3. Hardware description language AleC++

The object-oriented hardware description language AleC++ controls most of the Alecsis simulator operation. To support generality and avoid specialization, many activities otherwise located in the simulation engine are transferred to the input stage. In that sense, AleC++ serves not only as the modeling language, but also to customize the simulator and create supporting libraries.

Hardware description languages for digital simulation have been around for decades, VHDL and Verilog being one of the most elaborated examples. Although the area of analog modeling languages is not as crowded as for their digital counterparts, some of them, like MAST (SABER), M (Lsim), ALFA, Verilog-AMS and VHDL-AMS represent concepts that provide for the flexible analog behavioral modeling. However, we wanted to emphasize object-orientation of modeling. The object is an entity encapsulated within defined boundaries, so that its external features, which are accessible to other objects, are separated from its internal features, which are hidden from them. From this definition, one can see that the problem of modeling is clearly object-oriented. Property inheritance, operator overloading, access control, are all the features of C++ that fit nicely to the modeling problem.

Also, our intention was to create a language that is easy to learn, and the most common way to accomplish that goal was to upgrade some wide-known language, such as C++, with features that enable modeling and simulation. To keep language consistent with its core, we assured that additional constructs resemble C++ style to the highest possible extent. We also tried not to over-clutter the language with features that are not absolutely necessary.

3.1. Modules, processes and nets

The basic element of hierarchical system description in AleC++ is named **module**. Module is an entity used to describe components or subsystems with no specific domain specialization. This means that digital, analog, non-electrical or hybrid systems may all be expressed using one unified syntax. Module is the part of the simulated system that consists of other modules and/or built-in analog components. The module construction is also used to describe new basic components at the lowest hierarchy level. AleC++ description of a module consists of an **interface** and a **body**. The body in turn consists of declarative, structural and behavioral regions. The interface is used for communication between the module and its environment, while the module body represents its architecture. The architecture may be a simple description of interconnection of components/modules and nets (structural modeling), a set of statements (behavioral modeling) or a combination of both.

Alecsis uses the term **net** to mark a quantity that appears on a device (module) terminal. These quantities can be both continuous and discrete (mixed-signal simulation). There are five kinds of nets in AleC++: `node`, `current`, `charge`, `flow` and `signal`. The first three kinds refer to analog quantities obtained as a solution of nonlinear sets of circuit equations. The flows are similar, but they represent non-electrical analog quantities, such as light, temperature, pressure, and so on. In terms of across and through quantities (HDL-A [Pabs95]), `node` is across and `current` is through quantity. The net declared as `flow` can be used both as across and through quantity. The terminal quantity in discrete-event simulations is signal. The net kind appears in a net's declaration as an additional type qualifier.

The module architecture may be expressed in the structural region of the module body. Similar to C++ methodology, components and nets should be declared before they are used. Alternatively, the architecture may be described using behavioral constructs in the so called `action` block. The action block creates a new name space containing one or more concurrent processes. AleC++ syntax in this domain has been adopted from VHDL.

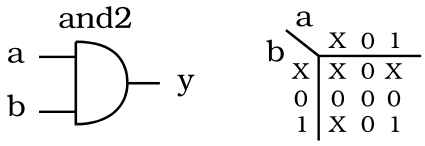

Fig. 3.1: Two-input AND gate model

The following AleC++ code shows an example of an AND gate model, see Fig. 3.1.

```
typedef enum { 'X', '0', '1' } three_t;
const three_t and_tab[][3]= {    { 'X', '0', 'X' },
                                 { '0', '0', '0' },
                                 { 'X', '0', '1' }   };
inline operator & (three_t op1, three_t op2) {
       return and_tab[op1][op2];
}
module and2 (signal three_t in a,b; signal three_t out y) {
    action (double delay) {
       hazard_detection: process {
          wait a,b while (a->stable || b->stable || a!=b);
          warning ("and2: static hazard");
       }
       and2beh: process (a, b) {
           y <- a & b after delay;
       }
    }
}
```

The above example shows several important features of AleC++. Firstly, there is no preferred logic value system. Type of every signal is given, and if an enumeration type is used for that purpose, legal logic value system for all the signals of the specified type is determined at the point where the type is defined. Like in VHDL, enumeration constants in AleC++ may be character literals. Vectors or matrices of enumeration types may be used to code lookup tables for standard Boolean operators. Bitwise logic operators may be easily overloaded for newly defined enumeration types using C++ operator overloading facilities. Module `and2` has two input signals and one output signal, two concurrent processes sensitive to both input signals and one `action` parameter `delay`. Action parameters serve as one of two methods for module parametrization in Alecsis, making modules more general. The first (passive) process waits until both input signals simultaneously

change to different values, using predefined signal attribute `stable` for detecting such a situation. Its function is to inform the user about hazardous situations. The second process has only one statement that drives the output signal.

Event-driven logic simulation in Alecsis is similar to that of VHDL. For example, AleC++ provides for definitions of bus resolution function and user-defined signal attributes. The user can supply AleC++ functions in which wired-or, wired-and or some other strategy will be applied to combine values of all the signal drivers into one resolved value.

One of AleC++ strengths is its SPICE compatibility. By defining special SPICE syntax regions in the input file, users may include the original SPICE model cards from their libraries. The most significant SPICE device models are supported: MOSFET, BJT, JFET and diode. For example, the next portion of AleC++ code inserts the SPICE model card of a diode named D1N4148 into the description of a circuit.

```
...
// AleC++ syntax ends, SPICE syntax starts:
spice {
.model D1N4148  D (Is=0.1p Rs=16 CJO=2p Tt=12n Bv=100
+                                                 Ibv=0.1p)
}
// AleC++ syntax continues:
d (14, 12) model = D1N4148;
...
```

System description in AleC++ has different syntax than SPICE. Nevertheless, the organization of the system description is the same in both SPICE and AleC++, and translation from one language to another can be automated.

3.2. Analog behavior modeling

AleC++ is an HDL that has inherited generality of the programming language from C++. HDLs that are made as analog extensions of digital modeling languages impose more limits in analog modeling. Flexibility of general programming language is therefore an advantage of AleC++. Nevertheless, for that reason, there is plenty of ways to solve the problem of modeling of some analog device or system. We will present here some preferred modeling styles, which lead to efficient and self-explanatory models. These styles can be combined in description of one model.

The simulation engine of Alecsis uses a modified nodal method (MNA) [Ho75]. Therefore, the user should describe a model's stamp, i.e. its contribution to the system of equations. However, direct definition of stamps may be inconvenient for the user, meaning that some more user-friendly way should be found for model description. Of course, that simplified description must be easily convertible into a stamp by the language or the simulator.

The first possibility is to represent the model by the use of the built-in or previously described components. Alecsis has no built-in logic devices, but does have a limited set of built-in analog elements. It was necessary to build in an ideal analog switch [Mr~a93, Mr~a96b], due to its importance in certain circuit classes (SC circuits, for instance), and its strong influence on the time step control mechanisms. We also considered it reasonable to create a built-in set of primitives like resistor, capacitor, diode, bipolar transistor, JFET, MOSFET, etc., to provide a SPICE-like base of analog components. Model described by the usage of built-in or previously described components is actually a subcircuit.

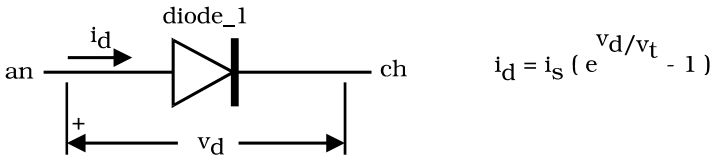

Fig. 3.2: Diode model

Another way is to use the language to describe a physically equivalent model of the given device. For instance, the nonlinear model can be represented by an equivalent linear circuit whose element values are computed in every iteration (companion element) [Anta95]. In this modeling method, the structure of the model is represented first, as it is composed of already defined or built-in elements. However, in the structural description, parameter values can be omitted, and can be calculated in the `action` block of the module. In this way, all the device parameters may be dynamically altered during the simulation using simple access conventions.

As a simple example, a semiconductor diode (Fig. 3.2) may be modeled as a resistor and a constant current source in parallel.

```
module diode_1 (node an, ch) {
       // declarative region
       resistor pnr;
       cgen pnc;
       // component mapping
       pnr (an, ch) 1Mohm;
       pnc (an, ch);
       // behavioral modeling region
       action (double is=1e-14) {
              process per_iteration {
                     double gm, id, vt = 25.8mV;
                     id = is*(exp((an-ch)/vt)-1.0);
                     gm = (is + id)/vt;
                     pnr->value = 1/gm;
                     pnc->value = id - gm*(an-ch);
              }
       }
}
```

In this simple pn junction model, values of the components in the linearized schematic are altered according to the Newton-Raphson iterative method.

Alternatively, in the second modeling style, the same device may be modeled using general nonlinear generators.

```
module diode_1 (node an, ch) {
    nlcgen pngen;
    pngen (an, ch, an, ch);
    action (double is=1e-14) {
       process per_iteration {
              double gm, id, vt = 25.8mv;
              id = is*(exp((an-ch)/vt)-1.0);
              gm = (is + id)/vt;
              nlcgen pngen = id {  @a = gm; @ch = -gm;  }
       }
    }
}
```

Nonlinear current and voltage generators (`nlcgen` and `nlvgen`) are used to conveniently model nonlinear systems with arbitrary number of controlling (input) terminals. If the partial derivatives are omitted, Alecsis makes a discrete approximation using the *secant* (finite difference) method. There is some penalty,

however, that the number of function evaluation increases since the order of convergence decreases (1.518 instead of 2).

In both ways of modeling shown above, processes activated in every iteration have been used in the `action` block. For that purpose, synchronization keyword `per_iteration` is used. If the component is linear but time dependent, calculation in every iteration is not necessary and one usually takes the calculation out of the inner, iterative, loop in order to save CPU time. In such case `process` in the `action` block can be synchronized `per_moment`. In this manner, several methods of synchronization are defined:

- `structural` process is executed once during the hierarchical design tree building
- `post_structural` executed once after the hierarchical design tree building
- `initial` executed once at the beginning of the simulation
- `per_moment` active in each new time point, before solving the system of equations
- `post_moment` the same, but after solving the equation set
- `per_iteration` active at each iteration, before solving the system of equations
- `final` executed once at the end of the simulation

Synchronization of processes is useful not only for decreasing simulation time, it can give to an experienced user plenty of possibilities for creating complex models. For instance, AleC++ allows creation of arrays of elements such as cascade of identical cells for example, by the use of loop constructs inherited from C++ and AleC++ `clone` command intended for cloning a declared element. Clearly, such process must be executed as `structural`, i.e., before actual simulation, during the building of data structures that describe system under consideration, since execution of such process defines structure of the system itself. The structural processes are, for instance, used in modeling the pressure sensor (see example in section 5.8) and in transmission line simulation.

Moreover, use of these synchronization methods is not restricted to analog modeling: by synchronizing `per_moment` the process labeled as `and2beh` in the example in section 3.1, the simulation becomes time-driven, instead of event-driven.

There is also a third way of modeling analog devices. It gives absolute freedom in modeling since the model stamp is specified directly, by the use of `eqn` statement. In this modeling method selected positions in the matrix may be filled

with regular expressions or using special `ddt` and `idt` operators, that represent time derivative and integral (useful for continuous time control system simulation), respectively. The following example shows model of a capacitor.

```
model new_capacitor (node j, k) {
       action (double value) {
              process per_moment {
                     eqn {j,k}.i = value * ddt{j,k}.v;
              }
       }
}
```

This model states that the current between nodes `j` and `k` equals `value` times the time derivative of the voltage between nodes `j` and `k`. The above code is not dependent on the actual numerical integration method used to solve differential equations.

Extension `i` in the equation above denotes current between nodes `j` and `k`. Extension `v` denotes voltage between nodes `j` and `k`. Similar construction can be found in HDL-A [Pabs95]. Similarly, one way to model an inductor in AleC++ would be as follows.

```
model new_inductor (node j, k; current i) {
       action (double value) {
              process per_moment {
                     eqn i,(j,k).v = value * ddt{i};
              }
       }
}
```

which states that the voltage between nodes `j` and `k` equals `value` times the derivative of the current `i` flowing between nodes `j` and `k`. If `j` and `k` are declared as `flow` (nonelectrical quantities), extensions `a` and `t` are used instead of `v` and `i`. Extension a stands for across and `t` for through quantity.

An operator for second derivative with respect to time, namely `d2dt2`, exists also in AleC++. It is introduced primarily for modeling of mechanical and micromechanical devices, since second derivative is useful for description of inertial properties. Applicability of Alecsis in this domain is already proven [Mr~a95a, Mr~a95b].

Using `eqn` statement, the entire built-in set of analog devices may be recreated as an external library.

3.3. Model cards as static objects

So far, no object-oriented language constructs were used in our examples. Those features may be normally included in any process or function in C++ fashion. Such an example is AleC++ upgrade of SPICE model card concept. SPICE users are familiar with its ability to group actual values of device model parameters into entities called **model cards**. Although AleC++ does provide means for parametrization through use of **action parameters**, it is more convenient to use some alternative methods in case of tens, even hundreds of parameters. AleC++ supports four SPICE model card classes: NMOS (PMOS), NPN (PNP), NJF (PJF) and D. Therefore, SPICE model cards can be used directly.

When the user defines a new model he can define a model card for it. There are certain requirements to be met in order to retain consistency:

- new model card class is defined using C++ class construct
- default values of model parameters should be set in the class constructor(s)
- beside constructor and destructor, there is third special class member function in AleC++, which does not exist in C++, called the preprocessor. Its purpose is to check the values of model parameters and preprocess them. It should have the same name as the enclosing class, with a '>' sign in front of it (similar to sign '~' in case of destructor). Preprocessing cannot be done in the constructor, since the actual model card is not known when the module is declared, but when it is used.

Let us define a simple MOSFET model card class called `simple_mos`. It will contain only six model parameters.

```
class simple_mos {
      // model parameters:
      enum mos_type { Ntype, Ptype } type;
      double vto, gamma, phi, lambda, beta;
  public:
      simple_mos ();      // constructor
      >simple_mos ();     // preprocessor
      double cdrain(double vgs, double vds, double vbs,
                    double *gm, double *gds, double *gmbs);
```

```
        double Type() { return (type==Ptype) ? -1.0 : 1.0; }
};
```

Arbitrary number of model cards of class `simple_mos` may be declared.

```
simple_mos:: model_1 { type=Ntype; vto=0.78; gamma=0.8;
                       phi=0.56; lambda=0.01; beta=1e-4; }
```

Thus obtained model cards may be viewed as static objects that are constructed at the beginning of the simulation and destructed at its end. The remaining task is to link the new model class and desired module (model).

```
module simple_mos:: smos (node drain, gate, source, bulk) {
    nlcgen ids;
    ids (drain, source, drain, gate, source, bulk);
    action (double l, double w) {
       process per_iteration {
          double vgs, vds, vbs, gm, gds, gmbs, id;
          vgs = Type() * (gate-source);
          vds = Type() * (drain-source);
          vbs = Type() * (bulk-source);
          id = this->cdrain(vgs,vds,vbs,&gm,&gds,&gmbs);
          nlcgen ids = Type() * id {
                    @drain=gds; @gate=gm; @bulk=gmbs;
                    @source = -gm -gds -gmbs;
          }
       }
    }
}
```

The model class is associated with the module using the scope resolution operator `::`. The keyword `this` refers to the model card object attached to a particular component via the mapping mechanism. It should be noted that all the model parameters are private to module `smos`. Module `smos` uses them indirectly, by a call to methods `cdrain` and `Type`. Data encapsulation and access control are here of the greatest importance, since more than one component may share the same model card object. An unwanted change of any model parameter would thus affect many components, creating a very dangerous side effect. Communication with model cards through class methods makes modeling cleaner and easier to follow and debug.

3.4. Model class inheritance in mixed-signal simulation

Alecsis automatically inserts special conversion devices at the point where analog and digital domains meet. Alecsis has no preferred system of logic states, they are defined in the model libraries and user can define his own set of states. Since D/A and A/D converters are closely related to the given set of logic states, they cannot be preprogrammed in the simulator. These converters are AleC++ modules that have to meet some special requirements:

- They must have at least one digital net (`signal`) and one analog net (`node`, `current`, `charge`, `flow`) in their interface.
- Since converter modules are always inserted on digital side, conversion is inherent to digital modules. In that sense, if converter modules accept model card classes, they must be either identical to the class of the parent digital module, or have to be its base class.

Domain converters are hierarchically inserted as children of the digital modules. One important property of Alecsis to propagate model card objects down the design tree when there are no explicit model cards attached, may be used to parametrize conversion modules. To accomplish this, model class inheritance should be used. One base class should contain conversion parameters (thresholds, levels, transition times, interface impedances, etc.). All other model classes with various delay modeling parameters for different logic devices should be then *derived* from the common base class, as shown in Fig. 3.3. In Fig. 3.3, base class `io` contains conversion parameters and functions needed for conversion. Few conversion modules are defined that take model cards of class `io`. All other classes (`sg` for standard gates, `ff` for flip-flops, `clkg` for clock generators, etc.) are derived from class `io`.

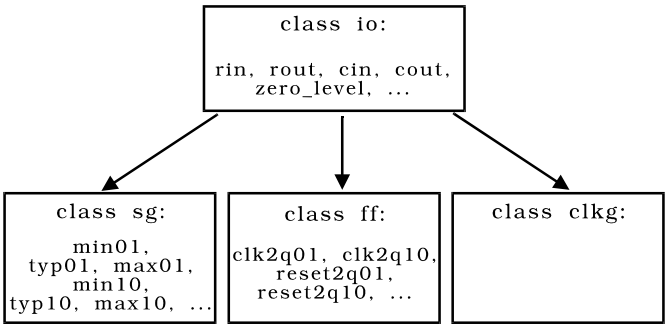

Fig. 3.3: Model class hierarchy used for mixed-signal circuit parametrization. Base class `io` contains A/D and D/A converter parameters, while derived classes contain timing parameters.

```
class io {
  protected:
       double rin, rout, cin, cout;
       double zero_level, one_level, x_level;
       double trans_time;
       ...
  public:
       io();  >io();
       inline double get_trans_time() { return trans_time; }
       ...
};
class sg : public io {
       double min01, typ01, max01;
       double min10, typ10, max10;
  public:
       sg();  >sg();
       double delay (three_t new_state, three_t old_state);
       ...
};
```

When describing a model card of some derived class, it is possible to set or modify both delay and conversion parameters.

```
model sg :: and2_model_1 {
       rin=50kohm; rout=200ohm; trans_time=0.2ns; ...
       min01=2.6ns; typ01=3ns; max01=3.5ns; ...
}
```

Conversion devices are assigned to digital modules through the use of special `conversion` specification.

```
module sg::and2 (signal three_t in a, b;
                                 signal three_t out y) {
       conversion { a2d = "cmos_a2d";   d2a = "cmos_d2a"; }
       ...
}
```

In this way, base class parameters are at disposal to digital modules. This useful property will be discussed in section 4.3.1. A model card object attached to the digital module will also be passed to the conversion module of which it may use only conversion parameters.

3.5. Signals-structures and signals-classes

Apart from enumerations, AleC++ signals may be of any other regular types (int, double, etc.). Signals may be composite, too. Composite signals may be classified into homogeneous (arrays) and heterogeneous (structures and classes). In case of arrays, composite signal is created as a collection of scalars, and signal subscript has all the properties of regular signals. Similar situation is with signals - structures. Let us examine the following example.

```
struct Line { three_t send, recv; ... };
signal Line port;
```

Signal `port` is a composite that may be broken into signals `port.send`, `port.recv`, etc. Each selected member behaves as a regular signal in all aspects. This property is desirable at the gate level, and tolerated at the **R**egister - **T**ransfer **L**evel simulation. However, at higher levels of abstraction, information channels between processes tend to carry more complex information which is transmitted in data blocks (packets, frames, etc.). It is therefore inappropriate to model it using structure types, since it will break the line into individual members, a situation that is in collision with the modeled hardware. Let us consider the following example:

```
class Line { public: three_t send, recv; ... };
signal Line port;
```

Signal `port` is now of the class type and represents entity that may not be further divided. Individual members like `port.send`, `port.recv`, etc. may be still accessed as usual, but they are not signals, just members of the signal `port`. From the simulation engine's point of view, signal `port` is scalar. This feature may be used to model complex connections between high-level subsystems, as will be shown in section 5.10.