sinelabore*RT* User Manual

(C) Peter Mueller

Revision: 1.02

Contents

1 Overview

A statechart (or state machine) diagram shows the dynamic behavior of an application. It is a graph of states and transitions that describe the response to events depending on the current state that it is in. 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 statecharts is popular because the behavior of devices in this domain can be often described very well with statecharts.

An important aspect of statecharts 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). Please note that the codegenerator is not certified in any way.

The generated code and the generation tool has to fulfill a number of requirements to be really useful for embedded software development:

- Generated code shall be human readable to allow debugging, validation and verification
- Generated code shall not enforce a specific system design (e.g. task based with queues for event delivery)
- Code shall be generated in a way that static code analyzers (e.g. lint) makes no trouble
- Generation process can be integrated into the build process

The sinelabore*RT* code-generator was built especially for embedded real-time developers and fulfills all of the above listed points. It focuses on just one task: code generation from statechart diagrams. A command line tool and a specification file is all what is needed.

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 analyzers.

sinelabore*RT* does not force you in any way 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.

The way sinelabore*RT* works is depicted in the next figure. From a statechart design file produced with the Cadifra UML editor the generation tool generates the complete statemachine implementation. For a specification file called `oven.cdd` the command line would look like `java -jar codegen.jar oven.cdd`.

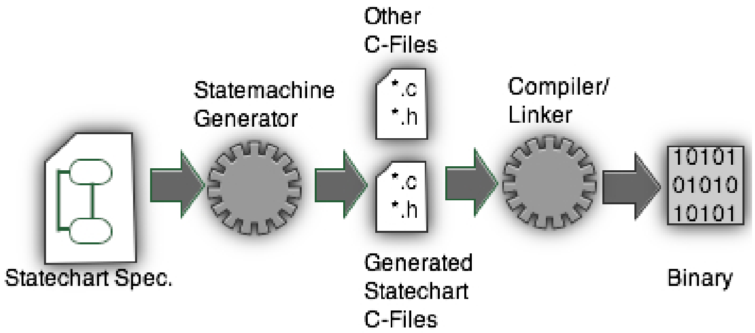


Figure 1: From design to code

1.1 What is new in this version?

Version (1.02) supports now 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 A.3 for more info.

How to go on from here?

If you are not familiar with the statechart notation or need a refresh consult appendix A. With the help of an application that allows you to interactively send events to a rather complex statechart you can learn how statecharts work. Furthermore the elements of a statechart are briefly explained. Section 2 describes how to install sinelabore*RT* on your computer. In the following section 3.1 a simple step by step example is presented. It starts with the design of a statechart and end with code generation. Then section 4 describes the different possibilities to influence the code generator. In appendix B different options are discussed on how to integrate the generated statechart into your code.

2 Installation

It is necessary to install both sinelabore*RT* and Cadifra UML Editor on your computer. The order doesn't matter. To start the installation of Cadifra UML double click the setup executable and follow the dialogs.

For sinelabore*RT* no installation script is provided. Simply copy the `codegen.jar` file and the license key file into a folder of your choice. It is recommended to place the files in a folder e.g. `bin` located in your project directory. This makes it simpler to access the generator from a Makefile and it can also be added to the project's version management if needed.

The code-generator of sinelabore*RT* is entirely written in Java. Therefore a Java runtime environment of version 1.5 or later is needed. If not already installed it can be downloaded from `java.sun.com`. Follow the installation steps as described there.

Finally the `jdom.jar` file must be copied into the `<JAVA_PATH>/lib/ext/` directory in the Java installation directory. On my system the file must be copied to: `C:\Program Files\Java\jre1.6.0_03\lib\ext`.

Due to the fact that the code-generator is written in Java it is possible to run it also on operating systems such as Linux.

3 Introduction

This section guides you through the whole development process from designing a statechart to integrating it into an application. This should help you to get familiar with the concepts. You will understand what the tools can do for you and where limitations exist.

3.1 A Microwave Oven

In this section we create the model of a simple microwave oven using a statechart diagram. A microwave oven was chosen because it is self-explanatory and not too complex to model. To keep this example as simple and clear as possible the hardware interaction routines are excluded. Consider the following image as example for a fictitious microwave oven.

The oven controller should be able do the following things:

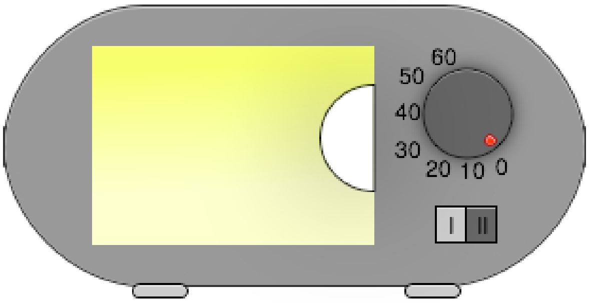


Figure 2: Fictitious microwave oven. With a wheel the cooking time can be adjusted between 0 – 60 seconds. The power can be set to high (II) or low (I).

1. Cooking time can be adjusted using a wheel between 0s and 60s.
2. Cooking starts if the cooking time is larger than zero. And the door is closed.
3. If the door is opened during cooking the microwave generator is switched off. Cooking time stops.
4. Cooking continuous if the cooking time is not over and the door is closed again
5. Cooking stops if the cooking time is over or the time is adjusted to zero.
6. Cooking time and power can be changed at any time.

3.2 Drawing the Initial Diagram

To start designing a statechart diagram, you first have to start the Cadifra UML editor and select the statechart mode either from the tool bar or from the menu entry *Diagram→State*. Now you can draw states and transitions. Right click to the drawing area to select the state chart element you want to use. Create step by step the complete start chart. The next figure shows an initial diagram fulfilling the above requirements. You can either draw it yourself or load it from the folder `example1` located in the installation directory.

Such an initial design is already useful. It can be used to discuss (e.g. with customers) if the requirements are fulfilled and it reacts to all events as expected. At this stage often unclear points in the the specification can be identified. E.g. in our design a user has to open the door once after the cooking time is over

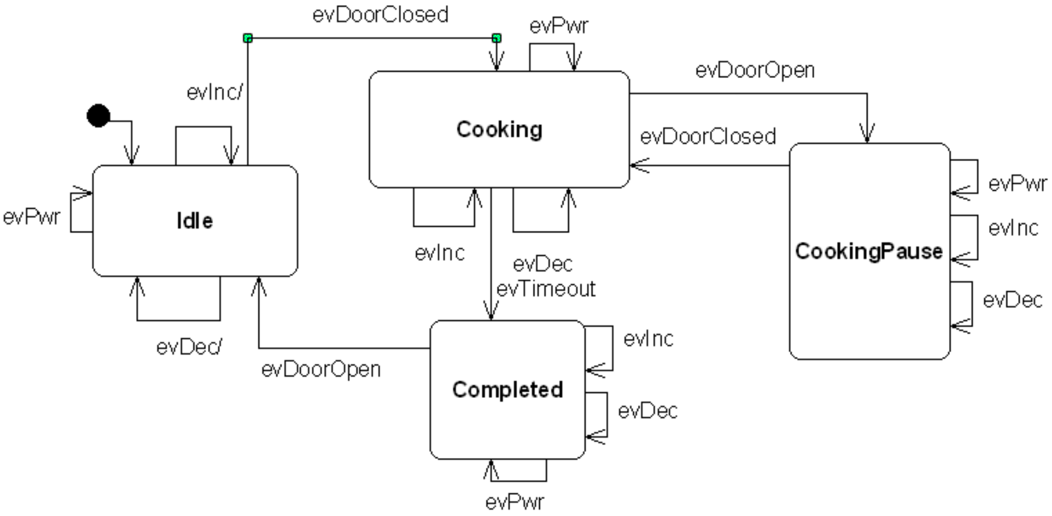


Figure 3: First statechart design of the microwave oven. Only states and events are modelled yet.

before cooking can be started again. This behavior is not explicitly specified in the requirements. It might be acceptable but it is also possible to go directly to state `idle` instead.

Our initial design is not optimal as you can easily see. Requirement six (power and cooking time can be adjusted at any time) leads to a lot of similar state transitions. To avoid this the design can be changed into a hierarchical one. The states already existing must be moved into the outer state and are now children of it. The power and time related events are now handled by the outer state. Please consider that the outer state needs to be a history state. This is necessary because we want to go back to the last inner state after event processing of events handled by the outer state.

So far the machine can receive events and change state as reaction. But some important details are still missing. For example a `close door` event in state `idle` causes a state change to `cooking` even if the cooking time was not set to a value $\neq$ zero. This is in opposite to requirement two. To avoid a state change a guard must be added. Furthermore action and entry/exit code is still missing allover the diagram. The next figure shows the complete statechart design. Functions with prefix `timer` are helper functions providing timer functionality. Functions with prefix `oven` are functions related to power control. See next section for further details.

In this example we touched the most important design elements of state charts.

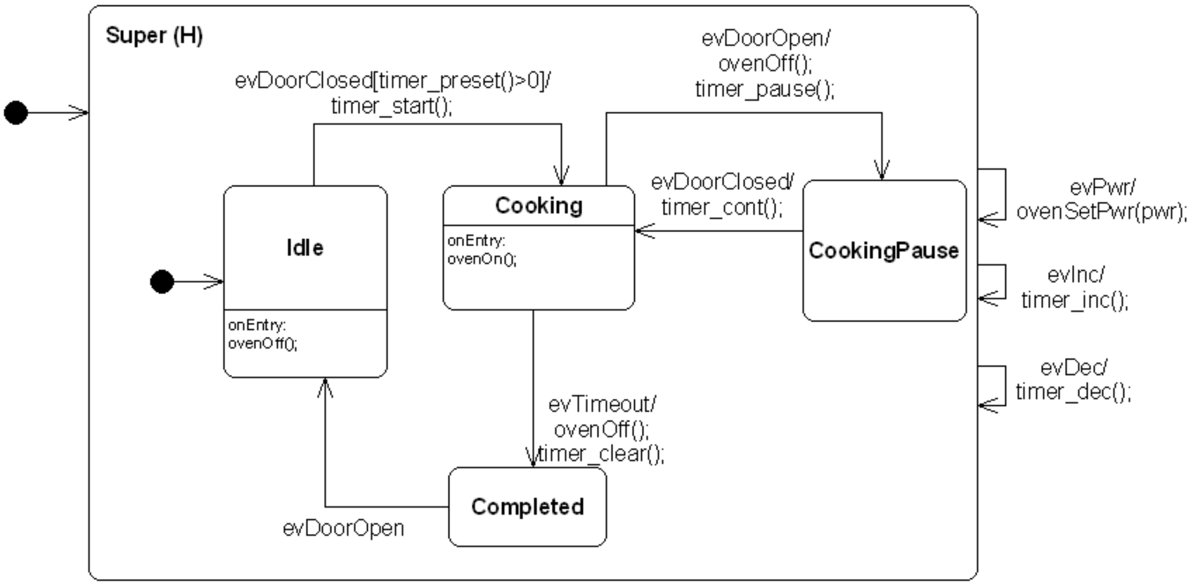


Figure 4: Complete statechart design of the microwave oven. All necessary actions, guards, entry and exit code was added.

In the next example we focus on code generation and execution of our design on a PC.

Please note that the design presented here is not the only possible solution for the given requirements. Also some functions of a real microwave oven – e.g. control of a lamp in the oven – are still missing.

3.3 Generating Code

In this section we will learn how to generate code from the statechart designed in the previous section 3.1. To test the generated code we develop a console based application. This application scans permanently the computer keyboard. Depending on the pressed key it then sends the corresponding event to the statemachine. Pressing for example `'+'` increases the cooking time by one second whereas `'-'` decreases the cooking time.

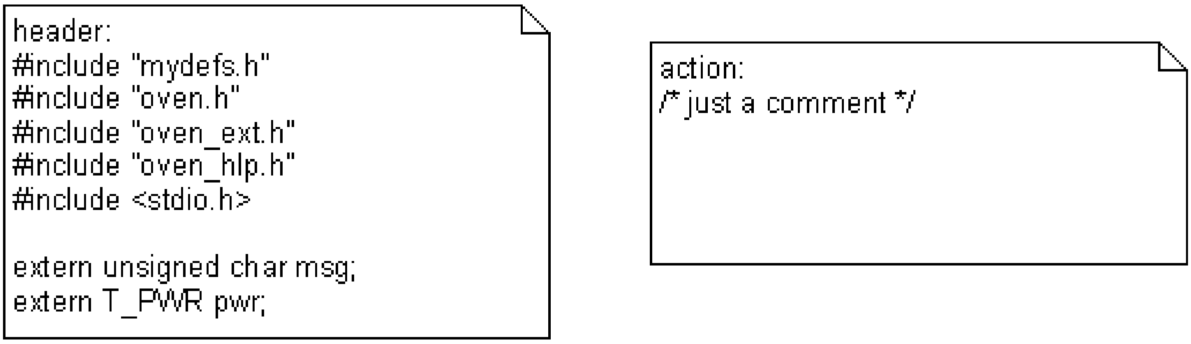
To be able to build the program on your computer a C compiler is needed. You are free to use whatever compiler you have installed. In this tutorial we assume that the Cygwin environment and GCC is installed on your computer. You can also follow this tutorial without a C compiler but then you can't build the code yourself. The final executables are provided for your convenience in the sample folder.

In principle it is possible to generate code from the statemachine specification that was designed so far. But in practice it is usually necessary to include some header files or to declare some variables or to execute some code before the statemachine code really begins.

Therefore you can add a note to your design that starts with the '`header:`' keyword. All the code that follows is then just copied to the begin of the C file implementing the statemachine. In the example it is used to include some header files and declare two external variables. The variable `msg` is the event that should be processed and the other variable (`pwr`) reflects the actual power selection.

For adding code that should be executed just before the statemachine add another note to your design that starts with the `'action:'` keyword. In our design it is only used for example purposes. But as mentioned it is very useful to add event handling code. See section B.1 for an example.

The figure below shows the mentioned notes from the design file.



```
header:
#include "mydefs.h"
#include "oven.h"
#include "oven_ext.h"
#include "oven_hlp.h"
#include <stdio.h>

extern unsigned char msg;
extern T_PWR pwr;
```

```
action:
/* just a comment */
```

Figure 5: Header and action code as used in the microwave oven example. With the `action` keyword code snippets can be introduced that are just inserted before the statemachine code. With the `header` keyword code at the beginning of the statemachine C file can be added.

Calling the codegenerator with the design from above `java -jar codegen.jar oven.cdd` produces the following three files:

- `oven.c` implements the statemachine as graphically specified in the `cdd` file.
- `oven_ext.c` defines the events that can be sent to the statemachine.
- `oven.h` defines the function prototypes, states etc. used in the statemachine. Also a macro is defined that can be used to initialize the state machine (see

5.2).

These three files realize the complete state machine. They are in human readable C code and can be understood and verified by every C/C++ programmer.

For a complete console application some further code is needed. The following files are already provided for you:

- `main.c` is the main entry. It initializes the statemachine and the keyboard, scans the keyboard and sends events to the statemachine.
- `oven_hlp.c` and `oven_hlp.h` defines some helper functions that are used in the state machine diagram such as the timer functions and the oven power control functions. Furthermore for the overall state machine and each hierachical state the state change functions are implemented here. In the simpliest case this is just a function seeting the new state. But for debugging purposes the user can add specific code here e.g. for logging purposes.
- `mydefs.h` defines types that are needed by the statemachine and can be adjusted to your platform needs (see 5.1).

Also a Makefile is available in the sample folder. Open a Cygwin shell window and change to the sample directory. Type in `make` there. You should see someline like this:

```
$ make
java -jar -ea "../../Cadifra CodeGen/testcases/codegen.jar"
     first_example_step3.cdd oven
Creating state-machine defined in first_example_step3.cdd.
        Output stored in oven.c / oven.h
Running in demo mode!
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
```

Now you can start playing with your design. Type in `./oven.exe` and send events with the keyboard.

Within this example you have seen how to generate code from a statemachine design file. The generated code was used in a simple interactive test program. Whenever you change the design simply type in `make` to rebuild the statemachine and the test application. Take a look in the main file. There is also code for automatic stimulation of the statemachine.

4 Commandline and Generator Flags

The code generator can be called to following way:

```
codegen input_file [machine_name]
```

The input file is the state chart file produced from the Cadifra UML Editor. The machine name - when specified - defines the name of the statemachen function and is used as prefix at many places in the code. If the machine name is not specified the input file name (without the cdd ending) is used instead.

The code generator requires a configuration file 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 specific needs.

The following table lists all the generator flags and explains their role during code generation.

5 Important Types and Helper Functions

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

5.1 User Defined Typedefs

Within the generated state machine code different flags and variables are used. You can define what types these variables should have. For example you want to define the state variable to be of type `unsigned int` when running on a 16Bit uC but of type `unsigned char` when running on a 8Bit uC. The following table lists all typedefs you have to specify. There are no default values defined.

5.2 Important (Type) Definitions

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

Key	**Value**
Copyright	Defines the text each generated file starts with. Use `'\n'` for multi line comments. Default is `/*\n * (c) Peter Mueller` ...
StateMachineFunctionPrefixHeader	Prefix of the state machine function in the C file. Default is void.
StateMachineFunctionPrefixCFile	Prefix of the state machine function in the header file. Default is void.
ChangeStateFunctionPrefixHeader	Prefix of the state change function in the C file. Default is void.
ChangeStateFunctionPrefixCFile	Prefix of the state change function in the header file. Default is void.
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.
EventFirstValue	Defines if event definitions start from zero or another value. Default is zero.
EventDeclarationType	Defines the C mechanism used for event definition. Options are 'define' or 'enum'. Default is ENUM
EventTypeInCaseOfDefine	In case the obove key is set to 'define' the event type can be specified here.
Realtab	Option 'yes' and 'no' select if real tabs or spaces are used for indentation.
Tabsize	In case of spaces are used for indentation the tabsize can be given here.

Table 1: Generator Flags

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.
`xxx_EVENT_T`	Type used for the events that can be sent to the state machine.

Table 2: Typedefs a user has to specify. The 'xxx' is replaced from the code generator with the name of the state machine.

(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 initialize the instance variable. Especially all state variables are set to their default states.

Table 3: Important (type) definitions the codegenerator creates. The 'xxx' is replaced from the code generator with the name of the state machine.

A Short Introduction into Statecharts

A.1 Interactive Statemachine

This section shows the possibilities that are presently supported from the code generator. This is done on basis of a rather complex statechart as shown on the next figure. A demo program with this statechart can be found in the folder `complex`. You can start the application `complex.exe` and type in events used in the statechart diagram (e.g. `'e1'`, `'e12'`) followed by a return. Then the program prints out the messages coded in the state diagram.

This program can be used to learn how a statemachine reacts to the input stimuli. Consider first what should happen if you type in a certain event and then check the messages on the console.

- Check in which order the entry and exit actions are executed
- Consider when e1 triggers a state change to S2 and when it triggers a self-transition to S11
- Make sure you understand the effect of the history marker in S2

A.2 Transitions

There are two types of transitions a) event based ones and b) conditional ones. An event based transition has the following syntax: `eventName[guardExpression]/action`.

From a transition like

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

the codegenerator generates the following code (taken from figure 4):

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

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

A conditional (or when) transition is not triggered from an event but a C expression that is evaluated to true. It has the syntax: `#condition/action`. From

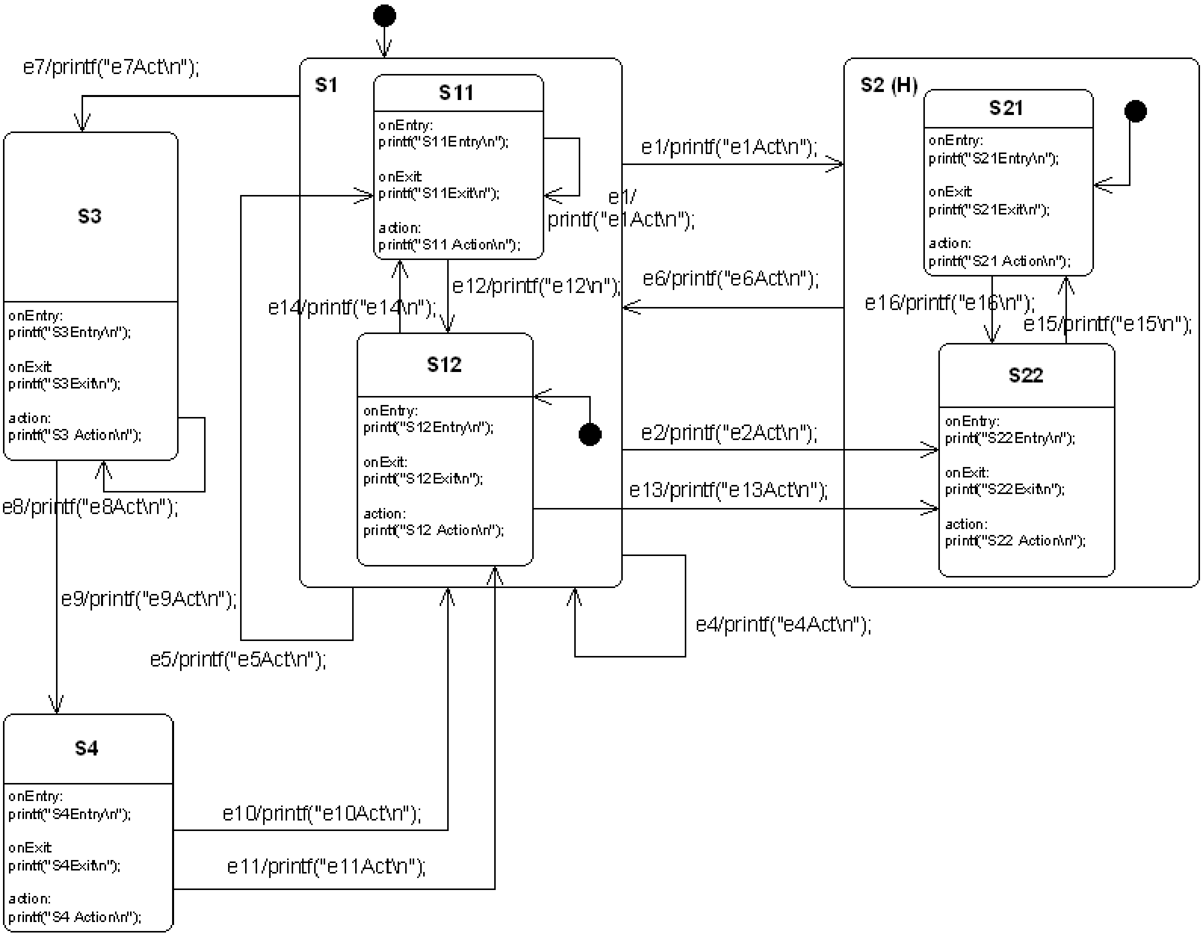


Figure 6: This is a rather complex statechart example with most of the features supported from the present version of the code generator or the Cadifra UML editor.

a transition like this `#i==1/printf("i==1\n");` the codegenerator generates the following code:

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

Action code defined in transitions must be non-blocking! Figure 7 shows examples for all supported transitions.

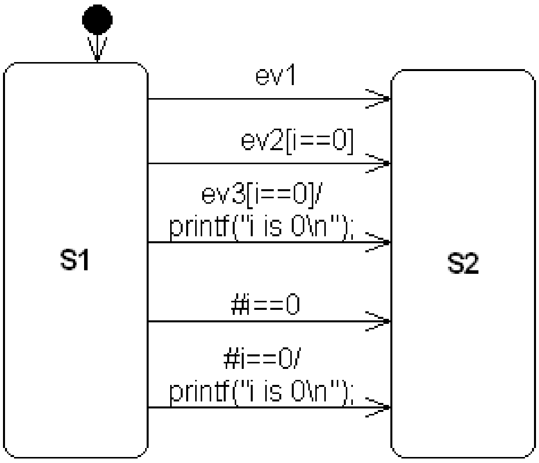


Figure 7: All possible transitions.

From top to bottom:

1. Simple event with no guard and no action
2. Event with guard. The guard expression enclosed in brackets([]) is denoting that this expression must be true for the transition to take place.
3. 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.
4. Conditional transition. The transition takes place if the variable i is zero.
5. Conditional transition with action. The transition takes place if the variable i is zero. Additionally the action code is executed.

A.3 States

State machines can be hierachical or flat. A state with substates is called a hierarchical statemachine. States can have entry code that is always executed if a state is entered. Exit code is executed whenever the state is left. Please note that the entry and exit code is also executed if a self transition takes place. If events shall be processed from a state without entry and exit actions being executed so called inner events can be used. If for a state no entry and exit actions were declared an inner event behaves exactly like a self transition.

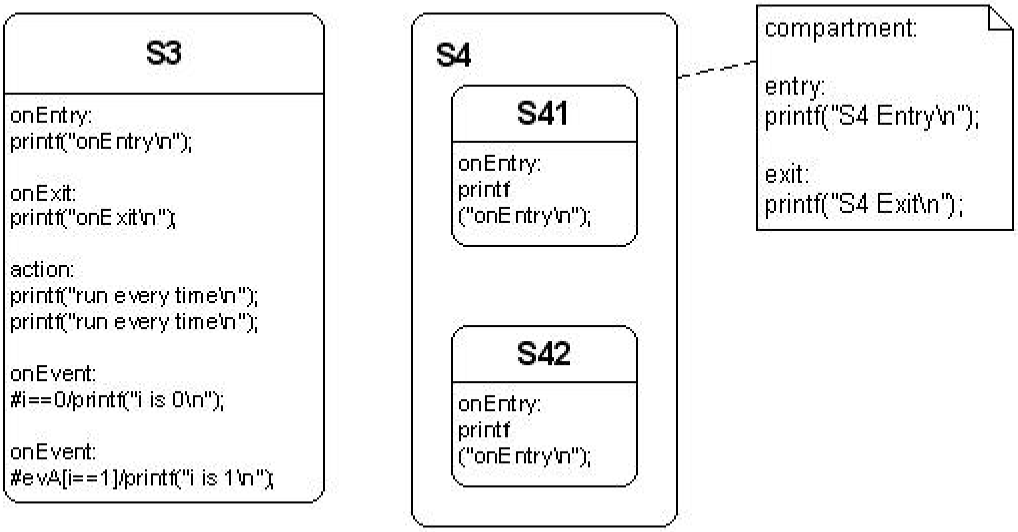
A state can also have action code. The action code is executed whenever the state is active just before event transitions are evaluated. This means that calculation results from the action code can be used as triggers for state transitions.

See figure 8 for an example. Code lines may span more than one line.

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 statemachine must be re-initialized to be reactive again.

The Cadifra UML editor does not allow to specify details for a complex state presently. Therefore entry and exit code for a state with substates must be defined in a linked note as shown in figure 8 on the right side. The note must start with the text `compartment:`.

Please note that not more than two levels of state hierarchy are supported yet.



```
onEntry:
printf("onEntry\n");

onExit:
printf("onExit\n");

action:
printf("run every time\n");
printf("run every time\n");

onEvent:
#i==0/printf("i is 0\n");

onEvent:
#evA[i==1]/printf("i is 1\n");
```

```
onEntry:
printf
("onEntry\n");
```

```
onEntry:
printf
("onEntry\n");
```

```
compartment:

entry:
printf("S4 Entry\n");

exit:
printf("S4 Exit\n");
```

Figure 8: Left: A state with entry-, exit-, action code and inner events. Right: Complex state with entry- and exit code specified in a linked note.

A.4 Include and Action Notes

To adapt the generated code to your needs you can add two notes to your design that have to start with either `include:` or `action:`.

All code following the include keyword is added at the begin of the generated statemachine code. This allows to include required header files or the definition of local variables needed within the statemachine.

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 B.1 this was used to receive events via a message queue.

B Design Questions

B.1 Running the statemachine in context of a RTOS

A frequently used design pattern with real-time operating systems is shown in the following figure 9.

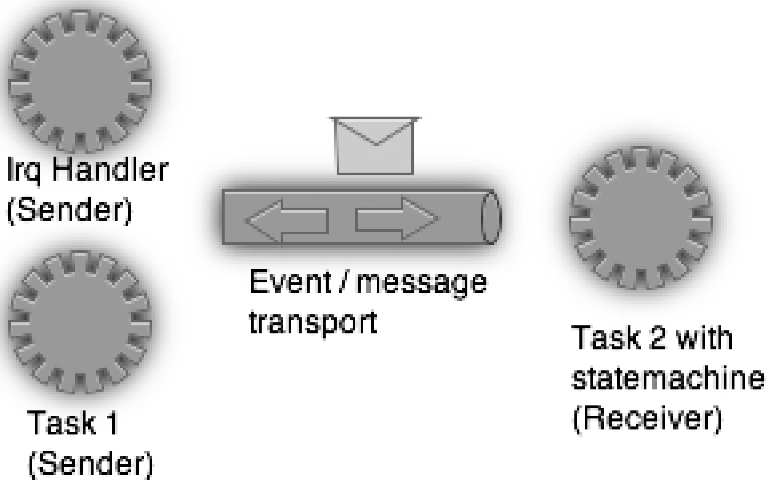


Figure 9: Communicating tasks exchanging messages between each other. At least one of the tasks executes a statemachine that reacts on the received events. As reaction new events to other tasks might be sent out.

1. A task executes a statemachine.
2. It waits for events by calling a blocking operating sytem 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 statemachine reacts on the new event
6. Jump to step 2

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

In folder `example2` the microwave oven statemachine 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 statemachine code which waits blocking until a new event is available. Figure 10 shows the sligly modified microwave oven specification file from section 3.1. 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 statemachine itself the machine reacts to the latest keyboard event.

B.2 Multiple Instances of a Statemachine

There are different options. First you can create multiple instances by declaring several instance variables. When calling the statemachine function handover the appropriate instance varaiable. Usually it is not simply possible to run multiple instances at the same time in different threads or tasks.

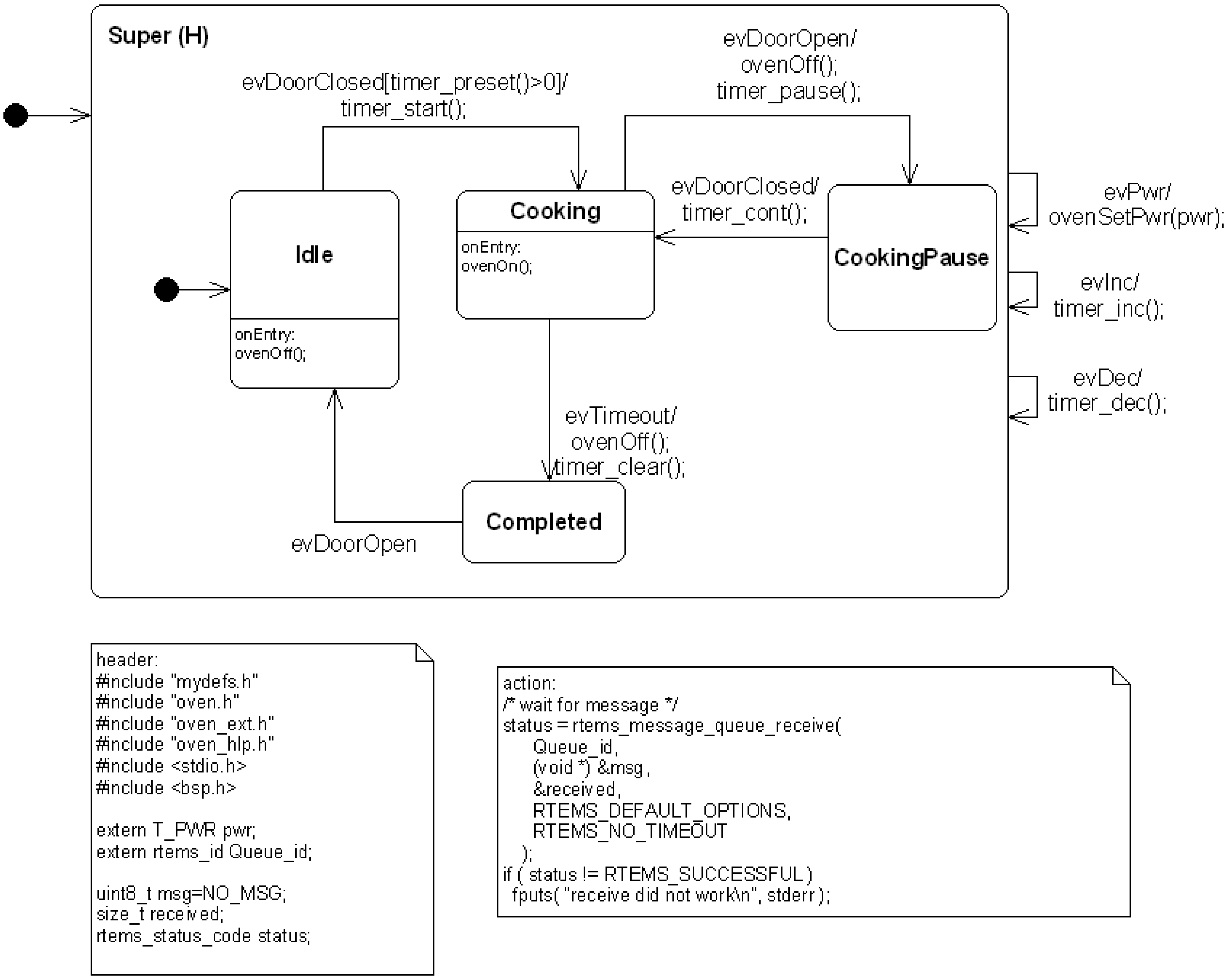
The other option is to generate the statemachine more than once using a different command line parameter for the machine name.

B.3 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 meat these requirements the keys `StateMachineFunctionPrefixHeader`, `StateMachineFunctionPrefixCFile` and `HsmFunctionWithInstanceParameters` can be adjusted according to your needs.

---

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



```
header:
#include "mydefs.h"
#include "oven.h"
#include "oven_ext.h"
#include "oven_hlp.h"
#include <stdio.h>
#include <bsp.h>

extern T_PWR pwr;
extern rtems_id Queue_id;

uint8_t msg=NO_MSG;
size_t received;
rtems_status_code status;
```

```
action:
/* wait for message */
status = rtems_message_queue_receive(
     Queue_id,
     (void *) &msg,
     &received,
     RTEMS_DEFAULT_OPTIONS,
     RTEMS_NO_TIMEOUT
   );
if ( status != RTEMS_SUCCESSFUL )
  fputs( "receive did not work\n", stderr );
```

Figure 10: Event handling code added as text note to the oven state machine.

The example below shows an interrupt service routine with the compiler specific extensions as required by mspgcc.[2]

```
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:

---

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

```
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.

C Drawing statecharts with Cadifra UML editor

The Cadifra UML editor 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.

C.1 Events

To add an event to a transition right click to the transition line and select `'New Text'` as shown in figure 11. For the event definition you must follow the syntax as described in section A.2. Only text associanted 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.

C.2 Hierarchical States

To draw hierarchical states it is best to set the `'Large'` flag in the outer state. With the this flag set the state name is shown in the upper left corner.

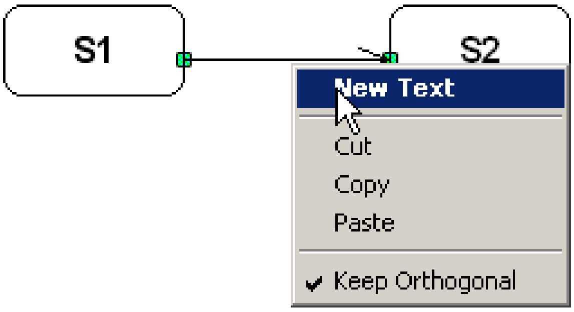


Figure 11: To enter events right click to the transition and use a text field to enter the event definition.

C.3 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 12. To edit the compartment double click on it and enter the definitions as needed. The definitions must follow the syntax as described in section A.3.

Presently state details can only be added to states without further substates.

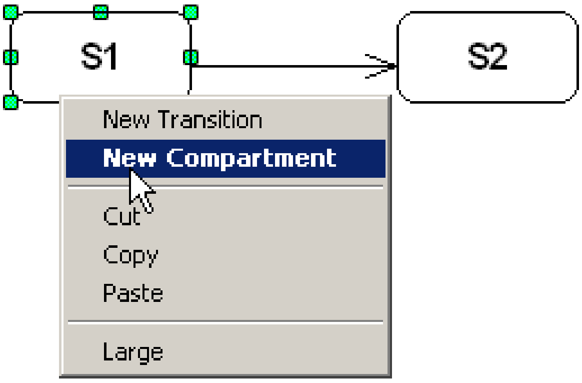


Figure 12: To enter state details right click to the state and add a compartment to it.