ext.dirs=...` java option was used sometimes the license file location could not be determined.

Version 2.38 *New features:*

– Zooming in the built-in editor uses the dot scale feature which results in sharper images.

– A validation function is now supported for the C-backend. This helps to detect serious errors happening outside the state machine code but effecting the correct execution of the the state machine. See section 3.3.7 for more details.

Bug fixes:

– Fix a problem with empty entry/exit/do actions in Enterprise Architect

– Print report if two final states have the same name

– Fix problem with tab handling

– Fix problem with prefix of main ChangeToState declaration

– ResetHistory and Change to state function declaration are from now on only included in the header if needed.

Version 2.40 *New features:*

– By default the C-backend now uses the types from stdint.h for simple data types. In the case other types must be used they can be changed in the configuration file.

– The *isIn* and *resetHistory* code are now provided as functions (not only macros). You can decide now if the instanceVar in these functions is accessed via pointer or variable. Use the UseInstancePointer parameter for configuration. This option addresses very memory constraint systems where pointer accesses shall be avoided where possible.

– The state machine image in the built in editor can now be copied to the clipboard (right click)

– The rank of the state machine layout in the built-in editor can be changed (right click). Depending on the state machine the top-down or left-right layout direction is better.

Version 2.41 *New features:*

– Support for Visual Paradigm added

– New feature to define events or states with a given hamming distance. This feature is only available for the CX backend.

Bug fixes:

– UModel: Fix a line-end problem with multiline actions

Version 2.5 *New features:*

– In the simulation and trace mode the achieved transition coverage is displayed as progressbar (0% . . . 100%). The tooltip shows the open transtions left to be taken to achieve 100% transition coverage.

– It is now possible to generate an Excel file with testcases to reach 100% transition coverage. The Excel file contains a sheet per route. Each line in a route represents a test step.

– The new keyword 'Constraints' allows to specify the expected output of a state. These constraints are listed in the Excel workbook as well and help testers to check if the test was sucessful. See section 3.19.4 for more info.

– C++ backend now generates const methods where possible (e.g. isIn()).

Bug fixes:

– Astah* backend has ignored inner events in previous versions.

Version 2.6 *New features and improvements:*

– A second test path generation algorithm added (-c1). This breadth-first tree search algorithm returns more but shorter tests routes.

– A reset command can be sent to the online visualization to indicate a restarted target.

- Entry/Exit/Do code can now be defined in a better way using Enterprise Architect. Multiline code pieces are supported now. The old way does still work.

Bug fixes:

- Problem when moving states in the built-in editor fixed.
- Robustness of the online visualization improved when receiving unknown events.
- XMI export of EA 9.2 has changed slightly but enough to stop the codegen from working. This is fixed again.

Version 2.7 *New features and improvements:*

- Support of a new key-word to make the generated Java code *Java 1.4 compatible*.
- Support for sub-machine states in Enterprise architect. See section G.11 for more details.

Bug fixes: -

Version 2.8 *New features and improvements:*

- Junctions are available now (details see 3.1.9)
- Support for nested namespaces added for the cppx backend
- In some cases empty 'if/else if' blocks were generated which can be problematic in some very resource constraint systems. Empty blocks are now avoided where possible.

Bug fixes:

- When using the cppx backend the factory and state class were generated in the wrong folder if the parameter of -o contained a path and filename.
- Improving code block indentation
- A missing empty 'else/* left empty */' was added in a specific case to satisfy static code checkers.

Version 2.8.1 *New features and improvements:*

- Support for multiple incoming transitions into a choice. See section 3.1.7 for more details.

Bug fixes:

- In previous versions the code for transitions leaving a state with flat history was not correctly generated under some circumstances.

Version 2.8.2 *New features and improvements:*

- Support for entry and exit points in diagrams modeled with EA. This allows to make better use of sub-machines. See section G.12
- New helper function available to return the innermost active state. This function is useful for debugging or tracing tasks and available for CX, CPPX and Java backends.
- Switching over to the new jdom2.jar version. Replace the existing jdom.jar file with the latest version coming with the codegen.

Version 2.8.3 *Bug fixes:*

- Fixing a problem when using the internal editor/simulator with a model containing a junction.

Version 2.9 *New features and improvements:*

- Support for entry and exit points and sub-machines in diagrams modeled with MD. See section C.9 and section C.9.

New features and improvements:

- A new editor component is used in the built–in state machine editor / simulator. The new editor supports folding, line numbers and other nice features. Thanks to Robert Futrell for his very nice component (see section copyright for more information).
- The $isIn()$ methods returns a *bool* now instead an *int* in C++.
- $getInnermostActiveState()$ is now a public method in Java
- Visual representation of $innerEvents$ in the built–in editor improved. They are now clearly represented in the graphics as well as in the tree view.
- Visualization of states in the built–in editor improved.

Bug fixes:

- Issue when simulating state machines with a history state fixed.

Version 3.0 *New features and improvements:*

- The Modelio UML modeling tool is supported now. Modelio is an open source modeling environment based on Eclipse. See `www.modelio.org` for more information about the tool.
- In this version only the following language back-ends are available anymore: CX, CPPX, Java. If you use one of the other language back-ends you have to stick with the previous codegen version.
- Support for regions in the CX language back-end

Bug fixes: —

Version 3.1

- The C++ language back-end now fully support the code generation from state machines with regions.
- Two new parameters for the C back-end were added. They help to avoid naming conflicts when calling multiple state machines from within one file or when using the same state names within different parent states. See section 2.2 parameters *PrefixStateNamesWithMachineName* and *PrefixStateNamesWithParentName*.

Version 3.11 *New features and improvements:*

- The simulator can now be used to simulate also state machines with regions.
- Improved visualization of state machines using Graphviz in the simulator

Bug fixes:

- Fix of some problems in the simulator

Bug fixes: —

Version 3.2 *New features and improvements:*

- Initial support for Metamill added

Version 3.3 *New features and improvements:*

- Support for C# added again
- Documenting `-L`*events* an `-L`*states* command line options

Version 3.31 *New features and improvements:*

- Support for "regions in regions" supported by the code generator in the C/C++ backend

Version 3.5

- Fixes a problem with transitions leaving a state which contains regions in regions.
- Handbook improvements especially in the Enterprise Architecture section.

Version 3.51 *New features and improvements:*

- Fully support of test case generation for state machines with regions
- Improved visual simulation and other smaller fixes related to regions
- Fully support of transitions from an initial pseudo-state to a choice state.

Version 3.6, 3.6.1 *Improvements:*

- Model checker improved to detect further problems on model level which led to null pointer exceptions before.

New features:

- Initial support for activity diagrams added. Only EA and C so far.
- Improvements in the Java backend (initial state to choice state added)

Version 3.6.2 *Improvements:*

- In the case the initial transition ended in a choice the generated initialization code was not always correct.

New features:

- UModel activity diagram support added. See section 4.5 for details.

Version 3.6.3 *Improvements:*

- Model checker of activity diagrams improved
- Fix a problem that leads to a crash of the integrated state-chart editor

New features:

- Astah activity diagram support added. See section 4.6 for details.

Version 3.6.4 *Improvements:*

- Initial code was eventually wrong in the c++ backend in the case the initial transition ended in a choice pseudostate.
- Visual editor and simulator did not start on some systems because of missing icons.

New features:

- -

Version 3.6.5 *Improvements:*

- The code generator performs some optimizations before generating code from an activity diagram (see section 4.3.2).

New features:

- -

Version 3.6.6 *Improvements:*

- In some cases the entry code was not executed. This problem was fixed.

New features:

- Region Support for Astah*
- Possibility to define events in comments attached to transitions (only Astah*)

Version 3.6.7 *Improvements:*

- Fixing a problem in the drag & drop function of the built-in editor.
- Fixing a problem that two or more exactly same triggers that have triggered a transition on different hierarchy levels of a state diagram could have lead to endless loops in the simulator engine (codegen threw an out of memory exception).

New features:

- Initial support for the new language Swift (see section 3.8).
- Undo/Redo feature added into the built-in editor. This is a great usability improvement!
- New parameter for the C++ backend `BackrefToMachineInStateClasses`. Now it is possible to automatically generate and set a reference to the state machine in the state classes during initialization of the machine. This makes it possible to easily access code of the state machine class (or its base class) from within state classes.

Version 3.6.8 *Improvements:*

- Compatibility problem with export format of Visual Paradigm fixed.

New features:

- New configuration file parameter to suppress the generation date in generated files. See section 2 parameter `IncludeDateFileHeaders`.

Version 3.6.9 *Improvements:*

- Fix a problem with XMI files exported from EA12 using sub-machines.

New features:

- –

Version 3.6.10 *Improvements:*

- –

New features:

- Added the new `OptimizeExitCode` parameter which if set to yes generates exit code only once even if there are multiple transitions leaving a state.

Version 3.6.11 *Improvements:*

- Problem with Modelio 3.3 fixed.

New features:

- Activity diagrams can now be generated using Modelio.

Version 3.6.12 *Improvements:*

- Update to latest version of `rsyntaxtextarea.jar` which is the text filed library used for the built-in editor. It now supports folding and contains several bug-fixes. Replace the old jar with the new one installed on your computer.

New features:

– Increased number of transitions / states in the demo mode to allow for better testing before taking a decisions.

Version 3.6.13 *Improvements:*

– -

New features:

– New command line parameter `-L` to define the path to the license file.

Version 3.6.14 *Improvements:*

– Support for astah SysML added

New features:

– – file.

Version 3.7 *Improvements:*

– -

New features:

– Python code generation added

Version 3.7.1 *Improvements:*

– C# now supports init to choice transitions. A transition from an initial pseudostate can now end in a choice pseudostate. The real init state is then evaluated at runtime depending on the guard conditions. See section 3.11 for more information.

– New *Fontname* keyword to define font in the built in editor and simulator.

New features:

– -

Version 3.7.2 *Improvements:*

– C++ backend bugfix (initialize code)

New features:

– -

Version 3.7.2.2 *Improvements:*

– Maintenance release

New features:

– -

Version 3.7.3 *Improvements:*

– Changed layout of integrated editor to better use space on wide screen displays

– Bug fix related to redo/undo of integrated editor

New features:

– -

Version 3.7.4 *Improvements:*

– Improved Python backend

– Improved visual diagram representation in the built-in editor

– Improved C++ backend

New features:

– For the Cadifra tool connectors can now be used to make complex state diagrams more readable.

Version 4.0 *Improvements:*

– Several new configuration parameters for the CPPX backend added. With the new parameters there is much more flexibility to adjust the generated code towards own needs. See table 2.2 for details.

Version 4.1 *New features:*

– Lua added as new back-end 3.10

Version 4.2 *Improvements:*

– Trace and validate code was reworked to improve type safety in C/C++

– Usage of const where possible in the C-backend

- Region handlers defined static. Not included in state machine header anymore.

New features:

- New config file parameter *IncludeTransitionsIntoStatesTheyStartFrom* to export the state machine in a hierarchical manner from the internal state machine editor.
- Introduction of *partial* keyword for the C# backend

Version 4.3 *Improvements:*

- Update of the c-section in the manual to better explain the various parameters relevant for C-code generation.

New features:

- New config file parameter *HsmFunctionUserDefinedEventParameter*. This parameter allows to define a user defined type as parameter for the state machine handler. This is useful if you want to hand over data in addition to the event. E.g. data received from a communication interface or the like. See section 3.3 for details and examples.

Version 4.3.1 *Improvements:*

- Bug fix in the activity generator backend for C. A problem occurred if the output contained a path and not just the output file name.

New features:

- -

Version 5.0.2 *Improvements:*

- Bug fix in the Lua backend

New features:

- Completely rewritten built in state diagram editor. The new version is much more user friendly and allows to create state machines and generate code from them within minutes.
- UnknownEventHandler: Allows to define code that is executed if an event could not processed in a state. Can be used for debug purposes for example

Version 5.1

- Initial support for regions in the built-in editor
- Several improvements and bug fixes of the built-in editor

Version 5.2 *New features:*

- Region support in the built in visual editor
- Support for displaying regions in separate graph windows. This provides a better overview in bigger state machines
- Tree / graphics relation. Clicking on a state in the state tree shows the related diagram. Clicing on a state in a diagram shows the state in the state tree.
- Several bug fixes and improvements in the built-in editor

Improvements:

- Bug fix in the validate code of the C-backend
- Bug fix in the activity generator

Version 5.3 *New features:*

- State/region notes can be displayed in graphical editor. A configuration parameter can be used to enable / disable the visualization of state/region notes
- Export / import of parts of the state diagram into another instance of the graphical editor or the clipboard. This allows to save parts of the state machine model into a text file and to create a library of pre-defined state diagram solutions (library of solutions)
- Create *.bak file each time the model is saved
- Code generation to ssc xml in obfuscated format. I.e. removing real state name, event names, ... This allows to share models for bug tracking or other purposes without giving away confidential information. A configuration parameter is available for that purpose.

Improvements:

- Improvement of the simulation engine to avoid deadlocks in special situations
- Prevent usage of already used state names without loosing the changed state details
- Improvements when working with separate tabs per region in the built-in graphical editor

- Bugfix in the C-backend related to validation code (wrong order of state names)
- Improvement in the code generation backend related to new Java features
- Changes in the scc xml parser related to Final States and Choice States. This may create inconsitencies with existing ssc xml files. Request details on how to fix your xml model file in case you run into this issue.
- General bugfixes and improvements

Version 5.4.1 *New features:*

- Windows and Mac installer that allows to open a state diagram file created with the built-in editor with a double-click or a right-click (open with). This makes life easier for users who are not at ease with the command line - esp. when using the built-in state machine editor.
- ValidationCall parameter now also supported by the C++ backend. The validation code itself has to be provided by the user.
- codegen.cfg file can be stored in the users home folder to make general settings valid for all project (e.g. dot path)

Improvements:

- General bugfixes and improvements

Version 5.4.2 *New features:*

- Attributes added to the class will be considered in the code generation and added to the instance data (only for Enterprise Architect/UModel and C backend). Details see section G.10 and D.8
- For all other tools the possibility exists to add attributes in the configuration file. They are added to the instance data (only C backend). See table 2.2 row INSTANCE_ATTRIBUTE_x etc.
- Operations added to the class will be considered in the code generation and mapped to functions in the generated state machine C - and header file. It is also possible to enter the function body code (only for Enterprise Architect and C backend). Details see section G.10. If no body is given the function is only generated in the state machine header file and can be implemented separately.

Version 5.5.4 *New features:*

- Added a template parameter option for the state handler method in C++. Various possiblities exist now for the state handler signature. See section 3.4.4.
- Added the option to pass an object to the state handler method in C# that can contain any data but also the event to be handled.
- Added validation code to C# if enabled
- Improved MagicDraw XMI parser.

Version 5.5.5.2 *New features:*

- Improved region handling for own instance data types. Recommended update for all C backend users.
- Improved XMI parser for the various supported UML tools

Version 5.5.5.3 *New features:*

- Simulation of timers was added to the integrated state machine editor. See online manual for more information.

Version 5.5.6.3515 *New features:*

- Rust code can now be generated from UML state machines. See chapter 3.11.
- Improved code window in the built-in state diagram editor
- Many detail improvements of the integrated state diagram editor

Version 6.0.2

- Additional parameters `NotUseRedundantVoidArgument` and `EnableTrailingReturnType` to modernize the generated c++ code and pass clang-tidy with `modernize-*` test enabled.
- Python generator calls init of a specified base class.
- Many detail improvements in the integrated state diagram editor
- Support for Eclipse Papyrus[a]dded. See details in section I.

Verison 6.1

- Bugfixes and improvements in the built in state-diagram editor
- Use of a project specific configuration file in the graphical editor
- New C-backend related parameters to generate code following the opaque object pattern to hide the state machine implementation from other user code and thereof reduce dependencies.
- Improvements in the section about the c-backend and reorganisation of the examples folder in the download.

New features:

- Experimental parsing for the DrawIO Editor added
- Events marked *external* are now exported in a separate file to allow further external processing. This allows to generate one file containing all state machine events (only with events as defines possible).

Version 6.2 *New features:*

- Support of the GO language as new code generator backend
- Support for the GO language in the integrated state diagram editor

Version 6.2.1 *New features:*

- Support of the JavaScript language as new code generator backend
- Support for the JavaScript language in the integrated state diagram editor

Version 6.2.2 *New features:*

- Improvements in the JS backend. IsIn() and getInnermostActiveStates() works now with regions.
- Improvements in the C++ backend. IsIn() and getInnermostActiveStates() works now with regions.

Version 6.2.3 *New features:*

- Python backend now uses enums for states / events and function annotations.
- Bugfix in the C Sharp backend.

Version 6.3.4 *New features:*

- Added option to return a bool value from the `processEvent()` method in C++
- Other minor changes to improve the robustness and clarity of the console messages

Version 6.4 *New features:*

- Extended history state handling added. It is now possible to end a transition at a history state. Only if a transition ends in a history state will the previous history be considered when entering the state set. Otherwise the default entry chain is used. The new **`TransitionsCanEndInHistoryStates`** parameter has been added to enable this feature. Section 3.1.11 describes the new history state handling in more detail.

Version 6.5 *New features:*

- Bugfixes and improvements in the built in state-diagram editor
- General improvements

Version 6.5.1

- Added support for additional command-line parameters when using `'-l cppx'`. You can now optionally specify `'-std c++11'` or `'-std c++14'` to enable all necessary parameters for the corresponding C++ language features. This simplifies configuration for new users. For more details, see section "Generating C++ Code" 3.4.

Bibliography

[1] Martin Gomez, *www.embedded.com*, 12/2008

[2] Object Management Group, *Unified Modelling Language (UML) 1.3 specification*, 2000

[3] Object Management Group, *UML Superstructure Specification, v2.4.1*, 2014

[4] Graphviz is open source graph visualization software, http://www.graphviz.org/