5. Basics of simulation in Alecsis

The previous chapters were an overview of concepts already familiar from C/C++, and were meant as an introduction to AleC++. The topic of this and following chapters are constructs of AleC++ not found in C/C++.

Alecsis 2.0 is a hybrid simulator, which means it is capable to simulate both digital and analogue circuits. To state it more generally, Alecsis can simulate both discrete-event and time-continuous systems. This is not a trivial problem, since the techniques of solving these two kinds of systems differ very much. Analogue circuits are simulated using Kirchhoff's laws, thus solving systems of differential equations describing a particular circuit. Digital circuit are simulated using logic states on the inputs, and logic functions determining the output. Where these two types meet A/D or D/A conversion is necessary.

Alecsis is an integrated simulator, which means the algorithms for both kinds of simulation and the conversion process are inseparable. That is, Alecsis is not a case of two simulators "glued" together using special mechanisms of synchronization. AleC++ allows mixing of various analogue and digital elements and constructs, and the user is the only judge of the usefulness of this capability. In this Manual, we will point out the price of using certain constructs in terms of memory, speed, etc.

The simulation engine of Alecsis controls and manages two basic mechanisms, solving of systems of equations represented using sparse matrices, and synchronization of parallel processes and signals in digital components using the *next event* principle and *selective trace* principle. Both mechanisms use the services of the virtual processor, whose role is to execute (interpret) instruction generated by AleC++ compiler and linker. The topology, and characteristics of the circuit is defined by the user regardless of the type of the circuit in question.

5.1. Module concept

When the relative complexity of C++ is considered, the user might get the wrong impression that even the simplest simulation demands hundreds of pages of code,. The example will show what we mean:

```
root module example_1 () {
    resistor  r1;
    capacitor c1;
    vpwl      vin;

    r1  (input, output) 2k;
    c1  (output, 0)     5pf;
    vin (input, 0)      { 0ns, 0v; 1ns, 1v; }

    plot { node input; node output; }
    timing { tstop = 100ns; a_step = 1ns; }
}
```

This is an example of a simple RC circuit with the impulse input of 1V. The impulse is generated using a generator of piecewise-linear voltage `vin`. Besides the topology of the circuit, the `root` module contains the printout format and the data on the duration and the step of the simulation. Such special statements that control the simulation flow can appear in `root module` only. The result is shown in the Figure 5.1:

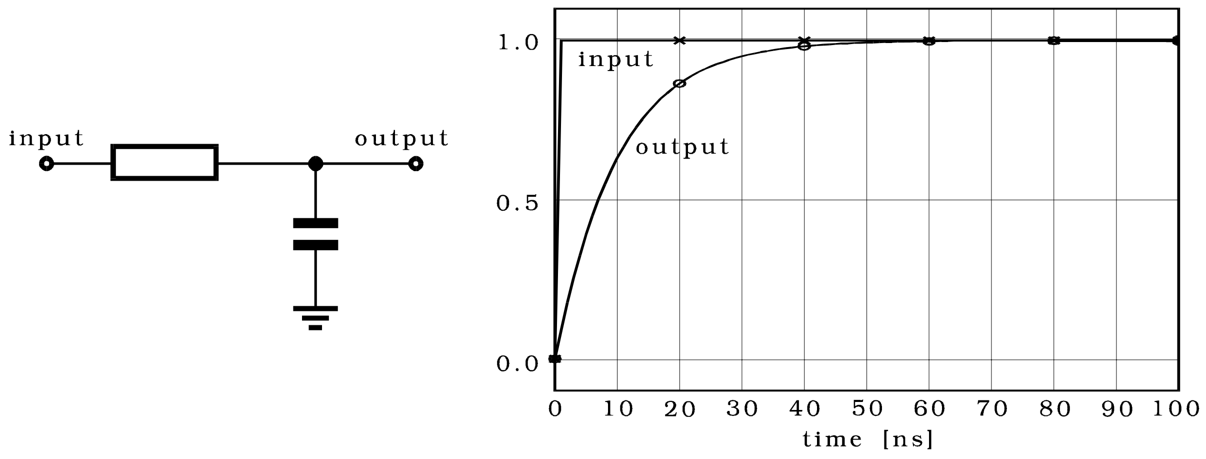

Figure 5.1: The result of the RC circuit simulation

The `module` is the basic element of hierarchy in the hardware description language AleC++. It is used for both discrete-event and time-continuous models, even for A/D (D/A) converters. In C, the execution of a program begins and ends within the `main` function. In Alecsis, the hardware hierarchy begins and ends in the `module` named `root`. Modules have a lot in common with the C-functions, beginning with the syntax, similar interface, to the visibility rules. The similarity is intended since the intention was to keep the spirit of C/C++. On the other hand, modules have something in common with classes as well, since they define new types of components, which can be used evenly with the built-in components.

5.2. Link

Link is the common term in AleC++ defining all types of entities used for connecting components. This term is vague by definition, since it is treated differently for the cases of analogue and digital components. There are five different links: **`node`, `current`**, **`charge`**, **`flow`**, and **`signal`**. These are key words used for link declarations, i.e. they represent link type. Links can be also of user-defined type, i.e. they can be structures, scalars, and vectors, can be initialized, etc. In these characteristics they remind us of variables, and in deed in particular context links can be used in expressions as variables. **However, links are very different from ordinary C/C++ variables, as they are unknowns in the circuit representations, and the simulator engine is used to solve for them.**

The first four mentioned link types (`node`, `current`, `charge`, and `flow`) are analogue. They are used for description of analogue components, and appear as unknowns in the system of ordinary differential equations. The value of the link is evaluated by solving the system of equations. These four link types have the same implementation, but differ in their physical meaning. `Node` is the basis for the analogue part of the simulator, since it represents the physical link - conductor, and because the system matrix is formed using nodal analysis method. The variables of link type `node` are actually representing the node voltage. This method is expanded in Alecsis (as well as in all well-known simulators, including SPICE) to include the elements of zero resistance (voltage sources, inductors, etc.) Such an element generates new (branch) equation, and the branch `current` appears as the variable. `Charge` is used for nonlinear capacitance modelling, to allow separate discretization and linearization of the model. `Flow` is a keyword for a general analogue link, whose physical meaning is not given in advance. This separations of analogue circuit unknowns is due to physical differences, in order to control the convergence tolerances separately. The absolute tolerance for flows is not given in advance, but can be defined by the user. When used in expressions, referencing the name of the analogue link relates to its value obtained in the last solution of the system of equations.

Signals differ significantly from other link types. Signals are links that carry some logic value. Digital systems in Alecsis are modelled as parallel processes. Signals are used for synchronization, too. Signals can be read-only, write-only, or both. Every signal has associated memory according to link data type, which contains its current value. That value is obtained if the signal name is used in an expression.

If the link type is `signal`, the link by default has **digital aspect** if not explicitly defined with **analogue aspect**. If a link is connected to a component whose fundamental type (A or D) does not agree with the link aspect, the component has a dual, or hybrid aspect. Links wit hybrid aspect make the simulator generate A/D (D/A) converters.

Note: The link declaration has the link type and the link data type. The link type can be `node`, `current`, `charge`, `flow`, `signal`, or some composite type made of these basic types. The link data type refers to type of its value (i.e. variable type - `double`, `int`, enumeration type, ..). The following declarations are legal:

```
node double X; signal three_t Y;
```

There are default link data types. It is clear that the `node` or `flow` would have value of type `double`, so there is no need to mention that explicitly. However, for signals it is usual to state the type explicitly, as it is normally some enumeration type.

5.3. Module declaration

Modules have prototypes in much the same way as functions do. Module has architecture, and interface for communication with the surroundings. The prototype declaration is related to its interface. The declaration allows the compiler to check the agreement of actual with formal links on when the component of the module type is referenced.

prototype_module:

module *<flow_sc> <library_name* **.***>module_name (<list of formal links>)*

flow_sc:
current
charge
flow

list_formal_links:
formal_group
list_formal_links **;** *formal_group*

formal_group:
< link_type > <type> <direction> list_links

list_links:
link
list_links **,** *link*

link:
link_declarator <= *constant_expression>*

link_declarator
identifier
link_declarator [<vector_dimensions>]

link_type:
signal
node
flow_sc

direction:
in
out
inout

Module has its name, which can be used evenly with the names of built-in types of components, such as `resistor`, `capacitor`, etc. Modules can return the link. If the link type is missing, type `node` is inserted by default, except in the case of the enumeration link, where `signal` is default. If the link data type is missing, the default for analogue links is `double`, and for digital `int`. Direction refers exclusively to signals, and is `in` by default. Initial values can be defined for signals as an option.

```
typedef enum { 'x', '0', '1' } digital3;
struct Net { digital3 send, recv, ack; };

module X (i, j);                     // two node links - i and j
module Y (digital3 in a, b; digital3 out y); // signals
module Z (signal digital3 in a[]="001"; signal Net out network);
module current misc.M (node i, j);    // library "misc"
```

These examples of module declaration encompass all of the rules. Module `Z` contains the declaration of the first formal link `a` as a digital vector, whose dimensions are not defined. As with functions, the dimensions of the vector will be defined when the actual vector comes to that position due to the module connecting. By default the dimensions of the vector is 3, based on the initialization expression - enumeration string `"001"`. Module `M` expects two nodes `i` and `j`, and returns the current under its name. Explicitly defined library (`misc`, and separator `.`) instructs the linker to look for the body of the module in the mentioned library (without searching all of the available libraries). If more libraries are visible, all containing a module with the same name the concept of the explicitly defined library will solve the dilemma.

Formal links do not represent unique entities, just references to real links on the module interface. These declarations are usually found in header files included by using command `include`. More than one declaration for the same module can be found in the same file with the condition that all those declarations overlap, or append each other.

Note: Module definition is valid also as a declaration for the following code, which is why carefully placed definitions can render declarations unnecessary. However, if modules are compiled, and stored in libraries, prototypes (declarations) of modules are necessary and have to be added using statement `include`, if they are gathered in one header file.

5.4. Module definition

Declaration of a module is helpful to the compiler during numerous checks. The definition of the module gives its actual content. The definition of a module can be compiled and stored in some library, since modules are external units, that is they are subject to linking operations.

Modules can be defined on the global level only, and cannot be nested. Note that more than one definition of a same module in same file is an error, which was not the case with declarations. Definitions repeat the declarative part, but also contain the body of the module. The body of a module is bounded by characters '`{`' and '`}`'.

The module architecture depends upon the modelling approach of a particular electronic component. The component can be digital, analogue, or hybrid. It can be represented structurally, as a collection of connected components, functionally, using constructs based on commands, or using a combination of both approaches. For each approach, a region exists in the body of the module where those constructs are allowed.

module_definition:
    module_declaration { module_body }
module_body:
    <declarative_part > < topological_part > < functional_part >

The body of a module can have all three, or none of the mentioned regions, but the former case has no practical implications.

5.4.1. Declarative part

The body of a module may contain some entities (links and components) that are local, i.e. used only inside that module. AleC++ needs them declared before they are used, which is a rule from the languages making its basis. Visibility of local links is the same as with local variables, being the body of the module, with the notion that local links duration is the whole the simulation (like static variables).

The syntax of the declaration is similar to the one for the interface, just the direction is not stated. Direction is by default `inout` for signals, and does not make sense for other links.

```
signal digital3 matrix[][4] = { "0011", "011x", "1101", "10xx" };
node vdd, vss;
digital3 clock = '0', mask[3:0] = "0010";
```

As with the variables, signals-vectors can be initialized and can have inverse dimensionality. If link data type is enumeration type, the link type can be omitted, as it must be signal. If the link data type is left out, defaults are the same like for formal links: `int` for signals, and `double` for others.

There are cases when local links need not be declared before they are used. If an undeclared name appears during the component connecting, a (scalar) `node` of the same name is implicitly defined, with the same characteristics. This rule makes it easier to use Alecsis for the users of SPICE, which does not demand nodal declarations.

Component declaration introduces names, and links them to the component type. The declaration consists of the type of the component, and the list of elements which can be used for modelling.

```
resistor r1, r2;
capacitor cload, cs;
module rsff ff1, ff2;
rsff ff3;
module ttllibrary.ff3;
module frsff (digital3 in reset, set; digital3 out q, qbar) ff4, ff5;
```

If the type of the component is module (i.e. not a built-in Alecsis component) it can be cited with or without the key word `module` (as with structures/classes, the key word needs to be used if the name `rsff` is masked). It is legal to give the name of the library (the row before the last), and even the complete declaration (the last row). If the complete declaration appeared somewhere in the previous text, it is sufficient to give the name of the module for the name of the component.

There is a possibility to skip the component declaration, again to make Alecsis comfortable to SPICE users. The compiler can, based on the name of the component, determine the type (see the paragraph on `implicit` constructs).

5.4.2. Structural part

We will focus on the component connecting and the definition of their parameters in this section. To connect components, we list their names and actual links, and perhaps the values of the parameters:

```
r1 (n1, n2) value = 2k;
r2 (n2, 0) 3.3k;
cload (load, 0) 15pF;
```

The syntax of connecting is:

component:
    name (<list_actual_links>) parameters

list_actual_links:
    static_link
    list_actual_links , static_link

static_link:
    name_of_link

> *static_link [constant_expression]*
> *static_link [constant_expression* **:** *<constant_expression>]*
> *static_link* **.** *structure_member*

parameters:
> *;*
> *paremeter_value ;*
> *{ list_parameters }*

list_parameters:
> *parameter_in_list*
> *list_parameters_of_ parmeter_in_list*

parameter_in_list:
> *parameter_name= parameter_value ;*

parameter_value:
> *constant_expression*
> *parameter_name = parameter_value*

Actual link can be a scalar, but also a composite link type (vector, structure ...). The link of composite type can be for used for connecting as a whole, by citing its name, or one can use only some of its members. Indexing can be used if the link is an array (vector, matrix, etc). The notion of a **static link** is applied either to the link, or to any of its parts, which can be fully determined during the compiling. For that reason, it is necessary that the index is a constant expression. We will return to static links when further describing syntax of AleC++. If the link which is an array is indexed using two expressions and character '`:`' between them, its dimensionally is not changed, but its boundaries are changed (reduced). The result of this operation is called **slice**.

```
signal digital3 s, v[10], m[3][4], v2[15:0];
signal Net data;
...
c (s, v[1], v, m, m[2], m[2][1], v2[9:7], data, data.ack);
```

In this example, component `c` is connected using 9 links: scalar `s`; scalar from the position `1` of vector `v`; vector `v` of length 10; matrix `m` of length 3x4; vector of length `4` from the position `2` of matrix `m`; scalar from position `2,1` of matrix `m`; slice of vector `v2` from position `9` to `7` (still a vector); structure `data`; and a scalar `ack`, member of the structure `data`. The compiler is responsible to determine if this list of arguments agrees with the prototype of module of component `c`. From the syntax standpoint, this is a legal list of actual links.

Alecsis is equipped with a few, but carefully chosen set of predefined (built-in) components. These are fundamental components, which are often used in electronic circuit. They can be also used as the basis for modelling of other components (you can find this explained in detail in the chapter on analogue simulation). The number of component parameters varies from one component to another. Simpler ones take only a single parameter of `double` type under the name `value` (resistor resistance, coil inductance, etc). In this case only a numerical value can be given. If there are more parameters, their list needs to be bounded by parentheses.

```
cload (load, 0) 15pF;
cs (n2, 0) value = 10pF;
mos1 (n1, n2, 0, 0) { model = nmos1; l=2u; w = 3u; }
```

Components of type `module` can have parameters, too. Their syntax does not differ from the one for built-in components, however the declaration needs to be expanded to include the names, and types of legal parameters. We did not talk about this up to this point, since this goes into functional simulation.

5.4.3. Functional part -- action block

Electronic component (subcircuit) can be defined functionally in several ways:

- By defining a logic function of the component . This is purely functional approach for the digital aspect.
- By stating the model equations. They contribute directly to the system of equations for the whole circuit (system). This is purely functional approach for the analogue aspect.
- By defining and equivalent circuit topology, and calculating the parameters of the components in that equivalent circuit in AleC++ code. For instance, nonlinear model can be represented as equivalent linear circuit whose parameters are changed in every iteration. This is a combined approach for the analogue aspect.
- By combining all approaches given above - combined approach for hybrid aspect.

To realize this functional description, we need a part of the module body where C/C++ -like code can appear. A region of the module body beginning with the keyword **`action`** is used for that. This functional region is bounded using parentheses, creating a narrower visibility area, which allows masking of elements, local and formal links, etc. If our module accept parameters, they are actually accepted by the `action` block. For that, in module definition, you need to give a list of `action` parameters as if a prototype of a function is created:

```
module X () {
    ...
    action (int n, double p, char *name="initial value")
    {   /* the code describing fucntional description */   }
}
```

As was the case with function parameters, action parameters can have default values. We recommend this approach, since it allows the correct work of the component even if the parameters are not set during connecting.

Absence of `action` parameters can be signalled by type `void`, parentheses without parameters '`()`', or by leaving out parentheses after the `action` keyword. Action can be defined with a variable number of parameters using symbol '`...`' as it is defined in the chapter on functions.

Action parameters are added to module declaration:

```
module and2 (digital3 in a, b; digital3 out y)
                      action (double tplh=10ns, double tphl=10ns);


module X () {
    digital3 s1, s2, s3;
    and2 a1, a2;

    a1 (s1, s2, s3) { tplh = 11ns; tphl = 13ns; }
    a2 (s3, s1, s2) action (11ns, 12ns);     // alternative method
}
```

As you can see in the example above, components that have `module` type (not built-in) can utilize another method of parameter setting, too. It resembles function calls, because a list of arguments follows the word `action`. Linking of parameters and arguments is done **by position**, which is different from the commonly used **associative method.** As was the case with the functions, parameters with initial values can be left out.

Note: **Action parameters** can be used bidirectionally, i.e. they can return the value. They **are passed *by reference***, unlike parameters of C function, which are passed *by value*. For one application of that feature see explanation of `plot` command in this Chapter.

> ⚠ If, for instance, `action` block in some module expects parameter of type `double`, and you connect that `module` in the parent `module` with the appropriate `action` parameter as integer constant, the simulator will not issue any warning, but will not give you expected results. The reason is that *an implicit conversion is performed*, since the parameter is passed by reference. This is, however, usually not what you wanted. **Be careful with `action` parameters -- always pass the constant or variable parameter of correct type.**
> In the following versions of Alecsis, warning will be issued for such cases.

Note: Action parameters of one component can be accessed from other components using indirection operator `->`. For instance, if `module X` in the example above has its own `action` block, inside that `action` block you can access action parameters of component `a1` as `a1->tplh`, `a1->tph`. (component name behaves as the pointer to the structure comprising of its `action` parameters). For another application of that feature, see section on `plot` command in this Chapter.

All variables local to the action area last throughout the simulation, and are therefore static. This is important, since `action` block can be executed many times during the simulation run, and the variables must not be reset. However, each separate component has a separate memory, different from the memory for other components of the same type. If needed, some variables can be defined `static`. These variables are similar to static members of classes in that memory reserved for them is common **for all components** of the same type in the circuit. This means change to the value of that variable in one component will ripple to all other components of the same type. Since this allows communication using the "back door", circumventing action parameters and links, great care needs to be exercised in order to avoid unwanted effects.

`Action` block is where functional behaviour of the `module` is defined. If the `action` block is left out, `module` is a set of components connected in a desired way. In that case, `module` represents only a subcircuit, and is used to describe the whole circuit hierarchically.

5.4.4. Modelling of parallel processes

It should be noted that all components in a circuit (system) are active simultaneously, or, in programmers' terminology, in parallel. To enable Alecsis users to describe that parallelism, and to control execution of parallel blocks, we have introduced `process` into the functional description. Processes are described inside the `action` region.

The `action` region is divided into the declarative part and the command part.

action_region:
    action *<trigger_sc> <(<parameter_declaration>)> <action_body }*

action_body:
    <declarations> <commands>

commands:
: *command_list*
: *process _list*

process_list:
: *process_command*
: *process_list process_command*

process_command:
: *<process_name :>* **process** *<sinhro> { <commands> }*

sinhro:
: *trigger_sc*
: *(sensitivity_list)*

trigger_sc:
: **structural**
: **post_structural**
: **initial**
: **per_moment**
: **post_moment**
: **per_iteration**
: **final**

sensitivity_list:
: *static_link*
: *sensitivity_list , static_link*

The `process` is the backbone of functional modelling in AleC++. The processes are given in the `action` block, and they consist of commands. Those commands are executed during the execution of the simulation. The processes ought to be synchronized.

In discrete-event simulation, signals synchronize processes. The **event** has happened when a state (value) of a signal is changed. All processes sensitive to that particular signal become active when such event happens (commands of the `process` are executed). After that, processes sensitive to that signal are in inactive state, or **latency,** until new event happens. This agrees with the concept of logic states, events, and transfer of signals throughout the circuit in digital (discrete-event) simulation. A `process` can be made sensitive using the sensitivity list, which lists all signals whose change can activate the `process`. As was the case with actual links, we can list signals, signals with indices, or, if the signal is a structure, a member of the structure. Process is activated if a change occurs **on any** of the listed signals. In the case of a composite signal, an event has happened if an event has happened on at least one of its scalar elements (positions of a vectors, or members of a structures).

Sensitivity list must be avoided if the process has **`wait`** command (see the section on the `wait` command).

If the `process` does not have a sensitivity list, `wait` command, and is not synchronized in any other way, it becomes a world on its own, and is useless as far as simulation is concerned.

More than one `process` can be sensitive to a particular signal. When an event happens on the signal, these processes are executed by the simulator one by one, but they appear as parallel from the point of view of circuit functionality.

In analogue circuit simulation, different synchronization mechanism needs to be implemented. Analogue components are active all the time, since they contribute to the system of equations that is repeatedly formed and solved during the simulation. As Alecsis is used for time-domain simulation of nonlinear circuits, there are two loops: the time-loop (outer), where simulation is executed in many discrete time instants; and iterative loop (inner), where, in every time instant, nonlienar circuit is solved iteratively. From that point of view, no synchronization is necessary. However, synchronization of processes is very useful in analogue simulation, too. If a component is linear and time-independent, as resistor, there is no need that it is executed in every time-instant, and in every iteration. The appropriate `process` can be executed once, before the actual simulation starts, which saves CPU time. If a model is linear, but time-dependent, the `process` can be synchronized to be executed in every time-instant, but out of the iterative loop.

For synchronization of analogue processes, some internal synchronization signals are generated and controlled. The value of these special signals cannot be accessed in expressions, it can be used for `process` synchronization only. These signals are:

♦ **`structural`** - activated before the simulation, during the creation of a hierarchical tree, that represent the circuit hierarchy in simulator memory. It is intended to be used for processes that contain command **`clone`**, that creates array of components (or subcircuits). As the command `clone` results in a change of the circuit structure, it has to be executed before the simulation starts, and before the circuit hierarchy is formed in the simulator memory.

♦ **`post_structural`** - Activates before the simulation, after the current hierarchical level is complete. It is used for modification of signal attributes (see section on user-defined attributes).

♦ **`initial`** - Activates only once, at the beginning of the simulation, when time $t=0$. Usually used for some intializations, but also to calculate contributions to the system matrix that are not changed during the simulation, i.e. that need not to be calculated inside the time loop and the iterative loop.

♦ **`per_moment`** - Activates in every new time instant of the simulation $t=t_{n+1}$ before solving the system of equations. It is intended to be used for modelling of linear but time-dependent contributions to the system matrix (e.g. linear capacitors, linear time-dependent voltage or current sources, etc.). Usage of link name in expressions in such `process` returns its value from the previous (last solved) time instant ($t=t_n$)

♦ **`post_moment`** - Activates in every time instant after reaching the solution of the system of equations. Processes, which need solution from the moment $t=t_{n+1}$, use this synchronization. If a circuit consists of digital elements, and links, solving the system is unnecessary, so this and the previous synchronization signal are activated simultaneously.

♦ **`per_iteration`** - If the circuit has analogue elements, this synchronization signal activates in every new iteration before solving system of equations. If there are n time instants, and m iterations in every time moment, processes sensitive to `per_iteration` signal are active m x n times. Used for nonlinear analogue elements, where contributions of the linearized model to the system of equations are calculated in every iteration, until convergence occurs. If the circuit does not have analogue elements, these processes will not activate.

♦ **`final`** - Pair with the **`initial`** signal - activates only once, at the end of the simulation. If a file is opened or memory is allocated in the `process initial`, this is the place where everything needs to be closed and ended.

Some of the above mentioned signals can be very useful in digital applications, too, especially at the beginning or an end of the simulation.

All legal AleC++ commands except **return** can appear in a process, since the process cannot terminate (the whole simulation can be terminated using exit, but this a very rough solution). Command **continue** needs a small modification of standard rules; considering the cyclic nature of a process: usage of this command outside a loop means a jump to the first command and restart of the process. If the process is sensitive to signals this causes halt of the process until an event occurs on signals. You cannot jump from a process to a process using command **goto**.

Every process creates a separate visibility area where local objects can be declared. All initializations of such object have to be static, that is have to contain constant expressions. Process objects, explicitly declared as static using key word static, are common for all copies of the process that are created by declaring more components of the given type. The rest of the objects, not explicitly declared as static, are unique for every copy of the process.

There can be more than one process in an action block. If there is only one process in an action block, the keyword process can be omitted. In that case, synchronization signal is given after the keyword action. If no synchronization is defined, per_iteration is used by default.

```
module X (digital3 in x, y; digital out z) {
    action (double delay) {
        int shared;       // common variable for all processes in
                          // this action block
        static nmodules;  // common for all components of type X
        p1: process initial {
             // initial activities
             ...
        }
        p2: process (x,y) {
             // activation on event on signals x or y
             ...
        }
        p3 : process final {    // names p1, p2, p3 can be omitted
             // final activities
             ...
        }
    }
}

module Y (i, j) {
    resistor r;
    capacitor c;
    action per_iteration {
        // ... an analogue process
    }
}
```

5.4.5. Variable number of action parameters

The number of parameters in the header of the action block of the module can be variable. As action header resembles the header of the function, the technique for variable number of parameters is the same as for the C/C++ -like function, described in Chapter 4.

```
#include <varargs.h>

module ResistorWithTemperatureCoefficients (node i,j) {
   resistor res;
   res (i,j);

   action post_structural (double resistance, ...)  {
      char *args;
      double tnom = 300., tc1,tc2,tce;
      tc1=tc2=tce=0.;

      va_start(args, resistance);   // setting pointer args

      if (tc1 = varargs(args, double))  // read other parameters
         if( tc = varargs(args, double))
            tce=varargs(args, double);

      if (tce)
         res->value = resistance*pow(1.01,tce*(temp-tnom));
      else if (tc1)
         res->value = resistance*(1+tc1*(temp-tnom)+
                                 tc2*(temp-tnom)*(temp-tnom));
      else
         res->value = value;
   }
}
```

Parameter `temp` in the example above is the user-defined temperature (see description of simulation `options` in this Chapter).

There are some differences in storing function formal parameters and `action` parameters. For that reason, alignment of parameters of type `double` using flag `DWORD_ALIGNMENT` is not necessary for `action` parameters, even for the computers that need that for C/C++-like functions. Moreover, you should be sure that this flag is not defined when file `varargs.h` is included

As we have pointed out for functions, functionality of macros defined in Alecsis `varargs.h` file depend on processor. Therefore, if you install Alecsis on computer that was not predefined in `Makefile`, some adaptations on varargs.h might be necessary.

5.5. Implicit declaration of components

SPICE deciphers the component type from the first few characters in the name of the component. All components in Alecsis need to be declared before connected. For some types of common components this can become rather tedious, and is not convenient for SPICE users. For that reason, we have enabled such implicit declarations in AleC++, but only as an option. Construct `implicit` is used to fulfil that.

implicit_command:
  implicit *{ association_list }* <;>

association_list:
  implicit_association
  association_list implicit_association

implicit_association:
  component_type list_names ;

component_type:
  built-in_type
  module_declaration

This `implicit` declaration defines one ore few characters that represent beginning of the component name. These characters are normally association of a component type. After such a declaration, all the components, whose names begin with these characters, have a declared type, and do not have to be explicitly declared. The name of the component has to be at least one character longer than the implicit symbol. In case of a conflict (two, or more suitable symbols) the longer is chosen. This command can be used many times in the text, but only on the global level. The care needs to be exercised not to make unwanted redefinitions.

```
implicit {
    resistor r, R;
    capacitor c, C;
    mosfet m, M;
    bjt q, Q;
    module rsff ff;
    rsff rsf;
}

module X (i, j, k, l) {
  r1 (i, 0) 2k;         // O.K. - resistor
  R2 (j, 0) 4k;         // also
  m1 (i, j, 0, 0) ...; // MOSFET
  ff (i, j, k, k);    // error - the name is the same as the symbol
  ff1 (i, k, k, k);     // O.K. - flip-flop
  rsf1 (i, j, j, l); /* resistor or rsff? - rsff, since it
                        is longer! */
}
```

5.6. The root module

Every hierarchical tree has a root. In AleC++, description of a hierarchy of an electronic circuit begins from the module named **`root`**. In this case, keyword `module` is not necessary. Everything said about modules is true for the `root module`, with certain modifications and additions. The `root module` must not have formal signals and/or `action` parameters, since due to its position on the top of the tree it cannot receive any. In the same fashion, it cannot return under its name any link. Finally, the `root module` has three additional constructs defining the conditions of the simulation and printing its results:

- **`plot`** - for printing out the results;
- **`timing`** - for timing control;
- **`options`** - for defining simulation conditions (e.g. tolerances).

Note: Commands `plot`, `timing` and `options` are placed between structural and functional part of the `root module`. In this space, these three commands can be given in any order.

Note: In the `root module`, functional part can be omitted, but the structural part must be described. Command `timing` cannot be omitted. Command `plot` can be omitted (but it does not make to much sense to simulate without printing out results). Command `options` can be omitted.

5.6.1. Print control -- command plot

Alecsis does not have waveform display capabilities that can be used to view the results of simulation. For that, separate program Agnu is used. Alecsis creates an output file, with the results of the simulation in numerical form. There is not much sense in saving states of all digital signals and values of all analogue variables in the circuit, because that would make the output file too big to handle. Only signals and variables specified by the user are saved. For that, the command **`plot`** is used. Alecsis creates the output file during the simulation. If the simulation is terminated for any reason before the final time-instant is reached, all time-instants solved until that moment are saved. The output file carries the same name as the input file, with the extension **`.ar`** (Alecsis results). (Input file has extension `.ac`.)

printing:
 plot *{ content_of_printing } <;>*

content_of_printing:
 element_of_printing
 content_of_printing element_of_printing

element_of_printing:
 caption *constant_string ;*
 link_type <type> <direction> link_list;
 sweep *<type> link;*

link_list:
 link
 link_list , link

link:
 static_link < (element) >
 absolute_path /static_link <element>
 identifier body_function

absolute_path:
 element_name
 absolute_path / element_name

Note: Keyword `plot` can be replaced with `out`, for compatibility with earlier versions of Alecis.

The links are specified with their link types. The link type comes from the set of legal types (`node`, `signal`, etc.). The link data type (`double`, `int`, etc. or some composite type) is not necessary for link local to the `root module`, but if you want the value of link situated somewhere else on the hierarchical tree, which is a vector, structure, or a digital signal, the link data type is necessary.

For link that is not local to the `root module`, the absolute path is given as the part of its name. The path is composed similarly to the path in UNIX operating system. The absolute path is the path from the `root module` to the internal link via names of the components making the path (i.e link `X` in submodule `Y` is given as

`Y/X`). The link itself can be an identifier, identifier with index (for arrays), or reference to a member of structure, by rules of static links. If the name is a composite link, the printout will consist of all its elements.

As said above, link data type has to be given for vectors, structures or signals that are not declared in the `root module`. If vector of nodes `U` of length `N` is declared in submodule `Y`, command:

```
    plot { node Y/U; }
```

results in printing only the first vector position `U[0]`, instead of the whole composite link, i.e. all `N` positions. The compiler does not see the declaration of vector `U`, which is lost after parsing the `module Y`. For that reason, `U` is treated as the scalar node, not as the vector. The problem can be solved if a vector type is defined on the global level, before the `root module`:

```
    typedef double tmpvec[N];
```

and printing is performed using:

```
    plot { node tmpvec Y/U; }
```

In this way, link data type is given explicitly, and the compiler knows that `U` is a vector.

The title to be passed to the program for graphical presentation can be controlled using keyword **`caption`**. If `caption` is omitted, the name of the `root module` is used as the title.

The variable on the x-axis is by default the time in seconds. However, any circuit variable can be set on the x-axis using keyword **`sweep`**.

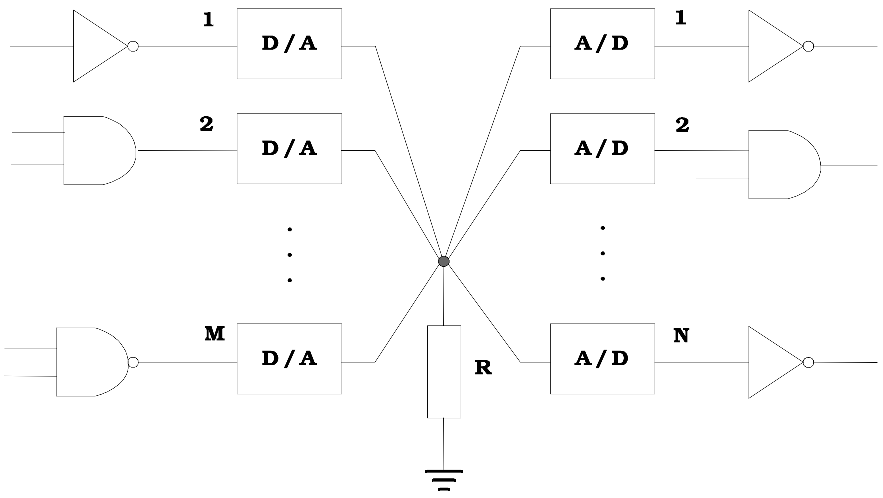

Figure 5.2: Problem of link with the hybrid aspect is solved by inserting converters

A link can have hybrid aspect if it is connected both to analogue and digital components. The case depicted in Figure 5.2 is a general one, since link is connected to analogue components (no matter how many of them) to *M* outputs of logic gates, and to *N* inputs of logic gates. In this case, Alecsis automatically generates *M* D/A converters and *N* A/D converters, as shown in Fig. 5.2. That means, converters are generated for every digital circuit, which is connected to an analogue link via input or output.

Such a link has an analogue aspect (unique analogue value), as *N* digital aspects, viewed from *N* A/D converters, and *M* digital aspects, or viewed from *M* D/A converters. We can print out all the aspects of such link. Analogue aspect is obtained by specifying the analogue link type (`node`, `flow`, etc). By specifying the direction indicator as `in`, and the link type `signal`, we get *N* solutions from the inputs of A/D converters, that is if the indicator is `out` - *M* solutions from the inputs of D/A converters.

```
struct S { three_t send, recv; };
root module test () {
   vgen vg;
   signal three_t v1, v2[4], v3[3:0]="0010";
   node n1, n2, n3, n4;
   module X x;
   signal S s1, s2;
   ...
   plot {
     caption "results of simulation of root module test";
     node n1, n2; //drawn in the same group - common scaling
     node n3; node n4; // drawn separately
     signal v1, v2[2], v3; //signals are always drawn separately
     current vg;          // current trough the voltage source vg
     signal three_t x/y/z/data;
     signal three_t in v1;   // all values           - A/D
     signal three_t out v1(e1), v2;  // all drivers  - D/A
     signal S s1, s2.send;
   }
}
```

Nodes `n1` and `n2` will be shown in the same group, which means they use the same scaling for y-axis. Nodes `n3` and `n4` are stated separately, and will be shown on separated waveforms. Signals `v1`, `v2`, `v3` are also given separately, one beneath the other, which is always the case with digital signals. The whole of the signal `v3` will be shown (from `v[0]` to `v[3]`). Since `vg` is a voltage source, compiler has generated the current of the same name flowing through the source (as in SPICE), which can be accessed using the keyword `current`. The next line defines the printing of the signal named `data`, which is reached using the listed path, as it is not on the root hierarchy level. In this case compiler accept the given link data type of the signal `data`, as the information about the real type is lost after parsing module `z`, where this signal is local (when preparing the data for simulation, simulator checks whether the signal `data` really exists). Supposing that some analogue components are connected to `v1` and `v2`, the following line enables us to get all results of A/D conversion for `v1`. After that the results of D/A conversion for `v2` are demanded, as well as the results of D/A conversion for `v1`, but only for the component `e1`. The last line defines the printing of all members of signal-structure `s1`, and member `send` of signal-structure `s2`.

In some cases, we do not need a value of the link as the simulation results, but the result of some computation with that values (e.g. difference of two node voltages, their ratio, etc.). In this case, instead of the link name, the body of some function that performs the computing is given, and the result is returned using command `return`.

```
plot {
      node double power { return (n1-n2)*vg; };
    }
```

In this way, the result named `power`, calculated as the difference of node voltages `n1` and `n2` multiplied by the current `vg`, is given in the output file. One of the two type declarations - of the link type (`node`) and the data type (`double`) - can be omitted, but not both of them. The declared data type must agree with returned type.

Note: It is more consistent to use only data type declaration (`double`), as power is a new variable, not a new link.

In the above example, computation is simple, but the user can implement more complex function bodies. The example can be rewritten as:

```
plot {
       node double power { double node1, node2, node_diff;
                           node1 = n1;  node2 = n2;
                           node_diff = node1-node2;
                           return node_diff*vg; };
    }
```

The problem arises with the links that are not on the root hierarchy level, i.e. that are not local to the `root module`. The symbol '`/`' used to define the path through the hierarchy is used for division in expressions, and would be understood as such in function body. Therefore, computation can be performed only with values of links local to the `root module`. That means, all links to be used in computation are to be declared in the `root module`, and than passed to submodules when they are invoked. The another way around is to have that links declared only in submodules, but to return their current value using `action` parameter. As already said in section on `action` block, **parameters of the `action` block can be used bidirectionally, i.e. they are passed by reference, unlike parameters of C functions;** and they can be accessed using indirection operator, i.e. **name of the component behaves as a pointer to the structure that comprise its `action` parameters.**

```
module Y (...) {
   node n1;
   action (double p1=0;) {
   ...
   process per_moment {
       p1=n1;
   }
}

root module X {
   Y y;
   y(...);
   ...
   plot { double n1_value { return y->p1; }
}
```

5.6.2. Timing control

timing:

timing *{ time_control }* <;>

time_control:

setting

time_control setting

setting:

parameter = timing_rhs ;

timing_rhs:

constant_expression

parameter = timing_rhs

This construct controls time parameters of a simulation. Simulator recognizes the following parameters.

Table 5.1. Parameters for simulation time control.

Name	Default value	Meaning
`tstop`	/	duration of the simulation (in seconds)
`a_step`	/	starting time step of analogue simulation (in seconds)
`a_stepmin`	`a_step/100`	minimal allowed time step of analogue simulation (sec)
`a_stepmax`	`min(a_step*100, tstop/100)`	maximal allowed time step of analogue simulation (in seconds)
`tprint`	0	printing step (in seconds)

For example:

```
timing {
    tstop=10ms;
    a_step=0.1ns;
    a_stepmax=0.1ms;
}
```

Of all these option only `tstop` applies to digital simulation. Digital simulator advances using the next-event technique. It performs the simulation only when an event happens The result of that simulation are new events, scheduled to happen in some future time. After that, the simulation time advances to the time of the next scheduled event. For that reason, time step does not exist for digital simulation - it jumps from one event to another. The results are printed for every event in output file, so the `tprint` parameter does not have effect, too. Digital simulator performs the simulation for all events scheduled before `tstop`. If there is no event scheduled exactly at time *t*=`tstop`, the simulator repeats the printout of the last state for the time *t*=`tstop` in order to complete the waveforms for the graphical presentation.

For analogue simulation, parameter `a_step` is obligatory, too. This value is used just to begin the simulation, since simulator alters the time step during the simulation, according to the dynamics in the circuit. Time step is chosen to have the maximal allowed value (to save CPU time), so that the accuracy of time-domain simulation is within limits determined by the tolerances `abs_LTE` and `rel_LTE` (see the following section).

The time step is kept in limits (`a_stepmin`, `a_stepmax`). Parameter `a_stepmax` is used to avoid to big time steps that makes waveforms, although accurate, to look discontinuous. The default value is hundred times bigger than the given `a_step`, but `a_stepmax` cannot be bigger than `t_step/100`. Parameter `a_stepmin` enables us to avoid too small time steps. Time step appears as denominator in reactive component models, and to small value can create numerical problems. Besides, if the time step is too small, the simulation can last very long, and the reason might be unimportant - for instance, rapid change of voltage on some parasitic capacitances. For that reason, it is useful to limit the smallest value of time step. However, if the simulator reaches value `a_stepmin`, the simulation error is not within given tolerances. The simulator issues warning message suggesting decreasing value of `a_stepmin` or increasing tolerances for numerical integration `abs_LTE` and `rel_LTE` (or, in case the circuit contains ideal switches, tollerances `SC_vtol`, `SL_itol`, or `SDDT_tol`).

Value of parameter `t_print` gives the minimal time difference of results printed in output file. It is very useful to limit the number of printed points on waveforms, since to big number of points results in very large output files, and often does not contribute to the readability of results. Default value is 0, when all computed time points are printed out.

5.6.3. Simulation options

Options control only the analogue aspect of the simulation. Here, many parameters that control the simulation run can be set. Clearly, this command is optional, as all these parameters have default values. We can divide this set of parameters in four groups:

- control of numerical integration;
- control of iterative process;
- control of sparse matrix solving;
- control of component models;

5.6.3.1. Control of simulation time (numerical integration)

Table 5.2. Control of numerical integration.

Name	Default value	Meaning
`method`	`Gear2` (2)	The method of numerical integration: can be `None` (`0`), `EulerBackward` (`1`), or `Gear2` (`2`).
`abs_LTE`	`1.0e-12`	Absolute tolerance of local truncation error (LTE)
`rel_LTE`	`0.001`	Relative tolerance of local truncation error.
`SC_vtol`	`1mV`	Accuracy of voltages of switched capacitors.
`SL_itol`	`1uA`	Accuracy of currents of switched inductors.
`SDDT_tol`	`1000`	Accuracy of quantities that are numerically integrated using `eqn` command if switches exist in circuit.

An example of statement `options` is:

```
options {
        method = EulerBackward;
        abs_LTE = 1.0e-11;
        SDDT_tol = SC_vtol = 0.01;
}
```

Note: Values of parameter `method` - `None`, `EulerBackward` and `Gear2` are actually integer values, defined in standard Alecsis header file `alec.h`. In that file, it is defined:

```
#define     None            0
#define     EulerBackward   1
#define     Gear2           2
```

Therefore, to use textual values of parameter `method`, you should have file `alec.h` file included before your `root module` definition, using command:

```
# include <alec.h>.
```

Parameter **`method`** represents the choice of numerical integration formula, used for reactive models (e.g. capacitor, inductor, etc.) The simplest formula is Euler-backward formula (or Gear 1 formula), where time derivative in the time instant t_{n+1} (time instant to be solved) is:

$$\left.\frac{dx}{dt}\right|_{t=t_{n+1}} = \frac{x^{n+1} - x^n}{h^n} \tag{5.1}$$

where x_{n+1} is the integrated quantity in new, ($n+1$)st time instant, and x_n is already solved quantity value from nth time instant. h_n equals time step t_{n+1}-t_n.

Gear2 formula is the most popular formula for electronic circuit simulation, and is default value of parameter `method`. This is a two step formula:

$$\left.\frac{dx}{dt}\right|_{t=t_{n+1}} = \frac{2h^n + h^{n-1}}{h^n(h^n + h^{n-1})}x^{n+1} - \frac{h^n + h^{n-1}}{h^n h^{n-1}}x^n + \frac{h^n}{h^{n-1}(h^n + h^{n-1})}x^{n-1} \tag{5.2}$$

where solutions from two previous time instants, as well as two last time steps are used.

If parameter `method` equals `None`, numerical integration is not performed. That means that capacitors are treated as open circuits, and inductors as short circuits. This option is usually useful in testing circuits, to see how the circuit behave with same input, but without (parasitic) reactive elements.

Note: If `method` equals `None`, numerical integration is not performed, and the time step remains constant throughout the simulation, with the user defined value `a_step`.

If you need to have constant time step, with reactive elements taken into account, you should state in timing command:

```
a_step = a_stepmin = a_stepmax = ...;
```

Parameters `abs_LTE` and `rel_LTE` are absolute and relative tolerance of numerical integration. LTE is the local truncation error, i.e. error in one time step. For Euler-backward formula, local truncation error can be estimated as:

$$\delta = \frac{h}{2}\frac{d^2x}{dt^2} \tag{5.3}$$

where h is current time step, and the current value of the second derivative is numerically estimated. For Gear2 formula, LTE is estimated as:

$$\delta = \frac{h^2}{3}\frac{d^3x}{dt^3} \tag{5.4}$$

The tolerance ε is calculated as:

$$\varepsilon = \varepsilon_T \max\left(\mathsf{rel\_LTE}\left|\frac{dx}{dt}\right| + \mathsf{abs\_LTE}, \frac{\mathsf{rel\_LTE}\,|x|}{h} \right) \tag{5.5}$$

where expression $\mathsf{rel\_LTE}\left|\frac{dx}{dt}\right| + \mathsf{abs\_LTE}$ takes into account both relative and absolute tolerance of local truncation error. Time derivative is numerically estimated. Expression $\frac{\mathsf{rel\_LTE}\,|x|}{h}$ is the correction that takes into account numerical error in iterative process (it increases ε for small time steps *h*, otherwise error in computing can lead to further decreasing of *h*). Value of ε_T is the correction factor, which is set to 10.

Error δ is compared to ε for every reactive element (*x* is capacitor voltage, inductor current, or value whose derivative is calculated in `eqn` command). If δ > ε for at least one reactive element, the solution is discarded. (*n+1*)st time instant is calculated again, with shorter time step. We shorten the time step to set δ to be nearly equal to ε, which gives maximal time step, and the error is still within tolerances. For Euler backward formula, when calculation using eqn. (3) is used, this gives new time step as:

$$h^{n+1} = \max\left(\frac{2\varepsilon}{\frac{d^2x}{dt^2}}, \frac{h^*}{10} \right) \tag{5.6}$$

For that calculation, *x* with highest error δ is used. h^* is the discarded time step, which means, that the time step cannot be shortened more than 10 times. For Gear2 integration method, usage of eqn. (4) gives:

$$h^{n+1} = \max\left(\frac{3\varepsilon}{\sqrt{\frac{d^3x}{dt^3}}}, \frac{h^*}{10} \right) \tag{5.7}$$

If δ < ε for all reactive elements, error is smaller than the tolerance, and the solution in the current, (*n+1*)st time instant is accepted. Counter *n* is increased, and next time step is increased. For Euler backward method, this new time step is calculated using again (critical) quantity *x* with highest error δ as:

$$h^{n+1} = \min\left(\frac{2\varepsilon}{\frac{d^2x}{dt^2}}, 2h^n \right) \tag{5.8}$$

and for Gear2 method using:

$$h^{n+1} = \max\left(\frac{3\varepsilon}{\sqrt{\frac{d^3x}{dt^3}}}, 2h^n \right) \tag{5.9}$$

which means that the new time step cannot be more than two times longer than the previous.

Parameters `SC_vtol`, `SL_itol` and `SDDT_tol` are used for circuits with ideal switches. Alecsis posses a built-in model of ideal switch, which is unique for this simulator. It has zero resistance for closed switch and infinite resistance for open switch, and every topology of circuit is allowed. Circuits can be nonlinear, too.

For circuits with capacitors and switches, circuit should be simulated exactly at the time instant of switch transition, but before the transition occurs, to set the capacitors' voltages to correct values. After switch transition, the circuit has new topology, but the capacitors "remember" the voltages before switching . If some internal circuit voltage controls the switches, the time of switching is not known in advance, as that internal circuit voltage is obtain as the result of simulation, too. The time instant of switching is found by an iterative process. Of course, the exact time instant of switching cannot be found, so we have to introduce some tolerance. As the accuracy of capacitor voltage is in question, we have introduced `SC_vtol` as maximal allowed difference of capacitor voltage in two last solved time instants before the switching occurs. If all capacitors have the change of voltage below `SC_vtol`, we consider that we have the capacitors' voltages correct enough, i.e. we have found the switching instant correctly enough. If at least one capacitor voltage have faster change rate, solution after switch transition is discarded, time step is reduced 5 times, and the simulator searches again for the switching instant. Parameter `SC_vtol` has no effect if the circuit does not have inductors or ideal switches.

Parameter `SL_itol` is used following the same philosophy, but for the accuracy of inductor current in the moment of switching. It has no effect if the circuit does not contain both inductors and ideal switches.

Parameter `SDDT_tol` is used to maintain the accuracy of the quantity that is differentiated in the eqn command, if the circuit contain ideal switches. Since `eqn` command is used for user-defined models, the simulator does not have a clue about the order of magnitude of that quantity. Therefore, default value of parameter `SDDT_tol` cannot be set to some usually needed value. (For built-in components, like capacitor or inductors, order of magnitude of electrical quantities is known, so the default values can be set.) `SDDT_tol` has default value of 1000, which is usually too high to have any effect. It should be set by the user, according to the actual application of the model.

5.6.3.2. Control of convergence (iterative process)

Table 5.3. Control of iterative process.

Name	**Default value**	**Meaning**
`vtol`	1μV	Absolute tolerance for node voltage.
`itol`	`1nA`	Absolute tolerance of branch current.
`qtol`	`1.e-20 C`	Absolute tolerance of (capacitor) charge.
`reltol`	`0.001`	Relative tolerance for all variables in the system of equations.
`maxiter`	`10`	Maximal number of iterations in one time instant.
`dump`	`0`	If different from 0, iterations are dumped.
`k`	`10`	Goes with `dump`. Iteration dumping factor.

`dcon`	`0`	If set to `1` helps the convergence in the first time instant (*t*=0) by adding (temporary) conductance between every circuit node and ground. If set to `2`, helps the convergence during whole transient simulation.
`max_weight`	`1.e-4`	Goes with `dcon`. Maximal conductance added to the main diagonal.
`min_weight`	`1.e-12`	Goes with `dcon`. Minimal conductance added to the main diagonal.
`p`	`6`	Goes with `dcon`. Parameter for calculating conductance in the next iteration. Higher value of `p` means that conductance value decreases faster.
`q`	`0.5`	Goes with `dcon`. Parameter for calculating conductance in the next iteration. Higher value of `q` means higher influence of current iteration number *m*, i.e. slower decrease of admittance value.
`maxdcon`	`10`	Goes with `dcon`. The maximal number of cycles.

An example is:

```
options {
        itol = 1.e-12; maxiter = 20; dcon = 2;
}
```

The first group of parameters for convergence control consists of tolerances. Alecsis checks both relative and absolute tolerances, using expression:

$$\left|x^{m+1} - x^{m}\right| < \left|x^{m}\right| \text{reltol} + tol \qquad (5.10)$$

where *m* is iteration number, x^m and x^{m+1} are quantity values obtained in two last solved iterations, `reltol` is the relative tolerance (same for all quantities) and *tol* is the absolute tolerance. From expression (10) one can conclude that for small values of x^m, absolute tolerance is checked, and for big values of x^m, relative tolerance is checked. For node voltages, parameter *tol* equals `vtol`, for branch currents it is `itol`, and for charges it is `qtol`. For links declared as flows, *tol*=0, as the simulator cannot estimate order of magnitude of non-electrical quantity. That means that only relative tolerance is checked for flows.

For every link, user can define its own absolute tolerance during its declaration:

```
node [0.001] v1, v2;
flow [1.e-5] pressure;
```

For `v1` and `v2`, *tol* would be equal 0.001, while for other nodes in the circuit, `vtol` would be used. For `pressure`, absolute tolerance would be 1.e-5. It is very useful to set absolute tolerances for all flows during their declaration, as relative tolerance check is not reliable for small values of x^m.

If (10) is satisfied for all links in the system, convergence is reached, and simulation can proceed to the next time step (*n* is increased, *m* is reset to 0). If (10) is not satisfied for at least one link, analysis is repeated for the next iteration. Counter *m* is increased, and nonlinear models are updated (calculated for new iteration).

Parameter `maxiter` limits the number of iteration per one time instant. It should not be a very big number. If simulator needs big number of iterations, it is very probable that the time step was too big, and the simulation result would be discarded because of local truncation error. Therefore, it makes sense to give up before reaching the convergence, and to shorten the time step. In such case, the time step is shortened 4 times in Alecsis.

If $|x^{m+1}|>10^{30}$ or $|x^{m+1}-x^{m}|>10^{30}$, Alecsis consider that there is an overflow, and the condition (10) is not checked at all. The simulator considers that also as no convergence case, and shortens the time step 4 times.

Alecsis posses two additional mechanisms to help the convergence. They are iteration dumping and node "grounding".

The user introduces iteration dumping by setting the option `dump`. When the dumping is introduced, starting point for the (m+1)st iteration is calculated as:

$$v^{m+1} \Leftarrow v^{m} + f(v^{m+1} - v^{m}) \tag{5.11}$$

where $f(x)$ is the dumping function, chosen so that $|f(x)|<|x|$. When no dumping is introduced, $f(x)=x$, which means that the starting point for (m+1)th iteration is v^{m+1} (solution from mth iteration). Dumping adds only part of the increment to v^m. In Alecsis, dumping function is implemented as:

$$f(x) = \frac{sign(x)}{k}\ln(1+k|x|) \tag{5.12}$$

This gives stronger dumping for bigger increments x. Eqn. (11) is applied on every vector member. Parameter k can be set by the user, and should be in the range (1,20).

Iteration dumping is useful if models in the system express strong nonlinearity (e.g. exponential), but it can slow convergence in other cases.

Another method of solving convergence process is node grounding. If grounding is used, conductances are connected from every node to the ground, and across PN junctions in the circuit. With high conductances, there is no doubt that the iterative process would converge easily. However, such solution is not correct, as this is not the original circuit. That is only an intermediate solution, which can be used as the starting point for solving the new circuit, where conductances are lower. In this way, conductances are decreased until their value is negligible, so we get the solution of the original circuit.

This option is activated if convergence is not obtained in `maxiter` iterations. Conductances are set to w=`max_weight`. If the solution converge, conductances are decreased in the following manner:

$$w \Leftarrow w10^{-\frac{\mathrm{p}}{1+s+\mathrm{q}m}} \tag{5.13}$$

Conductances are multiplied by factor smaller than 1. Parameters `p` and `q` can be changed by the user. With higher value of `p` conductances decrease faster, while `q` makes bigger influence of the number of iterations m. If m is higher, conductance decrease more slowly. Paramater s is firstly set to 0. After every successful convergence, eqn. (13) is applied again, until conductances reach value `min_weight`, when they are considered negligible.

If convergence is not reached for some value w, conductances are set to their previous value, s is increased by 1, and eqn. (13) is applied again. With higher s, conductances decrease slower. As this process can last very long, s is limited to 10, and number of successful application of eqn (13) to `maxdump` (if convergence is not reached, that trial is not counted).

Mechanism of node grounding shows to be very efficient and is able to help in solving most of the problematic circuits. If `dcon` is set to 2, this mechanism is applied whenever number of iterations m reaches `maxiter`. However, you should be aware that node grounding slows down the simulation considerably. Time step shortening, which is normally performed by Alecsis when m reaches value `maxiter` is usually faster.

If `dcon` is set to 1, node grounding is applied for the initial time instant (*t*=0) only. For this time instant, convergence probems are likely to occur, as a "good guess" from the previous time point solution is not available. There are other ways to help the simulator to find the initial solution. If you set:

```
node n1=6; n3=4;
```

when you are declaring these nodes. In this way, starting values for the iterative process in *t*=0 are set. (If you declare these nodes using:

```
node n1<-6; n3<-4;
```

you can set solution for the initial time instant, i.e. circuit is not solved for *t*=0 at all - user solution is accepted.)

One can monitor conductance values using verbose level 8. If Alecsis is invoked using:

```
alec -v8 circuit_name.ac
```

Alecsis prints out information on trial number, applied conductance value, and parameter *s* value. This can be useful information for adapting values of parameters.

Note: Conductances are applied only on nodes and on PN junctions in the current version of Alecsis. If you have nonelectrical system, and unknown quantities are declared as flows, option dcon cannot help.

Note: If some convergence problem occurs, options `dump` and `dcon` should **not** be used readily. **The reason for convergence problem is very often some error in the circuit or in models.** Therefore, you should firstly check your description.

5.6.3.3. Control of system of equations solver

Table 5.4. Control of sparse matrix solver.

Name	Default value	Meaning
`renum`	`Best` (2)	Quality of sparse matrix renumeration algorithm. It can be None (`0`), `Fast` (`1`), or `Best` (`2`).

There is only one option that controls sparse matrix solver, and that is `renum`. It can change the CPU time necessary for simulation. The number of nonzero elements in the system matrix generated during LU decomposition depends on the ordering of matrix rows and columns. This reordering is performed only once, at the beginning of simulation.

If you chose option `Best`, a variant of Berry's algorithm is used, when very detailed (and slow) reordering is performed. This is the default value, as reordering is performed only once, and good reordering guaranties fast simulation. With option `Fast`, a variant of Markowitz's algorithm is used, when reordering is performed much faster, with somewhat slower simulation in time domain afterwards. This option should be chosen for very large matrices (several hundreds of equations or more), since with Berry's algorithm, reordering can take more CPU time than time-domain simulation. When option `None` is chose, no reordering is performed. This is implemented for comparison only, it has no practical effect, since simulation can take too much time.

Note: Values of parameter `renum` - `None`, `Fast` and `Best` are actually integer values, defined in standard Alecsis header file `alec.h`. In that file, it is defined:

```
#define     None       0
```

```
#define     Fast       1
#define     Best       2
```

Therefore, to use textual values of parameter `renum`, you should have file `alec.h` file included before your `root module` definition, using command:

```
# include <alec.h>.
```

5.6.3.4. Control of models

Table 5.5. Control of component models.

Name	Default value	Meaning
`charge_model`	`0`	Applicable to built-in BSIM model of MOS transistor only. It can be 0 or 1.
`temp`	`300.0`	Ambient temperature.

Example:

```
options { temp = 400; }
```

Option `charge_model` is applicable to BSIM model of MOS transistor that is built in Alecsis. a nonlinear capacitance can be modelled correctly only over charges. If this is not performed, problem known as *charge non-conserving* can appear. In this way, new unknowns (charges) are added to the system of equation. Nevertheless, charges can be mathematically eliminated from the system of equations, when the model is still correct but the system of equation is smaller.

If `charge_model=0`, charges related to every terminal of MOS transistor are not appearing in the system of equations. As there are four such charges associated to every MOS transistor, size of the system of equations can be much smaller. If `charge_model=1`, system of equation is bigger, but accuracy of simulation is better controlled, as the convergence is checked also for charges, using parameter `qtol`.

Option `temp` sets the value of ambient temperature. It is passed to all built-in models in the system, where model parameters are recalculated for given temperature (as in SPICE). Temperature is given in Kelvin degrees. Besides, value of this option can be used in user defined models. If keyword `temp` appear in some expression in the code, it represents the temperature value set using command `option`.

In the current version of Alecsis, option `temp` does not work for built-in (SPICE-like) models. As temperature dependence is already programmed in built-in models, we would probably improve that very soon.
For user-defined models, option `temp` works correctly, i.e. keyword `temp` used in model code returns correct temperature.

5.7. Model cards

Action parameters enable for every component to receive certain parameters that determine its behaviour in the circuit. If the number of parameters in not very big, this is an easy-to-use method. However, with the increase in the number of parameters it becomes tedious to write parameters for every component separately, especially when more components share the same parameter values. It is much more convenient group those parameters, give the group a name, and then associate that name when connecting the component. This is the concept of model cards. This concept is familiar from the simulator SPICE, and is improved in AleC++ to allow object-oriented modelling.

AleC++ supports SPICE syntax of model cards for several analogue components - MOSFET, BJT, JFET, diode. This enables usage of available SPICE model cards in Alecsis. Keyword `spice` is used to switch on SPICE model card syntax.

```
module inverter (output, input, vdd, vss) {
   mosfet mup, mdown;

   mup (output, input, vdd, vdd) { model = MNPMOS; l=2u; w=6u; }
   mdown (output, input, vss, vss) { model = MNNMOS; l=w=2u; }
}

spice {               // transition to SPICE syntax
* SPICE syntax comments
.MODEL MNPMOS PMOS ( LEVEL=1
+ VTO = -0.92V GAMMA=0.9 LAMBDA=0.1 )

.MODEL MNNMOS NMOS ( LEVEL=1
+ VTO = 0.87V GAMMA=0.67 LAMBDA=0.078 )
}                     // back to AleC++ syntax
```

Cards formed this way can be associated desired number of times when connecting components, by listing the model name after the special parameter **`model`**. The name of the model is an external symbol. Therefore, the model card itself can be given before or after referencing in the text, or can be stored in a library. As was the case with modules, when referencing the name of a model you can specify the library name to solve conflicts with double names.

```
model = ttl_lib.short_nmos;
```

5.7.1. Model cards as static objects

SPICE syntax of model cards is used for built-in components. User-defined components must also have some way of grouping parameters into model cards. SPICE syntax of model cards is not convenient here, as user-defined models can be very complex, can be even composed of submodels that have their own model cards. And there is no need to follow SPICE syntax - user-defined models are not necessarily electrical.

AleC++ model cards are created using **classes**. C++ - like class is very convenient for this purpose. If the user wants to create a new type of model card for the new type of element (created by defining a module), he or she needs to do the following:

- ➢ Supply the information about the names and types of all parameters that can appear in the model card.
- ➢ Provide a mechanism that will allocate memory for the parameters (if they are pointers) and a mechanism to set parameters to default values.

- ➢ Provide a mechanism that will test the parameter values (if they are supposed to be in some range) before the simulation, and if necessary preprocess them before the simulation.
- ➢ Provide a mechanism to free the memory of a parameter-pointer
- ➢ Make parameters of a model card, associated with a module, visible in the functional part of the module.

Most of these reminds us of classes, constructors, destructors, and the visibility rules. This makes C++ syntax of classes a natural choice for model card definition.

Creating a new model card in AleC++ is the same as creating a class. Members of a class are parameters in the card. Constructor sets their initial values (default values of model card parameters), and destructor frees the memory. As in C++, constuctor has the same name as the class, and the descrutor the same name with prefix '~'. The only difference from C++ -like class is the new special function -preprocessor. Preprocessor is necessary, since testing of the parameter value range and the preprocessing cannot be performed in the constructor. Constructor is activated when the component is declared, and it gives default parameter values. Preprocessor has to be activated when the specific model card is used for the given component, and that is performed in SPICE-like style, when connecting the component. This new method has the same name as the class, but with the prefix '>'. Preprocessor does not return any result, and it does not accept any parameter.

We use the specific model card, i.e. set of parameter values, by invoking that model card when connecting components. However, we have to associate the component model with the given model card type before, when defining the component model. This is necessary to make the parameters defined in the model card visible when defining the component model. Association of a model card and a new module is performed using the name of a class, and the operator of the access resolution '`::`'.

```
class new_diode {
        double is;              // parameter is
        double eta;             // parameter eta
    public:
        new_diode();    // constructor - cannot be inline
        ~new_diode();   // destructor - not necessary
        >new_diode();   // processor
        double evaluate_current(vd);            // diode current
};

module new_diode::ndio (plus, minus) {
    ...
    action {
        process per_moment {
            double current;
            current = this.evaluate_current(plus-minus);
            ...
        }
    }
}
```

This example illustrates a few important characteristics:

– Special methods of a model class (constructor, preprocessor, destructor) cannot be `inline` because they are called by the simulator itself.

– It is recommended for the parameters to be private. More than one component can associate the same model card, so if more of them change the parameter values, an error can be created that is very difficult to debug.

– Note the difference in access rights between the **methods** of one class and **modules** that are associated to the class. Both module and models can access parameters either directly, or using the keyword `this`, since they have the same visibility area. However, **only methods have the right of access to all members of class**; `module ndio` from the example cannot access `private` or `protected` members. Therefore, all calculations (model equations) should be defined as `public`, so they can be accessed from the `process` code (as `evaluate_current` in the example above). Modules should use these methods, not the parameters directly. Note that all other methods, except special, can be `inline` (recommended for smaller functions).

Some users can find these limitations too restrictive. Restrictions are here to make the probability of an error smaller, and to narrow the space where you should search for an error. Nevertheless, you can declare a `module` as a `friend` of a model class. Such module has the right of access to all members.

```
class new_diode {
        ...
        friend module ndio;
};
```

If a module is to be associated to a model card, that has to be added to the module declaration. If a parent module accepts the same model card as the given module, the model card can be omitted during the declaration. In such case, the module uses the model card from the parent module. In this way the card can be passed down the model hierarchy, which can very useful for complex models.

5.7.2. Syntax of model card

Model cards are defined on the global level and are subject to external linking. The syntax of the definition is

model_card:
  model *class_name :: card_name <(<arguments_for_constructor>)> {<card_body>}*

card_body:
    parameter_setting
    card_body parameter_setting

parameter_setting:
    model_lhs = model_rhs ;

model_lhs:
    <class_name::>parameter_name

model_lhs . structure_member
    model_lhs [constant_expression]

model_rhs:
    initializing_pahrase
    model_lhs = model_rhs

We have explained how the model card type is defined. We have to explain how the particular instances of the model card (sets of parameter values) are given. The keyword `model` is used, followed by the association of the model card class and the particular model card name using the operator of resolution '`::`. (see the example below). If the constructor with arguments is used, the list of arguments in parentheses follows that association.

The body of a model card is in parentheses ('`{`' and '`}`'). The card consists of series of commands of assignment, which set the initial values of parameters. The same set of rules applies for initialization of parameters as was the case with the initialization of static objects (it is possible to do an aggregate initialization of composites and structures). You can initialize more parameters using the same command (e.g., `a=b=c=2;`). You can also initialize only a part of a vector or a structure. If a model card is derived (i.e. class is derived), that is inherits one or more base ones, and if some parameters share the same name, you can explicitly give the name of a class and operator and the access resolution operator '`::`' to clear which parameters is to be used. If some parameters are not assigned a value in the model card, they keep their default values given in the constructor. The whole model card can be empty, when all parameters keep their default values.

```
struct S { double a, b; };

class X {
        int a;
        double b;
        char *s, v[20];
        S s1, s2;
        double m1[2][2], m2[2][2], m3[2][2];
    public:
        X (int);
        ~X ();
        >X ();
};

model X::x (2) {           // definition of model card x of type X
    a = 2; b = 5.6; s="string1"; v="string2";
    X::s1 = { 2.2, 3.5 };
    s2.a =4.7;     s2.b = 6.8;
    m1={ {1,2}, {3,4} };  //automatically converted to type double
    m2[0] = {1,1}; m2[1] = {0, 3};
    m3[0][0]= m3[0][1] = m3[1][0] = m3[1][1] = 5.6;
}

module X::M () {            // module M uses model card type X
    module X::Y y1, y2, y3; // module Y uses model card type X
    module Z z;
    module K::MK k1, k2;    // module MK uses model card type K

    y1() model = x;    // O.K. - association of model x
    y2();              // O.K. - card inherited from a parent (M)
    z() model = x;     // ERROR - Z does not accept model cards
    k1() model = x;    // ERROR - MK accepts class K, not X
    k2();   // ERROR - parent takes class X, k2 expects class K
    y3() private model = x;   // copy, not a reference of model x
}
```

Association of a model card is passing by reference - address of the model card is accessed. That means that **no copy** of the model card is created when it is associated during connection of some component. Model cards are normally only read during the simulation, parameter values are not changed. By creating copies, we would spend memory without any need. However, someone can create class methods that change the values of parameters during the simulation run, which can cause problems with other components referencing it. In that case, it is better to make a **copy** and not reference using the keyword `private` before the word `model` (see connection of component `y3` in the example above). A copy of the model card `x` would be created, and constructor and preprocessor will be applied. Whenever the keyword `private` is used, a copy of the card is created. The rest of the components will have the reference to the original one. Compiler makes a shallow copy, but can do the deep copy if you define the copy constructor - `X(X&)`. By creating private copy of the model card, we are sure that changes in a private card do not affect other components. The price for this is higher usage of memory.

Note: In this context, keyword `private` is related to creation of private copies, and is not related to the rights of access to parameters - `public`, `private` and `protected` parameters keep their access attributes unchanged.

5.8. Modules with variable structure -- clone and allocate

Modules with variable structure (application of commands **`clone`** and **`allocate`**), are explained in the Chapter on analogue simulation. It applies without any difference to digital and hybrid simulation, too.

5.9. Visibility area (name masking)

Modules create a separate visibility area. Priority rules (masking of the entities with the same name) are the following, starting from the lowest priority level:

- global objects -- the lowest priority (the widest visibility area);
- parameters and methods from model class, if a module accepts a model card;
- formal links, if any;
- local links; components connected in the structural area of the module;
- `action` parameters, if any;
- local variables in the `action` block;
- local variables in `process`es, if any;
- local variables in every new block opened inside a `process`.

The priority increases from top to bottom of the list given above, while the appropriate visibility area decreases. Every declared entity can mask an entity of the same name, which was declared in the lower priority level.

Note: Compiling with option `-O` (using optimizer) gives warning whenever masking occurs.

Masking is not a redefinition, since redefinition means for two symbols of the same name to appear in the same visibility area. Redefinition is treated as error.