6. Digital simulation in Alecsis

The previous chapter has introduced basic terms and concepts used in Alecsis for both analogue, digital and hybrid circuit simulation. We are now going to focus on each group separately. This chapter discusses digital simulation in Alesis.

6.1. Alecsis support for digital simulation

Logic simulation can be performed using different systems of logic states. For each system of states, new logic gates have to be developed. We generalize this as a problem of discrete-event simulation, which is not necessarily electrical. Therefore, it is very useful if the user can create his own system of logic states, as well as logic operators (gates).

Alecsis was designed as an offspring of C/C++. These languages do not have too many specialized constructs, and can wok up to their full potential only in conjunction with various libraries, which cover numerous applications. In a similar manner, Alecsis does not have built-in digital component, or a formal system of states, or number of logic states. It provides only a discrete-event simulation engine, i.e. a mechanism that can process events and queue new events that come as results. If libraries of logic states and basic logic gates are not available this can be serious drawback. However if a library supporting different styles and types of digital simulation is available, this becomes a major advantage. User of already prepared libraries can consider Alecsis as an ordinary logic simulator. But and advanced user can create his own libraries and adapt Alecsis for his problems.

If we want to use Alecsis with a new system of logic states, we have firstly to prepare *program support*. That program support is used later in creation of appropriate logic gates. There are three steps in creation of the program support:

- creating the system of states (new enumeration type),

- creating new logic truth tables,
- overloading of appropriate logic operators, to accept links of the new type
- if model cards are to be used, declarations of model class and definition of methods are necessary.

In this way basis for simulation with new logic system is created, which can be used for creation of models.

6.1.1. Systems of states

System of state of a simulator describes legal states of a digital simulator. Total number of these states is called the cardinal number of the simulator. Many simulators have a built-in system of state. Alecsis does not have such a system, and it needs to be defined before the simulation (often used systems are available in the library).

You need to use enumeration types to define the system of state. To allow for representations of digital words (vectors), system of state needs to be defined with characters as symbols. Examples are:

```
typedef enum { 'x', '0', '1' } three_t;     //system with 3 states
typedef enum { 'x', '0', '1', 'z' } four_t; //system with 4 states
typedef enum { ' '=void, '_'=void,
               'x', 'X'='x', '0', '1', 'z', 'Z'='z'} four_t;
```

The last example defines a type with 4 states, two types of separators for enumeration strings, and is case-insensitive. Enumeration types in AleC++ occupy 1 byte (char), taking that the indexes of symbols are in range -128 to 127, otherwise they take 2 bytes.

Term "system of states" is used conditionally, since the construct `typedef` merely defines a new name for a enumeration set. Thus, an unlimited number of different "systems of states" can exist on the global level simultaneously, taking the names are different.

The order of symbols in the set is important for three reasons:

- If sets are non-initialized, values are given as in C/C++ - starting from value 0 for the first signal.
- Order of states is important for creating truth tables of logic functions;
- Signals (digital links) that are not initialized, have the starting value that is the first symbol in the set. In our example, all signals in the circuit of type `three_t` and `four_t` that are not initialized will have starting state `'x'`, since `'x'` is the first symbol in the set (separators do not count). Therefore, undefined state `'x'` should be always the first symbol, as it is usual to treat non-initalized signal as undefined when the simulation begins.

6.1.2. Truth tables

AleC++ has basic binary logic operators needed for logical simulation. However, they accept integer operands, not enumeration ones. This means that for every new system of state we have to define truth tables for all operators. Truth tables for unary operators are vectors of length N, for binary -- matrices NxN, where N is the number of states in the actual system of states (the cardinal number).

```
const three_t nottab[] =       { 'x', '1', '0' };
                             //   x    0    1

const three_t andtab[][3] = { { 'x', '0', 'x' },   // x
                              { '0', '0', '0' },   // 0
```

```
    { 'x', '0', '1' }    // 1
};
```

6.1.3. Overloading of operators

This tables alone are sufficient, and can directly be used for modelling of a digital circuit. It may be better, however to use operator overload:

```
inline three_t operator~ (three_t op) { return nottab[op]; }
inline three_t operator& (three_t op1, three_t op2)
                                    { return andtab[op1][op2]; }
foo () {
    three_t x = '1', y = '0', z = '1', r;

    r = ~(x & y & ~z);
}
```

Overloaded operators are very simple functions, which use truth tables, and return whatever was on a particular position. Being so short, they are suitable to be `inline`. It is usual for that definition of an `inline` operator to be in the same file with the definition of the system of states. That file can be included as a header file into a new file, where circuit description is, using `include` command:

```
# include <declaration_file.h>
```

Table declaration can be in the same file (as `extern`), but the definition (with the initialization) ought to be in a separate file. That may be a file storing all definitions for a particular system of states, and can be compiled into a library (using `-c` option), by invoking:

```
alec -c definition_file.ac
```

That library can be used later by Alecsis linker. If the user prefers common, and not inline operators, than the declaration of the operators remains in header file, and their definitions are compiled together with table definitions. As you can see, the intention is use C/C++ programming style - to group the declaration into one header file that can be included when necessary, while the definitions are compiled and stored as library.

Assignment operators can be also overloaded. It stores the result of a logical operation into the left operand (`&=`). The left operator need to be passed by reference to the operator function:

```
inline three_t operator &= (three_t& op1, three_t op2)
                           { return (op1 = andtab[op1][op2]); }
```

6.1.4. Overloading operators for vectors

All logic operators can be overloaded for a new system of state using appropriate tables. Nevertheless, if you want simulation on RT level (RTL -- register transfer level), you need to have operators for vector-operands overloaded, too. To make that possible, operator function needs to posses information about the vector dimension. A mechanism that is used for strings (`strlen` function) cannot be used. Symbols in systems of state can have a value of `'0'`, which renders the mechanism of classical strings (with `'\0'` as an *n+1* element) inoperable. For that reason, AleC++ has a special operator **`lengthof`**, which does not exist in C/C++.

6.1.4.1. Operator lengthof

This operator is syntactically very similar to the operator `sizeof`. It is a unary operator whose **operand needs to be a vector of enumeration type**. (vector of signals or vector of variables). The vector can be a formal variable, local variable, or a formal/local signal. In the case of a formal variable, AleC++ passes as a hidden parameter the length of the actual argument extracted by the operator. The situation is much simpler with signals -- a special instruction of the Alecsis virtual processor that returns the length of the signal-vector. Note that operand of `lengthof` operator cannot be a local pointer, because `lengthof` would return 0 in that case.

If a vector is declared with inverse dimensionality, the `lengthof` returns the **negative value** of its length. Usually, for overloading vector-operands, you need the absolute value of the length. Direction can have an important role with some specific operators (left shifting << or right shifting >>).

```
three_t v1[0:3], v2[3:0];
int L1 = lengthof (v1);   // L1 is 4
int L2 = lengthof (v2);   // L2 is -4
```

6.1.4.2. Buffers for temporary solutions

Logical operators often return a result, that can itself be an operand in a more complex expression. Since we are dealing with a vector, a temporary memory location needs to be reserved for the result, and another temporary memory to prevent the overwrite of that result by the new operation. The following example illustrates this:

```
# define Vreturn(v,size) {asm movq.l %d4, size; return v; }

# define BUFF_SIZE 512
three_t _op_buff[BUFF_SIZE], _res_buff[BUFF_SIZE];

three_t *operator& (three_t *op1, three_t *op2) {
    int size1 = abs (lengthof op1);
    int size2 = abs (lengthof op2);
    if (size1 != size2) {
      printf("incompatible lengths - op &, type three_t\n");
      exit(1);
    }
    for (int i=0; i< size1; i++) _op_buff[i] = op1[i] & op2[i];
    memcpy(_res_buff, _op_buff, size1);
    Vreturn(_res_buff, size1)
}

three_t *operator= (three_t *op1, three_t *op2) {
    int size = abs(lengthof op1);
    memcpy(op1, op2, size);
    memcpy(_res_buff, op1, size);
    Vreturn(_res_buff, size)
}

foo () {
    static three_t v1[]="0100", v2[]="11x0", v3[4], v4[4];
    v4 = v3 = v1 & v2;
}
```

The example illustrates a concept of using two buffers -- one for calculations, the other for the return of the result. In this particular example use of only one buffer would be a satisfactory solution. However, in some more complex operations it may happen for one of the operands to be a result of an overload operation, which may mean that we are reading and changing the same vector simultaneously. That may produce errors that are very difficult to

debug. It is better to use two buffers - the result is formed in `_op_buff` buffer, and is then copied into `_res_buff` buffer.

The return of a vector is rather uncommon -- we have used our macro `Vreturn`. Macro `Vreturn` returns not only the pointer to a vector, but also its length, which enables later usage of command `lengthof`. From `Vreturn`, usual command `return` is invoked as:

```
return v;
```

where pointer `v` is returned. Pointer is stored into lower register of the virtual processor accumulator %d0. But before invoking usual command `return`, we have stored the length of the vector into %d4 (the higher accumulator register). Assembler command is used for that. If this result is passed to a function as an operator, operator `lengthof` can determine the vector length from this higher register. Macro `Vreturn` is available in the standard header file "`alec.h`").

With a similar overload of operator =, you can handle enumeration vectors the same way as ordinary numbers. Therefore, the user of the logic simulator, which has a library already prepared, can be unaware of both `lengthof` operator and temporary buffers, as in function `foo` in the example above.

It is up to the designer if he or she is going to allow for the combination of vectors of different length and/or different direction in such binary operators.

Unary operators of left and right shift (<< and >>) can be overloaded in this manner, as well as increment, and decrement (++ and --), etc.

6.2. Synchronization of digital processes

Such program support, consisting of prepared systems of states, logic truth tables, and appropriate operators is now available to model digital components. On the other hand, we have available an internal *Alecsis discrete-event selective-trace* simulation engine, that is able to manage the logic events. The construct *discrete-event* means that the simulator does not solve the time-continuum, but only time instants when a logic event (change of logic state) happens. Outside of these time-instants, there are no changes in the circuit, i.e. nothing for the simulator to solve. *Selective-trace* means that only some models are executed when an event happens. When an event is generated at the output signal of some digital component, it activates only those components where the same signal acts as input. Therefore, there is no need to simulate the whole circuit, only activated components (models) are simulated. Therefore, we can consider execution of logic simulation as execution of *synchronized processes*, and *transmission of messages via signals*.

When we are defining digital models, we are using the program support from the library. We have to provide the information for the simulation engine how to synchronize the processes. Here is an example:

```
module and2 (three_t in a, b; three_t out y) {
    action (double delay) {
        process (a, b) {  y <- a & b after delay;  }
    }
}
```

We have modelled logic AND circuit with two inputs for system of states `three_t`. This `module` does not have structural constructs, i. e. we are not using previously defined components as submodels of this model. The functional part consists of one `process` sensitive to events at the inputs -- signals `a` and `b`. Whenever an event happens to either one of them, the `process` activates, and executes the signal assignment command (operator `<-`). We have used logic operator `&`, overloaded to accept signals of type `three_t`. The result will become the new value of signal `y`, but not immediately. A delay of `delay` seconds is introduced. Since that is the last command of the process, the control is transferred to the beginning, and the process is suspended until the next event on some of the inputs.

6.2.1. Interpretation of signals in expressions

The name of a signal can legally appear in the expressions of the process. (Be aware about the masking rules given in the previous chapter.) However, they must be treated differently than C/C++-like variables. Links, and therefore signals as digital links, are complex entities, which can be differently treated in different situations**. The treatment of the signal depends on the context.**

In all **non- assignment expressions**, the name of the signal refers to **the memory storing the current value of the signal**. The size, dimensionality, and the type of the memory depend upon the type of the signal (scalars, vector, or a structure). In other words, in such expressions, the name of the signal refers to it value - it is used as a variable.

Signals must not appear on the left side of the assignment operator (such as =, =+, ++, etc.) since they are not l-values. The change of the signal value can be performed exclusively using a special AleC++ assignment operator for signals `<-`. (Note that symbol `<-` is used differently for analogue links, where it can be used in analogue link declaration to set its value for the first time instant (t=0).) The simulation engine is responsible for assigning the value of the signals, as they are quantities that appear in the circuit connections. This information is given to the simulator by the use of the assignment operator `<-`. You cannot just write into signal.

The rules addressing the agreement of types, access to the members of structures, indexing of vectors, etc. are the same as for variables.

```
typedef enum { 'x', '0', '1' } three_t;
struct S { three_t send, recv; };
module X (three_t a, b[], c[][10]; S s1, s2[]) {
   action {
      process (a,b) {
         a;         // value of signal a - scalar
         b;         // pointer to position b[0] of vector b
         b[2];      // scalar  from position 2 of vector b
         c;         // matrix c (pointer to c[0][0])
         c[2];      // vector of length 10, position 2, matrix c
         c[2][3];   // scalar, position 2,3 matrix c
         s1;        // structure, type S
         s1.send;   // scalar - element of structure s1
         s2;        // vector of structures
         s2[1];     // structure at position 1, type S
         s2[1].send;//scalar - element of structure at position 1
      }
   }
}
```

Formal signals that are arrays do not need to have a specified first dimension (as `b[]`, `c[][10]`, and `s2[]` in the example above) since that dimension is not necessary for indexing (operator `lengthof` will determine it with enumerated variables) **Nevertheless, all other dimensions are necessary for arrays with more than one dimension.** Formal signals do not exist independently in the memory. Only global or local entities exist as independent entities in the memory. Formal signal is just the name used for matching the appropriate pair on the actual side.

Signals can be scalars, if composite homogeneous (vectors, matrices, etc.), and heterogeneous (structures), but they are also internally represented as collections of scalars. For that reason is the `process` that is sensitive on a signal-vector, actually sensitive to every scalar making the signal-vector.

6.2.2. Conversion of link type

If a link has more than one aspect, i.e. it is both digital and analogue, the **operator of link type conversion** can solve the ambiguity. This operator is very similar to C/C++ **cast** operator used for type conversion.

```
signal s;
action {
    process {
        double s_as_node;
        s_as_node = (node)s;
    }
}
```

We can perform link type conversion by stating (in parentheses) the desired type before the signal name. Priority and the associating rules of this conversion operator is the same as was with cast operator.

Note: A wrong assumption that signal `s` possesses analogue aspect will result in a usually harmless value of 0 during the simulation. However, if we try the reverse, to convert the node into a signal, where the node does not have digital aspect, effect can be harmful.

6.2.3. Conditional process suspension -- command wait

We have already said in the previous chapter that a `process` can be synhronized by giving the list of signals it is sensitive to. Another method of synchronizing a `process` is using command **`wait`**. Command `wait` suspends the current `process` for a period of time, giving t the same time conditions for its reactivation.

wait_command:
wait <*sensitivity_list*> <**while** *suspension_condition*> <**for** *time-out* >*;*

An empty command is also legal:

```
wait;
```

In this case, the `process` is suspended until the end of the simulation. The *sensitivity_list* is the same list of signals that can be defined in the `process` heading (the `process` using `wait` command must not be synchronized using some other method). The `process` execution stops at the `wait` command, until an event happens to some of the signals the command is sensitive to. If *suspension_condition* is given, it has to become zero, if the `process` is to activate, regardless of the events on some of the signals from the *sensitivity_list*. If *time_out* is defined, all of that can last up to *time_out* seconds.

```
enum Edge { RisingEdge, FallingEdge };

module (three_t in clock) {
    action (Edge mode) {
        process {
            ...
            switch (mode) {
                case RisingEdge:
                    wait clock while clock != '1' for 100ns;
                    break;
                case FallingEdge:
                    wait clock while clock != '0';
                    break;
            }
```

```
            // code activates at the edge of clock signal
            ...
        }
    }
}
```

In this example, `wait` command is sensitive to the clock signal, with the condition of a transition to `'1'` (if `action` parameter `mode` equals `RisingEdge`) or `'0'` (if `mode` equals `FallingEdge`). The appropriate edge of the clock activates the code that follows (omitted in the example above). At the end of the process, the control is back at its beginning, and the execution stops again at `wait` command, waiting the appropriate clock edge.

6.2.4. Predefined signal attributes

All signals, regardless of their data type, have a number of attributes, whose value can be accessed in a `process`. All attributes are of `int` type, and can be accessed using indirection operator `->` (signal name is treated as a pointer, i.e. address in memory). Signal attributes are:

- **`active`** -- 1 if the signal is active in the present moment, otherwise 0
- **`event`** -- 1 if an event is currently happening on the signal, otherwise 0
- **`quiet`** -- 1 if the signal is not processed in the present moment (`!active`), otherwise 0
- **`stable`** --1 if the signal is not changing value (`!event`), otherwise 0
- **`fanin`** -- number of processes sensitive to this signal
- **`fanout`** -- number of drivers of this signal
- **`hybrid`** -- 1 if the signal has also analogue aspect, otherwise 0

It can happen that the signal is processed, but the result is the same value as it was before. Processed signal is `active`, while an `event` happened to it only when the new value is different from the old one. The term sensitivity has already been explained, while the concept of drivers will be explained in the following section. The last attribute indicates the `hybrid` aspect of the present signal, that is the coupling of analogue elements (which can have effect on delay calculations and other activities).

Note: The indirection operator `->` can be used without restrictions, since the members of signals-structures can be accessed using character `'.'`, and **the signal cannot be declared as a pointer**.

Composite signals have attributes `active`, `event`, `quiet`, and `stable`. Attributes `hybrid`, `fanin`, and `fanout` are not allowed it the signal is not a scalar. Attributes `active` and `event` are obtained as a result of *or* operation on attributes of all scalars of the signal (for example, a vector is `active` if at least one position is `active`). Attribute `quiet` is negation of `active`, and `stable` is negation of `event`.

```
signal four_t v[4];

...
v[0]->fanin;  // o.k.
v->fanin;     // error - composite signals cannot that attribute
```

We use attributes in the following example for detection of static hazard - simultaneous and opposite change of the inputs of an AND circuit).

```
module and2 (three_t in a,b; three_t out y) {
   action (double delay) {
      process (a,b) {
         if (a->event && b->event && a != b)
            warning ("static hazard at inputs of AND2 circuit");
         y <- a & b after delay;
      }
   }
}
```

6.2.5. Signal assignment -- operator '<-'

The change in the signal value is possible only using operator <-. The difference between this, and an ordinary assignment of some value to a variable, is in that the ordinary assignment happens instantaneously, that is the next reference to the variable gives the new value. Assignment of signals is not performed instantaneously, it is information for the simulation engine to give a new value to the signal. It has a certain delay, so that the new value cannot appear before the next cycle of simulation. Such assignment can be delayed so the new value appears after a certain period of time expires. The delay models the inertia of the signal through a physical component in digital simulation (in analogue simulation, delay occurs indirectly by solving differential equations in transient simulation). In idealized case, when the delay is 0, the assigment is also not performed instantaneously, but in the next delta-cycle, i.e. next time when the simulator treats the events that are waiting in the queue to be processed.

The concept connected to signal assignment in AleC++ comes from a language for description of digital circuits VHDL. In the following section we describe properties of operator <- and its effects.

6.2.5.1. Drivers

When compiler reaches <- operator, it does not know whether some other process assigns the value to the same signal. That signal can be passed to a module using interface (formal signal), or can be a local signal that is passed to some submodule, so the simulator is not aware about all connections to that signal when executing the assignment statement. This technique of connecting is used for production of digital circuits known as WIRED-AND and WIRED-OUT, when outputs of more digital circuits are connected to a bus. In this way, logic function can be executed with savings in hardware and with reduced delay.

When it happens that more statements assign the value to the same signal, a resolution function is invoked. To enable this, assignment is never done directly, but using intermediaries -- drivers. **Every process creates a driver for that signal. Assigning into the same signal in two different processes creates two drivers. Signal with multiple drivers has to have a defined resolution function**, which will derive the final value of the signal from the combined values from drivers. Since a **driver is characteristics of a scalar signal**, the assignment to a signal-vector creates drivers for each particular scalar index.

```
signal three_t s1, s2[4], s3[2][4];
..
  process (...) {
    int i, j;
    three_t m[2][4] = { "1101", "0101" };

    s1 <- 'x';  // creates 1 driver for s1
    s2 <- "0100";       // 4 drivers - from s2[0] to s2[3]
    s2[3] <- '2';       // 1 driver for s2[3]
    s2[i] <- '2';       // 4 drivers - i is a constant
    s3 <- m;            // 2x4=8 drivers for s3
    s3[1] <- "0101";    // 4 drivers - from s3[1][0] to s2[1][3]
    s3[i] <- "0010";    // 8 drivers - i is not a constant
```

```
    s3[1][0] <- '0';     // 1 driver for s3[1][0]
    s3[1][i] <- '0';     // 4 drivers - i is not a constant
    s3[i][j] <- '0'; // 8 drivers - both i and j are not constants
  }
```

Drivers are created during compilation, and their number has to be known before the start of the simulation.

Signals on the right-hand side of `<-` operator can be composite, but not heterogeneous. During the type checking, compiler performs implicit conversion, as for all expressions.

6.2.5.2. Delay models

The previous assignment examples did not have a defined delay, which means that the new value will appear on the signals in the next cycles of the process (although the time indicator `now` is set to the same time-instant in all that cycles). This delay is a consequence of the simulation algorithm, which gathers the assignments, and then processes them. Nevertheless, assignment operator can be used with user-defined delay. This delays the processing for a specific period of time.

signal_assignment:
signal **<-** <**transport**> *new_value ;*

new_value:
expression <**after** *expression*>
list_of_new_values

list_of_new_values:
pair_expression_time
list_of_new_values , pair_expression_time

pair_expression_time:
expression **after** *expression*

A signal can be assigned a new value with or without the key word **`transport`**, which concerns the type of delay. If it is used, the delay that is applied is characteristic for transmission lines -- every impulse is transmitted, regardless of its length. If the keyword `transport` is not used, the delay is **inertial** -- impulses must last long enough to produce an effect. Interial delay is characteristic for digital components, where impulse must have enough energy to change the state of the digital component.

The expression after the keyword **`after`** regards the assignment delay with respect to the present time instant. Signal can accept a sequence of pairs *value-time*, as well.

New values of signals and their delays represent future events, which are stored on the internal list of every driver. These future events are ordered in the list by their time - ordering of ascending time. However, there are some rules of arrangement that depend upon the type of delay used:

- In case of **transport** delay, the new pair *value-time*, that is new events will be put on the list so that all pairs *value-time* whose *time* is greater than the time of the new pair will be erased;
- In case if **inertial** delay, all pairs on the list whose *time* is less than the *time* of the new pair will be erased. The exceptions are the pairs whose *value* is the same with the *value* of the new pair.

This type of arrangement of future events agrees with the behaviour of real digital components. The event caused by a short-lived impulse will be erased in the case if inertial delay before its time comes, so the component that owns the driver will not react to the impulse.

Simulator constantly controls the time of the first event on the list of every driver. If the time agrees with the present moment, the value of that event becomes the new value of the driver. In order for the new value of driver to become the new value of the signal, at least one of the two conditions needs to be met:

- Signal has only one driver;
- Signal has an appended resolution function, which will resolve the final value from driver values.

6.2.5.3. Resolution of conflicts on the bus (resolution function)

A signal with two drivers gives rise to **wired logic**. We can approximate a number of outputs of digital circuits using voltage sources with their output resistances. By superimposing these sources, we can arrive at the resulting value at a node. If the output resistances differ significantly, the resulting value in a node is the result of the source with the smallest output resistance. Speaking in wired-logic terms, signals can, beside the state, have **intensity**. The following conditions need to be met if the simulator is to model wired logic:

- System of states has to have a state of high impedance, indicating that the driver is currently off-line;
- Every multiple-driver signal has to have a resolution function.

```
typedef enum { 'x', '0', '1', 'z' } four_t;

four_t bus4res (const four_t *drv, int *report) {
    int ndrivers = lengthof drv;
    four_t result = drv[0];
    for (int i=1; i<ndrivers; i++) {
        switch (drv[i]) {
           case '0':
           case '1':
              result=(result=='z'||result==drv[i])?drv[i]:'x';
              break;
           case 'x':
              result = 'x';
              break;
           case 'z':
              break;
        }
    }
    if (ndrivers > 1 && result == 'x') *report = 1;
    return result;
}

signal four_t:bus4res bus[3:0];
signal four_t:bus4res line = '0';
```

Function `bus4res` given above makes the resolution for the system of four states. If at least only one driver is `'x'` (undefined), the result is also `'x'`. If none of the drivers is `'x'`, but there are at least two of them having different logic values `'0'` and `'1'`, the result is again `'x'`, as there are two drivers with different values and high intensity. The wired logic is normally used when only one driver drives the bus, which means that all others are at high impedance state `'z'`, that is, have weak intensity.

The resolution function is a global function fulfilling certain conditions regarding results and parameters. If a signal is of *link (signal) data type* `T`, its resolution function will return the same type. The function has to have two parameters, a pointer to data type `T`, and a pointer to `int` type. The former is a vector of instantaneous values of all drivers of a signal, and the second is a flag indicator. This flags conveys whether the resolution needs to report a conflict on the bus or not (the flag needs to be set to 1 for the message to appear).

An association of a signal and its resolution function is obtained on signal declaration, if the character `':'` and the name of the resolution function are listed after the signal data type. Another way is to create new type (using `typedef`) that includes the resolution function:

```
four_t bus4or  (const four_t *drv, int *report);
four_t bus4and (const four_t *drv, int *report);

typedef four_t:bus4or four_t_or;   // new, resolved type fout_t_or
typedef four_t:bus4and four_t_and; // second version

typedef struct {
    four_t send, recv;
    four_t:bus4and data[3:0];
}:bus4or Connector;
```

The last example shows the application of the resolution function in case of structures. The whole structure has a resolution function `bus4or`, which is applied on members `send` and `recv`. However, structure member `data` has its own resolution function `bus4and`.

For one system of states, you can define a desired number of resolution functions, e.g wired-AND, wired-OR. For system of states with high number of states it can be useful to create tables of resolutions as arrays.

If a signal has one driver, and its resolution function is defined, it is applied. However, signal with one driver needs not a resolution function. In such case, driver value is copied to the signal.

6.2.5.4. Driver initialization

If the system of states begins with `'x'` (the first enumerated symbol is indexed as 0), all non-initialized signals in the circuit would be set to `'x'` at the beginning of the simulation. For most of the cases, this is natural choice, as `'x'` denotes undefined state. Nevertheless, for modelling of switching logic this can be a problem. Transmission gate (switch) is actually a MOS transistor, and drivers for *source* and *drain* would be set to `'x'` even if the gate is open. Such transmission gates are usually used for connections to the bus. Undefined state `'x'` on any driver of the bus means usually that the state `'x'` would be resolved for the whole bus. However, correct model of real circuit behaviour demands that an open switch means that bus is driven with high impedance state, `'z'`.

The solution is found by initialization of formal signals.

```
module tgate (four_t in gate; four_t inout drain='z', source='z');
```

If formal signals have direction `in` or `inout`, and an initial value is given, then all drivers created by the `processes` of that `module` will have that initial value.

The problem can arise with initialization of multidimensional-array-signals. Their first dimension can be unknown, as they are formal signals, so one cannot utilize aggregate initial array of values with fixed length. In such cases, one can use operator `<-`, instead of `=`, which means that the given initial value is repeated to cover the whole length of the signal:

```
module X (four_t out y[] <- "z");
```

In the above example, all drivers created by `process`es of `module X`, for all indexes of vector `y`, will be in `'z'` state, whatever the size of the actual signal that corresponds to the formal signal `y` is.

6.2.5.5. Complex delay

There are many ways of modelling delay in logic simulators (unified, assigned, rise-fall, static, etc.). When the delay is not a predefined value, the best way is to express it as a result of a function. If the function is not too long, you can define it as `inline`:

```
double fdelay (three_t new_value, double tplh, double tphl) {
    if (new_value=='0') return tphl;
    if (new_value=='1') return tplh;
    return tplh > tphl ? tplh : tphl;
}

...
process (a,b) {
    three_t res = a & b;
    y <- res after fdelay (res, 10ns, 12ns);
}
```

When dealing with complex models of delay, such as minimum, typical, and maximum values, we recommend gathering of all parameters in model card. You can then define the functions that calculate the delay value as methods of model classes. Of course, other functions that are used for component modelling are suitable to be methods of model classes, too.

6.3. User-defined signal attribute -- operator '@'

Beside seven signal attributes generated and controlled by the simulator itself (`active`, `event`, `quiet`, `stable`, `hybrid`, `fanin`, and `fanout`) there is an user-defined attribute, too. Memory for this attribute is separately allocated. The name of the signal refers to its current value, while the name of the signal preceded by the operator '@' denotes the pointer to user-defined attribute. User-defined attribute is allowed for scalars signals only (including members of arrays or structure members).

Attribute can be of any legal type, including classes, but cannot be a vector, reference, or a pointer. Since one can declare a class as an attribute, there can be effectively more user-defined attributes (class members). Character '@' is used for attribute declaration, too:

```
signal three_t@int s, v[4];       // attribute of type int
...
process {
    s;          // value of signal s
    @s;         // pointer to an attribute of signal s
    s=1;        // wrong - s is a signal
    *@s=1;      // O.K. - recording int an attribute of signal s
    @v;         // wrong, v is a signal-vector
    @(v[1])     // O.K. - v[1] is a scalar signal
}
```

It is not legal to use operator '@' for signal without attributes. Attributes can be incorporated into data type using `typedef` command. Such data type can be later be used exclusively for signals.

Note: One can use attributes to pass information between processes. However, this cannot be a correct model of real circuit behaviour. Therefore, **attributes should not be used as another way of communication between processes**. You should set their values in the preparation phase of the simulation, and later only use these values.

Application of user-defined attributes will be illustrated on the problem of capacitance modelling in the digital circuit. Delay of a digital circuit depends linearly (as the first approximation) on the capacitance connected to its output. Capacitances in a digital system are input/output (terminal) capacitances of the circuit, and the parasitic capacitances of connections (links). The former can be considered as module parameters, and passed as such to modules. The later need to be connected to signals. This makes connection capacitance an ideal candidate for a signal attribute. These capacitances cannot be changed during the simulation, which means we should declare an attribute of as signal as a class, whose private member is a total capacitance of the signal. The capacitance value will come out in the preparatory phase of the simulation. During the simulation run, it will be used for calculation of the module delay.

```
typedef enum { 'x', '0', '1', 'z' } four_t;

class four_att {
        double Tcap;    // total capacitance of the signal
    public:
        four_att (double cap=0.0) { Tcap = cap; }  // constructor
        void add_tcap(double cap) { Tcap += cap; }
        double tcap () { return Tcap; }
};

typedef four_t @ four_att four_full;
```

The memory is allocated for local signals only -- formal signals just denote connections to other signals, which are declared as local in some modules that are on higher hierarchy levels. Therefore, constructor for attributes is activated with arguments that are given in local signal declaration.

```
four_t @ four_att(0.2p) bus;
four_full(0.1p) line[3:0];
```

This way of declaration gives certain capacitance to the signal (parasitic capacitance). The best moment for total capacitance calculation is after forming of the hierarchical tree. Therefore, process synchronized as `post_structural` should be used:

```
module and2 (four_full a, b; four_full out y) {
    action (double Cin, double delay, double skew) {
        process post_structural {
            (@a)->add_tcap (Cin);
            (@b)->add_tcap (Cin);
        }
        process (a, b) {
            y <- a & b after delay + skew * (@y)->tcap();
            ...
        }
    }
}
```

In every component of type `and2`, signals `a` and `b` would have input capacitance `Cin` (passed as `action` parameter) added. In reality, formal signals `a` and `b` are just connections (other names) to some actual signals elsewhere, that have that input capacitance added. Parameter `skew` is the coefficient of increase of total delay with the increase of capacitance load at the output. Therefore, multiplying value `skew` with the total capacitance of the signal `y`, we get the delay added to the parameter `delay` (parameter `delay` is the delay when the output is not loaded with capacitances). Therefore, the total logic circuit's delay depends on `fanin` (the number of modules linked to the output). We cannot access the value of capacitance, since it is declared as `private`. By the use of an

additional mechanism, we can ban the use of `add_tcap` method, and thus fix terminal capacitance to be read-only.

6.4. Leaving out actual signals -- void

Some formal signals can appear to be unnecessary during connecting. For example, one can use only non-inverting output of a flip-flop in the circuit, while the inverting output is not used -- it remains unconnected. In AleC++, this is enabled by inserting the word **`void`** instead the actual signal, when such component is connected. Compiler will generate an implicit signal whose dimensionality and initial value will be determined from the formal declaration. If the formal signal has direction `in`, processes sensitive to it will never activate (if we do not count one pass during the initialization). Signals with direction `inout` can still be active if they have drivers inside the `module` that is to be connected.

```
module rsff(three_t in reset='1', set='0';
                      three_t out q='0', qbar='1') rsff1;
signal three_t r='1', s='0', q;

ruffs (r, s, q, void);
```

The implicit signal, generated automatically as a consequence of omission of actual signal, cannot be referenced to in the printout control (command `plot`). Its activity are known only to processes on the formal side of the interface.

6.5. Variable number of formal signals -- operators '$' and '$$'

In C/C++, one can declare and define functions with a variable number of arguments. The number of formal parameters of a `module` can be variable, too:

```
module andx (three_t out y; three_t in ...) {
    action (double delay) {
        process structural {
            if ($$ < 2) warning("module andx has no inputs",1):
        }
        process ($2 ...) {
            three_t result = $2;
            int i;
            for (i=3; i<=$$; i++)  result &= $i;
            y <- result after delay;
        }
    }
}
```

The last example has modelled logic AND circuit with N (N≥1) inputs. The output to the circuit is declared first, while the inputs are given by type and direction, with `'...'` instead the name. This symbol is very important for compiler, and cannot be omitted in the declaration of such module. The compiler will not check the number and type of actual signals to the right of signal `y`, if this character is missing.

Operator '$$' does not demand operands and returns the total number of formal signals. To get the number of inputs, we need to subtract the number of fixed signals (1 in this case -- signal `y`) from the value returned by '$$'.

The main `process` is synchronized to be sensitive to all formal signals to the right of signal `y`. Symbol '$2' means the 'second formal signal by order.' The same signal appears in the code of the `process`, as well. In the `process` code it is legal to place an expression to the right of '$'. The result of the operation `$expression` is a formal signal at that position, with the type and direction from the declaration in the heading.

```
module andx (three_t out y; three_t in ...) action (double delay);

module X () {
    andx and1, and2, and3;
    signal three_t s1, s2, s3, s4, s5, s6;

    and1 (s3, s1, s2) delay = 10ns;
    and2 (s4, s1, s2, s3)  delay = 12ns;
    and3 (s6, s1, s2, s3, s4, s5)  delay=9.7ns;
}
```

Operator '$' has the same priority as all other unary operators, which mean that parentheses must be used when dealing with a complex expression.

```
$i + 1      // signal from position, added with 1 -- an error?
$(i + 1)    // signal from position i+1
```

6.6. Variable number of action parameters

The number of parameters in the header of the `action` block of the `module` can be variable, too. This is described in the Chapter 5, as it is the same for digital and analogue modules.

6.7. Array of components -- commands clone and allocate

In Alecsis, one can generate an array of previously declared components. Commands **`clone`** and **`allocate`** is used for that. Details of command `clone` are given in the chapter on analogue simulation, and are valid for digital simulation, too.

6.8. Structural aspect of digital circuits

The final discussion in this chapter concerns the structural approach to digital simulation. This was the only way with older digital simulators, since all digital components along with tables and system of states were built-in and fixed. Alecsis does not have any built-in digital component, but it does have mechanisms for creation of extensive libraries of components. Such components can be used as they are built-in.

Digital elements use the same syntax, parameter setting, declarations, etc., as analogue or hybrid components. AleC++ does not have a construct that would allow the compiler to guess if a component of some `module` type is analogue, digital, or hybrid.