SpecC for Beginners: The Example of a Sounding Dice

Dirk Jansen

Technical Report ICS-00-14
Department of Information and Computer Science
University of California, Irvine
Irvine, Ca 92697-3425, USA

MAY 2000

SpecC for Beginners: The Example of a Sounding Dice

Dirk Jansen

Technical Report ICS-00-xx
Department of Information and Computer Science
University of California, Irvine
Irvine, Ca 92697-3425, USA
001-949-824-8919

d.jansen@ics.uci.edu

Abstract:

In this document the SpecC-language is demonstrated in a low complexity example with main features of the laguage explained and evaluated. Purpose of this document is to give beginners a first feeling for the language constructs and its usage in actual modelling. Allthough the language was not developed for these small tasks in general, it is well suited to cover all kind of modelling tasks with the existing semantic. The example consist of several finite state maschines connected and communicating with each other. The sounding dice example is taken from an existing design out of the authors course, normal used for student education in the VHDL-classes of the University of Applied Science, Offenburg, Germany.

CONTENTS:

Introduction

SpecC is a new modeling language developed at the Center for Embedded Computing Systems CECS by Prof. Gajski [Gaj] within the last few years. The language uses most of the C-language semantics with additional features to allow system specification and modelling. The intention is to give the user, which may be familiar with the usage of the C-programming language, a fast and high performance capability to specify and model complex systems, this systems may they be realized as embedded systems or as systems on silicon (SOC).

The C-language was selected because there is a large amount of existing software with many different types of applications as well as a lot of existing tools for high efficient programming. **SpecC Technology Open Consortium** [STOC], with more than 30 members, has been founded to standardize the language and to further develop application areas and tools. Everybody who is interested can download the SpecC compiler from http://www.ics.uci.edu/~specC/ with detailed documentation and examples. The book on SpecC written by Gajski et.alt., Kluwer Accademic Publisher [Gaj2], is the best source to get a first glance of the methodology.

Before getting started, some remarks:

- SpecC is not intended to do detailed, cycle based simulation on register transfer level (RTL) or gate level, allthough it is possible to do that.
- SpecC uses sequential semantics on many tasks and has a partly imperative- style, and in other parts a declarative- style. There are significant differences to the VHDL-style modeling.
- SpecC is mainly a behavioral description methodology and not yet usable for direct behavior synthezis. It is topic of further work to bridge this gap between behavioral semantics and synthezisable code. The behaviors are aimed to be synthesized in hardware **and** software, not every construct of the C-language can be reflected in some hardware-assemblies.

SpecC is intended to be used to specify systems which contain hardware and software. It is also intended to do hardware / software codesign with design space exploration and conceptional definition of complex, interacting units.

Methodology

SpecC is a language. The design has to be captured in an ASCII-text (SpecC-source) in the language semantics. In the next step the source is compiled with the SpecC-compiler to an executable object-code and executed.

Graphical representation allows to get some overview over the system partioning. Taken from [Gaj1], there are some easy to understand pictorial views of the main construction „behavior“. This shows properties of a system-block, as well as can be used as a container for sequential code. There are definite ports for variables entering and leaving the behavior-container. The pictorial view of such a construct is chosen as a rectangle with rounded corners [Gaj3]. See Figure 1.

The identifier, or the name of the behavior is printed in the upper left corner. The behavior contains a code in a

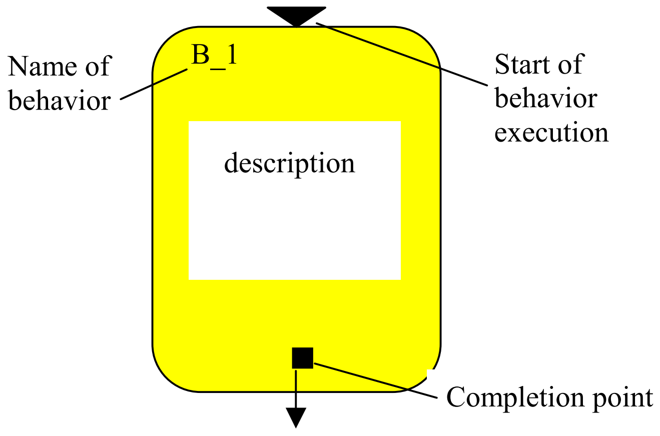

Figure 1: Graphical representation of a behavior

general sequential way, which may be represented by some pictorial or textual description in the body of the symbol. The entering of the behavior is marked with a **filled triangle** on the upper border of the behavior-scheme. The leaving point or, the **point of completion** of the behavior, is marked by a **black square**. These marks can be placed anywhere in the symbol

The above behavior-symbol in figure1 is a so-called **leaf-behavior**. There are no further hierachical levels below.

In general, the behavior is a **compound- behavior**, consisting of several hierarchical levels. From graphical representation, these child-behaviors are drawn with the same symbols inside the parent compound behavior. Child behaviors are instances of behavior-templates, which are placed inside the parents. They may be connected in parallel (*par{ }*-construct) or in serial manner and may be again compound behaviors etc., until there are only leaf-behavior-childs on the lowest level of hierarchy.

The highest behavior in the hierarchy is the „*Main*"-behavior, similar to the *main()* –function in each C-program. Normaly, this *Main*-behavior contains the testbench and the Device under Test (DUT), the system which has to be evaluated.

Each SpecC-Program starts with execution of the „Main"-behavior. The entering mark is the starting point for all behaviors inside the main.

The Task

To demonstrate the features of a lanuage it is well advised to use an easy to understand example. The example should not be a simple task in which everybody knows the solution in advance. But it should also not be too complex, to keep the reader interested and enjoing the way the ideas are modeled. The example should motivate him to do his own exercises. The „sounding dice"-example, which is a game, is well proven in student VHDL-education [FHO]. This example exists as a schematic design, a VHDL-design as well as a working VLSI-circuit [FHO]. Since it is not possible, to show all features and constructs of SpecC in one example, this tutorial is mainly targeting fsm-related digital designs.

For constructing an electronic dice we need:

- a **display**, formed by seven LED's, showing the points of the dice (see Figure 2),
- a **counter** with 6 states, giving the points of the dice,
- a **clock**, driving the counter.
- A **button**, which is pressed for a certain time, letting the counter run.

The result is unpredictable (random) because of the uncertainty of the time which the button remains down, together with the high clock-frequency (only by human use, not in simulation). It is even distributed in statistical sense.

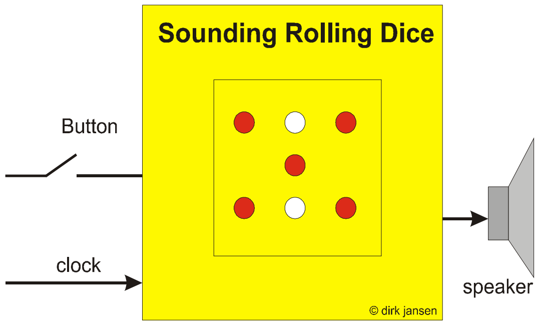

Figure 2: Block-diagram of the sounding, rolling dice.

A dice, designed in this simple way, will only get boring laughter from gambling people, because the result is directly presented after releasing the button. In order to make it more interesting, we add some additional requirements:

- The dice shall not stop rolling, after release of the button, it shall „roll out", meaning the display shall show a sequence of points with a frequency going down for some seconds until it stops at last on a number. This roll out should last some seconds.
- During roll out, each new number should be anounced by a „click"-sound.
- After roll out, there shall be a melody sound, depending on the result. For a „six" a 3 tone (or more) victory-melody, for a 2,3,4,5 only two tones (sorry, you lost…) and for a „one" some other 3 tone melody.
- The results should be shown for a certain time after stopping, after that a time-out should shut off the LEDs to spare battery.

There may be some more specific requirements like primary clock frequency, timing and frequency of the sounds and the click-sound, roll out perfomance, and power down etc.

Setting up the model

The first thing which must be thought of is, how to handle the inputs and outputs of the system and how to organize that which is commonly called a testbench.

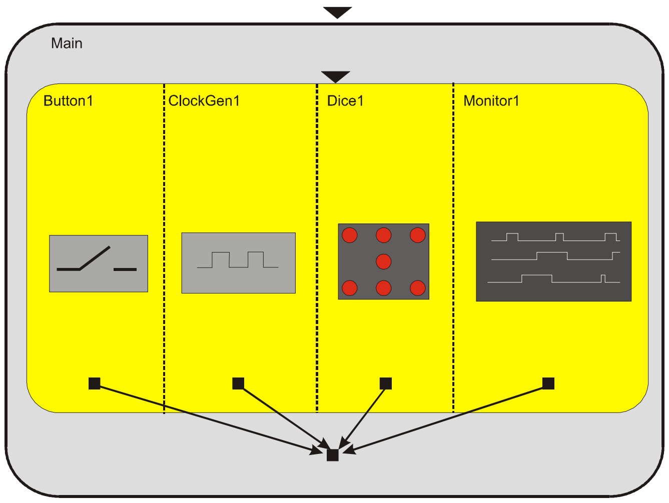

Figure 3:Modell-setup with testbench

Figure 3 shows a symbolic representation of the model-setup. As mentioned before, the Main – behavior is worked out as the testbench for the DUT, which is the behavior *Dice*. The testbench consists of the input device, which is a button, a clock-generator giving the main time reference and a monitor, which captures the signal of the testbench and responds from the DUT. Later we will see more of them. From the graphics you see that all 4 behaviors are working in parallel (concurrently) because they are existing independently from each other. This is a batch process, so it is not intended to make a real time simulation. The simulation starts with entering the `Main` and it ends with finishing the `Main`.

Lets have a look on the code for that (Figure 4):

```
//////////////////////////////////////////////////
////////////////////
// SpecC-Program of a dice with slow run out
dice2.sc
// Version No V 1.0
// 20.4.2000 Dirk Jansen
// simple testbench included
//////////////////////////////////////////////////
////////////////////
#include <stdio.h>
#include <sim.h>

#include "dice_header.h"
#include "dice_body.sc"
#include "monitor.sc"



//////////////////////////////////////////////////
///////////////////
//                      MAIN BEHAVIOR
//////////////////////////////////////////////////
///////////////////
behavior Main(void)                  //testbench
{
bool           roll_flg=      false,
               button_flg=    false,
               lights=        false,
               time_out_flg=  false;
bool           nco_flg=       false;

char           speaker=       '_';
int            Displ_State=   7;
int            Points=        9;
int            Nfreq=         20,
               accum=         254;

bit[6:0]       LEDs=          0000000B;


event          eclock;

//instances

Button         Butt1(button_flg);

ClockGen       ClockGen1(eclock);

Monitor Monitor1(eclock,button_flg,roll_flg,
nco_flg,time_out_flg,lights,Points,speaker,
Nfreq,accum,LEDs);

Dice Dice1(eclock,button_flg,nco_flg, roll_flg,
lights,time_out_flg, Points,speaker,
LEDs,Nfreq,accum);

// main

void main(void)
{
       printf("run simulation\n");
       pTrace=fopen("scc_traces.dat","w");

       par{
               Butt1.main();
               ClockGen1.main();
               Monitor1.main();

               Dice1.main();
          }

printf("\n\nfile scc_traces.dat written to
disc!\n\n\n");
       fclose(pTrace);
}

};
```

Figure 4:-Main-behavior with the testbench

The program starts with the *#includes* directives for including all the stuff needed to get the system run. It is good practice to separate your code into different files, such as they are presented below:

dice_header.h

dice_body.sc

monitor.sc

dice2.sc

These are the 4 files for the sources. The #includes <*stdio.h*> and <*sim.h*> are standard-includes when using the language. The suffix .sc is a **SpecC-specific suffix** invented to distinguish SpecC-code from common C-code-sources.

The included header file *dice_header.h* contains all the declarations of the later used behaviors as well as the global *event EndofSim.*Some constants such as MAXSIMTIME, CYCLETIME, STARTDELAY, TIMEOUT and BUTTONDNTIME, are also used to control the simulation. It is good practice to put all numerical simulation relevant constants here, so they can easily be changed (These numbers may also be imported via the command line, making the simulation again more flexible). The name of the above constants describe what they mean and what they are used for.

The *Main*-behavior (Figure 3) starts with the declaration of the used variables. Because we use a monitor on this high level of hierarchy, all variables which we want to have a look on have to be defined here and must be rooted to the related child-behaviors. Variables which we are not interested in monitoring may be declared on lower level of hierarchy.

A part of the declaration section is the declaration of the child-behaviors, which are now **instances** of the templates with the variables connecting the ports. The name for the instances are taken from the template by adding some numbering. Here the first instance of the template `button` is called `button1`. There may be more than one instance from the same template behavior.

This section has something of a netlist. It describes the interconnection of the instances via the variables. Variables are not signals, although there is some similarity with that and with the boolean- or bit-constructs it may behave somewhat like a behavior. But the C-language allows more transfer in the ports than only signals. In behavior modeling this can be effectively used for abstract formulations. We will later see that it is no problem to transfer char-types or even complete structures through the ports.

It also has to be mentioned that events have to be normaly routed through the ports. An exception of this rule is only the use of global variables, which **should be reserved** for certain things. So, in this case where we have a central clock defining the heartbeat of the dice, the clock-event named *eclock* is declarated in the *Main* and routed to all corresponding child-behaviors.

It is again good practice to name variables, behaviors and events related to their type, so events shall start with an „e..“, behaviors shall start with capitals etc.

The declaration section follows the main program. From point of view of the C-language the *main*-program is a definition of a sequential function and it is handled as such. The behavior starts executing at the beginning of the *main(void)*. There may be additional local declarations at this point. The following code starts with conventional C-code for printing a message to the terminal (`printf( …)`) and opening a file for capturing the monitor-data. This is written in the *main*-behavior but may also be done in the monitor-behavior.

The concurrent execution of the child-behaviors is done by the SpecC-construct `par {   }`. The main-functions of the instanced behaviors are executed. The construct ends, when all called `main()`-functions are finished. There is a final message to the terminal and closing of the opened file.

The simulation lasts as long as these behaviors are alive. If these behaviors are never-ending, there is a **deadlock-situation** and you get a warning from the runtime-system, making a **program abbort**. Consequentially this forces you to implement some superviser in your testbench, as we will soon see.

The Testbench

The testbench (see Figure 3 and 4) is setup in the *Main*-behavior. There the *Dice*-behavior is intanced together with the the *button-* and the *ClockGen*-behavior, which are needed to get the Dice rolling:

- ***Dice***: The main subject to test,
- ***button***: simulates the pressed buton with timing,
- ***ClockGen***: gnerates the clock-heartbeat of the system,
- ***monitor***: catches the status of all variable with each clock –event.

All codes which belong to the *Dice*-environment or the testbench are in the included *monitor.sc* – file. We will pick up some parts of this file, showing what can be done with the SpecC-language.

The code in Figure 5 describes the clock generator, emitting the main clock to the dice-model and to the monitor, catching up all data with each clock event.

Clocking and timing is something which is not existent in a conventional C-program. It makes sense to simulate timeless, which many C-programmers do every day. In the simulation of real things, time is a dominant parameter. Things are responding on events with certain behavior after certain time.

There is a general statement *(int)now()*, which picks up the internal simulation time. The *(int)* is a cast expression, allowing you to use this value as an integer, starting with „0". Sequential processing of functions and statements, or whatsoever, do not need (simulation) time. So, if there are no timing statements, internal time will stay at zero forever. You know that this is not very realistic. So SpecC contains some constructs for representing time, first and most easy is:

```
Waitfor(DELAY);
```

which inserts a delay of so many internal simulation unit timesteps as the constant DELAY says. With this construct you may now be able to setup some scheduling or asynchronous processing, which may be realistic enough for representing some designs. More detailed designs need a heartbeat, a central clock, or maybe even more than only one clock.

The waitfor() construct only stops the behavior from executing where the statement is placed. All others are running on their own (if concurrent). So there is a message mechanism via the event declaration, which allows to notify a behavior that an event has taken place:

```
notify e_abcd;
```

Please notice, events only make sense together with timing and delay-statements. Events may be declared as normal variables and routed in the same way. Everybody who is on the line and has access to the event, may react on it.

There is only one reaction type defined yet in SpecC:

```
wait e_abcd;
```

where *e_abcd* is the declared event. The actual behavior where this statement is placed will be put on wait until the event has taken place. This is done in a simulation timescale, there is no waiting in reality.

If there is a behavior waiting on an event, but there is no behavior active that may generate the event, there will be a **deadlock-warning** and an abbort of the program.

```
behavior ClockGen(out event clk)
{
void main(void)
 {
   do{
      waitfor(CYCLETIME);//Cycletime defines time
      notify (clk);     // between clock-events
     }while(SimActiv());
       notify EndofSim;// main end of simulation,
 return;
 }
};
```

Figure 5 :Code of the ClockGen-Behavior

The clock generator in Figure 4 is now easily done with these constructs. The behavior starts with a wait – statement, blocking execution for a certain delay time. After that, a notify message is generated. Because this is conventional C-language, all this is done in a *while*-loop which is executed as long as simulation lasts. For this the function

```
SimActiv()
```

is used in which it is declared as *bool* and is defined as (see Figure 6):

```
bool SimActiv()
{
 return ((int)now()<MAXSIMTIME);
}
```

Figure 6: Function for comparing actual simulation timewith a limit set by MAXSIMTIME

This is an easy comparison of the actual simulation time with the constant MAXSIMTIME. The constant is defined in the header.

This *while loop* loops in the ClockGen – code (Figure 5) until simulation time is finished. This behavior stays active and generates after each delay, which is CYCLETIME, an *eclock*-event. After finishing the loop, or after simulation time ends, an *EndofSim*-event is notified. This message can be effectively used to finish all hanging behaviors and to leave the program without error messages or deadlock-warnings.

This *EndofSim*- event is used in many places, it does not do anything else than closing open behaviors. So it makes sense, to put it off from the port-mappings and declare this event as a **global**. You can do this also with the *eclock*, if the system is similar and there are many synchronous concurrently working blocks. Here in the actual design, clocking has some functional behavior, so it is not used in a global fashion.

The *clock* is, of course, scaled in a timescale which is not predefined as nanoseconds, microseconds or whatsoever. The linkage to your real timescale has to be done by yourself. If you are using time constructs and want to model some internal delays in the behaviors, you are well advised to use a cycletime which is larger than 1, i.e. 10 or more. Because this will give you the timing resolution of your simulation.

The monitor (see Figure 7) is programmed in a straight forward manner. It should be alive all the time. It also contains this *while loop* with a wait-statement in the beginning:

```
Do {
{ wait (Dckl,EndofSim);
…….
fprintf(pTrace," ……", list of variable to print)
}while (SimActiv());
```

Figure 7: Code of the monitor-behavior

The `fprintf();` statement writes all stuff which is collected by the port to the disk-file. Here it is a text-file for easy reading. Of course, it would be nice to have a kind of trace-window, with cursors, zooming etc, but this comfort will be there sometime in future. Don't try to send this information to the terminal, there is a lot and it is gone! The monitor writes every clock-event all the information which it finds at its ports to the disk, so there will be **some thousand lines of text.**

We will make it a little more easier than it works in reality. The existing dice runs with a clock-frequency of 32 000Hz. Rollout lasts about 5 seconds, the melody additional 4 seconds, timeout is programmed to 16 seconds. So there would be a minimum of 30 seconds or about 1 million clock cycles to describe the detailed behavior. A textfile of that length is handable, but nobody would like it. Actualy you would have a problem looking into that file with a standard editor. So we have to simplify, which is done in such a way that the clock frequency is radically put down to a 100 Hz level, and this allows you to see many effects. What is not possible to simulate with that low frequency are the sound outputs, which are symbolized here by printing a character instead of emitting a sound.

The Dice Model

The Dice-model is a compound-behavior of four behaviors:

- Main_fsm1
- Cntrl1
- Fsm_dice1
- Display1,

These 4 behaviors consist again of sub-compound-behaviors (see Figure 8 and 9). See the code:

```
behavior Dice(event clock_int,in bool button_flg,
out bool nco_flg, out bool roll_flg, out bool
lights,out bool time_out_flg, out int Points, out
char speaker,out bit[6:0] LEDs, out int Nfreq,out
int accum)
{
  event DiceClock;

 // Instances of used behaviors

Main_fsm   Main_fsm1(clock_int,button_flg,nco_flg,
time_out_flg,Points, roll_flg,lights, speaker);

Cntrl      Cntrl1(button_flg,DiceClock, clock_int,
roll_flg, nco_flg, Nfreq, accum, time_out_flg);

fsm_dice   fsm_dice1(DiceClock, Points);

Display    Display1(clock_int,Points,LEDs);

void main(void)
  {

// concurrent operation

   par  {
              Cntrl1.main();
              fsm_dice1.main();
              Display1.main();
              Main_fsm1.main();
        }

   return;
  }
};
```

Figure 8: Code of the "Dice"- compound - behavior

The behavior starts with the intanciasation of the child-behaviors. The *main()* contains again a *par{}* statement for concurrent execution of the child-behaviors. There is no need for a *while-loop*, because this may be done if needed in the child-behaviors. The dice finishes when all the children have finished. An execution finishes from child to parents.

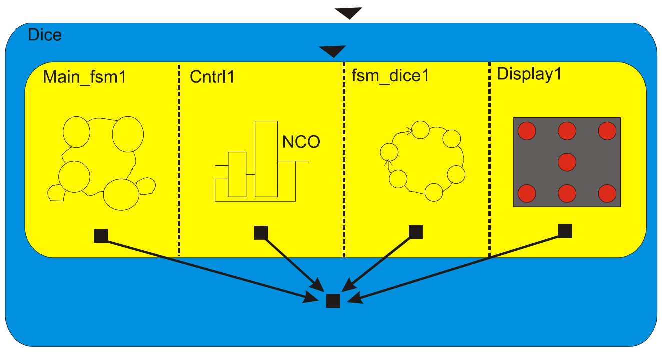

Figure 9 :The "Dice"- compound behavior

Although there may be other partitioning possible of the subject, the used one allows to demonstrate some constructs and often uses modeling tasks.

```
behavior Cntrl(in bool En ,out event DiceClk,in
event clk, inout bool rollflg,out bool ncoflg, out
int Nfreq, out int accum,out bool time_out)

  { int t=0;           //local time count

   void main()
   {
      do{

      wait clk,EndofSim ;// wait on clock-ev.

      accum+=Nfreq;            // basic NCO

      if (accum > 255) {accum=0;
                       if (En) Nfreq+=2;
                       else Nfreq-=2 ;
                           // freq up or down

        if (Nfreq > 31) Nfreq=31;
        if (Nfreq <= 0)
        {Nfreq=0;  //limiting

ncoflg=true;

        rollflg=false; // end of rolling out
                                             }
        notify(DiceClk); //generate NCO event
         };

t++; if (t*CYCLETIME>=TIMEOUT)
time_out=true; //Timeout

        }while(SimActiv());

       return;
   }
  };
```

Figure 10 :The code for behavior "cntrl", containing the description of a numerical controlled oscillator (NCO) .

`Main_fsm` is a finite state machine behavior (Figure 7), using the SpecC-construct. This behavior controls the other behaviors and interacts mainly with **`Cntrl`**. Later, it will be discussed in more depth.

`Cntrl` is the behavior, which generates the secondary roll-out dice-clock, using some NCO – ideas. It also generates the timeout and some other flags for controlling the **`Main_fsm`** (see Figure 10).

The behavior generates the *ncoflg*, which is a flag used in the *Main_fsm* to go to the next state which is playing the melody. This flag is set when the numerical controlled oscillator (NCO) goes to 0, or better, after roll out. This is the moment when the melody has to start. This may also be signaled by an event, but the finite state maschine construct in *Main_fsm* is only level-sensitive and needs flags for decision making.

In general, a flag in this case is a better selection than an event, which must be handled much more carefully. A flag may again be easier to show in the monitor-file, an event which can only be represented indirectly. The event, generated here, is the ***dice_clock***, which is notified when the *accum* -variable reaches the predefined limit of 256 and is reset to 0 (this is a very simple form of NCO). The value *Nfreq*, which is added with each *eclock*-event, defines the frequency of this NCO. If the number is large, 256 is reached in few accumulations, if *Nfreq* is small, it needs a larger number of accumulations to reach 256. So if *Nfrequ* is 1, which is the lowest frequency, it needs 256 accumulations maximum or 2.5 seconds in our simulation timescale to emit the *dice-clock event* (which drives the dice-fsm).

Time-out-flag is also generated in this routine, although it may be generated everywhere where there is *a while-loop* waiting on *eclock*. From the system side, this is again a finite state machine (a counter), flagging some status-signal for controlling something. Here the signal is used to shut off the lights in the related behavior.

```
behavior fsm_dice(in event Dclk, inout int state)
{       // main dice behavior,non synchr

void main(void)
 {     do
     {
       wait(Dclk,EndofSim);
       state++; if (state > 6) state=1;

     }while(SimActiv());
   return;
 }
};
```

Figure 11 :Code of the" fsm_dice" - behavior, describing a state machine with 6 states

`fsm_dice` is another state machine, but it uses a different approach for description (see Figure 11) . It contains the counter only and works very straight forward.

The behavior is blocked at the beginning by the wait-statement, which waits on the *Dclk*-event (which is not the *eclock*). As before described, it contains the *while loop*, executing this behavior every time there is a *Dclk*-event. The code contains a simple accumulation of the variable *state* and resetting it to 1 if the value amounts more than 6. This is a ring counter with 6 states, synchronized by *Dclk* and showing its states at the ports. This construction can be easily adopted to describe any kind of finite-state maschine, moore- or mealy-type.

`Display` is the easiest type of behavior (see Figure 12). This behavior only describes a mapping of the *(int) –states* to a binary representation. It shall demonstrate the handling of bit-vectors in SpecC and does not contain any timing. Also the behavior contains the *while-loop* for continous existence and executes with each *eclock*-event:

```
behavior Display(in event Dclk, in int Pts,
out bit[6:0] LDs)
{

void main(void)
 {
 do{
  wait Dclk,EndofSim;

  switch (Pts)
   {
        case 1: LDs=0001000B;break;
        case 2: LDs=1000001B;break;
        case 3: LDs=1001001B;break;
        case 4: LDs=1010101B;break;
        case 5: LDs=1011101B;break;
        case 6: LDs=1110111B;break;

        default: LDs=0000000B;break;
   };

   }while (SimActiv());
  return;
 }
};
```

Figure 12 :Code of the “Display” - behavior

The mapping is done in a classical way using the C-switch construct, giving a table-like concentrated code. The bit-declarated variables are directly loaded with the bit-constants written in a C-conformal style.

The *Main_fsm*: A Puppet in a Puppet

The most complicated behavior is the **`Main_fsm`**, which will now be discussed in more detail. It is again a compound-behavior, hosting the *fsm{}* construct of the SpecC-language. Figure 13 shows the code and Figure 14 gives some graphical representation of the Main_fsm-behavior.

```
behavior Main_fsm(in event clock_int,in bool
button,in bool nco_null,in bool time_out,in int
pts, out bool roll_dice_flg,out bool lights_on,
out char speaker)
{
Stop    Stop1(clock_int,button,lights_on);
RollDice RollDice1(clock_int,roll_dice_flg);
Melody         Melody1(pts,speaker);
WaitOnTime     WaitOnTime1(clock_int);
LightsOff      LightsOff1(lights_on);

void main(void)
{
fsm{

Stop1   : {if (button) goto RollDice1;goto Stop1;}
RollDice1:{if (nco_null) goto Melody1;
goto RollDice1;}
Melody1        : {goto WaitOnTime1;}
WaitOnTime1    : {if (time_out) goto LightsOff1;
       goto WaitOnTime1;}
LightsOff1     : {break;}

   };
return;
}
};  //end behavior main_fsm
```

Figure 13:Code of the „Main-fsm“- behavior. This behavior is a compound - behavior consisting of 4 subbehaviors.

After instanciation of the behaviors *Stop1,RollDice1, Melody1, WaitOnTime1* and *LightsOff1*, which represent the main successive state-behaviors of the „Dice“-system, the SpecC-construct *fsm{}* is used to schedule the performance.

- ***Stop1*** is the state at the beginning, before any button is pressed.
- ***RollDice1*** is the state, when the dice is rolling: rolling-up as well as rolling-down, until it stops.
- ***Melody1*** is a compound behavior containing the behaviors for generating the sound sequences.
- ***WaitOnTime1*** describes the situation when the dice has stopped and displays the point-results.
- ***LightsOff1*** is the situation when the display is shut off after all.

The controlling flags are *button*, which originates from the testbench. *nco_null*, originates from the *Cntrl*-behavior and signals the rollout of the dice and *time_out*, again from *Cntrl*. These flags are used to transfer from one (state-) behavior to the next. It stays in the behavior, if the flag is false. It is important to understand that there is no event causing this change in behavior flow but the flags coming in from the ports. All synchronization which may be needed has to be done in the child behaviors.

Let us look on the code of the first child behavior ***Stop1.*** See Figure 15:

```
behavior Stop  (in event clk,in bool but, out bool
lights)
{
void main(void)
 {wait clk,EndofSim;
  if (but) lights=true; else lights=false;
  printf("s");
 return;
 }
};
```

Figure 15 :Code for "Stop"- behavior

In the beginning this behavior contains the synchronization mechanism of a wait-statement. It waits for the *eclock*-event, which is the main clock. This behavior is processed synchronous to the main clock. But there is **no *while-loop*** and there should not be a *while-loop*.. After sensing the *eclock*-event, the behavior-statements are processed sequentially which is switching the lights on. For information for the user only, it prints an „s" (like stop-state) to the terminal. You can conclude from the terminal-output shown later, that this behavior is processed again and again as long as the button is pressed (counted in simulation cycles), giving a long „ssss…" in the terminal output. The „*goto*"- statement in the fsm{} construct points to the same behavior building a loop if the condition is not fullfilled. This is equivalent to a loop on itself in a state-bubble-diagram.

If one tries to take the wait statement from the loop, he

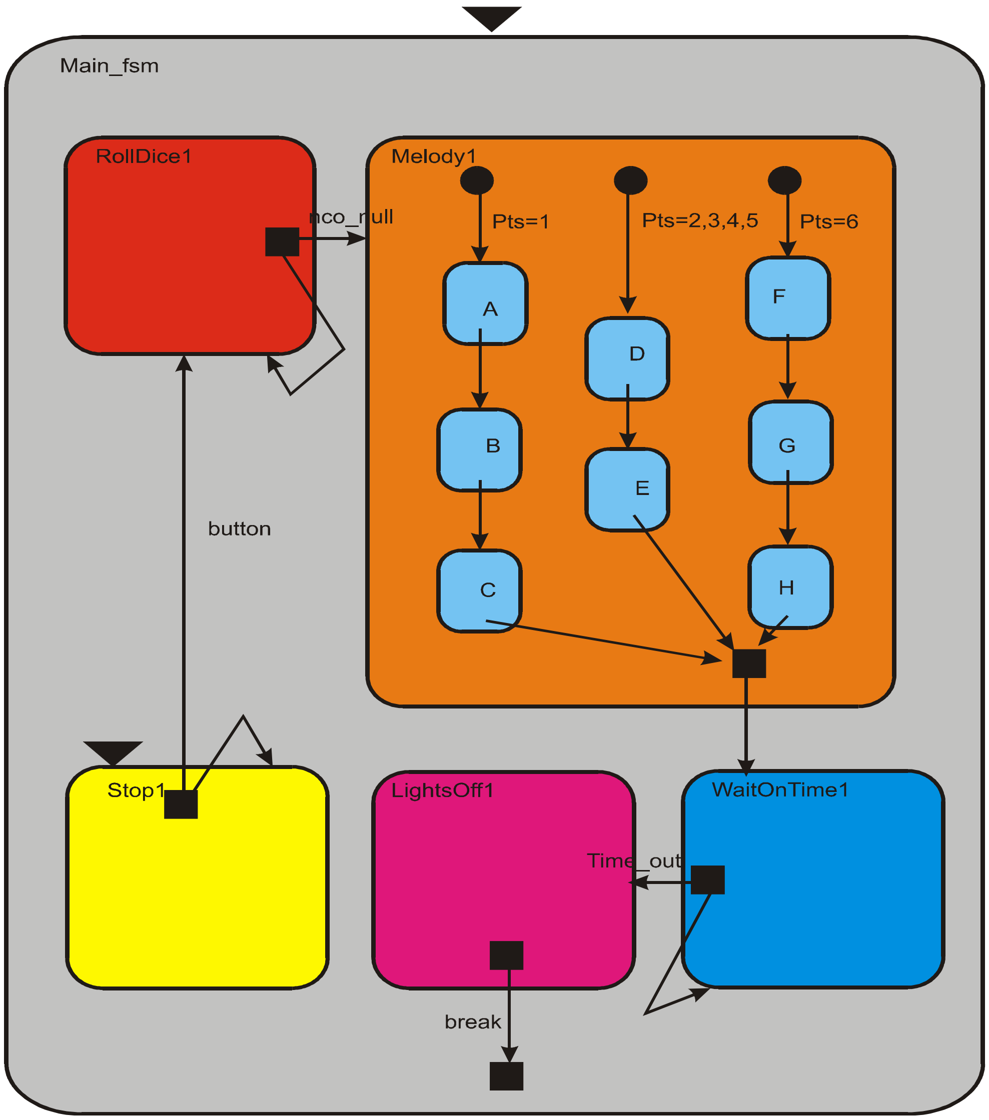

Figure 14: The „Main_fsm"- compound behavior

or she will get an infinite loop here, blocking out the overall program (only ^C helps). The same happens, if one tries to put into the behavior a *fscan()* function, getting something from the keyboard. This would be nice for interactivity, but keyboard is old fashion, so there will be a solution for that with a framework. It should be marked up as a rule, **never to introduce a stop-loop** into the behavior of a fsm – construct, which points on itself.

The child-behaviors of the *fsm{}* must not be synchronized. As an example the *Melody* – behavior Figure 16 may be taken. This behavior is only processed once so it doesn't need synchronization:

```
behavior Melody(in int pts)
{
void main(void)
  {
   waitfor(1);                //setup time
   printf("\nmelody out\n");

switch(pts)    {
 case 1: Sound_A.main(); Sound_B.main();
Sound_C.main(); break;
 case 2: Sound_D.main();Sound_E.main();break;
 case 3 : Sound_D.main();Sound_E.main();break;
 case 4 : Sound_D.main();Sound_E.main();break;
 case 5 : Sound_D.main();Sound_E.main();break;
 case 6  :Sound_F.main();Sound_G.main();
sound_H.main(); break;

 default        : break;
               };
  return;
  }
};
```

Figure 16 :Code of behavior "Melody" as an example for behavior sequencing

This behavior is a good example for serial processing of child-behaviors, such as the behavior *Sound*. *Sound* should create a sound, but in lack of interactivity only a character is output here. *Melody* is called with a parameter, which is used in the switch-statement to control the flow to the child-behaviors. So with *pts* = „6" there is a 3 - tone sound sequence. With 2,3,4,5 another sequence consists only of 2 tones. Each child *Sound_A*, *Sound_B* …- behavior is an instance of the template *Sound*, which is parameterized by a value passed through the port. Of course, the same may be done more easily using a C-function construct, but it is done here for demonstation of serial activation.

Compilation and Executing

After setting up the code, the source files are compiled with

> **scc dice2**

where *scc* is the call for the compiler and *dice2* is the main program source with suffix **.sc**. The header and all other files must be in the same directory. They are included in the compilation via the *#include* directives in the code. There are a lot of options together with the compiler call. Look into the *man*-pages or the *help*. After solving all error- and warning problems, (there may be a lot of them in the beginning), the compiler generates the following files:

dice2.si	**intermediate representation**
dice2.cc	**C++ output file**
dice2.o	**linkable objectfile**
dice2	**executable file.**

The file with the suffix *.si* is an intermediate file for experts only. The main output of the scc – compiler is the C++ -file with the suffix *.cc*. All SpecC-code is translated to C++-code. This code is than further processed using a standard C++ compiler,(i.e. the gnu++), which is done here in the makefile, getting out an executable which may be processed in the usual manner.

Results

The execution of the dice2 – Simulation generates the following terminal screen Figure 17:

```
run simulation
sssss…………………………………………………………………………………………….
……………………………………………………………………………………..
melody out
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,
file scc_traces.dat written to disc!
```

Figure 17 : Terminal screen, executing the dice-simulation

After simulation-start the program prints out the first message „run simulation", which we programmed in the beginning. The following five „s" are generated by the *Stop1*-behavior, as discussed before. This is, because we programmed a STARTDELAY = 10, with CYCLETIME = 2, 5 times the *Stop1*-behavior is processed. The following dots „." are a similar output from the *RollOut1*-behavior, printing each time a dot when *RollOut1* is active. After that there is a short message when the melody is produced. This behavior is processed only once. Now the *WaitOnTime1*-state is entered, each time printing a comma „ , „ . After time-out, all behaviors are closed and the message „`file scc_traces.dat written to disc!` “ is send to the terminal.

So look on the results:

```
|       2 |0 0 0 0 0 | 9 _  | 18   0 | 0000000b |
|       4 |0 0 0 0 0 | 1 _  | 18   0 | 0000000b |
|       6 |0 0 0 0 0 | 1 _  | 18  36 | 0001000b |
|       8 |0 0 0 0 0 | 1 _  | 18  36 | 0001000b |
|      10 |1 1 0 0 1 | 1 _  | 18  72 | 0001000b |
|      12 |1 1 0 0 1 | 1 _  | 18  72 | 0001000b |
|      14 |1 1 0 0 1 | 1 _  | 18 108 | 0001000b |
|      16 |1 1 0 0 1 | 1 _  | 18 108 | 0001000b |
|      18 |1 1 0 0 1 | 1 _  | 18 144 | 0001000b |
|      20 |1 1 0 0 1 | 1 _  | 18 144 | 0001000b |
|      22 |1 1 0 0 1 | 1 _  | 18 180 | 0001000b |
|      24 |1 1 0 0 1 | 1 _  | 18 180 | 0001000b |
|      26 |1 1 0 0 1 | 1 _  | 18 216 | 0001000b |
|      28 |1 1 0 0 1 | 1 _  | 18 216 | 0001000b |
|      30 |1 1 0 0 1 | 1 _  | 18 252 | 0001000b |
|      32 |1 1 0 0 1 | 1 _  | 18 252 | 0001000b |
|      34 |1 1 0 0 1 | 2 _  | 20  20 | 1000001b |
|      36 |1 1 0 0 1 | 2 _  | 20  20 | 1000001b |
|      38 |1 1 0 0 1 | 2 _  | 20  60 | 1000001b |
|      40 |1 1 0 0 1 | 2 _  | 20  60 | 1000001b |
|      42 |1 1 0 0 1 | 2 _  | 20 100 | 1000001b |
|      44 |1 1 0 0 1 | 2 _  | 20 100 | 1000001b |
|      46 |1 1 0 0 1 | 2 _  | 20 140 | 1000001b |
|      48 |1 1 0 0 1 | 2 _  | 20 140 | 1000001b |
|      50 |1 1 0 0 1 | 2 _  | 20 180 | 1000001b |
|      52 |1 1 0 0 1 | 2 _  | 20 180 | 1000001b |
|      54 |1 1 0 0 1 | 2 _  | 20 220 | 1000001b |
|      56 |1 1 0 0 1 | 2 _  | 20 220 | 1000001b |
|      58 |1 1 0 0 1 | 2 _  | 22   0 | 1000001b |
|      60 |1 1 0 0 1 | 3 _  | 22   0 | 1000001b |
|      62 |1 1 0 0 1 | 3 _  | 22  44 | 1001001b |
|      64 |1 1 0 0 1 | 3 _  | 22  44 | 1001001b |
|      66 |1 1 0 0 1 | 3 _  | 22  88 | 1001001b |
|      68 |1 1 0 0 1 | 3 _  | 22  88 | 1001001b |
|      70 |1 1 0 0 1 | 3 _  | 22 132 | 1001001b |
|      72 |1 1 0 0 1 | 3 _  | 22 132 | 1001001b |
|      74 |1 1 0 0 1 | 3 _  | 22 176 | 1001001b |
|      76 |1 1 0 0 1 | 3 _  | 22 176 | 1001001b |
|      78 |1 1 0 0 1 | 3 _  | 22 220 | 1001001b |
|      80 |1 1 0 0 1 | 3 _  | 22 220 | 1001001b |
|      82 |1 1 0 0 1 | 3 _  | 24   0 | 1001001b |
|      84 |1 1 0 0 1 | 4 _  | 24   0 | 1001001b |
|      86 |1 1 0 0 1 | 4 _  | 24  48 | 1010101b |
|      88 |1 1 0 0 1 | 4 _  | 24  48 | 1010101b |
|      90 |1 1 0 0 1 | 4 _  | 24  96 | 1010101b |
|      92 |1 1 0 0 1 | 4 _  | 24  96 | 1010101b |
|      94 |1 1 0 0 1 | 4 _  | 24 144 | 1010101b |
|      96 |1 1 0 0 1 | 4 _  | 24 144 | 1010101b |
|      98 |1 1 0 0 1 | 4 _  | 24 192 | 1010101b |
|     100 |1 1 0 0 1 | 4 _  | 24 192 | 1010101b |
|     102 |1 1 0 0 1 | 4 _  | 24 240 | 1010101b |
|     104 |1 1 0 0 1 | 4 _  | 24 240 | 1010101b |
```

Figure 18: Results of simulation (from timestep 1 to timestep 104), showing simulation start, taken from file scc_trace.dat.

The first column shows the simulation time. Defined as 10 msec each, only every second mark is printed because of CYCLETIME = 2. Each row is an *eclock*-event. The next five collumns are the flags:

button rollflg ncoflg tioutflg light

With simulation time 10, which is after the BUTTONDELAY = 10, the *button*-flag goes to 1 and also the *rollflag* which controls the „click-sound“goes to 1. The usage of the *rollflag* is no loger discussed here. Also *light* goes to 1 meaning the LEDs are switched on.

```
|    1716 |0 1 0 0 1 | 6 _  |   1 245 | 1110111b |
|    1718 |0 1 0 0 1 | 6 _  |   1 247 | 1110111b |
|    1720 |0 1 0 0 1 | 6 _  |   1 247 | 1110111b |
|    1722 |0 1 0 0 1 | 6 _  |   1 249 | 1110111b |
|    1724 |0 1 0 0 1 | 6 _  |   1 249 | 1110111b |
|    1726 |0 1 0 0 1 | 6 _  |   1 251 | 1110111b |
|    1728 |0 1 0 0 1 | 6 _  |   1 251 | 1110111b |
|    1730 |0 1 0 0 1 | 6 _  |   1 253 | 1110111b |
|    1732 |0 1 0 0 1 | 6 _  |   1 253 | 1110111b |
|    1734 |0 1 0 0 1 | 6 _  |   1 255 | 1110111b |
|    1736 |0 1 0 0 1 | 6 _  |   1 255 | 1110111b |
|    1738 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1740 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1742 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1744 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1746 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1748 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1750 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1752 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1754 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1756 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1758 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1760 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1762 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1764 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1766 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1768 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1770 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1772 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1774 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1776 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1778 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1780 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1782 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1784 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1786 |0 0 1 0 1 | 1 a  |   0   0 | 0001000b |
|    1788 |0 0 1 0 1 | 1 b  |   0   0 | 0001000b |
|    1790 |0 0 1 0 1 | 1 b  |   0   0 | 0001000b |
|    1792 |0 0 1 0 1 | 1 b  |   0   0 | 0001000b |
|    1794 |0 0 1 0 1 | 1 b  |   0   0 | 0001000b |
|    1796 |0 0 1 0 1 | 1 b  |   0   0 | 0001000b |
```

Figure19: Results of simulation (from timestep 1716 to 1796), showing melody output, selection from scc_trace.dat.

Next column shows the *points* from the dice-counter, starting to run up. NCO- preformance is shown in the 2 next columns, with the initial start frequency *Nfreq* set to 18 in the beginning. The *accum*-value of the NCO starting with 0 and incrementing by the value of *Nfreq*, is 18 in the beginning. At timestamp 34 the NCO is reset and a *dice_clk*-event is generated, causing the dice counter to count to *points* = 2 etc.The last two columns show the bits of the LEDs, converted to bitvector from *points*. The overall *scc_traces.dat* – file is 1250 lines long, there shall be only a one second slice shown

This slice taken from the output file shows the situation, when rollout is finished, ncoflg is set (2nd column) and the melody, which is shown here by the character „a", later „b" etc. is put to the output.

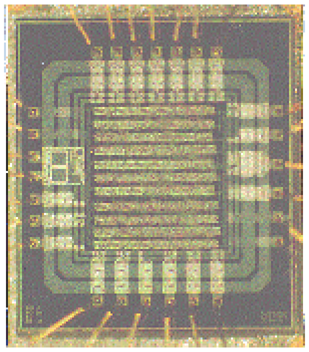
Figure 20: Standard-cell ASIC of the "Sounding Dice", taken from the authors laboratory [FHO]

The trace-file allows us to study or optimize the behavior of the whole system, which is the main task of system simulation. On the other side, it is clear from this example that there have to be more easy to use tools for monitoring and stimulating the simulation. There have to be some interactivity, because it is difficult to decide, if the melody „*a,b,c*" sounds better than „*f,g,h*". You really have to hear it! The same is with the timing of roll-out, time-out etc.. Everywhere a human interface is involved.

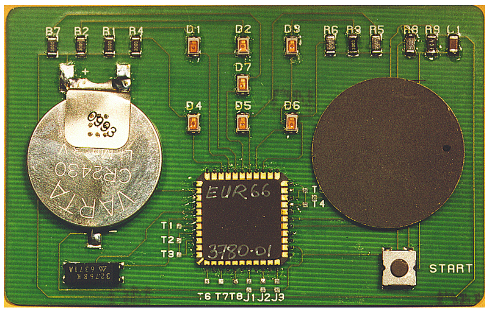

Figure 21: Chip-card size printed circuit board with the "Sounding Dice"-ASIC (in the middle), a speaker (right) and a battery (left), taken from the authors laboratory [FHO].

The dice-circuit may be further refined and synthezised to VHDL and placed in a standard-cell design ASIC (see Figure 20). The ASIC may then be placed on a printed circuit board with a battery, a ceramic speaker and a button (see Fig 21). Both pictures are taken from the existing design [FHO] and [Jan]. The „Sounding Dice"-design was **awarded as the second best student design** on the EUROCHIP-conference 1994 in Dresden, Germany.

Conclusion

This report demonstrates the easy to understand example of a sounding and rolling dice main programming constructs of the SpecC – language. The behaviors of the example are discussed and detailed. Advice to organize and setup the program are given. The performance of the main features of the „sounding dice" are demonstrated in a text-file, generated by the program as a monitor. The system may be optimized and refined in further design steps. This report is intended as a tutorial to the language. It has shown how to model a system, to describe it in the SpecC-language and to simulate it. The complete files are given in the appendix.

References

[Gaj1] *Daniel D. Gajski, Gaurav Aggarwal, En-Shou Chang, Rainer Domer et.alt.*: „Methodology for Co-Design of Embedded Systems". UC Irvine, Technical Report ICS-TR-98-02, March 1998.

[Gaj2] *Daniel D. Gajski, Jianwen Zhu, et.alt.*:"SpecC Specification language and Methodology", Kluwer Academic Publishers, Boston 2000.

[Gaj3] *Daniel D. Gajski,Frank Vahid, Sanjiv Narayan, Jie Gong*: „Specification and Design of Embedded Systems". Prentice Hall, Inc., Englewood Cliffs, 1994

[STOC] http://www.specc.gr.jp/eng/index.htm

[FHO] http://www.asic.fh-offenburg.de

[Jan] *Dirk Jansen et. alt.*: „Handbuch der Electronic-Design-Automation", Hanser Verlag, Leipzig, coming out 10/ 2000.

Appendix

```
#if !dice_header
#define dice_header dice_header.h

//Macros and Constants

#define        MAXSIMTIME     2500
#define        BUTTONDNTIME   500
#define        STARTDELAY     10
#define        CYCLETIME      2
#define        TIMEOUT        2200

// utility function declarations

bool SimActiv();

// file declaration

FILE           *pTrace;

// globals

event          EndofSim;


//////////////// behavior
declarations/////////////////////////////////////
// testbench
behavior Button(out bool Bstate);
behavior ClockGen(out event clk);
behavior Monitor(in event Dclk, in bool enable,in
bool rollflg, in bool ncoflg,in bool toflg,
                 in bool lightflg, in int state,
in char spk,
                 in int freq, in int accum,in
bit[6:0] Ls);
// Dice DUT

behavior Dice(event clock_int,in bool
button_flg,out bool nco_flg,
             out bool roll_flg, out bool
lights,out bool time_out_flg,
             out int Points,out char speaker,out
bit[6:0] LEDs,
            out int Nfreq,out int accum);

behavior Cntrl(in bool En,out event DiceClk,in
event clk,
               inout bool rollflg,out bool ncoflg,
               out int Nfreq, out int accum,out
bool time_out);

behavior fsm_dice(in event clk, inout int state);

behavior Main_fsm(in event clock_int,in bool
button,in bool nco_null,
                 in bool time_out,in int pts,
                  out bool roll_dice_flg,out bool
lights_on,
                  out char speaker);
behavior Stop  (in event clk,in bool but,out bool
lights);
behavior RollDice(in event clk,out bool
roll_dice_flg);
behavior Melody(in int pts,out char speaker);
behavior Sound(in char ch,out char spk );
behavior WaitOnTime(in event clk);
behavior LightsOff(inout bool lght);
behavior Display(in event clk, in int Pts, out
bit[6:0] LDs);


#endif


///////////// utility functions, body
////////////////////////////////////

bool SimActiv()
{
 return ((int)now()<MAXSIMTIME);
}
///////////////////// Button
/////////////////////////////////////////////

behavior Button(out bool Bstate)      // generates
timing-pattern
  {    bool ButtonDn=true;
       int Zeit = BUTTONDNTIME;

  void main(void)
     {
       waitfor(STARTDELAY);
       Bstate=ButtonDn;
       waitfor(Zeit);
       Bstate=!ButtonDn;

      }
  };


behavior ClockGen(out event clk)
{
void main(void)
 {
   do{
        waitfor(CYCLETIME);
        //Cycletime defines time
        notify (clk);                         //
between clock-events
     }while(SimActiv());
        notify EndofSim;                      //
main end of simulation, global
 return;
 }
};
//////////////////// Monitor
/////////////////////////////////////////

behavior Monitor(in event Dclk, in bool enable,in
bool rollflg, in bool ncoflg,in bool toflg,
                 in bool lightflg, in int state,
in char spk,
                 in int freq, in int accum,in
bit[6:0] Ls)
{

// utility function to print bitvectors

  const char *Bit2String(bit[6:0] b)
  {
    static char s[8];
    int i;

  for(i=0;i<7;i++)
    s[i]= (b[6-i] ?'1':'0');
    s[7]='\0';
  return(s);
 }
```

```
void main(void)
 {
  do
  {
  wait(Dclk,EndofSim);               //
synchronization




  fprintf(pTrace,"| %5d |%d %d %d %d %d | %d %c
|%3u %3u | %sb |\n",

(int)now(),(int)enable,(int)rollflg,(int)ncoflg,(i
nt)toflg,
              (int)lightflg, state, spk,
freq,accum,Bit2String(Ls));

                              // write
portvariables to disc
  } while (SimActiv());
  return;
 }

};



//////////////////////////////////////////////////
////////////////////
// SpecC-Program of a dice with slow run out
dice2.sc
// Version No V 1.0
// 20.4.2000 Dirk Jansen
// simple testbench included
//////////////////////////////////////////////////
///////////////////
#include <stdio.h>
#include <sim.h>

#include "dice_header.h"
#include "dice_body.sc"
#include "monitor.sc"




//////////////////////////////////////////////////
///////////////////
//                      MAIN BEHAVIOR
//////////////////////////////////////////////////
///////////////////

behavior Main(void)                  //testbench
{
bool           roll_flg=      false,
               button_flg=    false,
               lights=        false,
               time_out_flg=  false;
bool           nco_flg=       false;

char           speaker=       '_';
int            Displ_State=   7;
int            Points=        9;
int            Nfreq=         20,
               accum=         254;

bit[6:0]       LEDs=          0000000B;
event          eclock;

//instances

Button         Butt1(button_flg);
ClockGen       ClockGen1(eclock);
Monitor
       Monitor1(eclock,button_flg,roll_flg,nco_flg
,time_out_flg,lights,
                      Points,speaker,
Nfreq,accum,LEDs);
Dice
       Dice1(eclock,button_flg,nco_flg,roll_flg,li
ghts,time_out_flg,

Points,speaker,LEDs,Nfreq,accum);

// main

void main(void)
{
       printf("run simulation\n");
       pTrace=fopen("scc_traces.dat","w");

       par{
              Butt1.main();
              ClockGen1.main();
              Monitor1.main();

              Dice1.main();
          }

       printf("\n\nfile scc_traces.dat written to
disc!\n\n\n");
       fclose(pTrace);
}

};


// /////////////////////Sub-
Behaviors/////////////////////////////////

behavior Dice(event clock_int,in bool
button_flg,out bool nco_flg,
             out bool roll_flg, out bool
lights,out bool time_out_flg,
             out int Points, out char speaker,out
bit[6:0] LEDs,
             out int Nfreq,out int accum)
{
  event DiceClock;


 // Instances of used behaviors


 Main_fsm
       Main_fsm1(clock_int,button_flg,nco_flg,time
_out_flg,Points,
                         roll_flg,lights,
speaker);

 Cntrl
       Cntrl1(button_flg,DiceClock,clock_int,
                roll_flg,nco_flg, Nfreq, accum,
time_out_flg);

 fsm_dice      fsm_dice1(DiceClock, Points);
 Display       Display1(clock_int,Points,LEDs);




void main(void)
  {


// concurrent operation

   par  {
          Cntrl1.main();
          fsm_dice1.main();
         Display1.main();
         Main_fsm1.main();
        }


   return;
  }
};



behavior Cntrl(in bool En ,out event DiceClk,in
event clk,
               inout bool rollflg,out bool ncoflg,
              out int Nfreq, out int accum,out
bool time_out)


  { int t=0;                          //local
timecount

   void main()
   {

      do{

      wait clk,EndofSim ;                     //
wait on clock-event

      accum+=Nfreq;
        // basic NCO

      if (accum > 255) {accum=0;
                     if (En) Nfreq+=2;
                     else Nfreq-=2 ;
        // freq up or down

                     if (Nfreq > 31) Nfreq=31;
                     if (Nfreq <= 0) {Nfreq=0;
        //limiting
                                    ncoflg=true;

        rollflg=false; // end of rolling out
                                          }
                      notify(DiceClk);
              //generate NCO event
                      };



      t++; if (t*CYCLETIME>=TIMEOUT)
time_out=true; //Timeout

      }while(SimActiv());
```

```
       return;
     }
   };

behavior Main_fsm(in event clock_int,in bool
button,in bool nco_null,
                  in bool time_out,in int pts,
                  out bool roll_dice_flg,out bool
lights_on,
                  out char speaker)
{
Stop           Stop1(clock_int,button,lights_on);
RollDice       RollDice1(clock_int,roll_dice_flg);
Melody         Melody1(pts,speaker);
WaitOnTime     WaitOnTime1(clock_int);
LightsOff      LightsOff1(lights_on);


void main(void)
{
       fsm{

              Stop1          : {if (button) goto
RollDice1;goto Stop1;}
              RollDice1      : {if (nco_null)
goto Melody1;goto RollDice1;}
              Melody1        : {goto
WaitOnTime1;}
              WaitOnTime1    : {if (time_out)
goto LightsOff1; goto WaitOnTime1;}
              LightsOff1     : {break;}

          };



return;
}
};  //end behavior main_fsm

behavior Stop  (in event clk,in bool but, out bool
lights)
{
void main(void)
 {wait clk,EndofSim;
  if (but) lights=true; else lights=false;
  printf("s");
 return;
 }
};

behavior RollDice(in event Dclk,out bool
roll_dice_flg)
{

void main(void)
        {
              roll_dice_flg=true;
              printf(".");
               wait Dclk;

 return;}
};

behavior Melody(in int pts,out char speaker)
{
       char
a='a',b='b',c='c',d='d',e='e',f='f',g='g',h='h';

  Sound Sound_A(a,speaker),
       Sound_B(b,speaker),
       Sound_C(c,speaker),
       Sound_D(d,speaker),
       Sound_E(e,speaker),
       Sound_F(f,speaker),
       Sound_G(g,speaker),
       Sound_H(h,speaker);

void main(void)
  {
   waitfor(1);                //setup time
   printf("\nmelody out\n");
   switch(pts) {

    case 1     :
Sound_A.main();Sound_B.main();Sound_C.main();break
;
    case 2     :
Sound_D.main();Sound_E.main();break;
    case 3     :
Sound_D.main();Sound_E.main();break;
    case 4     :
Sound_D.main();Sound_E.main();break;
    case 5     :
Sound_D.main();Sound_E.main();break;
    case 6     :
Sound_F.main();Sound_G.main();Sound_H.main();break
;

    default    : break;
               };
  return;
  }
};

behavior Sound(in char ch,out char spk )
{
void main(void)
  {

  spk=ch;
  waitfor(50);          // sound ch for 50 units
  spk='_';

return;
  }
};

behavior WaitOnTime(in event clk)
{
void main(void)
 {
  wait clk,EndofSim;
  printf(",");
 return;
 }
};


behavior LightsOff(inout bool lght)
{
void main(void)
 {
  lght=false;

  return;
 }
};


behavior fsm_dice(in event Dclk, inout int state)
{                                     // main dice
behavior,non synchr

void main(void)
 {    do
    {
      wait(Dclk,EndofSim);
      state++; if (state > 6) state=1;     // on
trigger


    }while(SimActiv());
  return;
 }
};

behavior Display(in event Dclk, in int Pts, out
bit[6:0] LDs)
{

void main(void)
 {
 do{
  wait Dclk,EndofSim;

  switch (Pts)
   {
        case 1: LDs=0001000B;break;
        case 2: LDs=1000001B;break;
        case 3: LDs=1001001B;break;
        case 4: LDs=1010101B;break;
        case 5: LDs=1011101B;break;
        case 6: LDs=1110111B;break;

        default: LDs=0000000B;break;
   };

   }while (SimActiv());
  return;
 }
};
```

```
|      2 |0 0 0 0 0 | 9 _  | 18   0 | 0000000b |
|      4 |0 0 0 0 0 | 1 _  | 18   0 | 0000000b |
|      6 |0 0 0 0 0 | 1 _  | 18  36 | 0001000b |
|      8 |0 0 0 0 0 | 1 _  | 18  36 | 0001000b |
|     10 |1 1 0 0 1 | 1 _  | 18  72 | 0001000b |
|     12 |1 1 0 0 1 | 1 _  | 18  72 | 0001000b |
|     14 |1 1 0 0 1 | 1 _  | 18 108 | 0001000b |
|     16 |1 1 0 0 1 | 1 _  | 18 108 | 0001000b |
|     18 |1 1 0 0 1 | 1 _  | 18 144 | 0001000b |
|     20 |1 1 0 0 1 | 1 _  | 18 144 | 0001000b |
|     22 |1 1 0 0 1 | 1 _  | 18 180 | 0001000b |
|     24 |1 1 0 0 1 | 1 _  | 18 180 | 0001000b |
|     26 |1 1 0 0 1 | 1 _  | 18 216 | 0001000b |
|     28 |1 1 0 0 1 | 1 _  | 18 216 | 0001000b |
|     30 |1 1 0 0 1 | 1 _  | 18 252 | 0001000b |
|     32 |1 1 0 0 1 | 1 _  | 18 252 | 0001000b |
|     34 |1 1 0 0 1 | 2 _  | 20  20 | 1000001b |
|     36 |1 1 0 0 1 | 2 _  | 20  20 | 1000001b |
|     38 |1 1 0 0 1 | 2 _  | 20  60 | 1000001b |
|     40 |1 1 0 0 1 | 2 _  | 20  60 | 1000001b |
|     42 |1 1 0 0 1 | 2 _  | 20 100 | 1000001b |
|     44 |1 1 0 0 1 | 2 _  | 20 100 | 1000001b |
|     46 |1 1 0 0 1 | 2 _  | 20 140 | 1000001b |
|     48 |1 1 0 0 1 | 2 _  | 20 140 | 1000001b |
|     50 |1 1 0 0 1 | 2 _  | 20 180 | 1000001b |
|     52 |1 1 0 0 1 | 2 _  | 20 180 | 1000001b |
|     54 |1 1 0 0 1 | 2 _  | 20 220 | 1000001b |
|     56 |1 1 0 0 1 | 2 _  | 20 220 | 1000001b |
|     58 |1 1 0 0 1 | 2 _  | 22   0 | 1000001b |
|     60 |1 1 0 0 1 | 3 _  | 22   0 | 1000001b |
|     62 |1 1 0 0 1 | 3 _  | 22  44 | 1001001b |
|     64 |1 1 0 0 1 | 3 _  | 22  44 | 1001001b |
|     66 |1 1 0 0 1 | 3 _  | 22  88 | 1001001b |
|     68 |1 1 0 0 1 | 3 _  | 22  88 | 1001001b |
|     70 |1 1 0 0 1 | 3 _  | 22 132 | 1001001b |
|     72 |1 1 0 0 1 | 3 _  | 22 132 | 1001001b |
|     74 |1 1 0 0 1 | 3 _  | 22 176 | 1001001b |
|     76 |1 1 0 0 1 | 3 _  | 22 176 | 1001001b |
|     78 |1 1 0 0 1 | 3 _  | 22 220 | 1001001b |
|     80 |1 1 0 0 1 | 3 _  | 22 220 | 1001001b |
|     82 |1 1 0 0 1 | 3 _  | 24   0 | 1001001b |
|     84 |1 1 0 0 1 | 4 _  | 24   0 | 1001001b |
|     86 |1 1 0 0 1 | 4 _  | 24  48 | 1010101b |
|     88 |1 1 0 0 1 | 4 _  | 24  48 | 1010101b |
|     90 |1 1 0 0 1 | 4 _  | 24  96 | 1010101b |
|     92 |1 1 0 0 1 | 4 _  | 24  96 | 1010101b |
|     94 |1 1 0 0 1 | 4 _  | 24 144 | 1010101b |
|     96 |1 1 0 0 1 | 4 _  | 24 144 | 1010101b |
|     98 |1 1 0 0 1 | 4 _  | 24 192 | 1010101b |
|    100 |1 1 0 0 1 | 4 _  | 24 192 | 1010101b |
|    102 |1 1 0 0 1 | 4 _  | 24 240 | 1010101b |
|    104 |1 1 0 0 1 | 4 _  | 24 240 | 1010101b |
```

……