Appendix 10
PSpice to Alecsis converter

A10.1. Why PSpice2Alecsis conversion?

PSpice is an integrated mixed-mode (hybrid) simulator of the electronic circuit. Therefore, it can be used for analogue, digital and hybrid circuits. The input language of the PSpice simulation is based on the input language of SPICE simulator complemented by the mechanisms for digital circuit description.

PSpice2Alecsis is a program that converts PSpice description of hybrid electronic circuit into the equivalent description in AleC++. The execution version of PSpice2Alecsis program is called `p2a`.

There are three main reasons for development of this program:

- Providing a possibility to compare simulation results of the same circuit obtained by PSpice and Alecsis.
- Possibility to use big number of already developed and available PSpice libraries in Alecsis.

- **PSpice2Alecsis enables PSpice/Alecsis co-simulation**. It means that one part of the circuit is described by using components and commands of the input language of PSpice, while the other part of the circuit is described by using AleC++ constructions and principles.

Why is PSpice/Alecsis co-simulation needed? PSpice is a language for electronic circuit description, but without a possibility to describe new models. AleC++ possesses mechanisms for modelling of new components. Besides, systems that are not of electrical nature can be described in AleC++ and simulated in Alecsis, such as micromechanical systems, computer networks, neural networks etc. Therefore, if a complex system includes standard electronic components, that are well characterized in PSpice, but also components that cannot be found in PSpice, PSpice/Alecsis co-simulation can be the best solution.

Note: PSpice2Alecsis is under active development. Therefore, in the moment when you are reading this, new features may be added already. Besides, some of the limitations are due to Alecsis, and not the `p2a` converter. However, Alecsis is going to be improved to allow better compatibility with PSpice, too.

A10.2. Use of PSpice2Alecsis

The program is invoked from the command line as:

```
p2a file_name.ext
```

Where '`.ext`' is extension for PSpice input file (usually '`.cir`'.). This extension has to be specified when invoking `p2a`, if it exists in the name of the PSpice input file. As a result, two output files are created:

```
file_name.ac
file_name.cmm
```

AleC++ description is stored in `file_name.ac`, while comments on the conversion are stored `file_name.cmm`. These comments include PSpice command list from `file_name.ext`, which do not have its equivalent command (construction) in AleC++, nor can they be realised in AleC++ in some other way, as well as some commands whose conversion has not been implemented yet. In some specific situations, it can also contain some instructions for PSpice description, which may lead to better conversion to AleC++. If the conversion is successful, file for comments `file_name.cmm` is empty.

In `file_name.ext`, commands for including other files (or their parts) can be used:

```
.inc "inc_file_name.ext"
.lib "lib_file_name.ext"
```

can be used. In such case, conversion will be performed for these files, too. Therefore, files `inc_file_name.ac` , `inc_file_name.cmm`, `lib_file_file.ac`, `lib_file_name.cmm`, are also created.

In the beginning of each file that is a product of `p2a` program, following information is given: version of the program, date and time of current file generation, input file name, output file name with extension '`.ac`' and output file name with extension '`.cmm`'.

One of the major parts of PSpice2Alecsis converter is the parser, which identifies input language commands of PSpice simulator. Depending on which command is identified, an equivalent AleC++ command or language construction is generated. Two groups of commands could be identified in input language of PSpice simulator:

- commands describing circuit topology (components and their connections);
- commands for simulation control.

Commands for circuit topology description start with a letter (the first letter of particular component name), whereas commands for flow control analysis start with a point '`.`'.

PSpice identifies 22 different type of elements. They are not declared, since the first letter in the component name denotes the type of the component. There are three types of components:

- ♦ analogue,
- ♦ digital (whose name starts with `U`)
- ♦ A/D and D/A converters for hybrid circuit (`N`, `O`)

A10.2.1. Analogue circuit conversion

PSpice2Alecsis converter reads the description in PSpice and creates description in AleC++. Alecsis equivalents (AleC++ modules) of PSpice analogue components are already prepared in the library, which is named `PSpicecomp`. This library is already compiled, and is automatically included in resulting file using `library` command. Therefore, in each AleC++ source `file_name.ac` that is created by `p2a`, following commands are found:

```
#include "RLC_model.h"
#include "PSpicecomp.h"
library PSpicecomp;
```

It is supposed that these files are in directory visible to Alecsis compiler (see *Appendix 1* on Alecsis installation and usage). Files with extension '`.h`' are appropriate header files, containing necessary declarations for definitions stored in the library.

In `PSpicecomp` library, there are the following modules, which correspond to certain components from input language of PSpice simulator:

`resistorPSp`	resistor (**R***name* component)
`capacitorPSp`	capacitor (**C***name* component)
`inductorPSp`	inductor (**L***name* component)
`vsingen`	voltage source for a sinusoidal waveform which returns branch current on its name

`vsingenv`	voltage source for a sinusoidal waveform
`vexpgen`	voltage source for an exponential waveform which returns branch current on its name
`vexpgenv`	voltage source for an exponential waveform
`vsffmgen`	voltage source for a frequency-modulated waveform which returns branch current on its name
`vsffmgenv`	voltage source for a frequency -modulated waveform
`vpulgen`	voltage source for a pulse waveform which returns branch current on its name
`vpulgenv`	voltage source for a pulse waveform
`csingen`	current source for a sinusoidal waveform which returns (branch) current on its name
`csingen0`	current source for a sinusoidal waveform
`cexpgen`	current source for an exponential waveform which returns (branch) current on its name
`cexpgen0`	current source for an exponential waveform
`csffmgen`	current source for a frequency-modulated waveform which returns (branch) current on its name
`csffmgen0`	current source for a frequency-modulated waveform
`cpulgen`	current source for a pulse waveform which returns (branch) current in its name
`cpulgen0`	current source for a pulse waveform
`ccswPSp`	current-controlled switch (**`W`***name* component)
`vcswPSp`	voltage-controlled switch (**`S`***name* component)

In case there are more files containing circuit descriptions, these files are included in the resulting AleC++ file containing `root module`.

A10.2.2. Digital circuit conversion

All digital components are sorted as one component type in PSpice, and their name starts with `U`. Models of digital components that are implemented in PSPICE are prepared in libraries. There are three libraries, and two files containing necessary declarations and definitions:

`PSpicedef.h`	header file that contains used structure declarations, global data, functions and modules
`PSpicefun`	library that contains functions used in modelling (late functions, resolution functions etc.)
`PSpicestr`	library that contains digital `module` definition
`PSpicemod`	library that contains redefined model cards

`.alecrc` header file that contains certain variable definitions

These five files ought to be visible by Alecsis, and are included in AleC++ file created by `p2a`:

```
#include "PSpicedef.h"
#include ".alecrc"
library PSpicemod, PSpicesr, PSpicefun;
```

In case there are more files containing circuit descriptions, these files are included in the resulting AleC++ file containing `root module`.

A10.2.3. Hybrid circuit conversion

Hybrid simulation means that a described circuit contains both analogue and digital components. Due to different nature of mechanisms for analogue and digital simulation, analogue and digital domains need to be divided by A/D and D/A converter insertion. Converter insertion is executed by PSpice simulator itself (the same is valid for Alecsis, too). It can be also done manually, by the user. A converter that is inserted is a subcircuit, which contains **O***name* or **N***name* component for A/D or D/A converter, respectively.

A10.3. Conversion of PSpice commands with examples

A10.3.1. Conversion of components

C*name* command:

This command is used to specify a capacitor in the input language of PSpice simulator.

Example 1:

In PSpice input language, capacitor named `c1` is defined using:

```
c1   11 12 3.498E-12 ;
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
root ...// or module
{
  // declartion part
  capacitor c1;
  ...
  // structure part
```

```
  c1 (11, 12) 3.498e-12;
  ...
}
```

Example 2:

In PSpice input language, capacitor named `c14` and its model card `capmodel` are defined as:

```
c14 21 22  capmodel 300nF IC = 2.0V
.model capmodel cap (vc1=1.0 vc2=2.0 tc1=3.0 tc2=4.0)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "pspcomp.h"
library "pspcomp";

//model card for new component capacitorPSp
model CPSp::capmodel{
   vc1=1.0;
   vc2=2.0;
   tc1=3.0;
   tc2=4.0;
}
root ... // or module
{
  // declaration part
  capacitorPSp c14;
  ...
  // structure part
  c14 (20, 21)  { model = cmodel; value = 300; }
  ...
}
```

R*name* command:

This command is used to specify a resistor in input language of PSpice simulator.

Example 1:

In PSpice input language, resistor named `r705` is defined using:

```
r705 c1 c2  55
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
root ... // or module
{
  // declaration part
  resistor r705;
  ...
  // structure part
```

```
  r705 (c1, c2)  { value = 55; }
  ...
}
```

Example 2:

In PSpice input language, resistor named `r604` and its `TC` parameter are defined using:

```
r604 13 10  24.87 TC=10,1
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "pspcomp.h"
library "pspcomp";

//model card for new component resistorPSp
model RPSp::TCr604{
   tc1=10;
   tc2=1;
}

root ... // or module
{
  // declaration part
  resistorPSp r603;
  ...
  // structure part
  r603 (13, 10)  { model = resmod; value = 24.87; }
  ...
}
```

Example 3:

In PSpice input language an resistor named `r603`, its `TC` parameter and its model card `resmod` are defined as:

```
r603 13 10 resmod  24.87 TC=10,1
* command for specifying model card
model resmod res (r=1.5 tc1=0.02 tc2=0.005
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
include "pspcomp.h"
library "pspcomp";

// model card for new component resistorPSp
model RPSp::resmod{
   r=1.5;
   tc1=0.02;
   tc2=0.005;
}

root ... // or module
{
  // declaration part
```

```
  resistorPSp r603;
  ...
  // structure part
  r603 (13, 10)  { model = resmod; value = 24.87; }
}
```

D*name* command:

This command is used to specify a diode in input language of PSpice simulator.

Example:

In PSpice input language diode named `d12` and its model card are defined as:

```
d12 c31 c32 dmodel AREA = 20.9
.model dmodel D (Is=1E-13 Vj = 0.7)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
spice {
.model dmodel d ( is=1e-13 vj=0.7 ) }
root ... // or module
{
  // declaration part
  diode d12;
  ...
  // structure part
  d12 (c31, c32) { model = dmodel; area = 20.9; }
  ...
}
```

G*name* command with specification `poly:`

In the input language of PSpice simulator, this command is used to specify a voltage-controlled current source (with polynomial dependence).

Example:

In PSpice input language, an voltage-controlled current source named `g983` which has three pairs of nodes for voltage control, and eight coefficients, is defined as:

```
g983 983 0 poly(3)(1 2)(3 4)(5 6) 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
module gpoly3 (node i,j, n1, n2, n3, n4, n5, n6)
{
   cgen genc;
   genc (i,j);
```

```
   action per_moment (double p0=0.0, double p1=0.0, double p2=0.0,
double p3=0.0, double p4=0.0, double p5=0.0, double p6=0.0, double
p7=0.0, double p8=0.0, double p9=0.0)

   {
      genc->value = p0 + p1*(n1-n2) + p2*(n3-n4) + p3*(n5-n6) +
p4*(n1-n2)*(n1-n2) + p5*(n1-n2)*(n3-n4) + p6*(n1-n2)*(n5-n6) +
p7*(n3-n4)*(n3-n4) + p8*(n3-n4)*(n5-n6) + p9*(n5-n6)*(n5-n6);
   }
}
root ...{ // or module
  // declaration part
  gpoly3 g983;
  ...
  // structure part
  g983 (983, 0, 1, 2, 3, 4, 5, 6) {p0=1.0; p1=2.0; p2=3.0; p3=4.0;
  p4=5.0; p5=6.0; p6=7.0; }
}
```

Note 1:

`Poly` specification of E*name* and G*name* commands has two syntax options:

with brackets:

```
G983 983 0 poly(3) (1 2)(3 4)(5 6)  1.0 2.0 3.0 4.0 5.0 6.0 7.0
```

without brackets:

```
G983 983 0 poly(3)  1 2  3 4  5 6   1.0 2.0 3.0 4.0 5.0 6.0 7.0
```

Note 2:

Value of the controlling variable, having polynomial dependence on V_1, V_2, …, V_n, is defined :

$$\begin{aligned} V_{out}(V_1,V_2,\ldots,V_n) = {} & P_0 + \\ & P_1 * V_1 + P_2 * V_2 + \ldots + P_n * V_n + \\ & P_{n+1} * V_1^{\,2} + P_{n+2} * V_1 * V_2 + \ldots + P_{n+n} * V_1 * V_n + \\ & P_{2n+1} * V_2 * V_2 + P_{2n+2} * V_2 * V_3 + \ldots + P_{2n+n-1} * V_2 * V_n + \\ & \ldots + \\ & P_{n!/(2(n-2)!+2n} * V_n * V_n \end{aligned}$$

Where: $P_0, P_1, P_2, \ldots$ are polynomial coefficients.

E*name* command with specification `value`:

This command is used in input language of PSpice simulator to specify a voltage source, whose value is controlled by a function.

Example:

In PSpice input language, a function that controls voltage source named `esum` is defined using following command:

```
esum 4  0  value = {v(2)*v(6)*i(vr)*v(5)*i(vg)}
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
module  vvalue1 (node i,j; node n1; node n2; current c1; node n3;
current c2)
{
   vgen genv;
   genv (i,j);

   action per_moment ()
   {
      genv->value =  n1*n2*c1*n3*c2;
   }
}

root ... // or module
{
  // declaration part
  vvalue1 esum;
  ...
  // structure part
  esum (4, 0, 2, 6, vr, 5, vg);
  ...
}
```

E*name* command with specification `table`:

This command is used to specify a voltage source, whose control is given as a table.

Example:

In PSpice input language, voltage source named `erele` whose value is controlled by a function, is defined with the following command:

```
erele  2  0  table {V(1)} = (2, -1) (2.01, 1)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
module current vtable1 (node i,j; node n1)
{
   vgen vtable1;
   return vtable1 (i,j) 0.0;

   action per_moment ()
   {
     int loop;
     int i, j;
```

```
        double izlaz, x1, x2, y1, y2;
        static const double table[2][2] = {
            2, -1,
            2.01, 1,
        };

        loop = 1;
        i = 0;
        while (loop == 1)
        {
           if ( n1 > table[i][0])
           {
             if (i < 1)
               i++;
             else
             {
               loop = 0;
               izlaz = table[1][1];
             }
          }
           else
           {
             loop = 0;
             if (i != 0)
             {
                x1 = table[i-1][0];
                x2 = table[i][0];
                y1 = table[i-1][1];
                y2 = table[i][1];
                izlaz = (( n1 - x1)/(x2 - x1))*(y2 - y1) + y1;
             }
             else if( n1 < table[i][0])
                izlaz = table[i][1];
           }
        }
        vtable1->value = izlaz;

     }
  }

  root ... // or module
  {
    // declaration part
    vtable1 erele;
    ...
    // structure part
    erele (2, 0, 1);
    ...
  }
```

V*name* (I*name*) command

V*name* command is used to specify an independent voltage source in the input language of PSpice simulator. I*name* command is used to specify an independent voltage source in the input language of PSpice simulator.

Example:

In PSpice input language an independent voltage source with sinusoidal waveform named `vsin is` defined with the following command:

```
vsin 10 5 sin(2 2 5Hz 1 10 30)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "pspcomp.h"
library "pspcomp";
root ... // or module
{
  // declaration part
  vsingen vsin;
  ...
  // structure part
 vsin (10,5) { voff=2.0v; vampl=4.0v; freq=50hz; td=1msec; df=10;
 phase=30; }
  ...
}
```

X*name* command:

In the input language of PSpice simulator, this command is used to specify call of a subcircuit.

Example:

In PSpice input language, call of subcircuit named `xcomp` is defined with using following command (called subcircuit has five nodes (`0`, `3`, `nvdd`, `nvss`, `4`) and subcircuit's definition has name `tl064/ti`):

```
xcomp  0  3  nvdd  nvss  4  tl064/ti
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
root ... // or module
{
  // declaration part
  tl064_slash_ti xcomp;
  ...
  // structure part
  xcomp (0, 3, nvdd, nvss, 4);
}
```

M*name* command:

This command is used to specify a MOSFET in the input language of PSpice simulator.

Example:

In PSpice input language, MOSFET named `mmosfet` and its model card `modmos` are defined as:

```
mmosfet  0  2  100  100  modmos L=33u W=12u
+ AD=288p AS=288p PD=60u PS=60u NRD=14 NRS=24 NRG=10
.model modmos nmos ( lambda=2 )
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
spice {
.model modmos nmos ( lambda=2 ) }
root ... // or module
{
  // declaration part
  mosfet mmosfet;
  ...
  // structure part
  mmosfet (0, 2, 100, 100) {model = nweak; l=33u; w=12u; ad=288p;
  as=288p; pd=60u; ps=60u; nrd=14; nrs=24; nrg=10;}
  ...
}
```

Q*name* command:

This command is used to specify a BJT in the input language of PSpice simulator.

Example:

In PSpice input language, BJT named `mmosfet` and its model card `modbjt` are defined as:

```
qbjt  a  b  c  model605  area=24.87
.model modbjt npn (Is=17.01E-12 Bf=110 Vje=0.85)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
spice {
.model modbjt npn ( is=17.01e-12 bf=110 vje=0.85 ) }
root ... // or module
{
  // declaration part
  bjt qbjt;
  ...
  // structure part
  qbjt (a, b, c) { model = model604; area = 24.87;  }
  ...
}
```

J*name* command:

This command is used to specify a JFET in the input language of PSpice simulator.

Example:

In PSpice input language, JFET named `jjfet` and its model card `modjfet` are defined as:

```
Jjfet  a  b  c  modjfet  area=24.87
.model modjfet pjf(Is=17.01E-12 Beta=110.5E-6 Vto=-1)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
spice {
.model modjfet pjf ( is=17.01e-12 beta=110.5e-6 vto=1 ) }
root ... // or module
{
  // declaration part
  junctionfet jjfet;
  ...
  // structure part
  jjfet (a, b, c) { model = model603; area = 24.87; }
}
```

S*name* command:

This command is used to specify a voltage-controlled switch in the input language of PSpice simulator.

Example:

In PSpice input language, voltage controlled switch named `svcs` and its model card `sw1mod` are defined as:

```
svcs 13 17 2 0 sw1mod
.model sw1mod vswitch (ron=1.0 roff=1e+6 von=1.0 voff=0.0)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include <alec.h>
#include "pspcomp.h"
library "pspcomp";

// Model card for new component vcswPSp.Coressponds to the .model
// in example.
model vSWPSp::sw1mod{
   ron=1.0;
   roff=1e+6;
   von=1.0;
   voff=0.0;
}

root ... // or module
{
  // declaration part
  vcswPSp svcs;
```

```
  // structure part
  ...
  svcs (13, 17, 2, 0)  model = sw1mod;
  ...
}
```

W*name* command:

This command is used to specify a current-controlled switch in the input language of PSpice simulator.

Example:

In PSpice input language, current-controlled switch named `wccs` and its model card `wmod` are defined as:

```
wccs 13 17 vc wmod
.model wmod iswitch (ron=2.0 roff=1e+9 ion=1e-2 ioff=1e-6)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include <alec.h>
#include "pspcomp.h"
library "pspcomp";

// Model card for new component AleC++ ccswPSp. Coressponds to the
//.model in example.
model cSWPSp::wmod{
   ron=2.0;
   roff=1e+9;
   ion=1e-2;
   ioff=1e-6;
}

root ... // or module
{
  // declaration part
  ccswPSp wccs;
  ...
  // structure part
  wccs (13, 17, vc)  model = wmod;
}
```

U*name* command:

This command is used to specify digital components in the input language of PSpice simulator.

Example 1:

In PSpice input language, an AND logical circuit named `uand21`, and its model cards `io1` (input/output model card) and `tm01` (timing model card), are defined as:

```
.model io1 UIO (inld=0.1 outld=0.2 drvh=1.0 drvl=2.0 drvz=3.0)
.model tm01 UADC (tphlmn=1 tphlty=2 tphlmx=3 tplhmn=0.1 tplhty=0.2
tplhmx=0.3)
uand21 AND(2) $G_DPWR $G_DGND in0 in1 out tm01 io1
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "pscdef.h"
# include ".alecrc"
library pscmod, pscstr, pscfun;

model uadc::tm01_io1{
   tphlmn=1.0
   tphlty=2.0
   tphlmx=3.0
   tplhmn=0.1;
   tplhty=0.2;
   tplhmx=0.3;
   inld=0.1;
   outld=0.2;
   drvh=1.0;
   drvl=2.0;
   drvz=3.0;
}

root ...  // or module
{
  module ugate::and_2 uand21;
  ...
  uand21(out,in0,in1) model = tm01_io1;
  ...
}
```

Example 2:

In PSpice input language, a digital stimulus generator with `LOOP` specification named `ustim` is defined with the following command.

```
Ustim STIM(4,13)
+ $G_DPWR $G_DGND
+ 4 3 2 1 IO_STIM5 TIMESTEP=10ns
+ 0c 00
+ 5c 03
+ LABEL=STARTLOOP1
+ 100ns decr by 01
+ 200ns goto startloop1  until lt 00
+ +10ns  13
+ 700ns  06
+ LABEL=STARTLOOP2
+ 720ns  07
+ 800ns  10
```

```
+ 900ns  goto startloop2 2 times
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "pscdef.h"
#include ".alecrc"

library pscmod, pscstr, pscfun;
module STIM1ustim (signal ps_full out out1, out2, out3, out4)
{
   signal four_full y[1:4];
   action ()
   {
       process
       {
         int init = 1;

         if (init) { y <- "0000"; init = 0; wait y; }
         else
           y <- "0011" after 50ns; wait y;
           y <- "0010" after 50ns; wait y;
           y <- "0001" after 100ns; wait y;
           y <- "1011" after 110ns; wait y;
           y <- "0110" after 390ns; wait y;
           y <- "0111" after 20ns; wait y;
           y <- "1000" after 80ns; wait y;
           y <- "0111" after 100ns; wait y;
           y <- "1000" after 80ns; wait y;
           y <- "0111" after 100ns; wait y;
           y <- "1000" after 80ns; wait y;

       } //process

       process {
         int init = 1;
         if (init)
         {
            out1 <- '0';
            out2 <- '0';
            out3 <- '0';
            out4 <- '0';
            init = 0;
            wait y;
         }

         out1 <- y[1];
         out2 <- y[2];
         out3 <- y[3];
         out4 <- y[4];
         wait y;
       } // process

   } //action
}


root ... // or module
{
```

```
  module STIM1ustim uname;
  ...
  ustim (4, 3, 2, 1) { model = io_stim5;}
  ...
}
```

A10.3.2. Conversion of simulation control statements

`.TRAN` statement:

The `.tran` statement causes a transient analysis to be performed. The general form of the statement is:

```
.tran[/OP] <print_step> <final_time> [no_print [step_celling]] [UIC]
```

The transient analysis calculates the circuit's behaviour over time, starting at TIME = 0 and going to `<final time>`. Alecsis, in its current release, performs transient analysis only, so this command can be realized in AleC++.

`OP` specification demands printing of the complete information about DC analysis in textual output file. This has not its equivalent in AleC++.

`UIC` specification orders simulator to set the voltage across the capacitors and the current through the inductors at DC analysis, which is used to determine limit conditions for transient analysis. Since Alecsis in this release does not possess a mechanism for setting the voltage across the capacitors and the current through the inductors, `UIC` specification cannot be realised either.

AleC++ equivalent for PSpice `.tran` command is is `timing` command. As we have already explained, `timing` command does not support all parameters supported by `.tran` command. `.tran` command parameters supported by `timing` command are:

print_step --- *tprint* (Alecsis)

final_time --- *tstop* (Alecsis)

step _celling --- *a_stepmax* (Alecsis)

`timing` command does not support *no_print* parameter. However, `timing` command demands a parameter which does not exist in `.tran` command - *a_step* parameter. It is calculated for `timing` command on the basis of *print_step* parameter from `.tran` command, as *print_step* divided by 5.

Example 1:

In PSpice input language `tran` statement in basic form (with two parameters that cannot be omitted) is defined as.

```
.tran 2u 10m UIC
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
timing { tprint = 2u; a_step = 2u/5; tstop = 10m;  }
```

Example 2:

In PSpice input language `.tran` statement in complete form (with all parameters) is defined as:

```
.tran  2u  20m  1u  2u  UIC
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
timing { tprint = 2u; a_step = 2u/5; tstop = 20m; a_stepmax = 2u;}
```

`.TEMP` statement:

The `.temp` statement sets the temperature at which simulation is performed. If more than one temperature is given, then simulation is repeated for each temperature. The general form of `.temp` statement is:

```
.temp <temperature_value>*
```

The equivalent command to `.temp` command from PSpice in Alecsis is `temp` option in `options` command. The limitation of Alecsis is that it does not possess a mechanism to execute simulation for more than one temperature. Because of that, if more than one temperature value appears in `.temp` command PSpice2Alecsis will take only the first one, and in the file for commenting conversion (a file with '`.cmm`' extension) a note will be found that `.temp` command does not have a fully equivalent command in Alecsis.

An example of `.temp` command conversion into `.options` command follows. It should be noted that temperature in PSpice is specified in Centigrade degrees, while in Alecsis it is in Kelvin degrees.

Example:

In PSpice input language temperature is defined using the following command:

```
.temp 50 75
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
options {temp = 323.000; }
```

`.SUBCKT` statement:

This statement begins the definition of a subcircuit. The definition is ended with `.ends` statement. All the statements between `.subckt` and `.ends` form the subcircuit the definition.

Subcircuit definition statements should contain only topology description (statements without a leading '.'), and possibly `.model` statements.

The general form of `.subckts` statement and complete form of subcircuit are :

```
.subckt <name_subcircuit> [node]*
       [optional : <<interface_node> = <default_value>>*]
       [params : <<name> = <value>>*]
       [text : <<name> = <text> = <value>>*]
       ; structure block
 .ends [name_subcircuit]
```

A subcircuit from PSpice corresponds to Alecsis `module`.

Note:

`optional` and `text` specifications are not realised in converter.

Example:

In PSpice input language, beginning of a subcircuit named `ICL7652/TI` is defined using the following command.

```
.subckt ICL7652/TI 1 2 3 4 5
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
module icl7652_slash_ti  (node 1; node 2; node 3; node 4; node5) {
...//structure of module (subcircuit)
}
```

`.PRINT` and `.PROBE` statements:

The `.print` statement prints out results from dc, ac, noise, or transient analysis in the form of table, referred to as print tables. The `.print/dgtlchg` form is for digital output variables only. The general form of the statement is:

```
.print [/DGTLCHG][DC][AC][NOISE][TRAN][(output_variable)]*
```

The `.probe` statement writes the results from dc, ac, and transient analyses to a file `probe.dat` for use by the *Probe* graphic postprocessor. The general form of the statement is:

```
.probe [/csdf] [output_variable]*
```

Note:

For showing simulation results in PSpice we can use either `.print` command or `.probe` command, or both simultaneously. Both commands can be used without arguments, and in that case complete simulation results are included (values of all circuit quantities). (Conversion into AleC++ code is not supported for this case.). `.print` and `.probe` commands could print a great number of output value categories. In this release of PSpice2Alecsis, conversion of only one category - node voltage (e.g. `V(1)`, `V(a_node)`) - is supported.

Example:

In PSpice input language, `.PRINT` and `.PROBE` statements are defined using following two statements.

```
.print tran v(305) v(505) v(105 505) v(308 108) v(1408 708)
.probe v(608) v(1349 321) v(1402) v(305) v(505) i(lname) i(vname)
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
plot {
       //PRINT output :
       node 305; node 505;
       //PROBE output :
       node 608; node 1402; node 305; node 505;
}
```

`.PARAM` statement:

This command defines global parameters of simulation. A global parameter can be a constant or expression. This command is realised in AleC++ by means of preprocessor command `#define`.

Example:

In PSpice input language a `.param` statement is defined with the following command.

```
.param e19 = {1 / (6.28 * sqrt(ll * cc)) }
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#define e19 1/((6.28*sqrt(ll*cc))
```

`.INC` statement:

The `.inc` statement is used to insert (include) the content of another file into the current file. Including files is the same as simply bringing the file's text into the current file. Included files may contain all statements with these exceptions: no title lines is allowed (use a comment); `.end` statement is not allowed. The general form of `.inc` statement is:

```
.inc "file_name"
```

Note:

The current converter realisation allows that only subcircuits and `.inc` command can be found in the included file.

Example:

In PSpice input language, `.inc` statement is defined with as.

```
.inc "dat1.mod"
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "dat1.ac"
```

`.LIB` statement:

The `.lib` statement is used to reference a model or subcircuit library in another file. The convenience (with respect to `.inc` command) is that the complete library is not read through, but only needed objects are found and included in circuit description.

The general form of the statement is:

```
.lib "file_name"
```

Note:

For the time being, `.lib` statement is realised in the same way as `.inc` command - like `#include` preprocessor command.

Example:

In PSpice input language `.lib` statement is defined as:

```
.lib "dat1.mod"
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#include "dat1.ac"
```

`.FUNC` statement:

This statement is used to define "functions" that may be used in expressions. This command is realised in AleC++ like the macro by using `#define` preprocessor command.

Example:

In PSpice input language `max` functions is defined using the following command.

```
.func max (a,b) (a + b + abs(a - b)) / 2
```

AleC++ code obtained after conversion with PSpice2Alecsis is:

```
#define max (a,b) ((a)+(b)+abs((a)-(b))))/2.0
```

A10.4. Limitations of PSpice2Alecsis converter

This chapter deals with the restrictions of PSpice2Alecsis program. There are two causes for restrictions:

- some commands and specifications of the simulator input language can not be realised in AleC++, due to differences in simulator engines.
- conversion for certain commands and command specifications of PSpice simulator input language is not yet implemented in PSpice2Alecsis

Limitations are divided into three groups:

- limitations of commands describing circuit topology (components and their connections);
- limitations of commands for simulation control;
- general limitations.

In the next sections, following description is used:

yes - PSpice command is completely supported

no - PSpice commands is only recognised, but not supported

yes* - PSpice command is partly supported

A10.4.1. Limitations of circuit topology description

B*name* --- no
C*name* --- yes* (`IC` specification is not supported.)
D*name* --- yes
E*name* --- yes* (`FREQ` specification is not supported.
`LAPLACE` specification is supported only under the following conditions:

- transfer function can be : $\frac{A*s+B}{C*s^2+D*s+E}$,
- A, B, C, and D coefficients can constants or variables, but more complex expressions are not allowed.

– expression which shows control input can only be voltage or current.)

F*name* --- yes

G*name* --- yes* (FREQ specification is not supported.
LAPLACE specification is supported only under the following conditions:
– transfer function can be : $\frac{A*s+B}{C*s^2+D*s+E}$,
– A, B, C, and D coefficients can constants or variables, but more complex expressions are not allowed.
– expression which shows control input can only be voltage or current.)

H*name* --- yes

I*name* --- yes* (AC and DC specifications are not supported.)

J*name* --- yes

K*name* --- no

L*name* --- yes* (IC specification is not supported.)

M*name* --- yes

N*name* --- yes

O*name* --- yes

Q*name* --- yes

R*name* --- yes

S*name* --- yes

T*name* --- no

U*name* --- yes* (MNTYMXDLY and IO_LEVEL specification are not supported.)

V*name* --- yes* (AC and DC specifications are not supported.)

W*name* --- yes

X*name* --- yes* (TEXT specification is not supported.)

A10.4.2. Limitations of commands for simulation control

.ac --- no

.dc --- no

.distribution --- no

.end --- yes

.ends --- yes

.four --- no

.func --- yes

.ic --- yes* (Assignment of voltage between two nodes is not supported, because this version of Alecsis does not possess the convenient mechanism. For the same reason, IC specifications in commands for inductor and capacitor description are not supported.)

.inc --- yes* (Included library, according to the PSpice syntax, can contain all PSpice commands except title lines and .end command. However, in an included file, PSpice2Alecsis supports only commands that can be found in the subcircuit (commands for description of PSpice components and/or subcircuits) and .inc command.)

.lib --- yes* (For the time being, .lib command is realised the same way as .inc command - as #include preprocessor command.)

.loadbias --- no
.mc --- no
.model --- yes
.nodeset --- no
.noise --- no
.op --- no

.options --- yes* (Only temperature parameter is supported.)
.param --- yes
.plot --- no

.print --- yes* (See example for .print command in the section A10.3.2.)
.probe --- yes* (See example for .probe command in the section A10.3.2.)
.savebias --- no
.sens --- no
.step --- no

.subckt --- yes* (OPTIONAL and TEXT specification are not supported.)
.temp --- yes* (More than one temperature value is not supported, because Alecsis does not possess the convenient mechanism to repeat simulation.)
.tf --- no

.tran --- yes* (no_print parameter is not supported,
UIC specification is not supported.
.watch --- no
.wcase --- no
.width --- no
.text --- no

A10.4.3. General limitations

- At conversion, capital letters become small ones (small letters stay the same).
- In AleC++, characters '$', '%', '*', and '/' cannot be used in names of variables and nodes. For that reason, when PSpice2Alecsis reads such signs, it converts them into '_dollar_', '_percentage_', '_slash_', and '_star_', respectively.
- Voltage between two nodes cannot be initialised in this version of Alecsis (only node votlage with respect to ground node can be initialzied). It means that the command

```
.ic v(1, 2) = 5.0v.
```

  cannot be realised in IC specification in L*name* and C*name* commands.
- Connection of text lines using '+' sign is not made possible for every command. For instance, in .subckt command, it is not supported that each node name can be found in separate line. The same applies to STIM specification of U*name* command.
- PSpice2Alecsis does not recognise line comment in lines that are the continuation of the previous ones (beginning with '+')

- PSpice2Alecsis does not allow usage of following keywords as names of variables and nodes: `TIMSTEP`, `MNTYMXLY` and `IO_LEVEL`
- PSpice2Alecsis does not allow usage of keyword `FILE` as the name of a variable or a node in X*name* and `.subckt` commands.