Alecsis
the simulator

Dejan B. Glozić
Željko M. Mrčarica
Dejan M. Maksimović
Tihomir R. Ilić
Željko B. Dimić
Dejan S. Aleksić
Bojan Anđelković
Milan Savić
Miona Andrejević
Vančo B. Litovski

Laboratory for Electronic Design Automation
University of Niš
Faculty of Electronic Engineering

Internal publication

LEDA 3/96

reprinted as 2/97 (ver. 6/2003)

http://leda.elfak.ni.ac.yu

Corresponding address:

Prof. Vančo B. Litovski
Univerzitet u Nišu
Elektronski Fakultet
Beogradska 14
18000 Niš
Yugoslavia

phone: +381 18 355 878
fax: +381 18 46 180
e-mail: vanco@elfak.ni.ac.yu

Foreword

Alecsis (Analogue and Logic Electronic Circuit Simulation System) has been under development at the Faculty of Electronic Engineering in Niš since 1991. It is now installed at several European universities, and is mostly used for analysis of nonelectrical or mixed physical domain microsystems. Among the successfully simulated systems one should mention analog circuits, switched-capacitor circuits, neural networks, fuzzy-logic systems, magnetic circuits, microelectromechanical systems such as pressure and flow sensors, digital circuits, computer networks, mixed-signal circuits, etc. This book gives an overview of use of Alecsis in different problem domains.

Alecsis 2.4 is a simulation environment that connects digital and analog simulation engine to a modeling interface. It comprises tools for solving analogue designs (mechanisms for solving system of nonlinear differential equations characterized by a sparse matrix), as well as discrete-event simulation subroutines. The user interface to the simulator is a modeling (hardware description) language called AleC++. The language is used both for modeling and for customizing the simulator to different environments and purposes. The simulator and the language support electrical, logic, but also mixed-signal, mixed-mode and mixed-domain simulation.

AleC++ is a superset of C++ and inherits its properties. It is an object-oriented language providing data encapsulation, inheritance and polymorphism through the use of C++ class construction that appears to be a powerful mechanism for modeling. AleC++ is compatible with SPICE device models. It upgrades SPICE model card concept and proposes a general method for mixed-signal model parametrization in an object-oriented manner. Language AleC++ can be used in both compiled and in interpreted mode.

Several examples from different domains (analog, discrete-event, mixed-mode, mixed-signal, and mixed-domain) illustrate the modeling power of AleC++ and simulation capabilities of Alecsis 2.4.

Although we are continuing the development of our simulator, we consider version 2.4 to be stable and accurate, and we decided to inform research and industrial community about it. The simulator is distributed in the scope of research cooperation with our partners. These projects are in the domain of electronics, micromechanics and mechatronics, which proves that Alecsis can be successfully customized for different applications.

For Alecsis installation, one needs a UNIX workstation with C compiler, YACC and LEX tools and XWindows graphical interface. Up to now, the simulator is installed on HP9000s300/400, HP9000s700/800, IBM RS6000, Silicon Graphics, and SUN SPARC workstations, as well as on PCs running under LINUX. User's Manual which explains installation and usage of the simulator is available. If you have interest for cooperation in this field, you can contact us using the address given on the front page.

authors

Contents

1. Introduction

Electronic circuit simulation constantly attracts attention of the scientific community. Algorithms and methods that proved to be efficient for circuits with 100000 gates, run out of steam when faced with 1 million gates, while others cannot cope with heterogeneous circuits with micro-mechanical and/or optical devices and systems, or hardware/software co-simulation problems. It is well known that circuit simulation is very resource-intensive and that requirements for simulation of modern electronic circuits are always one step ahead of the state-of-the-art memory and CPU capabilities. Modern ASIC (Application Specific Integrated Circuits) more and more frequently contain both analog and logic subsystems, embedded software, and are sometimes used with optical and/or micro-mechanical devices. To be able to handle an entire system, a modern simulator must have the ability to express all these kinds of sub-systems in the most efficient way, with a desired level of accuracy. As an expected consequence, it is likely that the simulator itself will incorporate several different algorithms and concepts, resembling the heterogeneous nature of simulation problems.

If we assume that non-electrical devices may be expressed using sets of nonlinear differential equations, the problem of modern VLSI ASIC chip simulation essentially requires two different algorithms - one for solving the sets of nonlinear differential equations and one for discrete-event simulation. Thus, analog electrical and non-electrical portions of a system are analyzed using accurate and detailed, but quite slow and resource-intensive SPICE-like algorithms [Nage75], while logic subsystems are expressed using logic values and states, with reverse accuracy and efficiency figures. Besides, simulator should be able to interpret software routines during the simulation run, to enable description of systems with embedded software (SoC – system-on-a-chip). It is therefore necessary to have a behavioral simulator that handles mixed-mode systems, and a powerful hardware description language that enables description of both hardware modules and software routines..

There are several simulators available that can handle different classes of circuits, levels of accuracy and generality. In the presence of well-defined and loosely coupled analog and digital domains, it is possible to simulate the system using two separated simulators synchronized with some sort of kernel that can handle inter-process communication [Corm88]. Many commercial CAD frameworks have the option of coupling several different analog and logic simulators for that purpose [Smit92]. However, the approach is not general, since it cannot handle circuits with strong feedback loops. More appropriate is the algorithm that integrates analog and logic subroutines into one simulator. Integrated mixed-signal simulators like DIANA [DeMa80], SPLICE [Newt78], MOTIS [Chen84], SAMSON [Saka85], all combine analog system solving techniques and event-driven, selective trace methods controlled by one central agent to simulate mixed-mode circuits. In this way, large digital portions of ASIC systems may be efficiently simulated using fast logic simulation techniques, while accuracy and details are reserved for isolated analog blocks. It was shown, however, that those analog pieces, being modeled at the transistor level, use most of the simulators' time and resources and degrade the overall performance. It is therefore essential to have behavioral analog macro-modeling capabilities to resolve this bottleneck. Some of the analog and hybrid simulators offer these capabilities through their analog modeling languages, usually named hardware description languages (HDL). Such languages as M of Lsim [Odry86], MAST of SABER [Getr89], ALFA [Kazm92], S++SDL [Brow92], Verilog-AMS [Kund96] and HDL-A [Pabs95, Roch96], which is the working version of the standard VHDL-AMS [Vach95], [IEEE99], [Chri99]. Languages MAST, ALFA, Verilog-AMS and VHDL-AMS also have the ability to model non-electrical devices by expressing their behavior using algebraic and/or ordinary differential equations (ODE). Development of the language MHDL was sponsored by the MDHL Study Group, a part of the IEEE Standards Coordinating Committee - 30 (SSC-30). Verilog-AMS is proposed analog extension to a co-standard IEEE 1364 Verilog language [Thom91], [OVI], and VHDL-AMS is a standard developed by the IEEE 1076.1 committee as the analog extension of VHDL [Lips89]. An overview of this standardization activity can be found in [Rhod96]. General idea of all the solutions that are under standardization is separate development of the HDL and the simulation engine (engines). The languages are designed to be tool independent. They should support not only the simulation, but also other design tasks such as synthesis, layout generation, etc., which is already the case with digital HDLs (VHDL and Verilog). Nevertheless, automated design of analogue and mixed-signal circuits is on much lower level then of digital, and these languages are still used exclusively for simulation.

In the industrial community, it is already clear that standardization of the analog and mixed-signals extensions of discrete-event languages is not going to solve all problems. There is already a SystemC initiative [SysC], governed by a need to develop a language that is suitable for describing both hardware and software needed for a system-on-a-chip (SoC) designs. Importance of having C/C++ based modeling environment is recognized by the industry [Bore99].

Languages as VHDL-AMS and Verilog-AMS are intended to be universal tools for modeling and documentation of both analog and digital devices and physical models from other domains (mixed-domain simulation). Nevertheless, they are both regarded as suited for design representation of "big-D-little-A" systems, i.e. for systems with more digital than analog content [Anta96]. They are developed as analog extensions of digital (discrete-event) languages. Moreover, they cannot be used for harware/software co-simulation.

Our approach is essentially different. We have developed our mixed-signal simulator Alecsis (**A**nalog and **L**ogic **E**lectronic **C**ircuit **SI**mulation **S**ystem), and the object-oriented HDL named AleC++ as a united system. Alecsis is an integrated **mixed-mode** simulator, which can handle arbitrarily complex combinations of different kinds of devices and subsystems with no limitations concerning closed feedback loops. It can handle various kinds of quantities that appear as physical connections between devices (**mixed-signal**), from analog nodes and logic discrete signals, to non-electrical quantities such as pressure, light, etc. It also enables both analog and logic blocks to be described at any level of abstraction, according to required precision and efficiency. It is not used for electronic circuits only. The generality of the language AleC++ has enabled behavioral description of different physical or abstract systems. Different kinds of systems can be simulated (**mixed-domain** simulation).

AleC++ is designed to be a superset of C++, hence its object-oriented features. However, it can be still used just as another C++ compiler in which case the produced object code is immediately executed or stored into a simulation library to be linked later. One of the main qualities of our simulation system is that AleC++ is an integrated part of Alecsis. The organization of the language will be explained in more details in the next chapter. It integrates the power of C++, the ability to model concurrent processes and VHDL-like typed signals, and compatibility with the SPICE device models. It also upgrades the SPICE model card concept and proposes a general method for hybrid block parametrization in an object-oriented manner. All HDLs have some characteristics of the programming languages - if-else branches and basic computing is necessary for describing even the simplest models. However, features of a powerful object-oriented language like C++ can be used to advantage in the modeling process. As pointed out in the description of a C++ based simulator Sframe [Melv92, Melv93a, Moin94, Caba99, Li00], inheritance properties and polymorphism can be very helpful in modeling. In AleC++, a model can be derived as a derived class of a base model, which gives an efficient way of model code reuse, and helps significantly in the modeling process. As Alecsis can interpret C/C++ routines while executing hardware models, it is very suited for descriptions of systems with embedded software.

We have developed new HDL, but we could not neglect the fact that the standard for digital system description already exists. In order to enable use of rich VHDL model libraries, we decided to support simulation of digital systems described in VHDL using Alecsis. Alecsis-VHDL co-simulation is achieved using the simulation kernel of Alecsis simulator (virtual processor). We have developed a compiler that converts VHDL source code into the object code for Alecsis virtual processor. Alecsis object code is designed for mixed-mode simulation, and it supports almost all VHDL modeling mechanisms.

2. Alecsis 2.3 simulator - an overview

The organization of Alecsis is shown in Fig. 2.1. The simulator essentially consists of language compiler, linker/loader and the simulation engine. The compiler converts high-level source files into data structures and instruction streams required by the simulation engine. The information obtained in this way may be deposited into the design library, or directly passed to the linker. Purpose of the linker is to resolve all global symbol references, since the system under simulation may be composed of pieces from many different libraries. The engine controls the simulation flow, a task that includes handling concurrent processes, updating signal values, solving sets of nonlinear differential equations and advancing the simulation time using various time step control strategies. In the following text, each mentioned block will be described in more details.

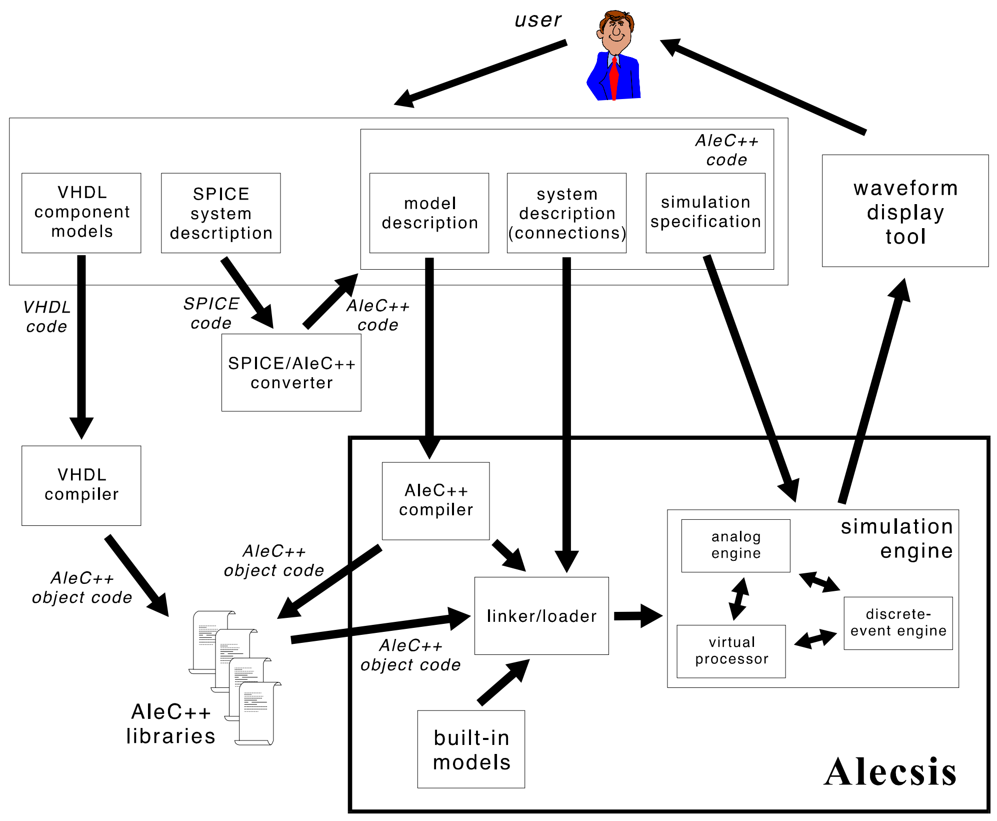


Fig. 2.1: Organization of Alecsis mixed-signal simulator.

2.1. AleC++ compiler

An important feature of AleC++ is that it can be used in interpreted and in compiled mode. If a HDL is based on a programming language, a straightforward solution would be to compile the HDL code into the appropriate programming language code and to link it with the simulation engine. Such a solution is used in another important C++ based modeling language ASL of simulator ARCHSIM [Anta95].

The solution that is simpler for use, but more complex for program implementation, is to create an interpreted language. The model development is much simpler if model code needs not to be linked to the simulation engine every time when it is modified. In an interpreted language, a model code would be executed directly after reading, command after command. However, such way of code execution is slow, since code optimization is not performed. Optimization is very important, since model code is executed many times during the simulation run.

We wanted to create a modeling language and a simulation program that are simple for use and efficient. From the user's point of view, AleC++ can behave both as an interpreted and as a compiled language. However, even if the user chooses interpreted mode, the language is compiled from the AleC++ source code into object code that can be optimized. That object code is interpreted by the virtual processor (Fig. 2.1).

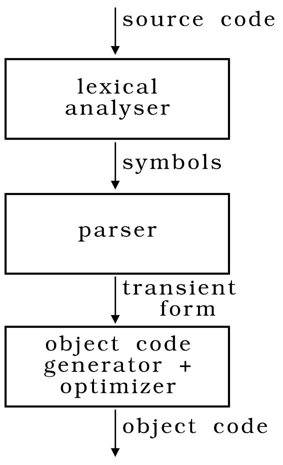


Fig. 2.2: AleC++ compiler organization.

The compiler represents the front-end of Alecsis simulation system. Its duty is to create data structures and instruction streams from an AleC++ input file. Since AleC++ is the superset of C++, its compiler consists of usual building blocks - lexical analyzer, intermediate form generator and object code generator, as shown in Fig. 2.2. After a preprocessor pass, the lexical analyzer groups individual characters from the preprocessed input stream into tokens, according to the lexical language rules. Tokens are then grouped and checked against AleC++ grammar in parser. The parser constructs binary trees, followed by an intermediate form (also known as "three-address code" [Aho86]). Such a conversion of user defined expression into a tree, where constant or variables are represented by leaf vertices, can be met in another C++ based language, created by Melville *et al.* [Melv93b]. In Alecsis, that tree is used to feed the object-code generator that also serves as an optimizer. Information may be passed further to the linker, or stored in the design library in the Alecsis object format (Fig. 2.1). Therefore, the user can use the code directly (interpreted mode), or store it in a library (compiled mode). In both cases, the instruction streams are not linked to the simulation engine, but are executed (interpreted) by a virtual processor. If an input file is not a model but an ordinary C++ program or subroutine, it can be interpreted by simulator Alecsis, too.

2.2. Linker/loader and library manager

Alecsis was designed to be robust and to provide for simulation of large circuits and systems. Such systems are usually composed of parts from many different libraries. The binding is performed by the linker/loader. There are four types of entities with external linkage in Alecsis - functions, global variables, modules and model cards. Functions and globals are terms borrowed from C++. Modules play the central role in circuit hierarchy description capabilities, while model cards behave similarly as in SPICE. They all may be stored in, or retrieved from the libraries, using simple or elaborated naming conventions. Alecsis employs type-safe linkage, which means that function calls with incorrect number and/or type of arguments will be caught by the linker due to a simple, yet effective strategy known as the name mangling. This strategy is also used to provide for function and operator overloading, a very useful C++ feature inherited by AleC++.

Using the services of Alecsis Library Manager (ALM), libraries may be merged, and items may be erased, copied or moved within or between libraries. It should be noted that linker performs early binding, while late binding, a feature peculiar to C++ and other object-oriented languages, is implemented using run-time mechanisms.

2.3. Simulation engine

Alecsis is an integrated mixed-signal simulator, and thus has only one control agent for both analog and logic simulation parts (mixed-mode simulation). Due to its generality, the engine controls both the elementary mechanisms for solving of sparse nonlinear equation sets and discrete-event simulation subroutines. To be able to cover a large variety of different circuit and system classes, only the most general algorithms are used. Nonlinear equation sets are solved using sparse matrix techniques and standard Newton-Raphson iterative method. Time integration of first-order differential equations is performed using Euler-backward and Gear2 methods. Logic entities are represented as concurrent processes that communicate via typed channels called signals. The role of the engine in that domain is to control the state of each process, activate or suspend processes according to their state, and propagate signal values as scheduled.

The data structures created by the compiler are used by the control agent. The instruction streams are executed as they are by the **virtual processor**. It is an instruction interpreter that loads and executes one instruction at a time. It was shown [Wang87] that simulators may be as much as three times less efficient when instructions are interpreted. However, we advocate our approach because of the following observations:

- Compiled-code simulators are known to generate large general-purpose language files, with as much as tens of thousands of lines. Thus, an efficient, final form is obtained at the price of slow compilation. In case of repetitive simulations (e.g. model development), the compilation turns into a real bottleneck, degrading the performance of the entire simulation chain.
- Since most of the analog models are nonlinear, it is likely that the modeling subroutines will be executed many times in an iterative process. The use of an interpreter in such a situation may degrade simulator performance. Fortunately, analog models tend to have many calls to mathematical library functions. If both the compiled and interpreted versions of the analog model use the same math routines, the resulting performance degradation may be smaller than expected, since those routines are the most critical from the efficiency point of view.
- The actual efficiency of an interpreter depends on its design and the level of code it interprets. Carefully crafted interpreters that operate on atomic instructions rather then the high-level code may be fast enough to be preferred solution in all but a few computationally intensive situations.

Led by the above mentioned observations, we have designed an efficient low-level interpreter that is used in Alecsis simulation engine whenever there is a need for instruction execution. Instructions are created by the AleC++ compiler and supplied by the linker/loader. The interpreter, called the virtual processor, has an instruction set similar to Motorola MC68xxx processors. Instructions are handled in two steps - *fetch* and *execute*. Fast fetching is the key to virtual processor efficiency, and it is obtained through the use of the resource tables, which are vectors of pointers to all the individual memory fields the operand may be loaded from. To verify the efficiency of the virtual processor, we compared many compiled and interpreted versions of the same code. For example, SPICE level-2 MOSFET model with hundreds of source lines executes only 1.2 to 1.3 times slower when interpreted, a performance that may easily be accepted for an interpreter.

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.

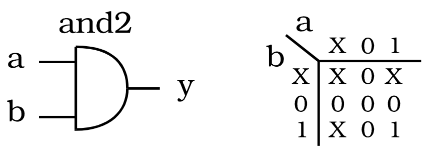


Fig. 3.1: Two-input AND gate model

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.

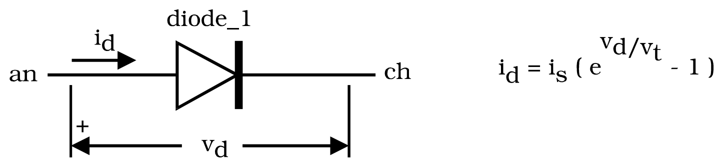


Fig. 3.2: Diode model

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`.

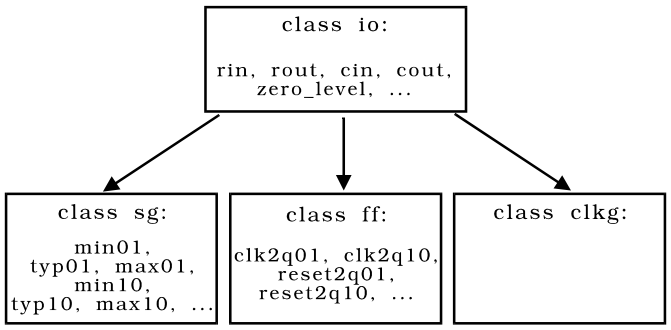

class io:
rin, rout, cin, cout, zero_level, ...
class sg:
min01, typ01, max01, min10, typ10, max10, ...
class ff:
clk2q01, clk2q10, reset2q01, reset2q10, ...
class clkg:

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.

4. Simulation algorithms

4.1. Analog engine

The use of modified nodal analysis (MNA) [Ho75] enables separation of the modeling process from the mathematical apparatus used for simulation. Contribution of each analog component to the system of equation is determined as the model “stamp”. In a behavioral simulator, model (i.e. stamp) can be defined by the user. On the other hand, the simulation algorithms are preprogrammed in a simulation engine. Alecsis have SPICE-like simulation algorithms implemented. In particular, we have used Gear methods for numerical integration of ordinary differential equations [Gear71], Newton-Raphson method for nonlinear equations, and modified Berry’s algorithm for solving system of equations characterized by sparse matrices [Berr71].

The user should not be aware of particular algorithms in the simulation engine when describing the model. However, the numerical integration of differential equations and the linearization of nonlinear equations are performed in the stamp description. Therefore, the modeling procedure cannot be quite independent from the algorithms in the simulation engine. However, we have tried to hide the algorithms of the simulation engine from the model designer. For instance, user can invoke operators `ddt` and `d2dt2` when he wants to describe differential equations. The method for numerical integration would be then invoked automatically. The same concept can be found in other analog hardware description languages [Getr89, Kazm92, Pabs95].

We will describe here shortly the algorithm of time advancing in the analog engine. This is important, since this algorithm has to be synchronized with the time-control of a discrete-event simulator. The analog simulation time is driven by the local truncation error (LTE) of the numerical integration method. This LTE is calculated after obtaining the convergence for one time instant. When the LTE is smaller than the allowed limit, the analog solution is accepted and the relation between LTE and the allowed limit is used to determine the new integration step (new simulation time). If the LTE is larger than the limit, the solution is discarded, and the simulation is repeated for a shorter time-step (backtracking in time). That shorter time step is again calculated on the basis of LTE value. Therefore, backtracking in the analog simulator is possible, but not more than for one time step.

4.2. Discrete-event engine

The logic simulation engine of Alecsis simulator uses the data structures built up by the compiler to propagate events through the digital nets - signals. An event represents a change in the value of a signal. When an event on a signal occurs, processes that are sensitive to that signal are activated. Each process that assigns states to a signal creates a driver for that signal. The driver is a time-ordered linked list that contains information of future events on a signal in the form of (time, state) pairs. Well-known inertial and transport delay mechanisms [Lips89] are used in assigning the future events to the driver. A signal can have more than one driver in which case the resolution function is needed to resolve the signal state. It is user's responsibility to provide the resolution function (applying preferred resolution strategy) and attach it to the signal that has more than one driver. In addition to events that convey updates in signal values, the logic simulation machine of Alecsis simulator also processes the control events imposed by the synchronization signals (`initial`, `per_moment`, `final`, ...) at appropriate moments during the simulation.

All events scheduled to occur in the future (relative to the current simulation time) are said to be pending and are maintained in a time-ordered linked list called the global event list (GEL). Besides the time of the event, an entry to the GEL contains the pointer to the signal driver that store the particular event information. GEL is used by the algorithm for advancing the time. The current time is the time of the first GEL entry. Simulation proceeds by processing all events queued at the current time (deleting them from the queue after processing) and then moving to process events at the next time in the GEL. Simulation ends when the GEL is empty or when user specified simulation duration is exceeded. When the process execution results in scheduling an event with zero propagation delay, the "delta delay" is used - this event is processed in the next simulation cycle, but the simulation time remains the same.

The changes in the values of the primary input signals are defined using `initial` processes. All the events defining the applied stimuli are inserted in the global event list and primary input signals drivers before the simulation. Clock generators can be defined that periodically insert events during the simulation.

Logic simulation algorithm consists of initialization and simulation phases. Initialization phase assumes activation of all processes and evaluation of states of all signals. Process activation is automatic, as well as the assigning the default or user defined values to the signals, but further initialization procedure must be built in the circuit model by the designer. The simulation phase assumes propagation of events from primary inputs and internal clock generators to the outputs of the circuit. Both phases use the same event handling procedures that can be divided into two basic operations:

1) evaluation of the new logic state of the signal and

2) execution of the activated process.

Before assigning the new state to the signal, the simulation machine applies the resolution function if the particular signal has more than one driver. If there is a change in signal state, the signal state is updated and its fan-out list is followed to determine the activated processes. When executed, in the next step, processes assign the new events to some signals. Those events are scheduled in drivers and GEL, and simulation proceeds by taking the next event from the GEL.

4.3. Hybrid simulation

For the mixed-signal simulation, analog and discrete-event engines should be connected into one simulation engine. There are two aspects of this connection:

- signal conversion (A/D or D/A) (mixed-signal simulation) and
- time synchronization of analog and discrete-event engines (mixed-mode simulation)..

4.3.1. A/D and D/A interfaces

A net to which both analog and digital components are connected is called hybrid net. After the compiler had built all the data structures representing the simulated circuit, Alecsis detects and eliminates hybrid nets. In order to make distinction between analog and digital portion of the circuit, simulator automatically inserts interface circuits called analogue to digital (A/D) and digital to analogue (D/A) converters. Models of interface circuits are part of system model and are closely related to the digital modules as described earlier.

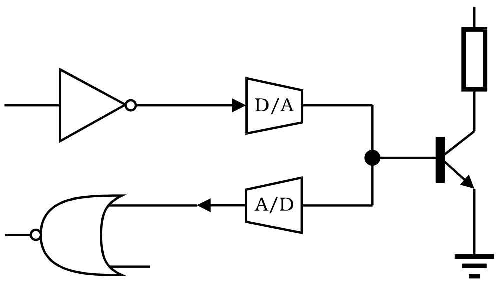


Fig. 4.1: Automatic A/D and D/A converter insertion for hybrid net to which both inputs and outputs of digital modules are connected.

A/D converter is inserted where an input of a digital module is connected to a hybrid node. D/A converter module is inserted where an output of a digital module is connected to a hybrid node. If there are both digital modules inputs and outputs connected to a hybrid node, both types of converters are inserted as illustrated in Fig. 4.1 [Nich92]. A possible A/D module structure for CMOS logic gates is shown in Fig. 4.2. The gate input is seen from the analog portion of the circuit as an input capacitance, while module `cmos_a2d` assigns new states to the digital output signal when the value of analog node crosses the defined thresholds. A/D converter module is parametrized in terms of threshold voltages and input capacitance. A simple model of D/A module for CMOS logic gate output is shown in Fig. 4.3. The gate output impedance is modeled by an RC circuit, while the signal transfer is expressed by a current source controlled by the logic states at the digital output. The interface is parametrized in terms of the output impedances and transition times. An 'X' (unknown) state from the logic circuit cannot be mapped directly into voltages, currents or impedances. The D/A converter module can be parametrized to convert an 'X' to '0', '1' or the previous determinate state.

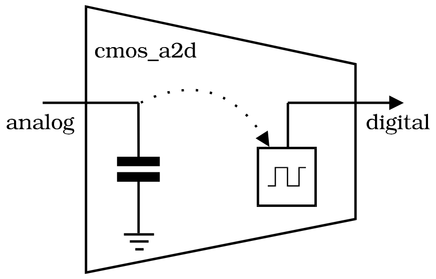


Fig. 4.2: Simple A/D converter model for CMOS gates. Input capacitance of CMOS gate is inserted at the analog side, and digital events are transmitted to the digital side when analog variable crosses the specified thresholds.

In order to produce a smooth ramp from the previous logic state to the new one, the converter should start generating the ramp *before* the actual event occurs. Since it is not possible in logic simulation machine to backtrack the current simulation time, digital modules are bound to assign events to the hybrid nets 1/2 of the ramp duration time (transition time) before it actually occurs. Therefore, it is necessary that the transition time (parameter of the converter) is a member of the model card of the digital module. This advocates the used model class inheritance concept described earlier. Detection if the net has a hybrid character is provided by the signal attribute `hybrid`.

```
        if (y->hybrid)
              y <- a&b after this->delay(a&b, y) -
                          this->get_trans_time() / 2.0;
        else
              y <- a&b after delay;
```

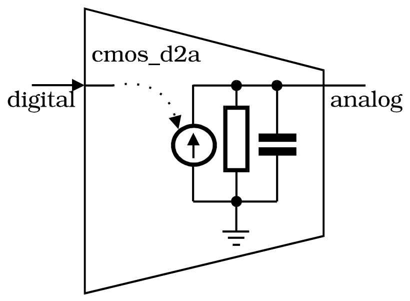


Fig. 4.3: Simple D/A converter model for CMOS gates. The output of CMOS gate is modeled with output resistance and capacitance, while its driving capability is modeled with current source controlled by the gate output logic state.

After inserting converter modules, digital and analog portions of the circuit can be separately simulated.

4.3.2. Initialization

The initialization phase in the mixed-signal circuit concerns both digital and analog part. Simulation of analog circuits requires a consistent initial state, otherwise the whole analysis may fail. Since analog simulation cannot cope with unknown state, the simulator performs initialization in the following manner:

1) Firstly, initialize the digital portion, i.e. compute the initial values of signals.

2) Than, execute a DC operation point computation in analog portion.

Thanks to modeling capabilities of AleC++, it is possible to include into digital modules the initialization mode as well as the normal operational mode. Initialization of digital part of the circuit is performed by the zero-delay logic simulation. To assure that zero delay is used while digital part settles in a stable state, user can check current simulation time (control parameter now), as in the next example.

```
if (now==0.0s) {  //  initialization mode
  out <- a & b after 0.0s;
}
else {  //  normal operational mode
  out <- a & b after delay_func(out, a & b, trise, tfall);
}
```

If digital simulation agent fails to initialize some signals that are needed for analog part initialization due to unknown input states that comes from analog part of the circuit, different initialization strategies can be applied, but those strategies depend on modeler and cannot be automated.

4.3.3. Time control mechanism

As we have described above, analog and logic simulation engines have completely different mechanisms of advancing the time. When a mixed-signal simulator is implemented, these two algorithms should be unified. In our implementation, the analog engine is the main one, it behaves also as a synchronization agent. When it is necessary, the analog engine invokes the discrete-event engine (Fig.2.1). The usual LTE-driven time algorithm of the analog engine is modified to allow for synchronization with the discrete-event engine:

- If the logic event is waiting to be executed in a particular time in the event list, analog simulation time cannot advance over that time. That means, analog time step would be shortened to match the time instant when the logic event happens. In that time instant, both engines are active (iteratively, if necessary, until all logic events for that time instant are executed).
- When the analog engine solves given time instant, that solution can give conditions for generating a logic event. However, the actual time instant when the analog value passes the threshold is not known precisely, since the analog time step is finite [Brow91]. For that reason, this analog time step can be discarded and the time step shortened. In this way, time-instant when the given analog value passes the threshold, i.e. when the logic event is generated on A/D node can be found with better accuracy.

Therefore, in mixed-signal simulation, logic engine is not responsible for advancing the time. When necessary, the analog engine invokes the logic engine to execute one delta-cycle of events in a given time-instant.

5. Application notes

Introduction

During the development, Alecsis was constantly verified on a large number of different examples. Also, it was used in some real projects, where different electronic circuits have been designed. We will give here some of the typical simulation examples where Alecsis is presented as a circuit and system (mixed-domain) simulator, used in different analogue, discrete-event and mixed-signal applications.

The first example is an analog multiplier, where Alecsis is used as a circuit simulator, i.e. equivalently as SPICE. SPICE compatibility is enabled through the usage of similar syntax rules and same syntax of model card. In the next two examples of SC filter and switching flyback converter, Alecsis is used also as a circuit simulator. Nevertheless, an ideal switch model that is a peculiar feature of Alecsis is used there. Such simulations cannot be so easily performed in other circuit simulators.

After that, five examples are given that depict analogue modeling features of AleC++. Firstly, a representation of MOS transistor modeling using neural network is given. After that, example of fuzzy-logic control system is presented.

Variables used there for modeling are not only of electronic, but also of very abstract nature. An example of control system modeling using transfer functions is given after that. Later on, example of electromechanical system modeling is presented through a nonlinear electromagnetic circuit. As a last example of analogue modeling features of Alecsis, a pressure sensing system is used. In this electromechanical simulation, space-continuous models, i.e. partial differential equations, are described in AleC++.

Ninth and tenth example in this section are of discrete-event simulation. In the systolic array circuit, AleC++ is used for modeling of logic circuits. After that, in LAN network simulation, models of more abstract discrete-event systems are described.

Three examples of mixed-mode and mixed-signal simulation are also presented. For A/D converter simulation, analogue and logic circuitry are coupled. In the next example, sigma-delta modulator is simulated, which is considered to be one of the benchmark tests for mixed-mode simulation. The last example of this group is pressure sensor simulation, with analogue readout circuitry and analog signal conversion into a discrete value. Therefore, in this example mixed-domain, mixed-signal and mixed-mode simulation is performed.

In discrete-event system modeling, Alecsis is able to support not also AleC++, but also VHDL. The last example in this chapter gives such AleC++ / VHDL co-simulation.

In the examples throughout this chapter some characteristic parts of model code are given. They are used just to illustrate usage of the modeling language AleC++. For learning the syntax of the language, Alecsis User's Manual is available.

5.1. Analogue multiplier simulation

The common way of analog systems modeling is structural description that stands for coupling of components with links and definition of components' parameters. As an example of pure structural description, the analog multiplier shown in Fig. 5.1 will be considered. It consists of 19 MOS transistors, and has the following functional blocks: pre-distortion circuit, voltage controlled current source, and two differential stages. Multiplier output is between nodes 18 and 19.

Multiplier is described structurally in a way that circuit elements are first declared:

```
resistor r1,r2;
```

meaning that components `r1` and `r2` are resistors. It is also possible to use implicit declaration:

```
implicit { resistor r; capacitor c; }
```

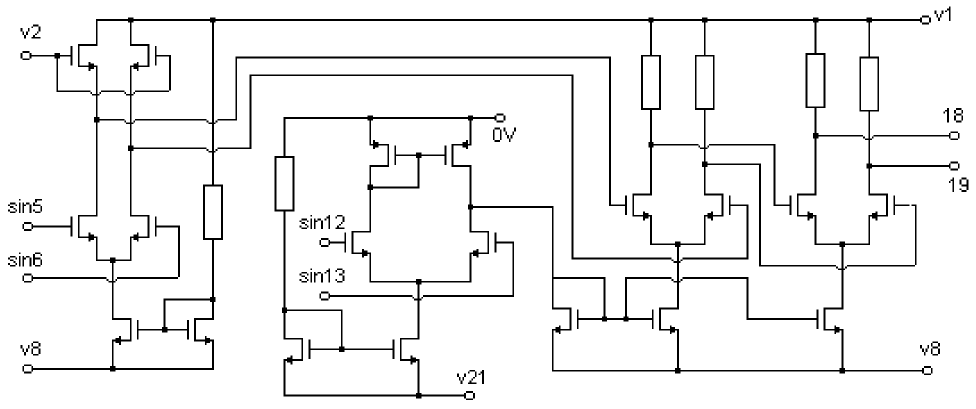


Fig. 5.1: Analog multiplier with MOS transistors

which means that all elements beginning with `r` are resistors, and all ones beginning with `c` are capacitors.

After declaring the components to be used, structural description of the multiplier is similar to SPICE circuit description. Nodes where the component is plugged are given in parenthesis, and parameters (including the name of the model card) are given afterwards. A part of the code describing the multiplier is given that shows the instantiation of an MOS transistor.

```
mp2 (11,10,0,0) { model=PC_PM1; w=1.7u; l=1.2u;
                  ad=150p; as=100p; pd=10p; ps=10p; }
```

Of course, the model card `PC_PM1` must be defined earlier in the simulation library. It has the same syntax as SPICE MOS model, so it will not be described in details here.

To demonstrate the simulation of the multiplier depicted in Fig. 5.1, two sine wave signals with different frequencies are brought to its inputs. Multiplier output is an amplitude-modulated signal. The simulation results are given in Fig. 5.2.

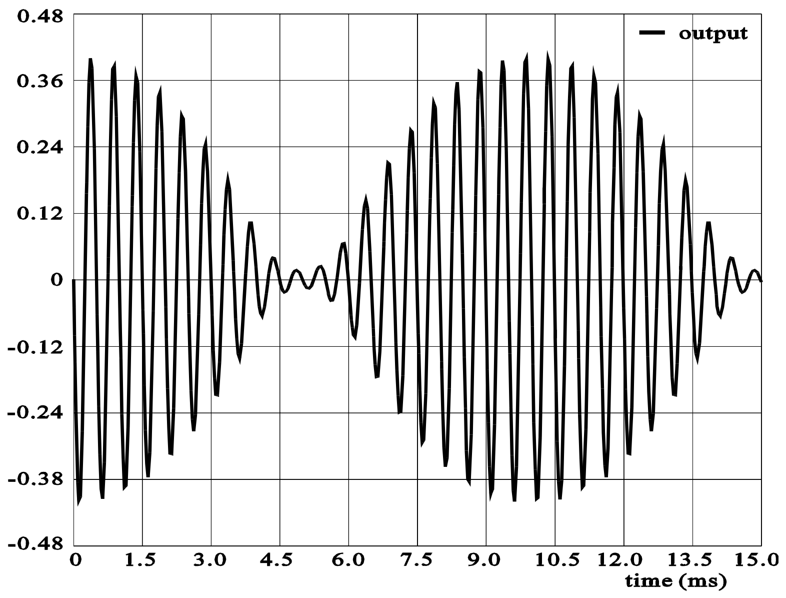


Fig. 5.2: Results of analog multiplier simulation

5.2. SC filter

This example also employs only structural description. The circuit shown in Fig. 5.3 contains switches, therefore the ideal switch, an Alecsis built-in component, is used. It should be noted that a truly ideal switch is used, with resistance zero for closed switch and infinite resistance for open switch [Mr~a93]

The parameters `val_on` and `val_off` determine voltages that turn switch on and off, respectively. There is another parameter `hyst` that defines switch with hysteresis if set on 1. The switch description requires four nodes: the first two are topological information of switch position in the circuit, and last two are controlling nodes. The double switch from Fig. 5.3 is described as a module named `Double_Switch`. It consists of two switches that charge and discharge capacitors making them behave as they were resistors.

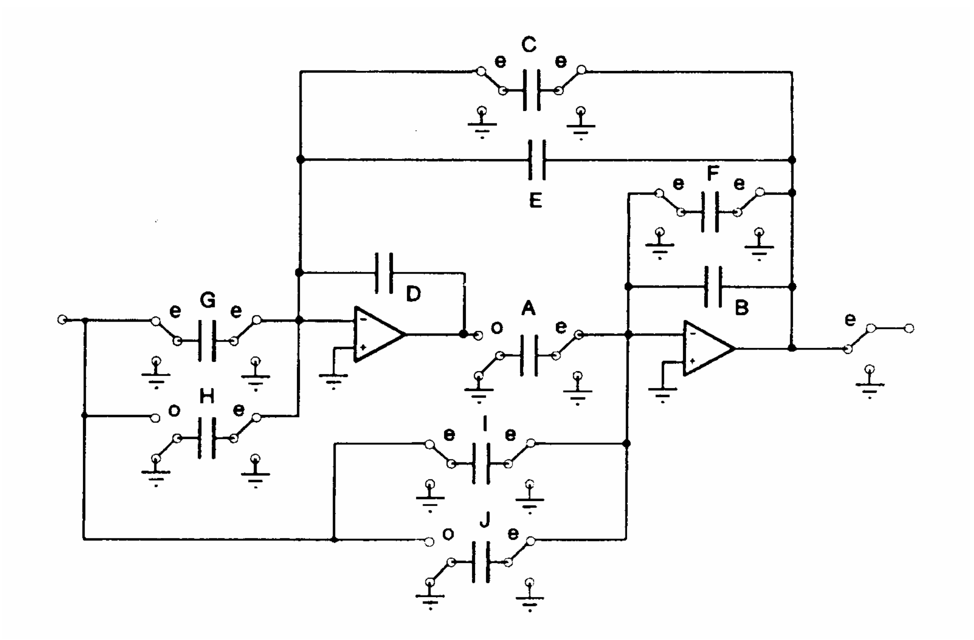


Fig. 5.3: SC filter circuit.

```
module Double_Switch (even,odd,common,commutation) {
        // Declaration section
```

```
                switch se,so;
              // Structural section
                se (common,even,commutation,0) val_on=val_off=0;
                so (common,odd,0,commutation) val_on=val_off=0;
        }
```

The SC filter circuit is excited through a sample and hold circuit, shown in Fig 5.4.

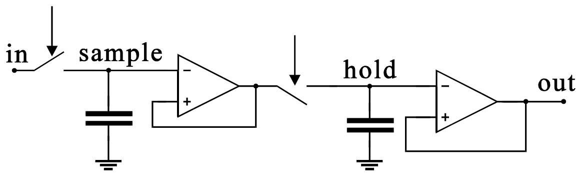


Fig. 5.4: Sample and hold circuit

The sample/hold circuit is also described as a separate module `SH_circuit`, with three interface nodes.

```
        module SH_circuit (node input, output, commutation) {
          switch se,so;
          capacitor c1,c2;
          opamp opamp1,opamp2;

          se (input,sample,commutation,0) {val_on=val_off=0;}
          c1 (sample,0) 1pF;
          opamp1 (internal,sample,internal);
          so (internal,hold,0,commutation) {val_on=val_off=0;}
          c2 (hold,0) 1pF;
          opamp2 (output,hold,output) ;
        }
```

Switches are controlled by the voltage at the node called `commutation`. Its value, depending on weather it is greater or less than zero, determines state of the switches. Operational amplifiers are modeled as ideal (single voltage controlled source).

For the simulation, sine wave of frequency 128kHz is used to control the switches. In Fig. 5.5 simulation results are shown: excitation and response of the SC filter circuit.

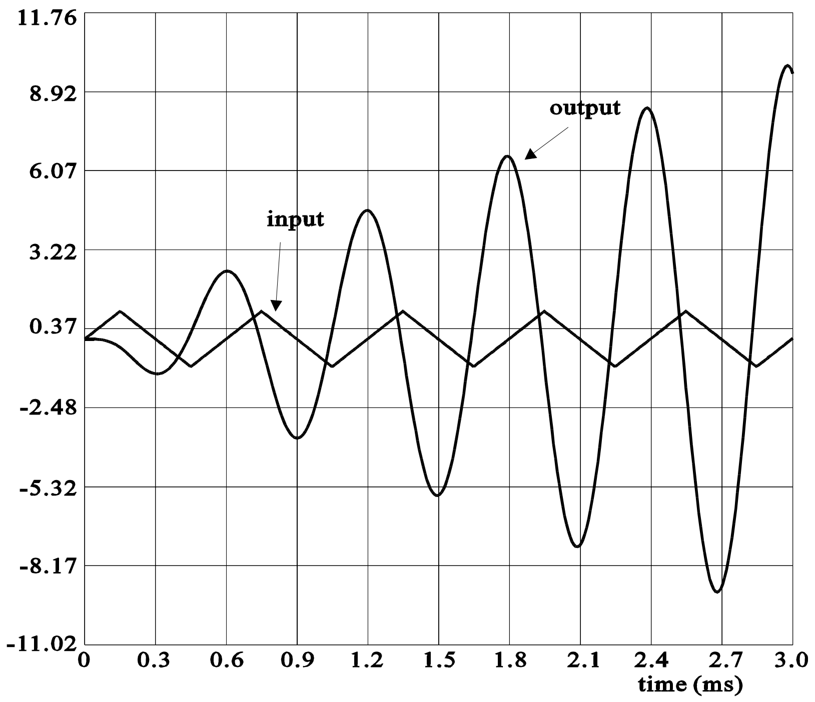


Fig. 5.5: Simulation results for SC circuit excited by triangular input signal.

5.3. Switching voltage regulators

The ideal switch model is suited for all types of circuits. A particularly important class of switched circuits is that of switching voltage regulators. One of the main problems in simulation of such class of circuits is occurrence of inconsistent initial conditions. These conditions occur after switching, when instantaneous change of the capacitor voltage or inductor currents can happen. In such cases, current through capacitor and voltage over inductor have infinite values, but such impulses have infinitely short duration (Dirac impulses). Such problems are usually solved by the application of special algorithms for switched networks [Opal90].

With our model, we have modeled the transition of the switch state, i.e. the switching is not represented just as a replacement of one network topology by another. With our nonlinear model, the switch transition is performed through number of iterations, where, in every iteration, both Kirchhoff laws are satisfied [Mr~a99]. Networks with inconsistent initial conditions are solved using standard circuit simulation algorithms. This enables changes in the level of abstraction used in modeling, without change of the simulation environment.

As identified in [Bedr92, Vlac95], the problem of inconsistent initial conditions is not only to conserve the charge and the flux. It is very important to take into account Dirac impulses that can occur in the moment of switching, especially if the network contains internally controlled switches. A Dirac impulse can itself change the states of some switches in the network, thus changing the network topology once more in the same time instant.

The problem will be depicted by the ideal flyback switching converter, given in Fig. 5.6(a). The switch *s* represents the transistor that is externally controlled. The diode *D* is also modeled as ideal switch, but internally controlled. One can model this diode using a control variable *p* [Bedr92]:

$$D\!:\begin{cases} ON & if\ p>0 \\ OFF & if\ p<0 \end{cases}, \qquad p=\begin{cases} i & if\ D\ \ is\ \ closed \\ v_j - v_k & if\ D\ \ is\ \ open \end{cases}. \tag{5.1}$$

Therefore, the state of the switch *D* is controlled by the variables in the circuit itself. Let us start the analysis with switch *s* closed and switch *D* open. The equivalent circuit is shown in Fig. 5.6(b). Inductor current i_L is linearly increasing. When *s* is externally opened, the inductor current has no closed loop. Therefore, i_L must drop instantaneously to zero. Because of that, a Dirac impulse of voltage appears at the inductor. This Dirac impulse changes the switch control variable *p* to a positive value (eqn. 5.1.), and D becomes closed. Since Dirac impulse has zero duration, *D* is closed in the same time instant when *s* is opened. When this happens, the current of the inductor has a closed loop to flow (Fig. 5.6(c)), and there is no discontinuity in the inductor current. Therefore, the

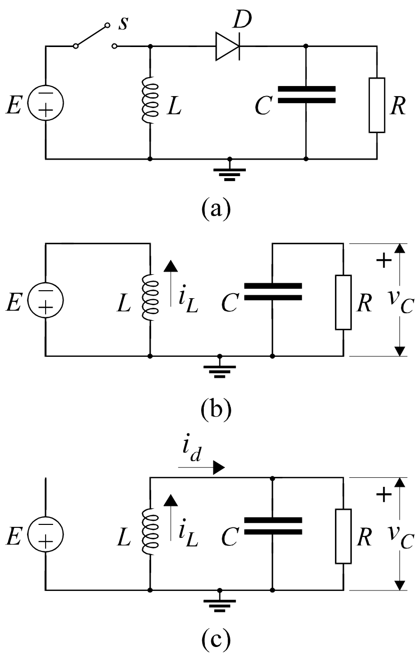


Fig. 5.6: (a) Ideal flyback switching converter. (b) Equivalent circuit for *s* closed and *D* open. (c) Equivalent circuits for *s* open and *D* closed.

Dirac impulse of the inductor voltage would be erased. There cannot be any impulse of the inductor voltage in simulation results, but the Dirac impulse must appear to switch the diode *D*, i.e. to change the value of the control variable *p*. Similar condition happens when *s* is externally closed. Dirac impulse of current appears through capacitor *C*, but it opens diode *D* in the same time instance.

In [Bedr92], special algorithm is developed for simulation of such networks, but with linear elements only. Our switch model enables simulation of such networks using exclusively standard simulation algorithms, and without any restriction in network topology, possible switch states and model linearity. The simulation results for the flyback converter are given in Fig. 5.7. The element values are *E*=1 V, *L*=150 mH, *C*=50 mF, *R*=10 W. The switching period for the switch *s* is 70ms, time when *s* is closed is 30ms, when it is open is 40ms (duty cycle is $\frac{3}{7}$). These parameters are not optimized from the point of view of circuit performance.

The results show correct transitions of the internally controlled switch *D*. The model is nonlinear, since it uses an iterative procedure for transition from one switch state into another. During this procedure, the existence of Dirac impulse is automatically taken into account, although no special algorithms are used for that. Therefore, the Dirac impulse occurs only in the iterative process, but not in the final solution given in Fig. 5.7, where current i_L and voltage v_C are continuous. However, the Dirac impulse can change the states of other switches in the circuit. The convergence can occur only when all switches in the circuit reach their final state. There is no need to create any special algorithm that would check for the possible occurrence of the Dirac impulses and their influence to the switch states. When the convergence is reached in the whole circuit, all switch transitions are finished and consistent state is obtained, with charge and flux conservation. Details of the switch implementation and analysis of this iterative procedure are given in `[Mr~a99]`.

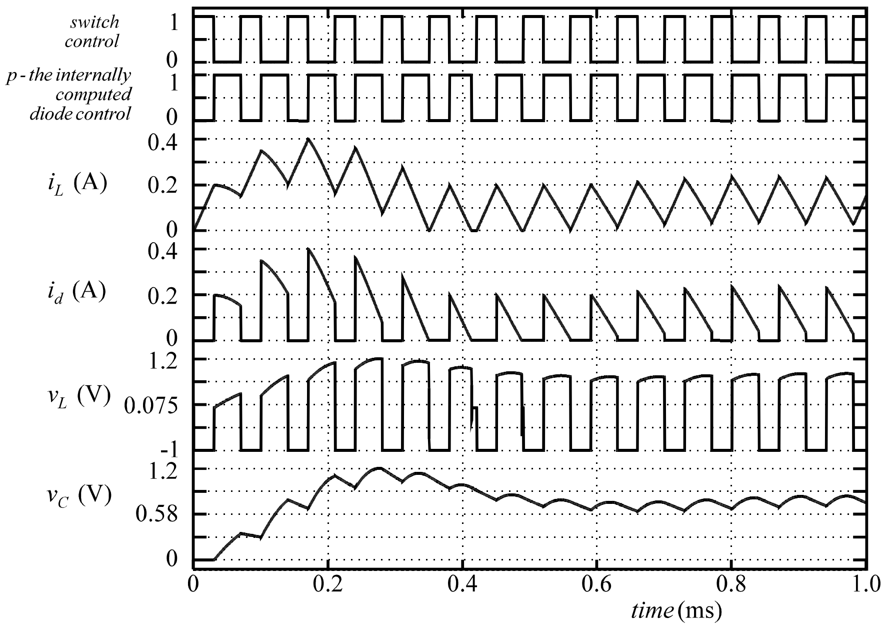


Fig. 5.7: Simulation results for the flyback converter.

5.4. MOSFET modeling using artificial neural network

As shown in section 3.3, modeling of nonlinear analog devices using model cards as static objects has the advantage of separating the nonlinear function and the enclosing process. As long as model class methods accept terminal voltages and return both nonlinear function value and partial derivatives, the actual method implementation is of no importance. We used this property to develop a model class whose methods model MOSFET drain current using artificial neural network. It was shown that an artificial feed-forward neural network may be trained to behave like a nonlinear device, given the sufficient input-output data set, obtained either by measurements or simulation [Lito92, Lito93]. Such a network has as many neurons in the input layer as there are input variables (in our case voltages Vgs and Vds), and one output neuron (Ids). Training has shown that one hidden layer with 10 neurons is sufficient to obtain neural network response with an acceptable error. The results of the training were weights and thresholds associated with the neurons, and they were introduced in the defined model card class as parameters. Once the neural network response method was implemented, the MOSFET model did not differ from the example given in section 3.3, with the identical interface type signature.

The following declaration was used to introduce a model class `annmos`:

```
class annmos {
  private:
    double w[32], t[12], x[50];   // weights, thresholds
    double oj[11], ojg[11];       // intermediate results
    int n, n0, np;                // network dimensions
    int hidden_transfer, outtransfer;   // transfer type
    double hidden_gain, outgain;        // transfer gains
    annmos(int, int, int);        // class constructor
    >annmos();                    // model card preprocessor
    void create_unified_neurodata();    // class unifier
  public:
    double vds_min, vgs_min;      // vds normalization
    double vds_max, vgs_max;      // vgs normalization
    double ids_min, ids_max;      // ids normalization
    double vgsd, vdsd, idsd;      // normalized values
    double Cgd, Cgs, Cgb;         // MOS capacitances
    double Cbd, Cbs;              // junction capacitances
    double type;                  // device polarization
    void neuro_response(double *, double *); // net response
};
```

A subcircuit that defines artificial neural network MOSFET model is defined by the use of structural modeling:

```
module annmos::neuromos ( node drain, gate, source ) {
    vccs e_gm, e_gds;
    cgen iaux;
    // voltage-controlled current sources
    e_gm (drain, source, gate, source);
    e_gds (drain, source, drain, source);
    iaux (drain, source);        // error current
    // capacitances
    cgs (gate, source);
    cgd (gate, drain);
    cds (drain, source);
    action {        // model of the behavior
        process initial { *cgs=Cgs; *cgd=Cgd; *cds=Cds; }
        process per_iteration {
            double inp[3], ok[4];
            double vg, vd, vs, vgs, vds, ids, gm, gds;
            // extract controlling voltages
            vd = drain;   vg = gate;   vs = source;
            vgd = type * (vd - vs);
            vgs = type * (vg - vs);
            // normalize input vector
            inp[INP_VGS] = (vgs - vgs_min)/vgsd;
            inp[INP_VDS] = (vds - vds_min)/vdsd;
            this->neuro_response ( ok, inp );
            ids = ok[1] * idsd + ids_min; // denormalize Id
            gds = ok[2] * idsd / vdsd;     // denormalize gds
            gm = ok[3] * idsd / vgsd;      // denormalize gm
            // update the linear model elements
            iaux->value = type * (ids - gm*vgs - gds*vds);
            e_gm->gm = gm;
            e_gds->gm = gds;
        }
    }
}
```

The simulation of a CMOS inverter composed of two ANNMOS (Artificial Neural Network MOS) devices is illustrated in the Fig. 5.8. Since the nonlinear function that models MOSFET drain current has continuous first derivatives in the entire range of interest (four orders of magnitude), no convergence problems occurred in the transition regions. It should be noted that this modeling method may be used for any nonlinear device or system, given sufficient input-output relation data.

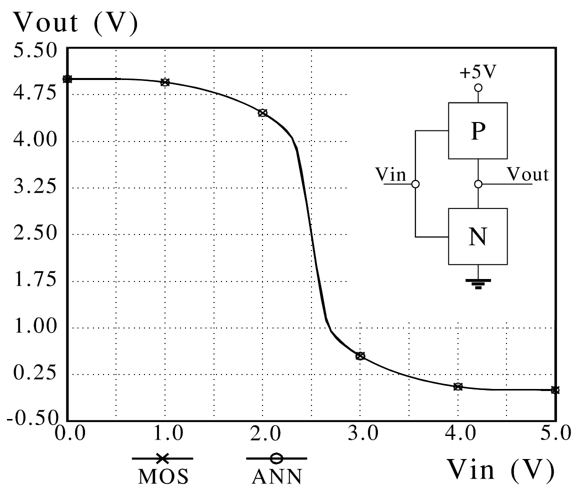


Fig. 5.8: Comparison of the simulation results of a CMOS inverter composed of standard transistor models (MOS) and artificial neural network models (ANN). Two curves evidently match in the whole voltage domain of interest.

5.5. Adaptive fuzzy-logic vehicle engine controller

Fuzzy logic control systems are particularly interesting as mixed-signal systems. Engineers often use general-purpose mathematical packages to develop fuzzy logic control systems. However, those systems eventually end up in hardware, when electronic circuit simulators should be used. Due to AleC++ modeling power, Alecsis is capable of supporting the entire design process, from the conceptual to the final stage. The example of such a system may be an adaptive fuzzy-logic vehicle engine controller, shown in Fig. 5.9.

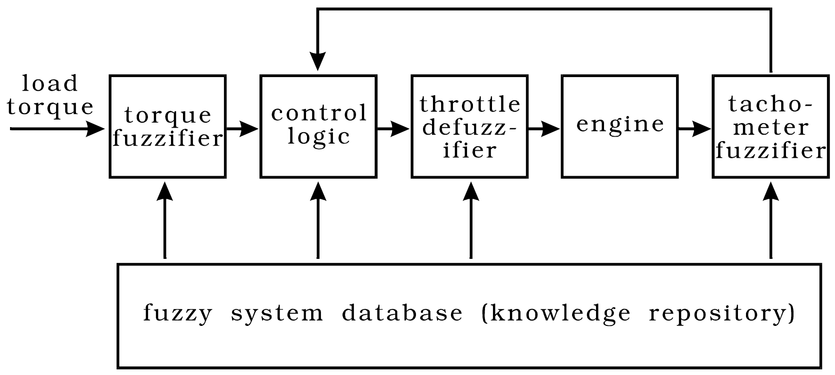


Fig. 5.9: Adaptive fuzzy-logic vehicle engine control system block diagram

The system controls a vehicle engine in order to respond to changes in the road grade [Cox93]. The engine speed is monitored using a tachometer. In order to solve the control problem using the fuzzy-logic approach, the tachometer readings and the road grade are first fuzzyfied. This information is used to produce a fuzzyfied engine throttle movement value in the control logic block. The exact (crisp) control value is obtained in the defuzzyfication module, and then applied to the engine. As expected, the entire system is parameterized by creating a new model class, containing data about fuzzy regions, domain knowledge in the form of a fuzzy associative matrix, rule weights and so on. Connections between the modules are modeled using variables of type `flow` for crisp values, and structures for fuzzy values. The entire system is self-adaptive, i.e. capable of responding to long-range changes in the environment. The behavior of all of the modules is implemented as model class methods, including the self-adaptation mechanism. The vehicle engine was modeled using an ordinary differential equation by means of AleC++ `eqn` statement.

The first step in modeling for Alecsis is to create a new class that describes resources needed in the simulation:

```
class fuzzy {
    int ntach;   // number of tachometer fuzzy values
    int nlt;     // number of load torque fuzzy signals
    int ntas;    // number of throttle mov. fuzzy values
    FUZZY_VALUE tach[TACHSIZE]; // tachometer fuzzy logic system
    FUZZY_VALUE torq[LTSIZE]; // load torque fuzzy logic system
    FUZZY_VALUE tas[TMSIZE];  // throttle action fuzzy log. sys.
    double tcmin, tcmax;// tachometer range
    double trqmin, trqmax;     // load torque range
    double tamin, tamax;// throttle action range
    int fam[LTSIZE][TACHSIZE];// fuzzy assoc. memory (knowledge)
    int center_of_response[2];// steady-state location
    double weights[LTSIZE][TACHSIZE];  // FAM weights matrix
    void fuzzy(int);           // class constructor
    void ~fuzzy(int);          // class destructor
    double engine_response;    // engine reaction
  public:
    double memf(FUZZY_VALUE*,double,double); // membership func.
    void evaluate_output(double*,double*,double*,int);// mapping
    double integrate(double*); // evaluation of centroid
    void update_weights();     // auto-adaptation of weights
};
```

After the definition of the model class, an unlimited number of model cards of that class may be constructed. An example may look like this:

```
model fuzzy::speed_cont {
    ntach=5;   nlt=4;      ntas=5;     tcmin=0.1;   tcmax=8.0;
    trqmin=0;  trqmax=40;  tamin=-60;  tamax=60;
    // tachometer fuzzy value system coordinates (feedback)
    tach = { { "very_slow", { 0.1,  0.1,  0.4,  0.8 } },
             { "slow",      { 0.5,  1.65, 1.65, 2.8 } },
             { "optimal",   { 2.0,  3.25, 3.25, 4.5 } },
             { "fast",      { 3.2,  4.7,  4.7,  6.2 } },
             { "very_fast", { 5.5,  7.0,  8.0,  8.0 } } };
    // load torque fuzzy value system coordinates (feed forward)
    torq = { { "zero",              {  0,    0,  2.5, 10 } },
             { "small_positive",    {  5, 12.5, 12.5, 20 } },
             { "moderate_positive", { 15, 22.5, 22.5, 30 } },
             { "large_positive",    { 25, 32.5,   40, 40 } } };
    // throttle action fuzzy value system coordinates (output)
    tas = { { "LN", { -60, -60, -45, -30 } },
            { "SN", { -40, -20, -20,  -2 } },
            { "ZR", { -10,   0,   0,  10 } },
            { "SP", {   2,  20,  20,  40 } },

            { "LP", {  30,  45,  60,  60 } } };
    // fuzzy associative memory (knowledge repository)
    fam = { { LP, SP, ZR, SN, LN },
            { LP, SP, ZR, ZR, SN },
            { LP, SP, SP, SP, ZR },
            { LP, LP, LP, SP, SP } };
    // initial equilibrium center on the control surface
    center_of_response = {Optimal, Zero };
    // initial rule contribution weights - all rules are equal
    weights = { { 1, 1, 1, 1, 1 },
                { 1, 1, 1, 1, 1 },
                { 1, 1, 1, 1, 1 },
                { 1, 1, 1, 1, 1 } };
}
```

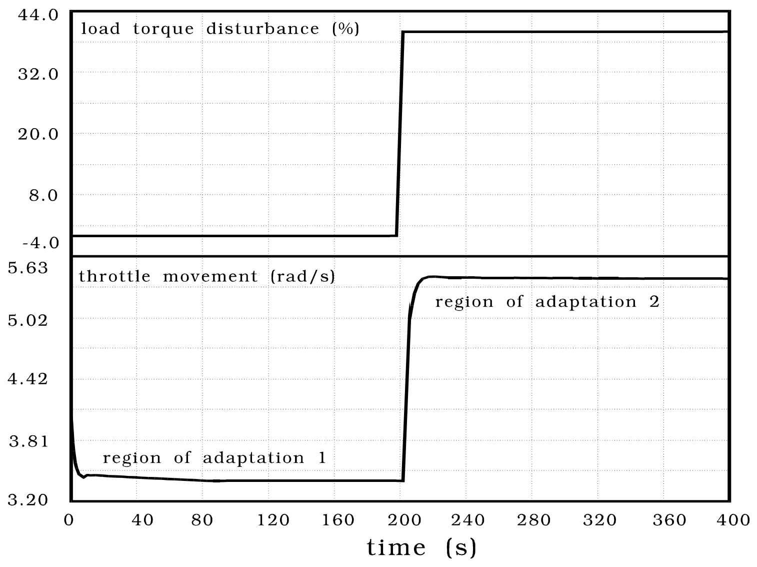


Fig. 5.10: Vehicle engine control system response to rapid change in the road grade

Fig. 5.10 shows the system response to a rapid change in the road grade. Two distinct regions exist in the response curve: one, that represents the primary response, and another, where the system slowly adjusts the rule weights to move the center of the response to a different location. This effect is shown in the weight surfaces in the Fig. 5.11. The center-of-response displacement is clearly visible on the surface contour lines.

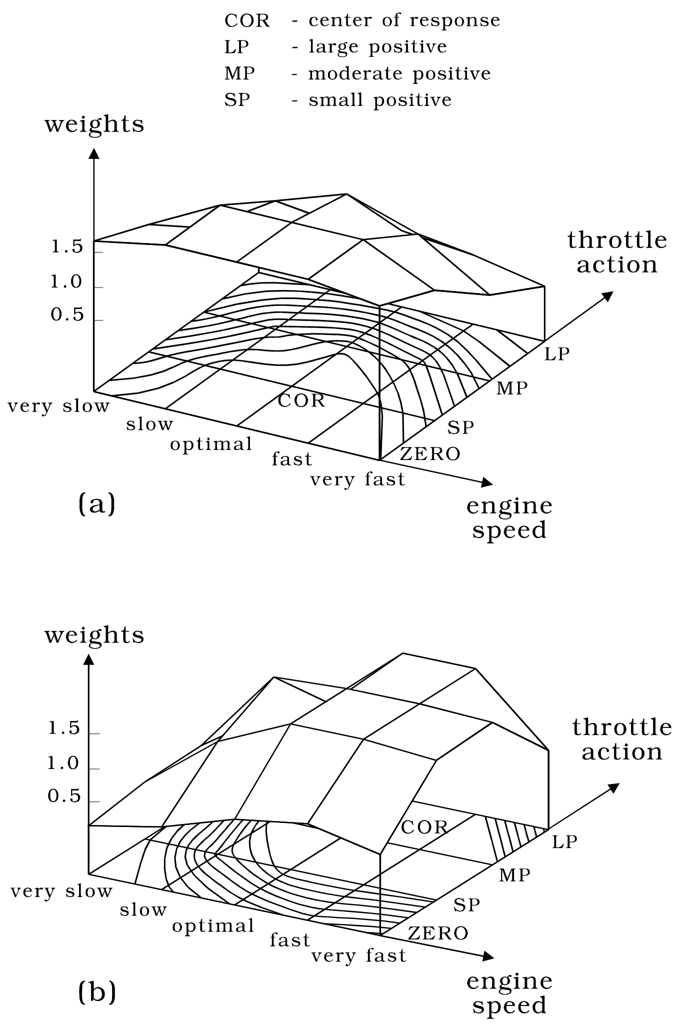


Fig. 5.11: Weight surfaces before (a) and after (b) the road grade change. The center of response has visibly moved towards the new equilibrium point.

5.6. Continuous time control system simulation

When systems are described on a higher level of abstractions, block diagrams can be used for their representation. Control systems are often represented using such description. A transfer function in *s*-domain of one subsystem (block) can be represented as a rational function:

$$H(s) = \frac{D(s)}{N(s)} = \frac{b_0 + b_1 s + \ldots + b_m s^m}{a_0 + a_1 s + \ldots + a_n s^n} \tag{5.2}$$

Such expressions can be realized as shown in Fig. 5.12, when the control form representation is used.

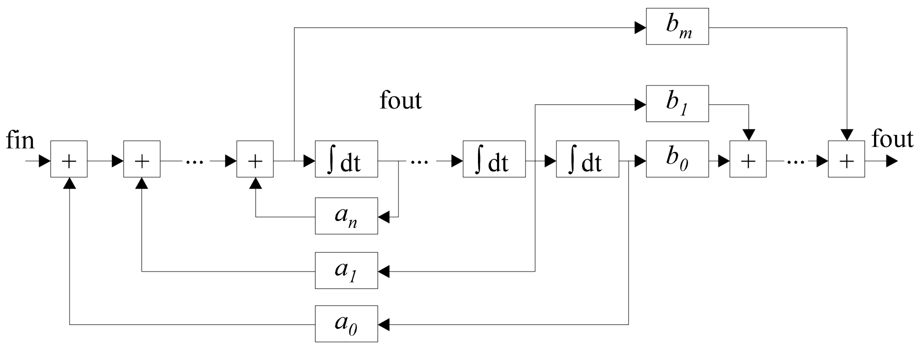


Fig. 5.12: Control form representation.

When writing transfer function module, we are using modules that describe smaller blocks in Fig. 5.12. For instance, integrator block is represented through:

```
module integral (flow fin, fout) {
   action (double w=1.0) {
      process per_iteration {
         eqn fout: { fout } - w * idt { fin } = 0;
      }
   }
}
```

The command `clone` was used to model an array of integrators, as well as gain stages that create inputs to the adders. This is contained inside the module `transfer` (not given here), representing the block scheme from the Fig. 5.12.

As an example of continuous-time system simulation the response of a pulse compressing allpass filter will be evaluated. The transfer function of the circuit is given by

$$T(s) = \prod_{i=1}^{6} \frac{s^2 - 2\sigma_i s + \left(\sigma_i^2 + \omega_i^2\right)}{s^2 + 2\sigma_i s + \left(\sigma_i^2 + \omega_i^2\right)},$$

$(\sigma, \omega)_i$ being the pole/zero coordinates as given in Table 5.1.

Table 5.1:

i	σ_i	ω_i
1	0.051690	0.568976
2	0.057089	0.646074
3	0.064608	0.731975
4	0.076094	0.830426
5	0.096704	0.948449
6	0.147276	1.098074

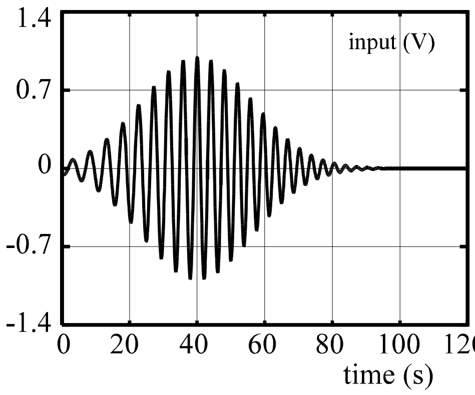


Fig 5.13.a: The input signal

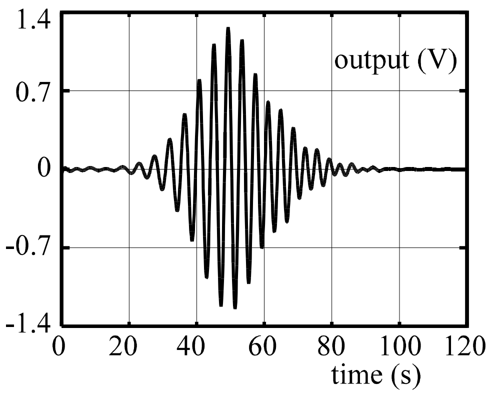


Fig 5.13.b: The output signal

The input signal is frequency modulated and gaussian shaped:

$$v_{in} = e^{-at^2} \cos\left(\omega_0 t + bt^2\right),$$

where ω_0 = 1.52344, a = 0.11033215 and b = 0.25794107 [Laz72], given in Fig 5.13.a.

The output (amplified and monochromatic) signal as obtained by simulation is depicted in Fig 5.13.b.

Here, again, the `clone` command was used in order to replicate the basic second order allpass cell six times.

5.7. Nonlinear magnetic circuit modeled using artificial neural network

Direct current magnets have found a wide range of applications, as actuating elements in devices with automatic control, in precision mechanics, etc. Latest applications of these magnets have set more rigorous requests with regard to increase of the switching speed and dimension miniaturization. It is very important to model and simulate electromagnets, in order to optimize their parameters. Especially important is transient analysis, where the goal is usually to find the optimal waveform of the input voltage in order to obtain fast switching of the armature, but to avoid damage of the contacts caused by collisions.

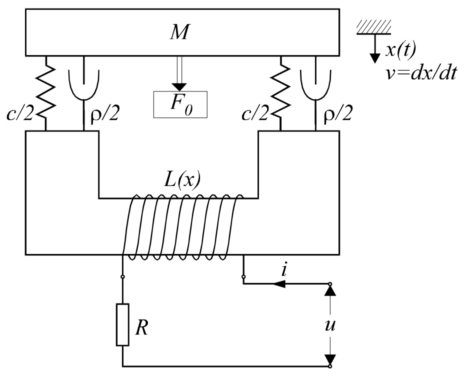


Fig 5.14: An electromagnetic circuit

Schematic representation of a DC magnet is shown in Fig. 5.14. Position of the electromagnet is determined by the equilibrium of mechanical and magnetic forces:

$$F_{mag} = F_{mech} \tag{5.4}$$

Mechanical force is a sum of four components. These may be expressed as shown in Table 5.2.

Notation used here is: M for mass, c for spring constant, ρ for friction resistance.

Table 5.2: Mechanical forces acting on the armature.

Mass	Spring	Friction	Constant force
$F = M \cdot \frac{dv}{dt}$	$\frac{dF}{dt} = c \cdot v$	$F = \rho \cdot v$	$F = F_0$

Total mechanical force is then:

$$F_{mech} = M \cdot \ddot{x} + \rho \cdot \dot{x} + c \cdot x + F_0 \tag{5.5}$$

The value of the magnetic force depends on electrical quantities. Electrical part of the device can be modeled as:

$$u = R \cdot i + \frac{d\psi}{dt} \tag{5.6}$$

where R represents ohmic resistance of the coil, and ψ is the magnetic flux. The flux is determined by the current through the coil and the size of the air gap between the movable armature and the magnet core.

$$\psi = f(i, x) \tag{5.7}$$

Magnetic force can be than calculated as:

$$F_{mag} = \frac{\partial E_{mag}}{\partial x} = \frac{\partial}{\partial x} \int_0^i \psi \cdot di = g(i, x) \tag{5.8}$$

where E_{mag} denotes electromagnetic energy in the air gap.

The equilibrium equation (1) can now be written as:

$$g(i, x) = M \cdot \ddot{x} + \rho \cdot \dot{x} + c \cdot x + F_0 \tag{5.9}$$

Equations (5.6) and (5.9) represent the model of the magnet that should be implemented in an HDL. Unfortunately, function that determines flux is, as is already mentioned, usually very difficult to obtain in a closed form. An artificial neural network (ANN) can help solving this problem. We used ANN shown in the Fig. 5.15 to approximate flux as a function of *i* and *x*. Moreover, neural network can be used to calculate directly *g(i,x)*, i.e. the magnetic force. Force is the function of the same inputs *i* and *x*, and the same network calculates both values. For that reason, network in Fig. 5.15 has two output neurons. Both values are necessary for implementation of the model in the simulator (equations (5.6) and (5.9)).

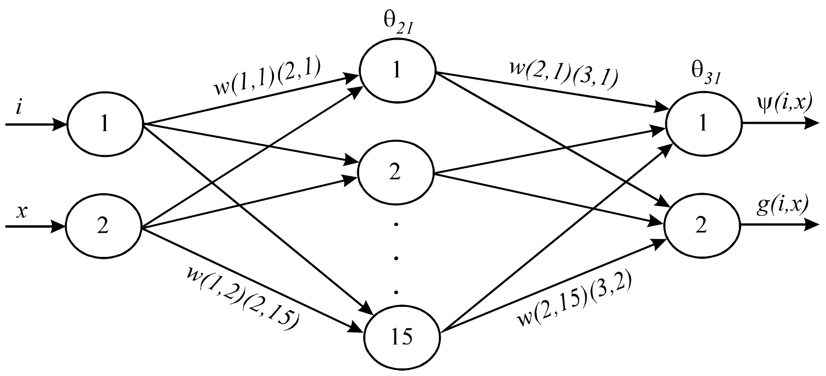
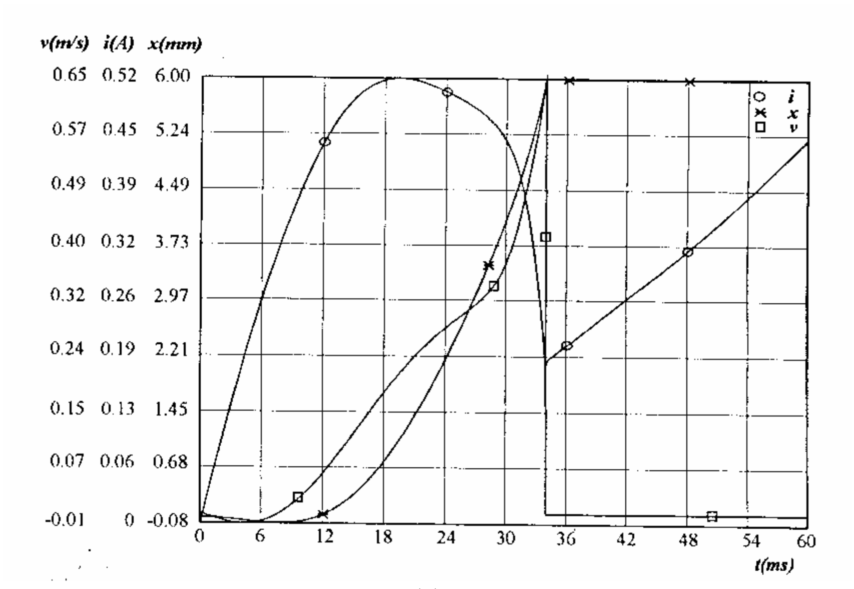


Fig. 5.15: ANN used in modeling magnetic characteristics

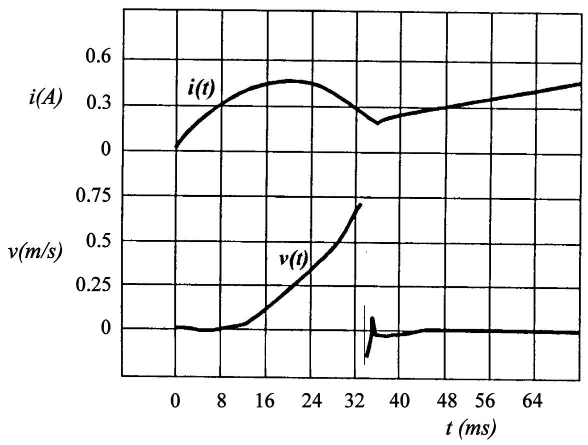
As mentioned in section 3.2, AleC++ have special operators for description of first and second order time derivatives. The latter one was developed particularly for description of mechanical systems, since second order differential equations are often in use there. For that reason, eqn. (5.9) can be described nearly as it is given in the text.

```
main_equation: process per_moment {
   eqn x:  -{Fmag}  +M*d2dt2{x}  +RO*ddt{x}  +c*{x} = -F0;
}
```

For the voltage excitation the step function is used. Its amplitude was 24V. All variables (x(t), v(t) and a(t)) were considered zero for t=0. In Fig. 5.16(a) the simulation results are given. As may be seen, the movable part of the core eventually reaches its goal with significant velocity. These results are to be compared with measured ones given in Fig. 5.16(b).



(a)



(b)

Fig. 5.16: (a) Simulation results. (b) Measured characteristics.

5.8. Pressure sensing system

AleC++ is a HDL created as a superset of the standard programming language. Its programming features are very helpful in defining complex models. In this example we have used them to model devices with distributed parameters.

In Fig. 5.17, a micro-electro-mechanical pressure sensing system is shown. A rectangular capacitor membrane is deformed when pressure is applied. Membrane deformation is modeled using a partial differential equation [Timo59]. That equation can be discretized and represented as a system of ordinary differential equations. Discretization is performed in the `for` loop in the `action` block of the module. That `for` loop must be in the structural process (types of process synchronization are given in the section 3.2), since the structure of the system of equations must be defined before the simulation, and the structural processes are performed before the actual simulation starts.

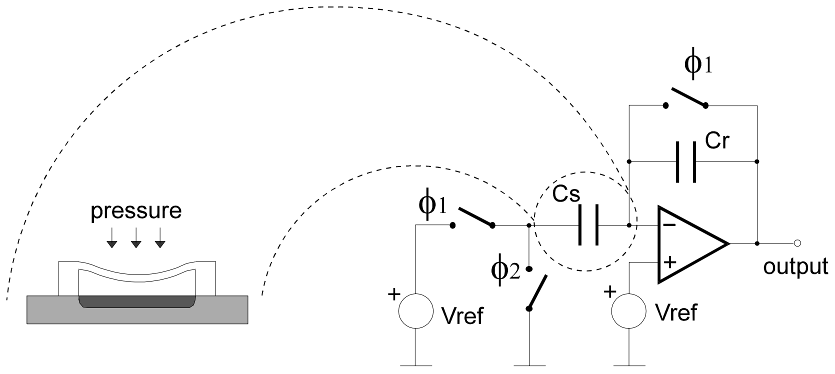


Fig. 5.17: Micro-electro-mechanical capacitive pressure sensing system.

The time-domain simulation results are given in the Fig. 5.18. The spatial displacement of the membrane for one time instant of time-domain simulation is given in Fig. 5.19. It is interesting to note that the graphical postprocessor that is used to create all time-domain simulation results is also a class described in AleC++ language and stored in the library. For spatial drawings, as given in Fig. 5.19, a new class is defined in AleC++, derived from the class for standard, time-domain waveforms. In this way, we did not have to write a new graphical postprocessor for this purpose. Functions from the base class are used, just some new had to be added to the derived class. Both the time-domain simulation results and the spatial drawings can be viewed during the simulation run.

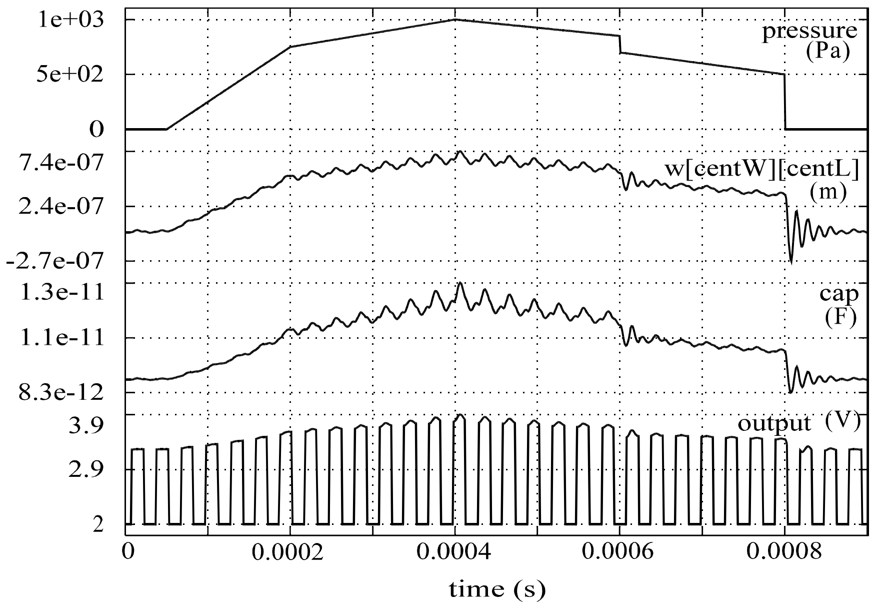


Fig. 5.18: Time-domain simulation results for the system given in Fig. 5.17. Traced signals are pressure, displacement of the pressure sensor center, sensor capacitance, and the output voltage.

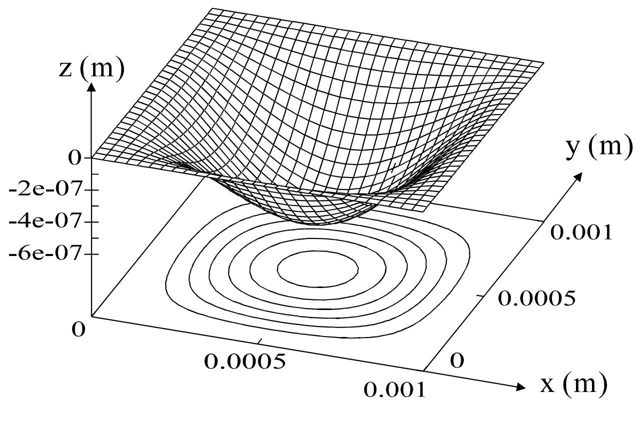


Fig. 5.19: Displacement of the sensor membrane for simulation time instant 0.0004s in Fig. 5.18.

5.9. Thermoelectrical simulation

In many microsystems and microelectronic circuits thermal effects have significant importance. Spatial distribution of the temperature on a chip should be simulated, together with electronic subsystem. The coupling is bidirectional, as electronics is influenced by the temperature, and electronic components represent heat sources.

Thermoelectrical simulation is performed in Alecsis using finite element (FE) method for spatial discretization. The user defines the geometry and placing of the components on the wafer by using commercial FE software with meshing capabilities. The strength of the method is in the fact that FE discretization can accurately describe the physical laws on irregular shapes with irregular mesh. The system of equations is extracted into the form applicable to analogue simulators. Afterwards, the electronic system or components described on higher level of abstraction are defined and coupled with thermal FE modules. Alecsis is employed to solve a system, assembled from particular element models previously programmed as libraries in AleC++.

The quality of the approach can be described on the flow sensor based on anemometry principle (system in Fig. 5.20, similar to [Jime98]). The heating resistor is employed for generating thermal gradient in the region of interest (wire, cantilever, bridge). The temperature difference between the referent and the heated position is proportional to the velocity of flow, since the flow disturbs the distribution of thermal gradients. It is a fluidic-thermal-electric system and requires simultaneous treatment of all participating physical domains.

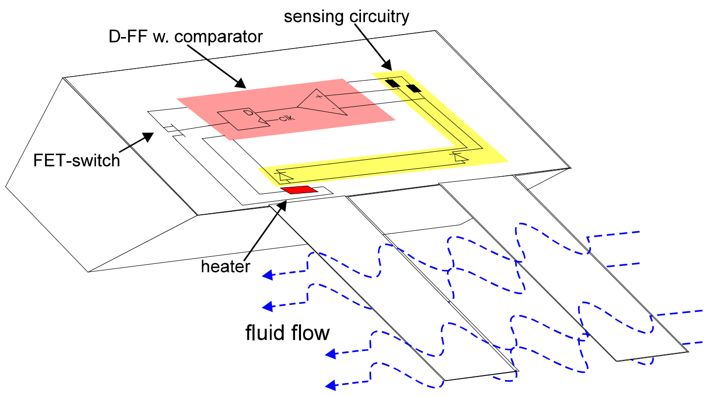


Fig. 5.20: The flow sensor with the circuitry.

The placement of the components is shown in Fig. 5.20. It describes the circuitry for defining constant average temperature difference between the heater and the reference. The fluid influences the thermal flow, and that flow is measured by the temperature-dependant diodes.

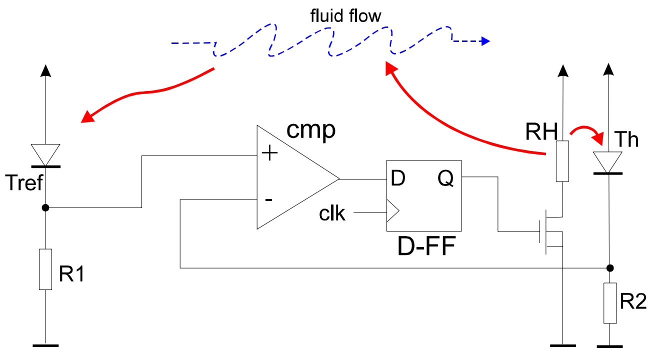


Fig. 5.21: Main thermal flows in the flow sensor.

This system defines a sigma-delta converter of the first order with the low-pass feedback incorporated in the thermal system (Fig 5.21). The FET transistor is switched depending on the temperature difference between the heater and the reference. The frequency of the switching is a measure of the flow velocity.

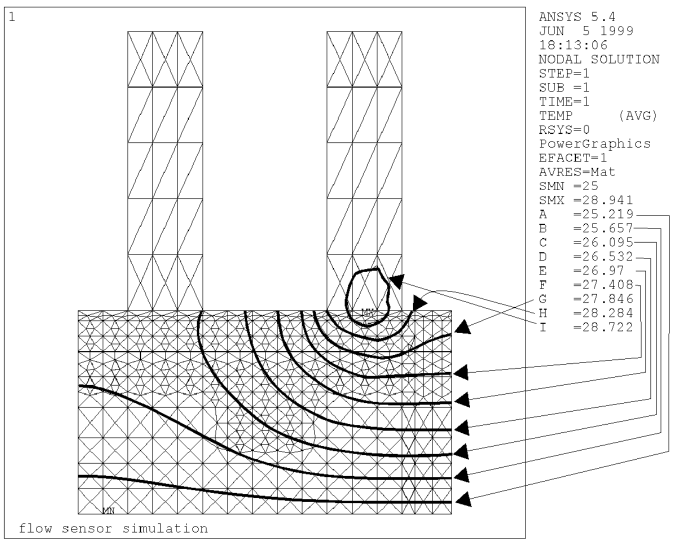

```
ANSYS 5.4
JUN  5 1999
18:13:06
NODAL SOLUTION
STEP=1
SUB =1
TIME=1
TEMP       (AVG)
RSYS=0
PowerGraphics
EFACET=1
AVRES=Mat
SMN =25
SMX =28.941
A   =25.219
B   =25.657
C   =26.095
D   =26.532
E   =26.97
F   =27.408
G   =27.846
H   =28.284
I   =28.722
```

Fig. 5.22: Temperature distribution on the chip.

The ANSYS v5.4 results for 2D thermal simulation with emphasized isotherms are shown in Fig. 5.22. Comparison with Alecsis results for the same thermal system shows relative error smaller than 0.002%. The 2D model was somewhat simplified. The bottom part of the structure in Fig. 5.22 has fixed temperature, which presumes the connection with an ideal sink. The system is modeled as adiabatic in z-plane, i.e. there is no heat transport in the direction of the z-axis.

Alecsis transient simulation results of the coupled electro-thermal system are shown in the Fig. 5.23. Traced signals are the heater temperature, the temperature of the cantilever close to the heater, the reference temperature, the measured fluid velocity, and the clock signal. Results show change of the frequency of the heat pulses with the change of the fluid flow velocity.

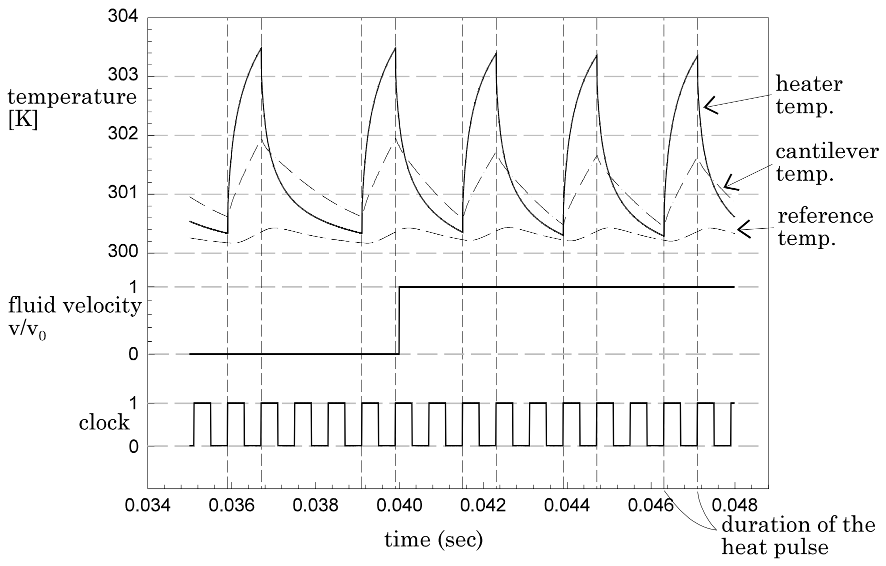


Fig. 5.23: Transient results for the behavior of the flow.

More details about electrothermal simulation in Alecsis can be found in [Jako99].

5.10. Digit-serial 8-bit systolic array multiplier

Like VHDL, AleC++ does not favor any logic value system. The logic value system and appropriate lookup tables, overloaded logic operators, basic logic gates, bus resolution functions, etc., can be described in AleC++ and conveniently stored in the library and retrieved when needed.

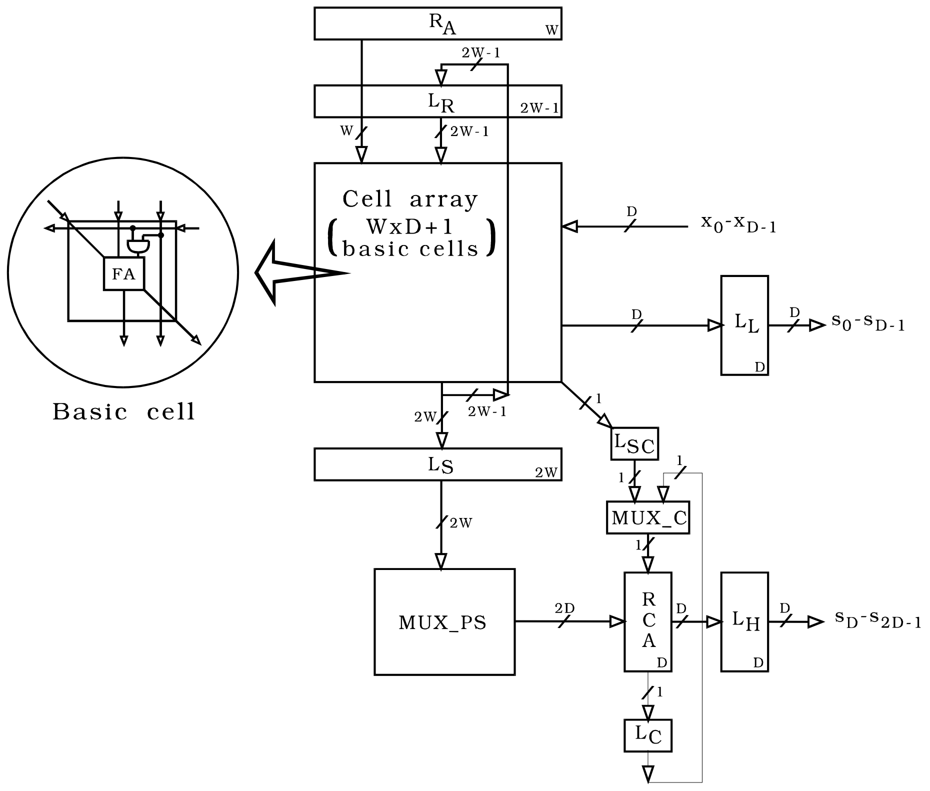


Fig. 5.24: Block diagram of digit-serial semi-systolic multiplier: R_A - register for the first operand of multiplication; L_R and L_S - latch arrays storing partial products; MUX_C - multiplexer 2 to 1; MUX_PS - multiplexer 2W to 2D; L_{SC} and L_C - latches; RCA - ripple carry adder; L_L and L_H - latch arrays storing lower and higher portions of the result; FA - full adder.

The digit-serial multiplier shown in the Fig. 5.24, the building element of more complex semi-systolic architectures [Mile95, Mile96], is simulated using a pre-compiled library developed to emulate the HILO logic simulator [Harr85]. The multiplier is composed of basic processing elements that communicate locally and work in synchronization. The first operand of the length W is available in parallel in register R_A, while the second (X) is fed in digit-by-digit, where the number of bits in a digit is marked as D. The multiplier operates in two phases: in the first phase the lower portion of the product is generated at the outputs s_0-s_{D-1}, while the other half is available in the second phase at the outputs s_D-s_{2D-1}.

Let us illustrate the multiplier modeling by showing the simplified model of full adder, which is part of the systolic array basic element, as shown in Fig. 5.20. Firstly, the model class `fadd` is defined.

```
// model parameters defined with min, typical and max value
typedef double param[3];
// different delays from inputs to outputs, direction flags
#      define FROM_AB           0
#      define FROM_CIN          1
#      define TO_SUM            0
#      define TO_COUT           1
#      define TAKE_MAX_DELAY   -1
// full adders model class
class fadd {
       param delay01[2][2];// rising edge propagation delay
       param delay10[2][2];// falling edge propagation delay
       fadd();
       >fadd();
  public:
       // delay function
       double ADDdelay (three_t, three_t, int, int);
       friend module fa;
};
```

Full adder module is described with the following code.

```
module fadd::fa (fift_t in a, b, c_in;
                 fift_t out sum, c_out) {
  action {
    process (a, b, c_in) {
      three_t a3, b3, c_in3, sum_result, cout_result;
      int from_in;
                    // detect active input


      if((a->event || b->event) && c_in->stable)
```

```
            from_in = FROM_AB;

          else if( c_in->event && a->stable && b->stable)
            from_in = FROM_CIN;
          else {        // simultaneous event at (a or b)/c_in
            warning(
               "fadd::fa - simultaneous (a or b)/c_in change");
            from_in = TAKE_MAX_DELAY;
          }
                        // convert input states to three_t
          // log. operations are not held in HILO's 15-st. log.
          // system (fift_t), but in 3-state system (three_t).
          a3 = Con15to3[a];
          b3 = Con15to3[b];
          c_in3 = Con15to3[c_in];
                        // evaluate logic function
          sum_result = a3 ^ b3 ^ c_in3;
          cout_result = (a3 & c_in3) | (a3 & b3) | (b3 & c_in3);
          sum <- Con3to15[sum_result] after
                 this->ADDdelay(sum_result, Con15to3[sum],
                                           from_in, TO_SUM);
          c_out <- Con3to15[cout_result)] after
                 this->ADDdelay(cout_result, Con15to3[c_out],
                                           from_in, TO_COUT);
        } // process (a, b, c_in)
      } // action
    } // module fadd::fa ()
```

The above code is part of mentioned Alecsis library for HILO emulation. In order to use the module `fa` in multiplier description for simulation, user has to connect the library file and to define its model card. One particular model card for a full adder module is as follows.

```
    model add15::fa_1 { // {min value, typ value, max value}
      delay01[FROM_AB][TO_COUT] = {0.2ns, 0.5ns, 1.0ns};
      delay10[FROM_AB][TO_COUT] = {0.3ns, 0.6ns, 1.5ns};
      delay01[FROM_CIN][TO_COUT]= {0.3ns, 0.5ns, 1.3ns};
      delay10[FROM_CIN][TO_COUT]= {0.3ns, 0.6ns, 1.5ns};
      delay01[FROM_AB][TO_SUM]  = {1.3ns, 2.4ns, 5.6ns};
      delay10[FROM_AB][TO_SUM]  = {0.9ns, 2.2ns, 6.0ns};
      delay01[FROM_CIN][TO_SUM] = {0.5ns, 0.9ns, 2.0ns};
      delay10[FROM_CIN][TO_SUM] = {0.4ns, 0.9ns, 2.7ns};
    }
```

The multiplier is simulated at gate level, using generic structures that enabled the word length to be passed as an action parameter. An example of the simulation results for the configuration with W=8 and D=4 is shown in the Fig. 5.25. The entire circuit contains more than 500 gates modeled with more than 1000 concurrent processes, while around 25000 events were handled during the simulation. It took 4.76 CPU seconds to simulate the circuit using Alecsis implementation on Hewlett-Packard Series 9000/375 platform with Motorola MC68020 processor. The simulation results obtained using HILO logic simulator were identical. Unfortunately, HILO was implemented on a different platform and simulation times were not comparable.

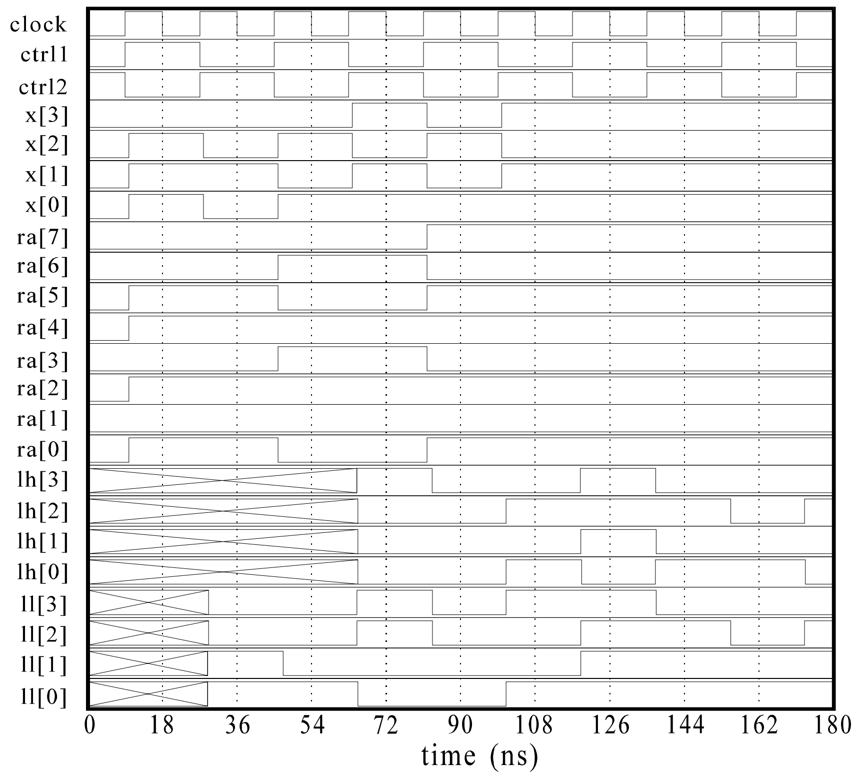


Fig. 5.25: Example of simulation results for 8-bit configuration of the multiplier from Fig. 5.24: clock, ctrl1 and ctrl2 are global control signals; x[0] - x[3] are inputs for second operand; ra[0] - ra[7] are R_A outputs (first operand); lh[0] - lh[3] are the outputs of L_H; ll[0] - ll[3] are the outputs of L_L.

5.11. LAN Ethernet network simulation

While the previous example shows Alecsis use in gate-level logic simulation, this one will illustrate its capabilities at higher levels of abstraction. Using signals typed as classes, it is possible to create complex inter-process communication. Fig. 5.26 shows Local Area Network (LAN) modeled using AleC++ discrete-event simulation constructs. Each workstation is represented as an autonomous concurrent process with its own activity pattern. LAN cable segments between stations are modeled as simple bi-directional buffers that pass information with some transport delay proportional to the cable length. After non-deterministic idle periods calculated using the uniform random distribution, the stations attempt to send their frames, according to CSMA/CD protocols [Watk93, Schw87]. After sending the frame, the station monitors the line for the specified period (*2*slot_size*), and then compare the original and the frame on the line. If they are not the same, the collision took place, and the frames are scheduled for retransmission after a period calculated using *binary exponential backoff* algorithm. According to CD (Collision Detection) strategy, the station is capable of truncating the corrupted frame without waiting *2*slot_size* period in case of collision.

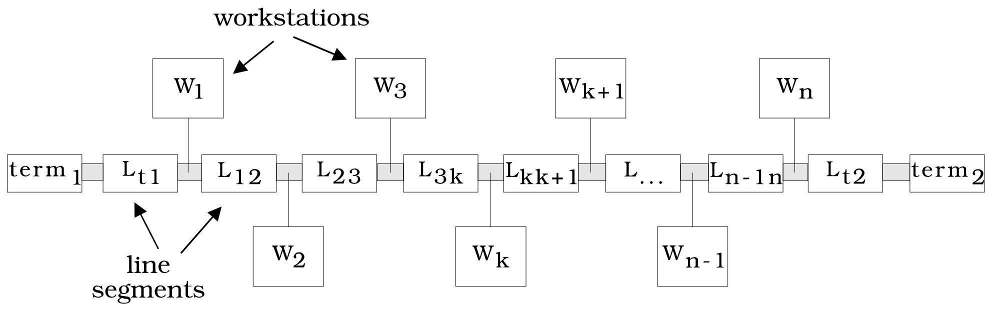


Fig. 5.26: Schematic of an abstract LAN Ethernet network

The workstation model in AleC++ is as follows.

```
enum StationState { NormalState, CollisionState };
```

```
module LAN::workstation(frame inout lan_port) {
    action (int id, const char *ws_name) {

        process {
            frame last_frame;
            double last_sent = 0.0;
            StationState ss = NormalState;
            int ntrials = 0;
            if (ss == NormalState) {
              wait for idle_period();
              last_sent = now;
              if ( !lan_port.status (FreeLine) )
                wait lan_port while
                                !lan_port.status(FreeLine);
            }
            last_frame.stamp(NormalFrame, id,
                           receiver(id, nst), now, ws_name);
            lan_port <- last_frame;
            last_sent = now;
            wait lan_port for listen_period();
            if ( last_frame != lan_port ) {
              if (ss == NormalState) {
                ntrials = 0;
                ss = CollisionState;
              }
              ntrials++;
              if(ntrials > max_trials() ) {
                ss = NormalState; // discard frame
                ntrials = 0;
              } else {
                lan_port <- frame(NoiseBurst, now);
                wait for burst_period();
                lan_port <- frame(FreeLine, now);
                wait for retrans_period(ntrials);
              }
            }
            else {  // successful transmission
              ss = NormalState;
              lan_port <- frame(NoiseBurst, now);
              wait for burst_period();
              lan_port <- frame(FreeLine, now);
              last_frame.stamp(FreeLine,0,0,now,ws_name);
            }
        }
    }
}
```

The complete modeling procedure for this problem is explained in [Maks95].

As in all previous examples, workstation modules are parametrized using model card class. That gives an opportunity to create model cards of real workstations, with different idle times and activity patterns. Fig. 5.27 shows the simulation results of an abstract LAN Ethernet network with 50 workstations interconnected with 5m cable segments. The stations were idle from 0 to 10ms, while the simulation covered 0.2s time period. The curve represents number of stations waiting to transmit their frames due to the occupied line. All the stations were capable of gathering the statistical data about network performance. Post processing of those data carried out in a process synchronized using implicit signal `final` showed that 712 frames were sent successfully, while 1319 frames were re-transmitted due to a collision. The total transmission delay was 2.43s, average delay per station 47.6ms, and average delay per frame 3.47ms.

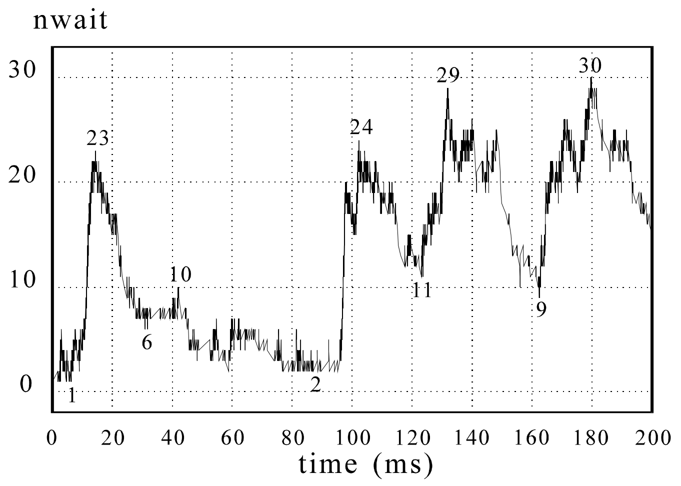


Fig. 5.27: Simulation results of the LAN Ethernet network with 50 workstations. The curve represents number of stations waiting to transmit their frames due to occupied line.

As the next example, the Ethernet bridge modeling and simulation will be considered.

The Ethernet bridge, as shown in Fig. 5.28, interconnects network segments. The network layer as well as all other (transport, session, presentation and application) layers are not affected by the bridge, so that the bridge is totally transparent to the user. Two main functions of the bridge are: filtering traffic (bridges provide for a filtering component to improve network performance) and bridging media. (Bridges frequently interconnect different networks. These networks are often Ethernet or token ring. The bridges merely duplicate all signals originating from all source networks and send them to other networks.)

In the example considered here, the bridge has two LAN-boards because it interconnects two segments. The received frame on one board (`port1`) has to be transferred by the bridge to another segment via appropriate LAN-board (`port2`). In operation mode, the bridge may be in one of the following five states:

1. **NormalState -** in this state the bridge waits to receive frames

2. **TransmisionState1 -** in this state bridge attempts to send a frame to port 1

3. **CollisionState1 -** in this state, after collision, the bridge resends the frame to port 1

4. **TransmisionState2 -** in this state the bridge attempts to send a frame to port 2

5. **CollisionState2 -** in this state, after collision, the bridge resends the frame to port 2.

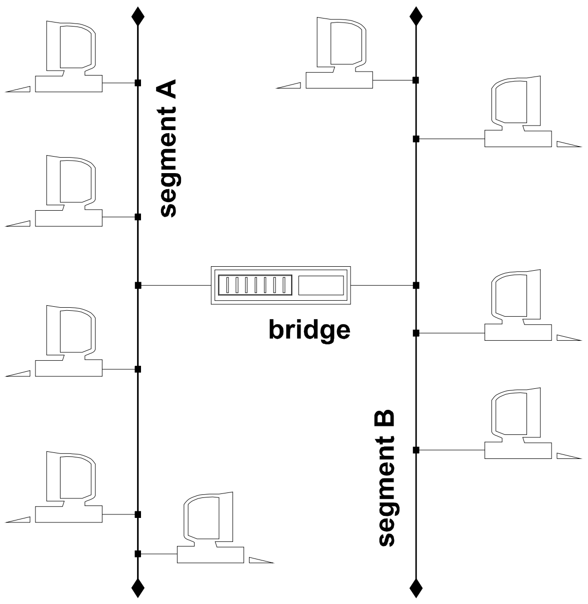


Fig. 5.28: Ethernet LAN with a bridge

The state-transition diagram used for modeling is presented in Fig. 5.29 in a form of an oriented graph. For simulation purposes, in addition to the network topology and functionality, the time properties for every network element and delay times for every state transition should be given. The next AleC++ code describes a bridge in a computer network.

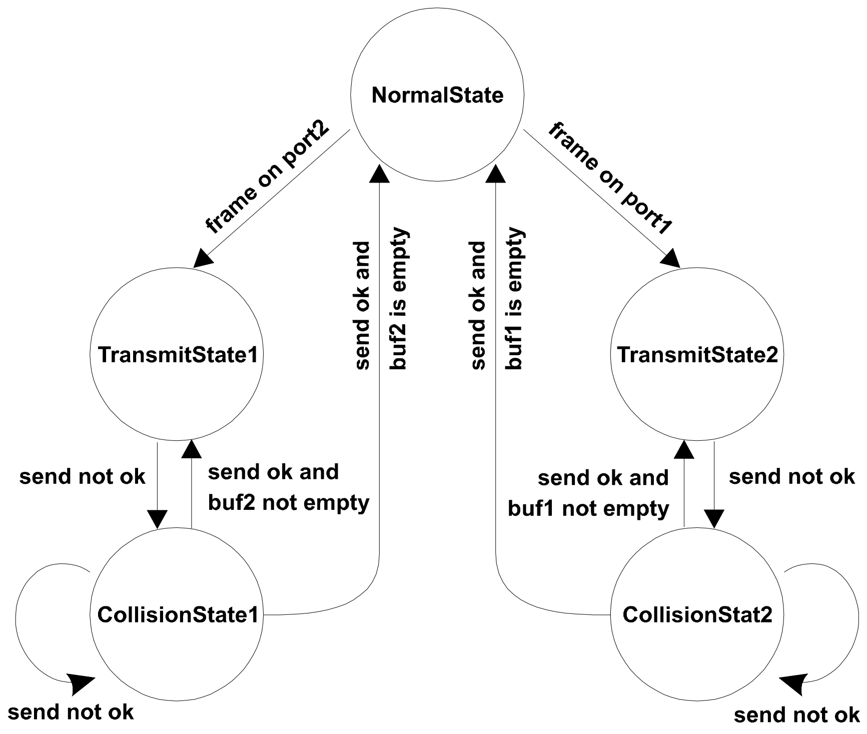


Fig. 5.29: State-transition diagram used for bridge modeling

```
//libr. lanb contains modules: cable,workstation,bridge,...
library lanb;
// model cards for stations and bridge
model LAN::sgi_indigo {
  frame_size=256; max_retrans=16; idle_time=IdleTime;
}
model LAN::hp9000 {
  frame_size=256; max_retrans=16; idle_time=IdleTime;
}
model LAN::bridge2 { frame_size=256; max_retrans=16; }
...


root module net_with_bridge ( ) {


  // declarations of workstations, cables and bridge
  module workstation w1, w2, ... ;
  cable c_1, c_2,  ... ;
```

```
         module bridge b1;
         ...
         // signal declarations
         signal cable_t w1_port={FreeLine}, w2_port={FreeLine};
         signal cable_t b1_port1={FreeLine}, b1_port2={FreeLine};
         signal cable_t term_1={FreeLine}, term_2={FreeLine};
         ...
         // instantiation
         w1(w1_port)
           {ws_name="indigo"; id=5; nst=numst; model=sgi_indigo;}
         w2(w2_port)
           {ws_name="hp";     id=6; nst=numst; model=hp9000;    }
         b1 (b1_port1, b1_port2)
           { bg_name="bg301"; id=20; model=bridge2;}
         c_5 (w1_port, w2_port)  cable_delay = Ldelay(4.0_meter);
         c_2 (w1_port, b1_port1) cable_delay = Ldelay(5.0_meter);
         c_6 (w2_port, term_2)   cable_delay = Ldelay(0.1_meter);
         ...
         timing { tstop = 13*IdleTime; }
         ...
       }
```

As simulation result, statistics are obtained related to all the stations, the bridge (see Table 5.3), and overall network. Table 5.3 contains two kinds of data: number of frames is given in the first five rows, while the last two rows represent delays caused by the data transfer through the bridge.

Table 5.3: Bridge activity statistics

	Bridge	Port 1	Port 2
received frames total	60	43	17
for transmission	25	18	7
successful	23	16	7
unsuccessful	2	2	0
repeated	14	12	2
total delay [s]	0.0062464		
mean delay [s/frame]	0.0001041		

5.12. Successive-approximation 8-bit serial A/D converter

This example represents the ability of Alecsis to support the top-down hierarchical refinement of complex hybrid systems. Fig. 5.30 shows the block diagram of a successive-approximation serial A/D converter. The converter can serve as a mixed signal simulation benchmark for the following reasons:

- the system contains analog (comparator), digital (control and weighting logic, shift register), and mixed-signal subsystems (sample and hold circuit, D/A converter macro model);
- there is a strong feedback loop with logic subsystems present;
- the system is too complex to be simulated at transistor level;
- pure behavioral simulation will not give important details such as possible oscillations or glitches.

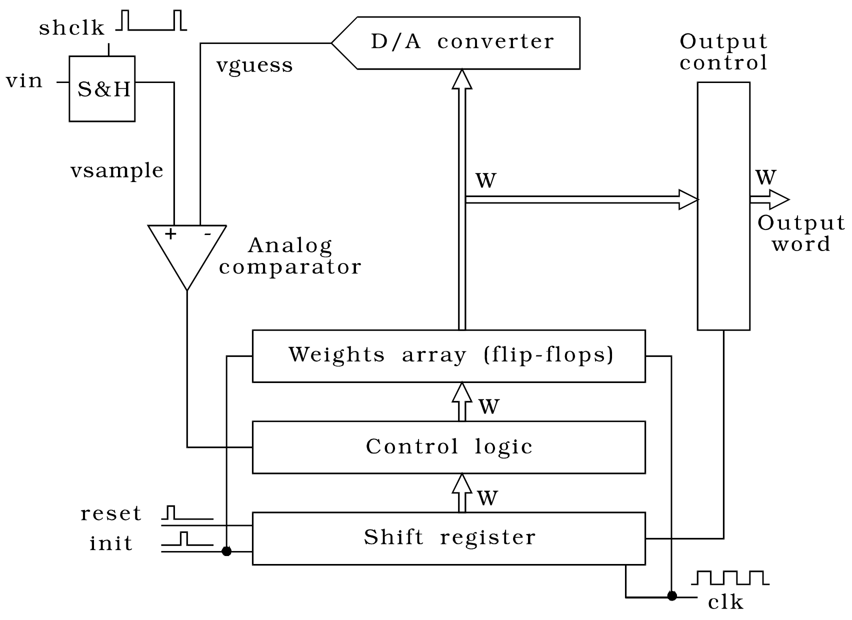


Fig. 5.30: Block diagram of an serial approximative A/D converter

The description of the serial A/D converter started with subsystem interface definition. All of the subsystems modules were parametrized in order to handle different output digital word lengths. In the first refinement, all the subsystems (analog, digital and hybrid) were implemented as behavioral to obtain crude results. All of the logic modules were modeled at the Register-Transfer Level, i.e. delay times were identical for all the bits in the register data paths. Transfer characteristic of the analog comparator was modeled using general nonlinear generator and *hyperbolic tangens* function, while D/A converter and sample and hold circuit employed dual-process strategy similar to the one used for standard D/A domain-coupling modules implicitly inserted by the simulator. The simulation results of the 8-bit configuration with the slow ramp input are shown in Fig. 5.31.

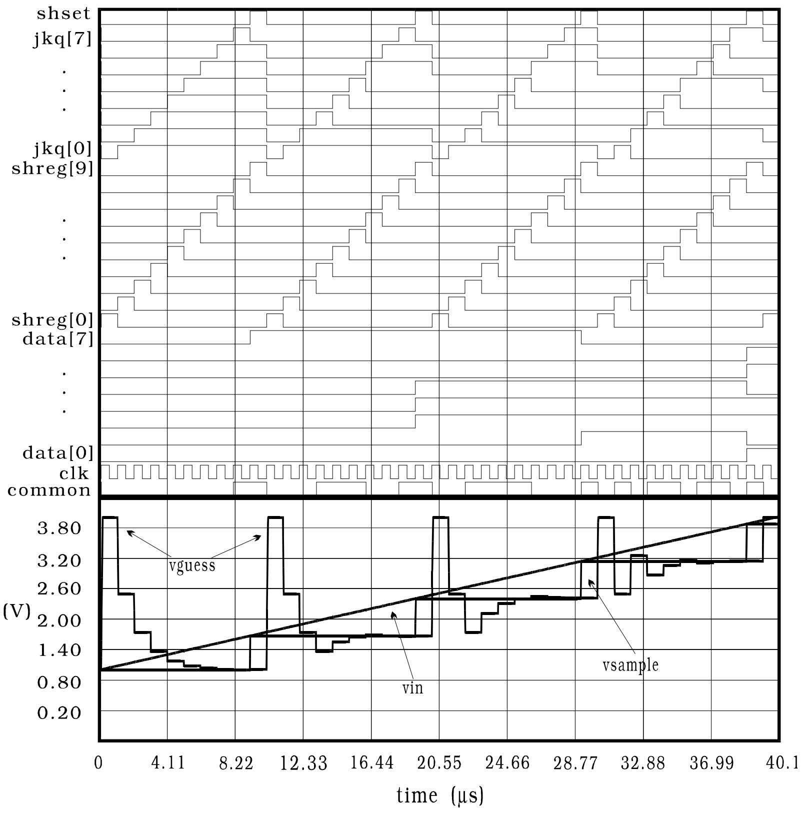
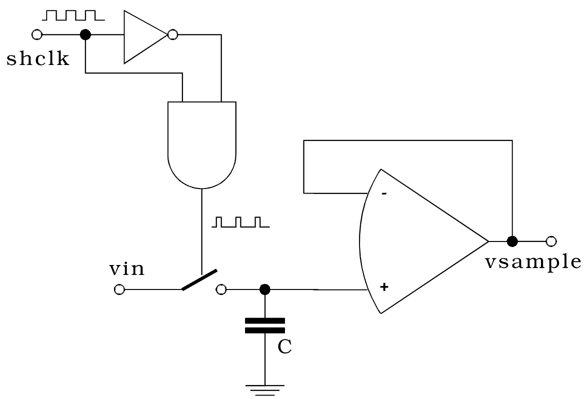
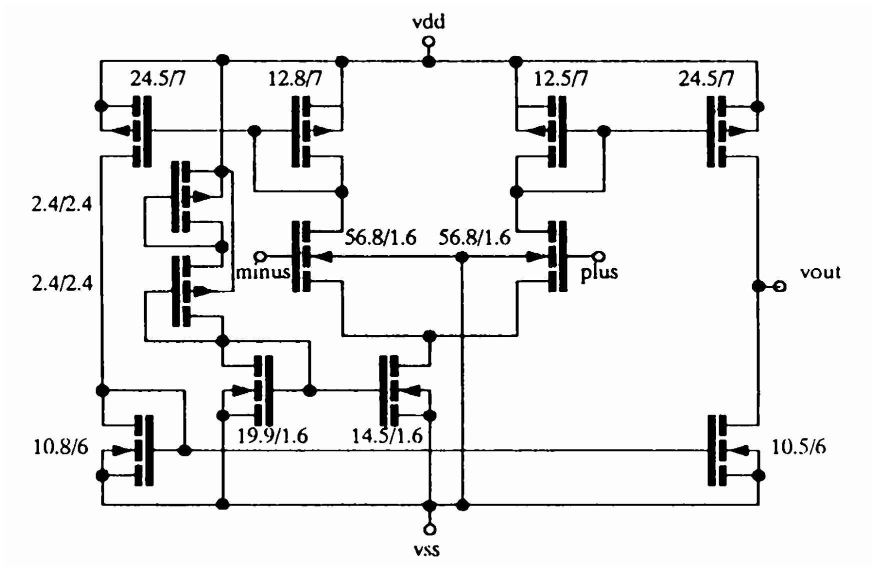


Fig. 5.31: Results of the A/D converter simulation using behavioral representations of the sub-blocks. Digital output variables are at separate plots while analog output variables are all at one plot. Signals data [0] - data[7] represent the output word.



(a)



(b)

Fig. 5.32: (a) Structural version of sample and hold sub-block used in the second A/D converter partitioning. The circuit consists of logic gates, internally controlled ideal switch, capacitor and operational amplifier described at transistor level.

(b) Operational amplifier structure. Transistor channel dimensions are given in μm.

The first simulation was very fast due to the high-level behavioral models. In order to obtain a better accuracy, some of the subsystems were further partitioned into smaller blocks. Digital blocks were modeled at the gate-level, using JK and D-type flip-flops and standard logic gates. The sample and hold circuit was replaced with the circuit shown in the Fig. 5.32(a), with ideal switch, capacitor and a CMOS operational amplifier described at transistor level as shown in Fig. 5.32(b). The same amplifier was used as a comparator. This partitioning helped to detect certain instabilities caused by the small phase margin of the operational amplifier. Another problem occurred due to different delay times of the individual JK flip-flops that form the digital output, resulting in noticeable glitches at the D/A converter output. Both effects are shown in the Fig. 5.33. The second simulation was much slower (455 sec with transistor-level CMOS opamps and logic gates as opposed to 4.7 sec in the case of the behavioral model) primarily because of the analog portion of the system.

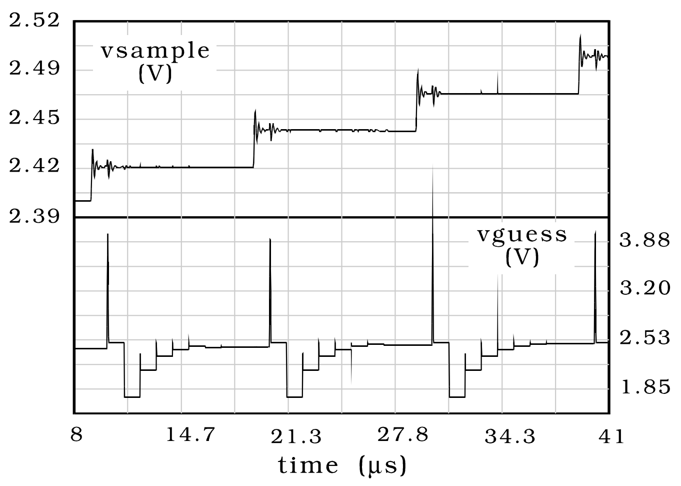


Fig. 5.33: Simulation results of the A/D converter with digital circuits at logic level, and analog at transistor level. Attenuated oscillations at the S&H output and glitches at D/A converter output are now clearly visible.

5.13. Second order sigma-delta modulator

The oversampling sigma-delta conversion technique offers an alternative method of producing high-accuracy, high-resolution ADC’s without the need for precise component matching and complex analog circuitry.

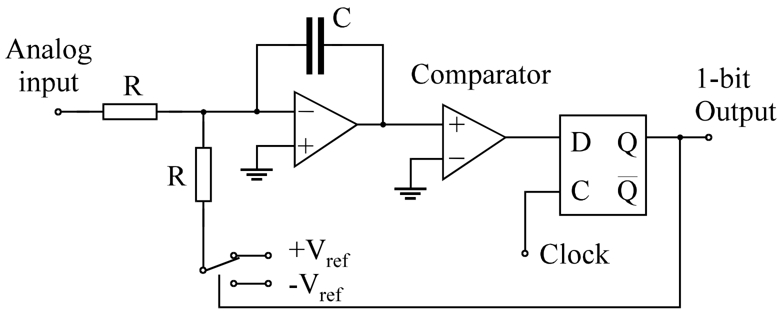


Fig. 5.34: First order sigma-delta modulator.

A first order sigma-delta modulator is shown in Fig. 5.34. The gain of the circuit is given by

$$G = \frac{1}{RC\, f_{clock}}\,, \tag{5.10}$$

which means that it is dependent on the sampling rate. If the primary gain is designed for the lowest sampling rate, the gain will decrease with increasing sampling rate, reducing the dynamic range of the modulator.

A way to keep the gain constant is to make the integrator charging time invariable with respect to clock rate. This means that the analog switch must be turned on for fixed time duration regardless of clock rate. One solution for achieving this is to use monostable multivibrator as a fixed-width pulse generator in the circuit. A second order sigma-delta modulator with variable sampling rate is shown in Fig. 5.35.

The monostable multivibrator between the clock input and switch control block functions as a pulse generator to produce control signals of fixed time duration. The pulse width is chosen such that the circuit operates at the maximum clock rate of 1.024 MHz. The reference voltages of this circuit are ±1.5 V.

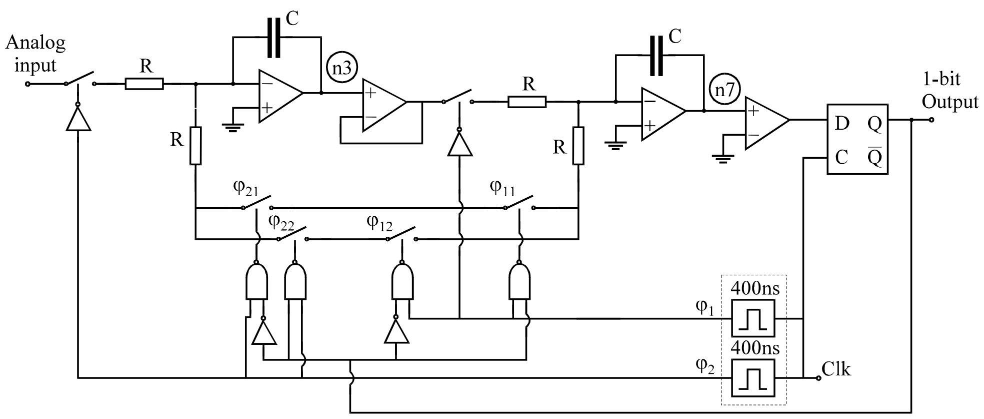


Fig. 5.35: Second order sigma-delta modulator.

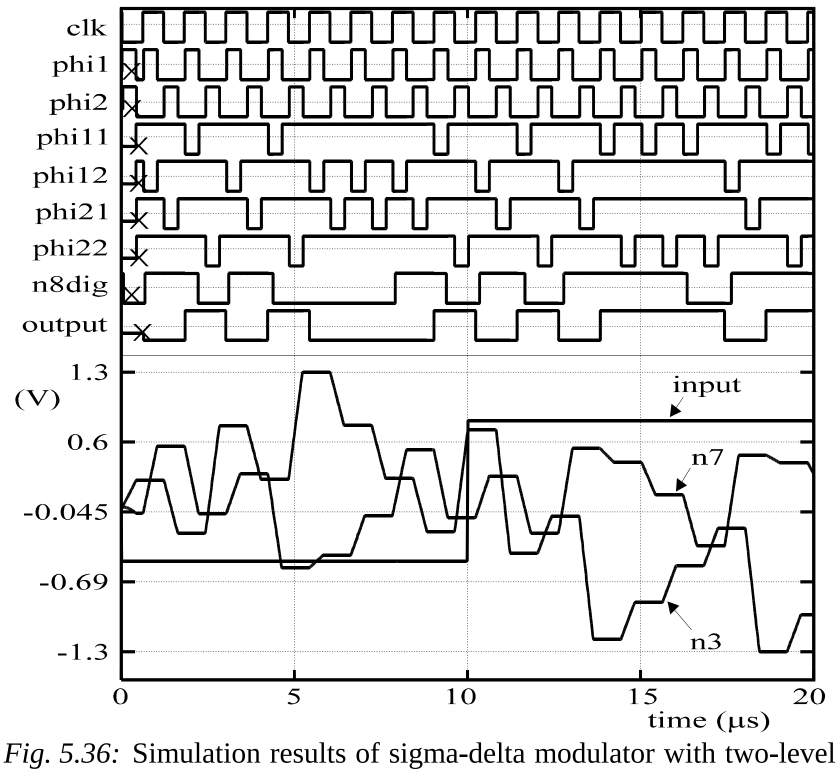


Fig. 5.36: Simulation results of sigma-delta modulator with two-level input excitation.

Results of the simulation, when two levels of a constant analog signal are brought to input, are given in Fig. 5.36. All digital signals in the circuit, and three analog signals (input, and two voltages at the output of the integrators) are plotted out.

Fig. 5.37 shows reaction of the system when the input is excited by a linear ramp. The simulation time is longer, and the changes of the output can be noticed.

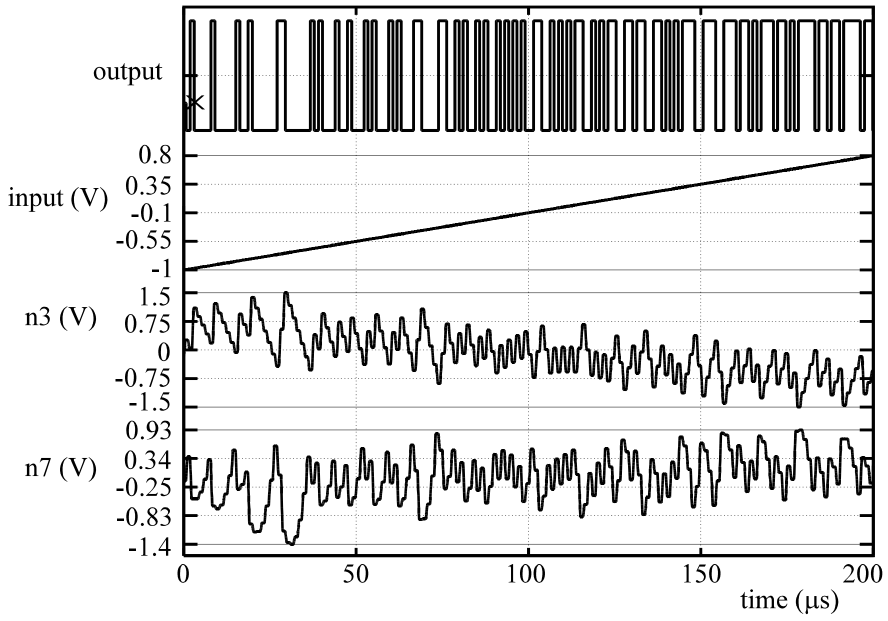


Fig 5.37: Simulation results for linear ramp excitation

5.14. Capacitive pressure sensor followed by A/D conversion

Microsystem development, miniaturization and integration demand more than purely electronic circuit simulator, since mechanical and electrical subsystems are to be fabricated together. Simulation of the complete system is often necessary and cannot be performed with a simulator solely dedicated to electronics or mechanics.

The system considered here is shown in Fig. 5.38.

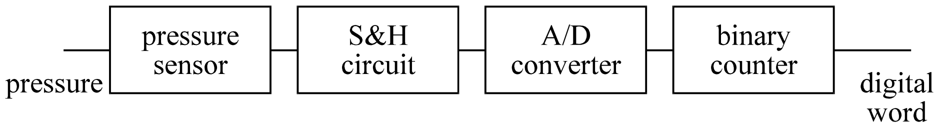


Fig. 5.38: Sensing system

The pressure sensor is a capacitor, where the capacitance is dependent on a plate displacement, and this displacement is a function of pressure. In this example, the plate has circular shape. We presume quasi-static conditions and constant pressure all over the plate. A switched-capacitor pressure sensing system is given in Fig. 5.17. Cs is the sensor capacitance and Cr a reference capacitance.

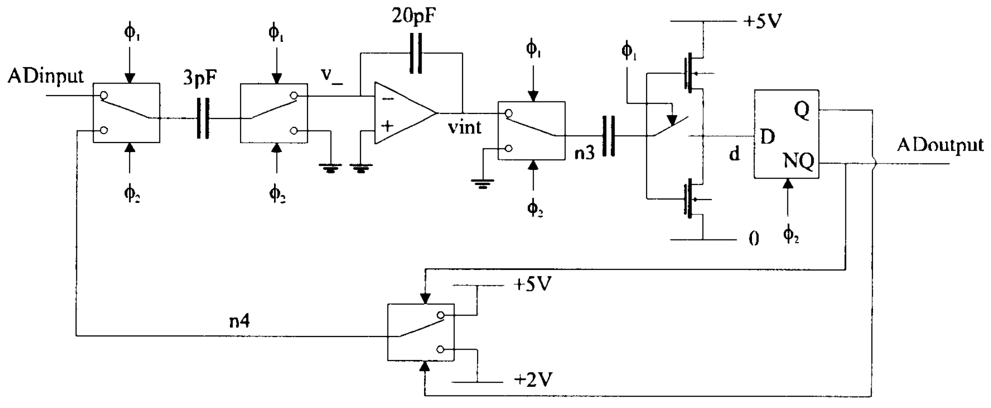


Fig 5.39: Sigma-delta modulator

In real systems there is a need to have digital information about the voltage which refers to input pressure. Since we have discussed quasi-static conditions, the so-called sigma-delta modulator would be a convenient way to perform A/D conversion. Fig. 5.39 shows detailed scheme of a sigma-delta modulator.

When uniform signal is brought to the input, the converter gives the number of pulses proportional to the level of the input signal. The resolution of converted signal is proportional to a time spent in conversion. Logic state of the output signal makes decision whether the signal coming to integrator would be increased or decreased.

The sample and hold circuit as well as the binary counter are described behaviorally. Their structure is not of interest, but their existence is required. All the parts of the system work with a clock signal of 10μs period, and sample and hold circuit has a clock pulse computed from:

$$\text{SHclk} = \text{resolution} \cdot 10\mu\text{s}$$

When resolution parameter is increased, the amount of time needed for gaining one value of pressure is also greater. Since we have discussed quasi-static conditions, this fact is not crucial.

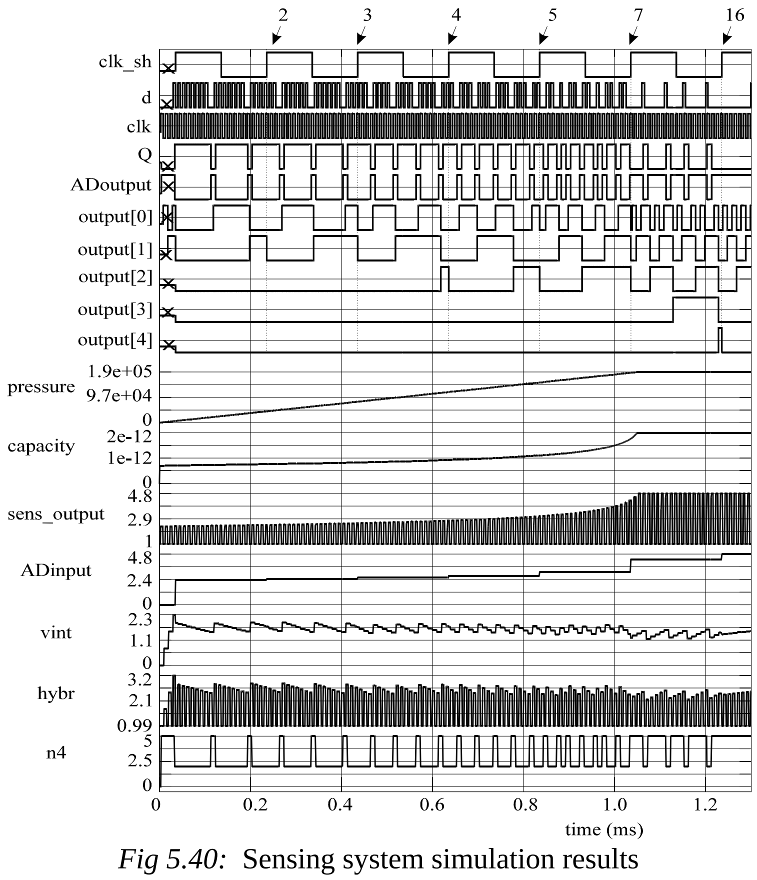


Fig 5.40: Sensing system simulation results

The results of the simulation, when the pressure is monotone and rising, are given in Fig 5.40. The following variables are shown: input pressure signal; capacitance of the nonlinear capacitor; pressure sensor output signal; AD converter input held value; internal converter waveforms - voltage at the integrator output (`vint`), voltage at the input of flip-flop (`hybr`), returned value (`n4`). Digital signals are: `clk_sh` - sample/hold circuit clock signal; `clk` - system clock; `d` - digital value of voltage at node `hybr` obtained at the output of automatically inserted A/D converter interface circuit; `ADoutput` - converter output; `output [0]` to `output[4]` - counter output signals. Counted values are marked below.

Fig. 5.41 shows the system response to sine input pressure, with resolution=100 (`ADinput` is held value and `counted` is counter output). It is easy to observe mapping of sensor nonlinearity to the measured value.

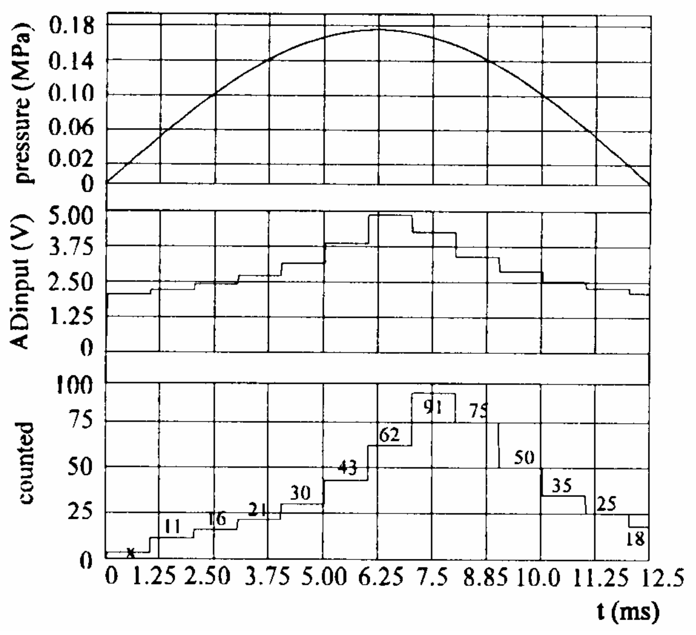


Fig. 5.41: The sensing system response to the sine input signal

5.15. Alecsis-VHDL co-simulation

In this chapter we shall demonstrate the possibility of Alecsis-VHDL co-simulation. To realize this idea we are developing a separate program – VHDL-AMS compiler. The most of VHDL constructs are already implemented, as will be illustrated on a simple example.

There are many digital model libraries developed in VHDL, and there are many experienced VHDL users, too. Therefore, our intention is to enable Alecsis to simulate digital systems described in VHDL. It also assumes simulation of mixed-signal systems keeping the advantage of using VHDL standard libraries for description of digital portion of the system.

The simplest way to achieve Alecsis-VHDL co-simulation was to use the existing simulation kernel of Alecsis simulator (virtual processor), and to develop a new compiler suited for VHDL language. Our VHDL-AMS compiler converts VHDL or VHDL-AMS source code into object code for Alecsis virtual processor. Alecsis object code is designed for mixed-mode simulation. Therefore it supports almost all VHDL and VHDL-AMS modeling mechanisms.

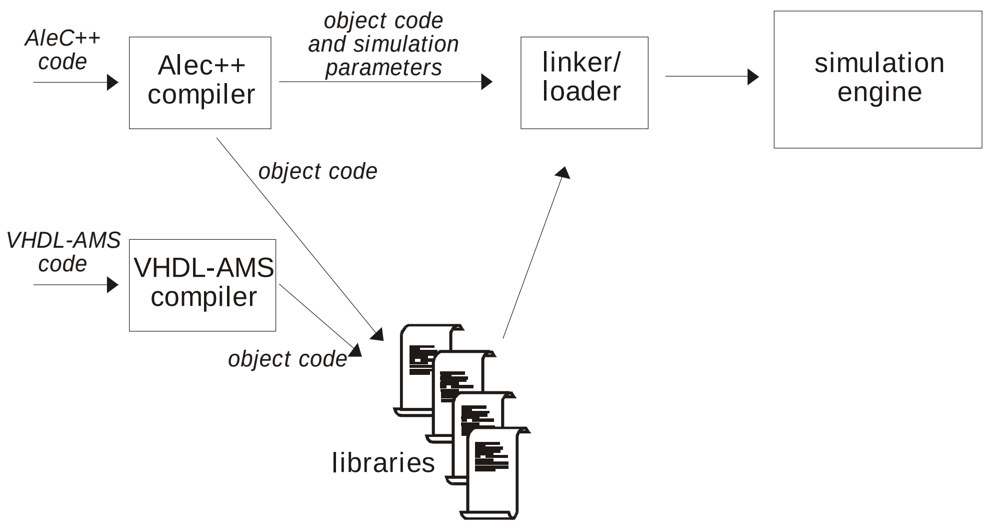


Fig. 5.42: Concept of Alecsis-VHDL-AMS co-simulation

Fig. 5.42 shows the co-simulation concept. VHDL source files are compiled with VHDL compiler and objects are stored into simulation libraries. AleC++ source files are also compiled and objects are stored into libraries of the same format. There is only one important difference. Simulation control parameters cannot be obtained from VHDL code. In other words, the root module must be described in AleC++. The simulation engine cannot make difference between library objects obtained from VHDL and AleC++ compiler.

The basic structural unit in VHDL - entity is converted into AleC++ structural unit - module. Each architecture of an entity becomes a new module with the name equal to the name of the architecture. Generic parameters correspond to module action parameters. The data types in VHDL and AleC++ have different names, but the correspondence can easily be established due to the same machine representation. Type *real* from VHDL corresponds to type *double* in AleC++, *integer* to *int*, *records* are related to *structures*, etc. Structural hierarchy is fully supported both in AleC++ and VHDL. It is possible in VHDL description to instantiate components and to call functions described in AleC++ and vice versa. References to those components and functions will be resolved by the linker/loader before the start of the simulation.

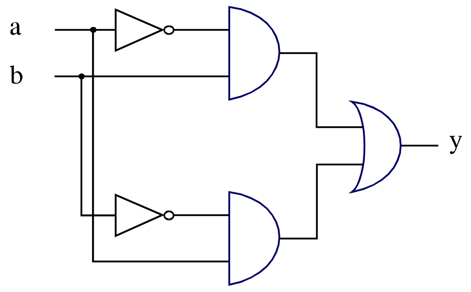


Fig. 5.43: An exclusive-OR circuit

For the illustration of Alecsis-VHDL co-simulation, we shall model a simple circuit shown in Fig. 5.43 with combination of AleC++ and VHDL code. The modeling technique will not be the optimal one, but is intended to be a good illustration of co-simulation possibilities. Fig. 5.44 explains model hierarchy. Inverter (module `inv`) is modeled in AleC++. AND circuit as well as OR circuit is modeled in VHDL (architectures `and2` and `or2`). Inverter and AND circuit are then combined into the AleC++ module named `inv_and`. The whole circuit shown in Fig. 5.43 is described in VHDL and named `xor2`. The delay `delay_f` function used both in AleC++ and VHDL is defined in VHDL.

The simulation is performed in 2-state logic system (signals are of `bit` type). The AleC++ description of module `inv` follows:

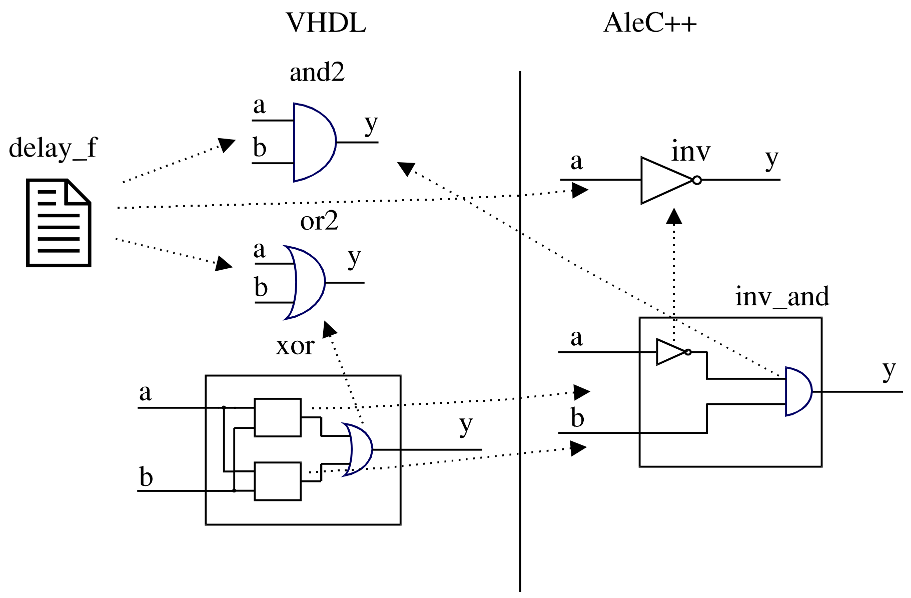


Fig. 5.44: Model hierarchy for circuit from Fig. 5.43

```
double delay_f (bit, double, double); // func. declaration

typedef enum { '0', '1' } bit;  // state system definition

bit const not_tab[] = { '1','0' };

bit operator ~ (bit op) { return not_tab[op]; }

module inv (bit out y; bit in a) {
  action (double tr, double tf){
    process (a) {
      y <- ~a after delay_f (~a, tr, tf) ;
    }
  }
}
```

The function `delay_f` is defined in VHDL. AleC++ only requires its declaration to be visible before the function call, because of type-checking mechanism. Function code is rather trivial.

```
--- Standard gate delay function:
function delay_f (state: bit; tr: real; tf: real)
                                     return real is
begin
  if state='0' then
    return tf;  --fall time
  else
    return tr;  --rise time
  end if;
end delay_f;
```

OR and AND circuits are modeled with the next VHDL code.

```
--- Two-input AND gate:
entity and2_e is
  generic (tr: real := 1.0e-9; tf: real := 1.0e-9);
  port (y: out bit; a,b: in bit);
end and2;

architecture and2 of and2_e is
begin
  process(a,b)
  begin
    y <= a and b after delay_f(a and b, tr, tf);
  end process;
end inv;

--- Two-input OR gate:
entity or2_e is
  generic (tr: real := 1.0e-9; tf: real := 1.0e-9);
  port (y: out bit; a: in bit; b: in bit);
end or2_e;

architecture or2 of or2_e is
begin
  process(a,b)
  begin
    y <= a or b after delay_f(a or b, tr, tf);
  end process;
end or2;
```

The block composed of an inverter and an AND circuit (`inv_and`) is described in AleC++ code that instantiates VHDL component `and2`. Structural modeling is used.

```
module inv_and ( bit out y; bit in a; bit in b) {
  signal bit inter; // internal signal, inverter output
  module inv inverter;     // modeled in AleC++
  module and2 and_circ;    // modeled in VHDL

  inverter (inter,a)   { tr=1ns;tf=0.9ns; }
  and_circ (y,inter,b) { tr=1ns;tf=0.9ns; }
}
```

The whole circuit is modeled in VHDL and named `xor2`. It is composed of two components `inv_and` and the component `or2`.

```
--- Two-input XOR gate:
entity xor2_e is
  port (y: out bit; a: in bit; b: in bit);
end or2_e;

architecture xor2 of xor2_e is
  component or2
    generic (tr: real; tf: real);
    port (y: out bit; a, b: in bit);
  end component;
  component inv_and
    port (y: out bit; a: in bit; b: in bit);
  end component;
  signal inter1,inter2: bit;           -- internal nodes
begin
  -- modeled in AleC++
  c1: inv_and port map (inter1,a,b);
  c2: inv_and port map (inter2,b,a);
   -- modeled in VHDL
  c3: or2 generic map (tr=>1.0e-9, tf=>0.8e-9)
          port map (y,inter1,inter2);
end xor2;
```

Finally, to simulate exclusive-OR circuit, it is necessary to have a root module in order to define circuit stimulus and simulation control parameters. The following code represents a simple root module for testing the module `xor2` by checking its output state for all input vectors.

```
library "xor";
root module xor_circuit_test (){
  signal bit a, b, y;
  module xor2 g1;


  g1 (y, a, b);
```

```
        out { signal bit a, b; signal bit y; }

        timing { tstop = 30ns; }
        action {
          process initial {
              a <- '1' after  5ns, '0' after 15ns;
              b <- '1' after 10ns, '0' after 20ns;
          }
        }
      }
```

Simulation results are shown in Fig. 5.45.

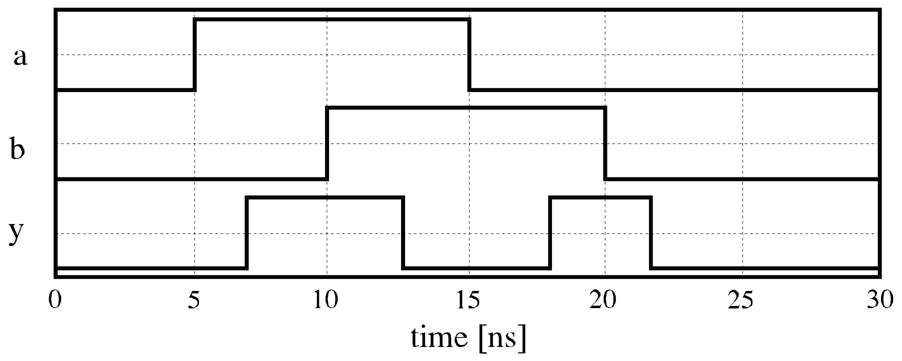


Fig. 5.45: Simulation results for exclusive-OR circuit shown in Fig. 5.43.

5.16 AleC++ - VHDL-AMS co-simulation

VHDL-AMS modeling environment in Alecsis is also supported by the compiler described in the previous section.. The co-simulation concept is the same as it is shown in Fig. 5.42.

Since AleC++ resembles the semantics of standard HDLs, such as VHDL-AMS, the correspondence between language elements can be easily established (Fig. 5.46).

ALEC++		VHDL-AMS
node	$\rightleftarrows$	terminal
current	$\rightleftarrows$	through quantity
flow	$\rightleftarrows$	free quantity
simple eqn, across eqn, through eqn	$\rightleftarrows$	simple simultaneous statement
ddt	$\rightleftarrows$	dot
idt	$\rightleftarrows$	integ

Fig. 5.46. Correspondence between AleC++ and VHDL-AMS

For describing of continuous systems VHDL-AMS uses the theory of differential and algebraic equations (DAE's). A new class of objects, the *quantity*, is introduced to represent the unknowns in the system of DAE's. Also, special kind of quantities called *branch quantities* are used for representing the unknowns in the equations describing conservative systems (systems obeying Kirchhoff's laws). There are two types of branch quantities: *across quantities* (for effort like effects such as voltage or pressure) and *through quantities* (for flow like effects such as current or fluid flow rate). They are declared with reference to two terminals representing nodes of the module. AleC++ uses a similar language element called *link* to describe quantities that appears on a module terminal and represent the unknowns in the behavioral system descriptions. There are five types of links in AleC++: *node*, *current*, *flow*, *charge*, and *signal*. Obviously, node corresponds to a terminal and current to a through quantity. The flows represent general analog quantities and correspond to free quantities in VHDL-AMS. Simple simultaneous statements used for notating DAE's in VHDL-AMS relate to appropriate AleC++ constructs for writing equations. Since the way of writing equations is almost the same in both languages VHDL-AMS compiler can easily determine contributions of the equations from VHDL-AMS model to the matrix of the system of equations describing the whole design. It is necessary in Alecsis to explicitly specify to which matrix row the equation contributes. In equations using free quantities that information can be provided as the equation's label. In equations containing branch quantities terminals can be used to determine matrix rows to which equations contribute.

VHDL-AMS provides conditional and selected forms of the simultaneous statement that allow changing set of equations in the model. Since AleC++ has similar constructs, VHDL-AMS compiler can easily translate those statements into the corresponding object code.

Both languages support signal attributes for derivative and integration over time used in differential equations. AleC++ also supports second-order time derivative attribute and it is implemented in VHDL-AMS compiler, too.

In order to illustrate AleC++/VHDL-AMS mixed-language description and simulation we shall model a mechanical system describing oscillating mass. The structure of the model is shown in Fig. 5.47.

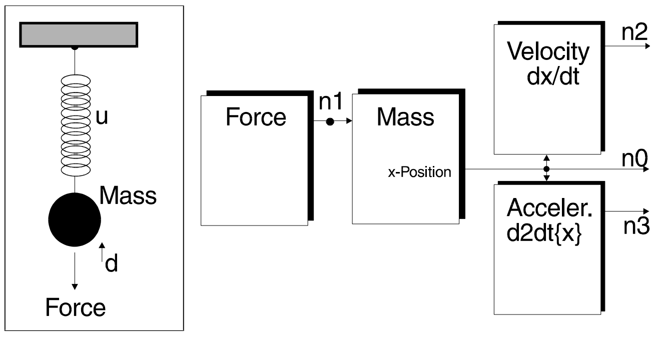


Fig. 5.47. The structure of oscillating mass model

The architecture of the mass is described in VHDL-AMS and defines mechanical equilibrium as a single second-order differential equation using a simple simultaneous statement:

```
entity mass_e is
      generic (m,u,d: real);
      port (quantity x: out real;
            quantity force: in real
);
end entity mass_e;
architecture mass of mass_e is
begin
     x:  m*x'dot'dot + d*x'dot + u*x - 1*force == 0;
end architecture mass;
```

Entities for calculating velocity and acceleration are not necessary for simulation, but are used just to print out the appropriate results:

```
entity velocity_e is
      port (quantity x: real;
            quantity v: real
);
end entity velocity_e;
architecture velocity of velocity_e is
begin
      v:  1*v - 1*x'dot == 0;
end architecture velocity;
```

```
entity acceleration_e is
      port (quantity x: real;
             quantity a: real
);
end entity acceleration_e;
architecture acceleration of acceleration_e is
begin
      a:  1*a - 1*x'dot'dot == 0;
end architecture acceleration;
```

The model stimulus is described in AleC++:

```
module Force (flow force) {
      action (double force_value) {
             double force_out;
             process per_moment {
                    force_out = force_value*exp(-now);
                    eqn force: {force} = force_out;
             }
      }
}
```

In order to simulate the system all models are instantiated and appropriate simulation control parameters are defined in a root module described in AleC++:

```
       #include <alec.h>
       #define Period 15. s
       module  mass_mod(flow  x,  force)  action(double  m,  double  u,
double d);
       module velocity_mod(flow x, v);
       module acceleration_mod(flow x, a);

       library "mass";
       library "velocity";
       library "acceleration";

       module Force (flow force) {
              action (double force_value) {
                     d ouble force_out;
                     process per_moment {
                            force_out = force_value*exp(-now);
                            eqn force: {force} = force_out;
                     }
              }
       }

       root eq() {
              flow n0, position, velocity, acceleration;
              mass_mod p;
              Force F;
              velocity_mod V;
              acceleration_mod A;

              p(position,n0) {m=1; u=1; d=0.35;}
              F(n0) {force_value=10;}
              V(position,velocity);
              A(position,acceleration);

              timing {tstop = Period; a_step = Period/1000;}
              plot  {flow  position;flow  velocity;flow  acceleration;}
}
```

VHDL-AMS models have to be compiled first into the AleC++ object code by using VHDL-AMS compiler. Then, the whole system is simulated using Alecsis.

The mixed-language simulation results are shown in Fig. 5.48. Traced signals are positon, velocity and acceleration, respectively.

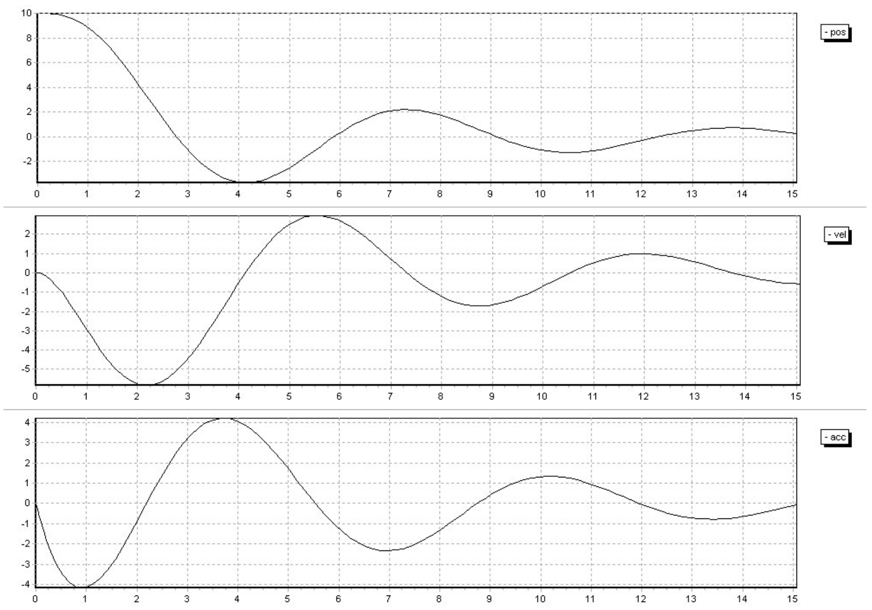


Fig. 5.48. Simulation results of the oscillating mass shown in Fig. 5.47.

5.17. Bouncing ball -- discontinuity example

Most of classical simulators cannot cope with the discontinuity of the system's behavior when it occurs during the simulation. This means that systems that have response that is not differentiable cannot be simulated with such simulators. Primary reason is the way in which a simulator works: for finding the solution in one time point it needs solution(s) in one (or more) past point(s), depending on the integration method.

Excellent, while simple example of discontinuity is bouncing ball. Discontinuity occurs when the ball reaches ground. The sign of velocity is changed and the ball rebounds. It is of crucial importance to extract the "discontinuity" instant, so that the simulator can react by calculating new initial conditions.

The idealized ball’s trajectory is described by the equation:

$$\frac{d^2x}{dt^2} + g + \rho \cdot \left(\frac{dx}{dt}\right)^2 = 0 \qquad (5.11)$$

where g is the gravity constant and ρ the air damping coefficient.

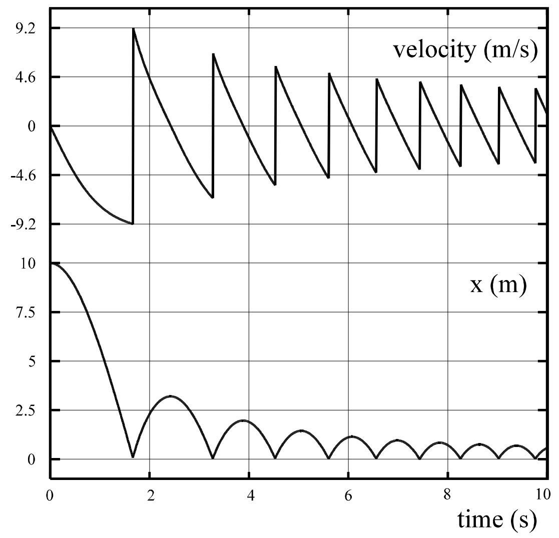


Fig. 5.49: Simulation results for bouncing ball

Discontinuity is determined when x falls down to zero. The results of the simulation are given in Fig 5.49. Lost of energy due to the impact is not considered in this example.

5.18 Modeling of D/A interface for mixed-mode behavioral simulation

The design of electronic and telecommunication integrated circuits is unavoidably faced with simulation of analogue subsystems of ever rising complexity thereby building more complex mixed-signal systems containing both analogue and digital parts. Design of such systems needs simulation tools that perform fast and accurate in the same time. Main obstacle to this requirement is related to the difficulties in high level modeling of the analogue part and accurately enough modeling of the digital-analogue (D/A) and analogue-digital (A/D) interfaces being frequently encountered in such systems. In fact at the (D/A) interface one needs to model the output circuit of the digital part in order to enable electrical excitation for the analogue load. In the opposite case, at the (A/D) interface, we need to model the input impedance of the digital part in order to establish conditions for computation of the voltage and current at the interface. Having in mind that the simulation is performed in the time domain, the fact that we are dealing with mixed-level simulation, and the complexity and non-linearity of the circuits involved, one generally applies behavioral modeling for these purposes.

We will consider here the situation when the signal is transmitted from the logic to the analogue element, and then we need digital to analogue conversion. Since the load is analogue, there is a problem in generating of the signal waveform on the output of the digital circuit. Modeling of D/A node is very complex because one needs to get the waveform of the signal that drives the analogue part of the circuit out of the set of logic states. The conversion algorithms are mostly based on the synthesis of the electronic circuit that replaces the logic element, and that is applied as excitation to the actual node. The propagations of the logic elements should also be considered.

The following solution is based on artificial neural networks. It is considered very convenient because the function is approximated using the measured values, and no electronic circuit synthesis is needed.

A new topology of the circuit is proposed for this purpose, being depicted in Fig. 5.50.

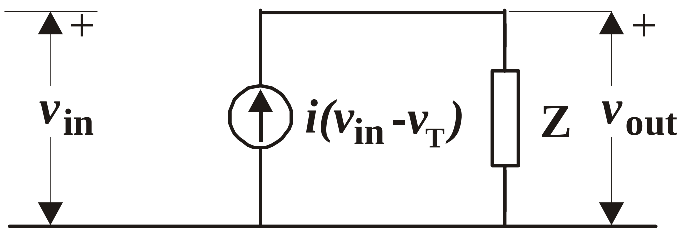


Fig. 5.50. Circuit representation of the model

An inverter is considered here, and v_{in} stands for a controlling ramp-shaped voltage-waveform,

$$i(v_{in}-v_T)=I_{max}\left[1-\text{th}(v_{in}-v_T)\right], \tag{5.12}$$

and *Z* is a time delay recurrent neural network approximating the function

$$v_{out}=Z(i) \tag{5.13}$$

I_{max} is the maximal supply current during the transition in the inverter, and v_T is (usually) equal $V_{DD}/2$, V_{DD} being the supply voltage. Obviously, the ANN model of *Z* has one input (current) and one output (voltage) terminal. The training of the network is performed based on training pairs (*i*(t), v_{out}(t)), where *i*(t) is calculated from (5.12) while v_{out}(t) is obtained from simulation of the circuit to be modeled (here inverter). The neural network is a time delay recurrent network (*Z*), with one hidden layer, five input, three hidden and one output neuron.

Inverter has only one input, so the value of v_{in} voltage is only one. When we are dealing with multi-input circuits, there exist more input voltages because for every combination of inputs there is corresponding output impedance. Also, there is a problem with sequential circuits because the output state depends on input states, as well as on previous output state. During the logic simulation, there always exists information about the events and time of their happening, so it is known what inputs caused certain state and which impedance should be applied to the certain node.

The first results are shown in Fig. 5.51. Here output waveforms of the original inverter and the model are shown in order to show the quality of the approximation procedure. Unloaded circuits are simulated. A behavioral simulator, Alecsis is used to exercise such model.

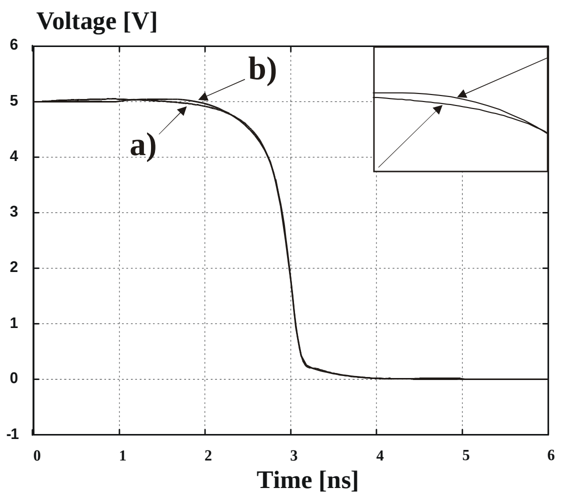


Fig. 5.51. Digital-analogue interface modeling a) response of an unloaded CMOS inverter (considered as digital output) and b) of the new model

The following three examples are intended to check the modeling procedure based on situations not present during the training procedure. Fig. 5.52 represents two responses. The first trace is the output voltage of an inverter (all modeled by regular transistor models) being loaded by inverter. The second one represents the response of the same circuit with ANN model used for the driving part and circuit model for the loading. This situation was unknown in the modeling process.

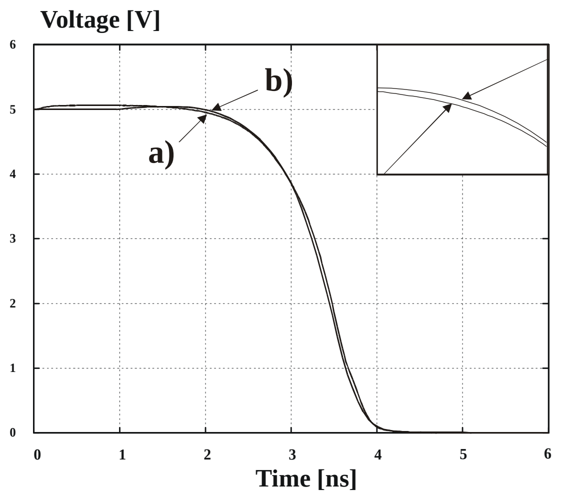


Fig. 5.52. Digital-analogue interface modeling. a) response of an inverter loaded by inverter and b) of a model loaded by inverter

Further, Fig. 5.53 represents similar comparison the loading element being a transmission line modeled by a π -*RC* network.

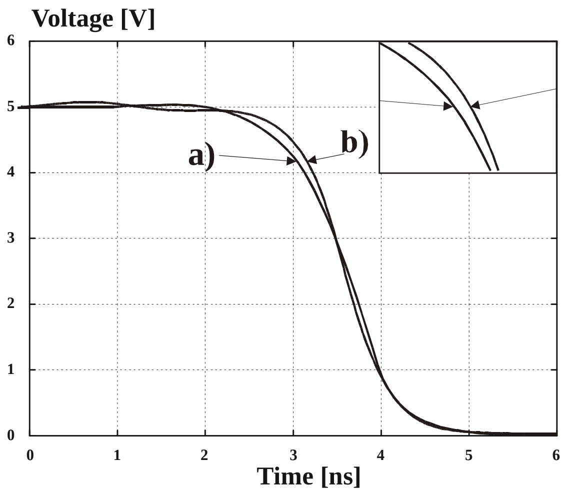


Fig. 5.53. Digital-analogue interface modeling. a) response of an inverter loaded by RC π -network and b) of a model loaded by RC π -network

Finally, a diode load was used to demonstrate the successfulness of the ANN model in the case of "large" non-linear dynamic load. The comparison of the circuit simulation and behavioral simulation are given in Fig. 5.54.

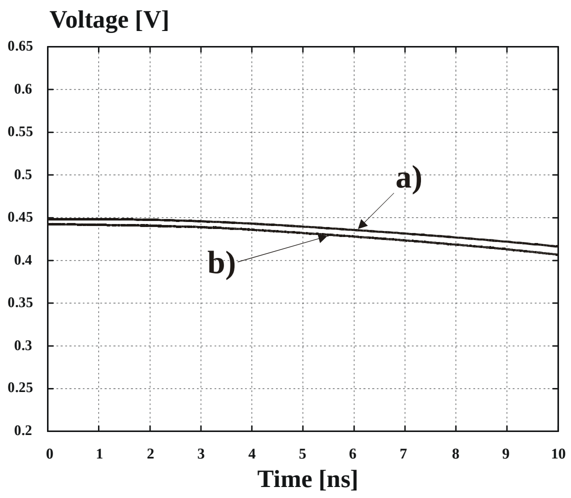


Fig. 5.54. Digital-analogue interface modeling. a) response of an inverter loaded by diode and b) of a model loaded by diode

5.19 Frequency domain simulation

In addition to time-domain simulation, Alecsis simulator also enables the frequency domain simulation. Following examples illustrate this feature.

In order to implement AC domain simulation input language of Alecsis simulator had to be extended. First a new command `ac` for AC analysis has been added. E.g.

```
ac {fscale=2;fstart=10;fstop=1G;fnum=100;}
```

The AC analysis works with complex numbers, and since AleC++ is an object oriented language, all the variables have been added following properties: `amp` (amplitude), `phase` (phase), `real` (real part) and `imag` (imaginary part). E.g. `output->phase` gives the phase of the output. Also a `dc_value` property is added providing the value of the variable after dc simulation, which can be very useful during small-signal modeling.

New types of independent generators have been added as well. Those are `vac` - voltage and `cac` - current generator. They accept following parameters: `amp` - amplitude and `phase` - phase of the signal.

The lists of parameters for any type of controlled sources have been extended. For example, voltage controlled voltage source, beside the `gain`, that now becomes real part, `igain` parameter has been added, representing the imaginary part of complex gain.

New type of process with `per_frequency` synchronization has been supplied, providing the user to write AleC++ code to be processed per frequency. Operator `currfreq` is introduced, returning the current frequency of the simulation. Operator `domain` returns the current simulation domain.

In order to enable the behavioral simulation in frequency domain, equations with complex matrix entries are implemented. There are three types of equations: *simple*, *through* and *across*.

Behavioral description of a capacitor using through `eqn` statement is:

```
module my_capacitor (node i, j){
  action (double value=1.e-15){
...
    process per_frequency {
      double y = 2*M_PI*value*currfreq;
      eqn {i, j}.i = y*{i->im, j->im}.v;
    }
  }
}
```

All these features, along with object-orientedness of the AleC++ provide a powerful tool for behavioral frequency domain simulation.

As a simulation example for passive circuits, a ten stage crystal band-pass filter shown in figure 5.55.a has been simulated. Obtained amplitude characteristic is shown in figure 5.55.b.

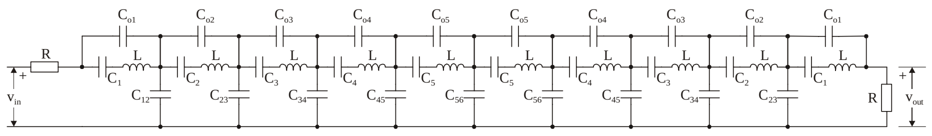


Fig. 5.55.a: Ten-stage crystal band-pass filter

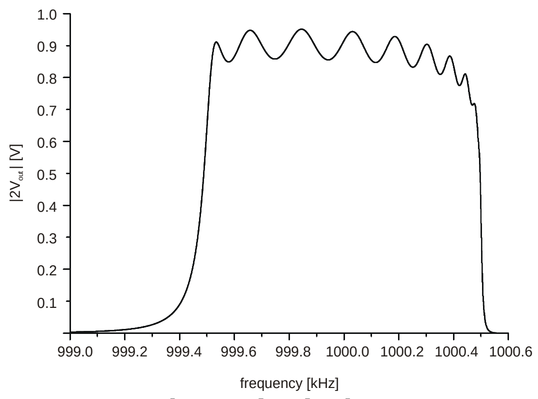


Fig. 5.55.b: Amplitude characteristic

As an example for simulation of active circuits, two-stage frequency compensated MOS amplifier, shown in Fig. 5.56, was simulated. Transistor M_{13} has a function of a zero-canceling resistor. Transistors M_{14}, M_{15} i M_{16} bias this transistor.

Fig. 5.57 gives the simulation results for variation of channel width of the M_{15} transistor.

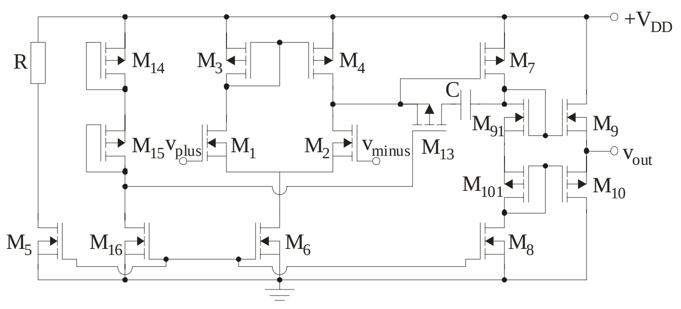


Fig. 5.56: Two-stage operational amplifier

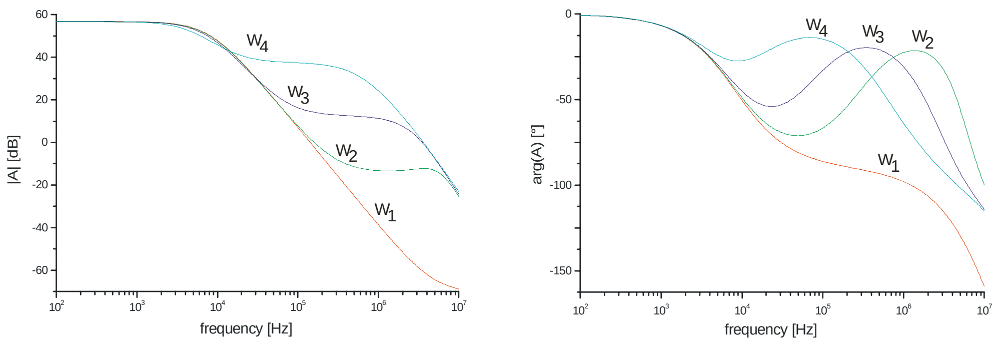


Fig. 5.57: Amplitude and phase for the variation of channel width of M_{15}

The following example illustrates the fully behavioral description using controlled sources. Fig. 5.58 shows the simulated circuit. Block named *core* can model virtually any circuit. All the parameters of the block (here h - parameters) can be assigned functions of arbitrary complexity. Combination of circuit and behavioral model is used. Sample of simulation code in AleC++ shows how one of the core parameters is calculated. The results of the simulation are depicted in Fig 5.59. Complex class enables complex arithmetics.

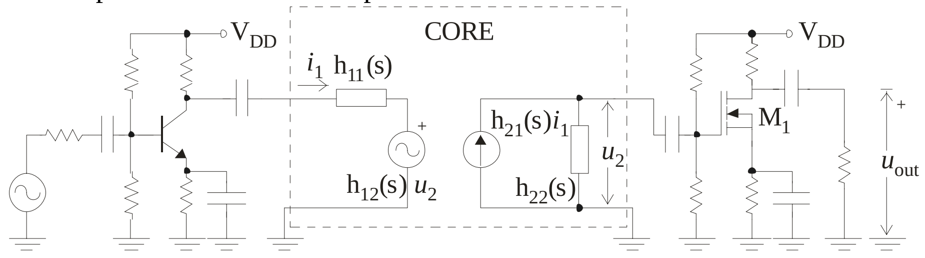
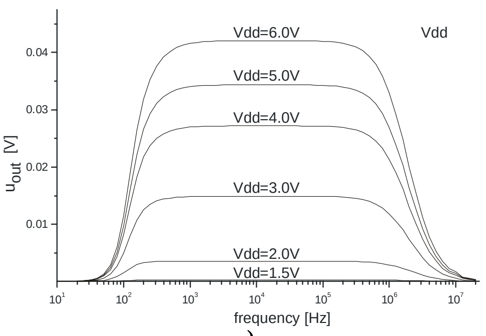
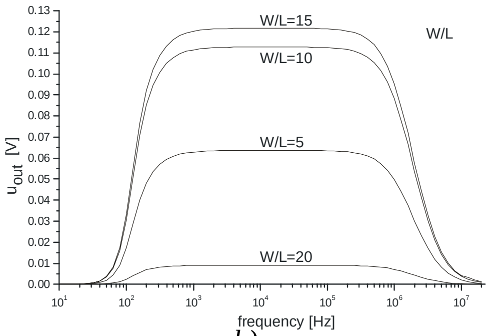


Fig. 5.58: Simulated circuit

```
#include <complex.ac> //complex class
...
root module test_circuit()
{
  IP_core ip;
...
  ip (3, 0, 4, 0);
...
  action per_frequency (){
    double h210 = 100 + 1.5*(VDD - 6);
    double w = currfreq/(2*M_PI);
    double wz1 = 10/(2*M_PI);
    complex s(0, w);
    complex h21 = h210*(s/wz1+1);
...
    ip->h21r = h21.real;
    ip->h21i = h21.imag;
...
  }
  ac{fscale=2; fstart=1; fstop=10MEG; fnum=10;}
}
```



a)



b)

Fig. 5.59: Simulated results: output voltage for variations of a) Vdd and b) W/L ratio of the M1 transistor

Another example represents the behavioral modelling of the BJT. Sample of the AleC++ code shows how capacitors are modeled behaviorally using eqn statements.

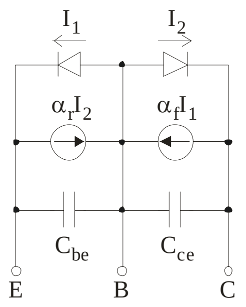


Fig. 5.60: Ebers-Moll model of the BJT

```
#include <alec.h> //standard header
...
     process per_frequency {
      double Vbe_dc, Vbc_dc, alpha, gdbe, gdbc, cbe, cbc;
      alpha = 1/vt;
      Vbe_dc = (node) b->dc_value - (node) e->dc_value;
      Vbc_dc = (node) b->dc_value - (node) c->dc_value;
      gdbe = i0/vt*exp(alpha*Vbe_dc);
      gdbc = i0/vt*exp(alpha*Vbc_dc);
      cbe = tau*gdbe //diffusion component
           + Cbe_junc*pow((1-Vbe_dc/vt), -0.5);
                     //junction component
      cbc = tau*gdbc //diffusion component
           + Cbc_junc*pow((1-Vbc_dc/vt), -0.5);
                     //junction component
      eqn {b, e}.i = gdbe*{b, e} - ar*gdbc*{b, c} +
                     2*M_PI*cbe*currfreq*{b->im, e->im}.v;
      eqn {b, c}.i = gdbc*{b, c} - ar*gdbe*{b, e} +
                     2*M_PI*cbc*currfreq*{b->im, c->im}.v;
    }
...
```

5.20 Space-continuous analogue systems

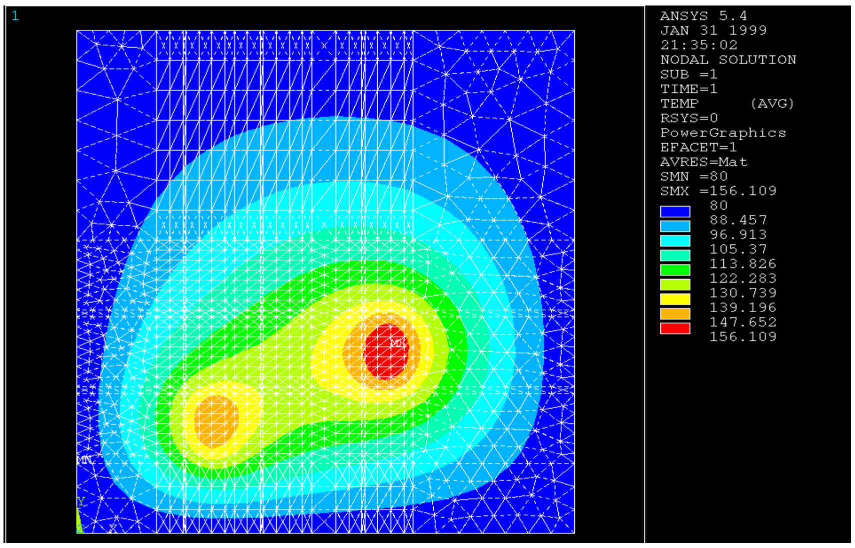
Circuit simulation normally uses so-called lumped models – continuous in time, but discrete in space. However, deep submicron designs and mixed-domain systems (semiconductor sensors and actuators) often demand introduction of models that cannot be correctly defined with lumped models.

A solution can be to define space-continuous models as a set of lumped models. In standard circuit simulators, this can lead to extremely large input files, which also makes design errors intraceable. However, AleC++ posses a mechanism to define arrays of lumped analogue models using simple syntax costructions. For instance, array of finite elements that describe thermal or mechanical behavior can be contained in a module named **elementary_matrix**. An array of such matrices can be defined in **process structural**:

```
module finite_element model( … ) {

   /* declaration */
   module el_matrix EM;
   node n1[auto]; // declaration of nodes
   node n2[auto]; // that will be dynamically
   ….             // allocated

   /* structure is here omitted, as it
      will be given in process structural */
   action(double size_n1,
          double size_n2,
          double no_of_elems) {
      process structural {
         allocate n1[size_n1];
         allocate n2[size_n2];
         int i;
         for (i=0; i<no_of_elems; i++) {
            clone EM[i] (n1[i-1],n2[i-1],...);
         }

         …
      }
   }
}
```

This feature had been extensively used in Alecsis for transmission line modeling, micromechanical systems simulation, thermoelectrical simulation, and smart-material analysis.

In Fig. 5.57, finite element simulation results obtained using ANSYS simulator for temperature distribution on a chip are shown. Fig. 5.58. shows the same results obtained using Alecsis. The relative error is smaller than 0.005% for static analyses. Comparable results may be achieved in transient simulation with coupled electronic components.



```
ANSYS 5.4
JAN 31 1999
21:35:02
NODAL SOLUTION
SUB =1
TIME=1
TEMP      (AVG)
RSYS=0
PowerGraphics
EFACET=1
AVRES=Mat
SMN =80
SMX =156.109
     80
     88.457
     96.913
     105.37
     113.826
     122.283
     130.739
     139.196
     147.652
     156.109
```

Fig. 5.57. Temperature distribution on the chip (ANSYS results)

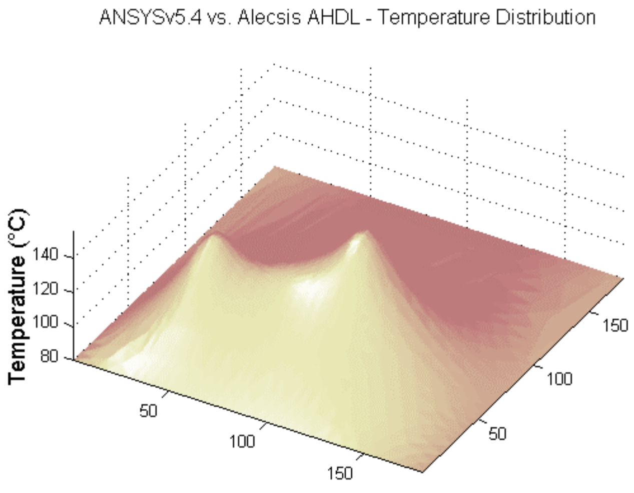


Fig. 5.58. Temperature distribution on the chip (Alecsis simulation/MathCAD visualizatison)

The application of mixed-domain simulator with analog hardware description languages and mixed analog/digital simulation seems to be promising for this class of engineering problems.

6. Summary

An object-oriented behavioral simulation system called Alecsis has been described. Its main advantages over the existing solutions are generality, wide range of application, object-oriented approach and C++ - based hardware description language AleC++. The system provides for separate compilation of analog, digital and hybrid modules, as well as C/C++ functions, and their storage in design libraries for further usage. The simulation engine contains the analog, digital and mixed-signal simulation algorithms. The hardware description language serves for problem description and also as the simulator engine customizing tool. The capabilities of Alecsis are demonstrated on digital, analog, mixed-signal, mixed-domain, and hardware/software modeling and simulation examples.

One of the specific features is the possibility to use AleC++ in interpreted mode, which simplifies model development. Language interpretation is fast, since the code that is interpreted is optimized.

Alecsis uses the concept of concurrent processes and signals similar to the one in VHDL. At the moment we are developing a VHDL compiler that will output standard Alecsis libraries. It will enable VHDL entities and architectures to be imported and used as regular AleC++ modules. Another compiler will import

SPICE input files. Both tools are expected to further increase applicability and flexibility of the Alecsis simulator.

References

[Aho86] A. V. Aho, R. Sethi, J. D. Ullman, "Compilers - principles, techniques and tools", Addison-Wesley Publishing Company, Massachusetts, 1986.

[Anta95] B. A. A. Antao, A. J. Brodersen, “Behavioral simulation for analog system design verification”, IEEE Trans. on VLSI Systems, Vol. 3, No. 3, pp. 417-429, September 1995.

[Anta96] B. A. A. Antao, "AHD languages - a must for time-critical designs", IEEE Circuits & Devices, pp. 12-17, July 1996.

[Bedr92] D. Bedrosian, J. Vlach, "Time-domain analysis of networks with internally controlled switches", IEEE Trans. on Circuits and Systems, Vol. 39, pp. 199-212, March 1992.

[Berr71] R. D. Berry, “An optimal ordering of electronic circuit equations for a sparse matrix solution”, IEEE Trans. on Circuit Theory, Vol. 18, No. 1, pp. 40-50, January 1971.

[Bore99] Joseph Borel, ed., “The MEDEA design automation roadmap”, Preliminary Version, MEDEA, 1999.

[Brow91] A. D. Brown, M. Zwolinski, K. G. Nichols, T. J. Kazmierski, “CONFIDENCE in mixed-mode circuit simulation”, 1991 Research

Journal, Dept. of Electronics and Computer Sciences, University of Southampton, pp. 177-179, 1991.

[Brow92] A. D. Brown, M. Zwolinski, K.G. Nichols, T. J. Kazmierski, "The design of a language for mixed-mode circuit simulation", 1992 Research Journal, Dept. of Electronics and Computer Science, University of Southampton, pp. 103-105, 1992.

[Caba99] D. Cabanis, “Application of object-orientation to HDL-basec designs”, PhD Thesis, Bournemouth University, 1999.

[Chen84] C-F. Chen, C-Y. Lo, H. N. Nham, P. Subramaniam, "The second generation MOTIS mixed-mode simulator", Proc. 21st Design Automation Conference, pp. 10-16, 1984.

[Chri99] E. Christen, K. Bakalar, “VHDL-AMS – A hardware description language for analog and mixed-signal applications”, IEEE Trans. on CAS II, Vol. 46, No. 10, October 1999, pp. 1263-1272.

[Corm88] T. Corman, M. U. Wimbrow, "Coupling a digital logic simulator and an analog circuit simulator", VLSI Systems Design, pp. 40-47, February 1988.

[Cox93] E. Cox, "Adaptive fuzzy systems", IEEE Spectrum, Vol. 30, No. 2, pp. 27-31, February 1993.

[DeMa80] H. J. De Man, J. Rabaey, G. Arnout, J. Vandewalle, "Practical implementation of a general computer aided design technique for switched-capacitor circuits", IEEE J. Solid State Circuits, Vol. SC-15, No. 2, pp. 190-200, April 1980.

[Gear71] C. W. Gear, ‘Simultaneous numerical solution of differential-algebraic equations”, IEEE Trans. on Circuit Theory, Vol. 18, No. 1, pp. 89-95, January 1971.

[Getr89] I. E. Getreu, "Behavioral modeling of analog blocks using the SABER simulator", 32nd Midwest Symposium on CAS, pp. 977-980, Illinois, August 1989.

[Harr85] R. L. Harris, A. N. Omid, "An improved version of the HILO simulator", Electronic Engineering, pp. 159-167, September 1985.

[Ho75] C.-W. Ho, A. E. Ruehli, P. A. Brennan, “The modified nodal approach to network analysis”, IEEE Trans. Circuits Syst., Vol. 22, pp. 504-509, June 1975.

[IEEE99] -, “Definition of Analog and Mixed Signal Extensions to IEEE standard VHDL”, IEEE Standard 1364-1995.

[Jako99] M. Jakovljević, Ž. Mrčarica, P.A. Fotiu, H. Detter, V. Litovski, "Analogue behavioural simulator as a tool for coupled electro-thermal analysis of microsystems", Proc. of IEEE Conf. MIEL'99.

[Jime98] V. Jimenez et al., "Simulation of flow sensor for home appliances", Microelectronics Journal, Vo. 29, p. 283, 1998.

[Kazm92] T. J. Kazmierski, A. D. Brown, K. G. Nichols, M. Zwolinski, "A general purpose network solving system", IFIP Transactions AI, VLSI-91, ed. A. Hallaes and P. B. Denyer, pp. 147-156, North-Holland, 1992.

[Kund96] K. Kundert, E. Filseth, “Modeling a wireless IF sampling circuit in Verilog-A”, Computer Design, pp. 102-104, January 1996.

[Laz72] Lazović, S. M., "Transfer function synthesis of allpass filters for pulse compression", Electronics Letters, Vol. 8, No. 3, 1972, pp. 77-79.

[Lips89] R. Lipsett, C. Schaefer, C. Ussery, "VHDL: hardware description and design", Kluwer Academic Publishers, New York, 1989.

[Lito92] V. B. Litovski, J. I. Radjenović, Ž. M. Mrčarica and S. Lj. Milenković, "MOS transistor modeling using neural network", Electronic Letters, Vol. 28, Issue 18, pp. 1766-1768, August 1992.

[Lito93] V. B. Litovski, Z. Ž. Panti}-Tanner, "Artificial neural network application in electronic circuit analysis", Proc. World Congress on Neural Networks, WCNN'93, pp. I-293-I-299, Portland, Oregon, July 1993.

[Li00] Y. Li, M. Leeser, “HML, a novel hardware description language and its translation to VHDL”, IEEE Trans. on VLSI Systems, Vol. 8, No. 1, pp. 1-8, February 2000.

[Maks95] D. M. Maksimović, D. B. Glozi}, V. B. Litovski, "Discrete event modeling and simulation in an object-oriented hybrid simulator", Proc. of the 14th IASTED International Conference on Modeling, Identification and Control, pp. 103-106, Igls, Innsbruck, Austria, February 20-22, 1995.

[Melv92] B. Melville, P. Feldmann, S. Moinian, “A C++ based environment for analog circuit simulation", Proc. of IC-CAD (International Conference on Computer Design), pp. 516-519, 1992.

[Melv93a] R. Melville, S. Moinian, P. Feldmann, “Sframe: an efficient system for detailed DC simulation of bipolar analog integrated circuits using continuation methods", Analog Integrated Circuits and Signal Processing , Vol. 3, pp. 163-180, 1993.

[Melv93b] R. C. Melville, Lj. Trajković, S.-C. Fang, L. T. Watson, “Artificial parameter homotopy methods for the DC operating point problem”, IEEE Trans. on Computer-Aided Design, Vol. 12, No. 6, pp. 861-877, June 1993.

[Mile95] I. Milentijević, M. Stojčev, D. Maksimović, "Digit-serial semi-systolic convolver", Facta Universitatis (Ni{), series: Electronics and Energetics, Vol. 8, No. 2, pp. 191-210, 1995.

[Mile96] I. Milentijević, M. Stojčev, D. Maksimović, "Configurable digit-serial convolver of type F", Microelectronics Journal, Vol. 27, No. 6, pp. 559-566, 1996.

[Moin94] S. Moinian, P. Feldmann, R. C. Melville, “Chip-level electro-thermal simulation of bipolar transistor circuits”, Bipolar/BiCMOS Circuits & Technology Meeting, pp. 123-126, 1994.

[Mrča93] Ž. M. Mrčarica, V. B. Litovski, "A new ideal switch model for time domain circuit analysis", International Journal of Electronics, Vol. 74, No. 2, pp. 241-250, February 1993.

[Mrča95a] Ž. M. Mrčarica, T. Ili}, D. Glozi}, V. Litovski, H. Detter, "Mechatronic simulation using Alecsis. Anatomy of the simulator", Proc. of Simulation Congress EUROSIM 95, Vienna, Austria, pp. 651-656, September 1995.

[Mrča95b] Ž. M. Mrčarica, D. Glozi}, V. Litovski, H. Detter, "Simulation of microsystems using a behavioural hybrid simulator Alecsis", MICROSIM 95, Southampton, UK, pp. 129-136, September 1995.

[Mrča96] Ž. Mrčarica, D. Glozi}, V. Litovski, H. Detter, “Describing space-continuous models of microelectromechanical devices for behavioural simulation”, European Design Automation Conference EURO-DAC with EURO-VHDL '96, Geneva, Switzerland, pp. 316-321, September 16-20, 1996.

[Mrča99] Ž. M. Mrčarica, T. R. Ili}, V. B. Litovski, "Time domain analysis of nonlinear switched analogue networks with internally controlled switches", IEEE Trans. on Circuits and Systems, March 1999, pp. 373-378.

[Nage75] L. W. Nagel, "SPICE 2: A computer program to simulate semiconductor circuits", Memorandum ERL-M520, University of California, Berkeley, May 1975.

[Newt78] A. R. Newton, D. O. Pederson, "A simulation program with large-scale integrated circuit emphasis", Proc. IEEE International Symposium on Circuits and Systems, pp. 1-4, New York, May 1978.

[Nich92] K. G. Nichols, A. D. Brown, M. Zwolinski, T. J. Kazmierski, “A logic-analog interface model”, 1992 Research Journal, Dept. of Electronics and Computer Sciences, University of Southampton, pp. 106-109, 1992.

[Odry86] P. Odryna, K. Nazareth, C. Christensen, "A workstation-based mixed mode circuit simulator", Proc. 23rd IEEE/ACM DAC, pp. 186-192, Las Vegas, 1986.

[Opal90] A. Opal, J. Vlach, "Consistent initial conditions of linear switched networks", IEEE Trans. on Circuits and Systems, Vol. 37, pp. 364-372, March 1990.

[OVI] Verilog-AMS, Open Verilog International, under development.

[Pabs95] D. Pabst, “HDL-A VHDL-based analog and mixed signal model description language”, Tutorial T1 of Simulation Congress EUROSIM ‘95, Vienna, Austria, September 1995.

[Rhod96] D. L. Rhodes, "A design language for analog circuits", IEEE Spectrum, pp. 43-48, October 1996.

[Roch96] S. Rochel, “Modeling data-acquisition components in HDL-A”, Computer Design, pp. 104-106, January 1996.

[Saka85] K. A. Sakallah, S. W. Director, "SAMSON2: An event driven VLSI circuit simulator", IEEE Trans. on CAD, vol. 4, No. 4, pp. 668-684, October 1985.

[Schw87] M. Schwartz, "Telecommunication networks: protocols, modeling and analysis", Adison-Wesley Publishing Company, 1987.

[Smit92] M. J. S. Smith, "More logic synthesis for ASICs", IEEE Spectrum, Vol. 29, No. 11, pp. 44-48, November 1992.

[SysC] http://www.systemc.org/

[Timo59] S. Timoschenko, S. Woinowsky-Krieger, ‘Theory of plates and shells”, McGraw-Hill Inc., 1959.

[Thom91] D. E. Thomas, P. Moorby, "The Verilog hardware description language", Kluwer Academic Publishers, New York, 1991.

[Vach95] A. Vachoux, "VHDL-A: A future standard for analog and mixed digital-analog description and simulation", Proc. of the IEEE Int. Symposium on Industrial Electronics, pp. 39-44, Athens, Greece, July 10-14, 1995.

[Vlac95] J. Vlach, J. M. Wojciechowski, A. Opal, "Analysis of nonlinear networks with inconsistent initial conditions", IEEE Trans. on Circuits and Systems, Vol. 42, pp. 195-200, April 1995.

[Wang87] L. T. Wang, N. E. Hoower, E. H. Porter, J. J. Zasio, "SSIM: a software levelized compiled-code simulator", Proc. 24th ACM/IEEE DAC, pp. 2-8, 1987.

[Watk93] K. Watkins, "Discrete event simulation in C", McGraw-Hill, England, 1993.

List of published papers

Simulation algorithms

Foreign conferences

Litovski, V., Petković, P., *"Time Domain Black-Box Modelling Of CMOS Structures And Analog Timing Simulation"*, 3rd Annual European Computer Conf., COMPEURO '89, Hamburg, May, 1989, pp. 5.142-5.143.

Mrčarica, Ž., Litovski, V., *"Some Considerations Of Short-Channel MOS Model Selection For An Analog Circuit Simulation Program"*, Microsystem '92, Bratislava, 17-19. November, 1992.

Maksimović, D., Glozić, D., Litovski, V., *"Logic Simulation Modeling Techniques In Hybrid Simulator Alecsis2.0"*, 15-16 International Annual School on Semiconductor and Hybrid Technologies, 1992-93, Sozopol, May, 1994, pp. 133-139.

Zwolinski, M., Litovski, V., *"Hybrid Simulation: The State-Of-The-Art"*, 2nd Serbian Conference on Microelectronics and Optoelectronics, MIOPEL '93, Niš, Srbija, 26-28 Oktobar, 1993, pp. 375-382.

Maksimović, D., Glozić, D., Litovski, V., *"Discrete Event Modeling And Simulation In An Object Oriented Hybrid Simulator"*, XIV IASTED International Conference: Modelling Identification and Control, Innsbruck, Feb., 1995, pp. 103-106.

Mrčarica, Ž., Ilić, T., Glozić, D., Detter, H., Litovski, V., *"Mechatronics Simulation Using Alecsis, Anatomy Of Simulator"*, EUROSIM '95, Vienna, 1995, pp. 601-604.

Mrčarica, Ž., Glozić, D., Ilić, T., Litovski, V., *"Capacitive Preassure Sensor For Quasi-Static Conditions Followed By A/D Conversion"*, 20th International Conf. on Microelectronics MIEL '95, Vol. 2, Niš, 12.-14. Sep., 1995, pp. 601-604.

Mrčarica, Ž., Glozić, D., Litovski, V., Detter, H., *"Simulation Of Microsystems Using A Behavioural Hybrid Simulator Alecsis"*, First Int. Conf. on Microsystems and Microstructures, MICROSIM'95, Southampton, 26-28 Sep., 1995, pp. 129-136. Printed also in: Adey, R.A., Lahrmann, A., Lemboe Ilmann, C., editors, “Simulation and Design of Microsystems and Microstructures” , Computational Mechanics Publication, Southampton, GB, 1995.

Mrčarica, Ž., Detter, H., Glozić, D., Litovski, V., *"Describing Space-Continous Models Of Microelectromechanical Devices For Behavioural Simulation"*, European Design Automation Conference, EURO-DAC '96 with EURO-VHDL '96, Geneva, 16-20 Sep., 1996, pp. 316-321.

Mrčarica, Ž., Ranđelović, Z., Jakovljević, M., Litovski, V., Detter, H., *"Methods For Description Of Microelectromechanical Device Models For System-Level Simulation"*, in Adey,R.A. and Renaud,Ph., ed. "Microsim II, Sec. Inter. Conf. on the Simulation and Design of Microsystems and Microstructures, Proceed.", Lausanne, Sep., 1997, pp. 271-280.

Mrčarica, Ž., Litovski, V., Ilić, T., *"Simulation Of The Pressure Sensor Interaction With Its Electronic Environment"*, Micromechanics Europe 1977, Eighth Workshop on Micromachining, Micromechanics and Microsystems, MME '97, Southampton, Kingdom, 31 Aug. - 2 Sep., 1997, pp. 246-249.

Aleksić, D., Litovski, V., Milenković, S., *"TCP/IP Protocol Modeling And Simulation Using Alecsis 2.3"*, 3rd Int. Conf. on Telecommunications in Modern Satellite, Cable and Broadcasting Services TELSIKS '97, Niš, Oct., 1997, pp. 564-567.

Mrčarica, Ž., Risojević, V., Lenczner, M., Jakovljević, M., Litovski, V., "*Integrated Simulator For Mems Using FEM Implementation In AHDL And Frontal Solver For Large Sparse Systems Of Equations*", Design, Test and Microfabrication of MEMS, MOEMS '98, CAD, Design and Test, Paris, 30. Mar. - 1. Apr., 1999, pp. 306-315.

Maksimović, D., Litovski, V., "*Inverse Circuit Simulation Method For Topological Delays Estimation With Logic Simulator*", 22nd Int. Conf. on Microelectronics, MIEL '99, Niš, May, 2000, pp. 707-710.

Jakovljević, M., Mrčarica, Ž., Fotiu, P., Detter, H., Litovski, V., "*Analogue Behavioural Simulator As A Tool For Coupled Electro-Thermal Analysis Of Microsystems*", Proc. of the 22nd Int. Conf. on Microelectronics, MIEL '99, Niš, Yugoslavia, May, 2000, pp. 561-564.

Zarković, K., Ilić, T., Savić, M., and Litovski, V., "*ANN application in Modeling of Dynamic Two-Terminal Linear Circuits*", Proceeding of ETAI'2000, September 2000, Ohrid, pp. E-27-E-30.

Jakovljević, M., Fotiu, P., Mrčarica, Ž., Litovski, V., and Detter, H., "*Electro-thermal simulation of microsystems with mixed abstraction modelling*", Proceedings of SSSS2000, Niš, Sept. 2000, pp. 13-22.

Ilić, T., Zarković, K., Litovski, V., and Damper, R., "*ANN Application in Modelling of Dynamic Linear Circuits*", Proceedings of SSSS2000, Niš, Sept. 2000, pp. 43-47.

Stefanović, D., Sokolović, M., Petković, P., and Litovski, V., "*T-Spice-Alecsis Co-simulation*", Proceedings of MIEL'02, Niš, 2002, pp. 625-628.

Andrejević, M., Milovanović, D., Petković, P., Litovski, V., "*Extraction of Frequency Characteristics of Switched-Capacitor Circuits Using Time-Domain Analysis*", Proceedings of MIEL'02, Niš, 2002 pp. 635-638.

Andrejević, M., Litovski, V., "*Electronic Modelling Using ANNs For Analogue and Mixed-Mode Behavioural Simulation*", Proc. of Neurel 2002, 2002, pp. 113-118.

Andrejević, M., Litovski, V., "*ANN Application In Modelling of Chua's Circuit*", Proc. of Neurel 2002, Belgrade, September 2002, pp. 119-122.

Andrejević, M., Litovski, V., "*Non-Linear Dynamic Network Modelling Using Neural Networks*", Int. Congress on Computional and Applied Mathematics, Leuven, Belgium, 22.July-26.July, 2002, pp. 16.

Savić, M., Litovski, V., "*Frequency Domain Electronic Circuit Analysis in an Object Oriented Environment*", Proc. of the 11th Int. Scientific and Appl. Sci. Conf. of Electronics ET2002, Sep., 2002, Sozopol, 2002. Vol. 2, pp. 65-70

Litovski V., Andrejević M., Damper. R., "*Modeling the d/a interface for mixed-mode behavioral simulation*", EUROCON 2003, Ljubljana, Sept. 2003, pp. A130-A133

Foreign papers

Mrčarica, Ž., Litovski, V., *"A New Ideal Switch Model For Time-Domain Circuit Analysis"*, International Journal of Electronics, Vol. 74, No. 2, 1993, pp. 241-250.

Mrčarica, Ž., Vujanić, A., Detter, H., Litovski, V., *"Simulation Of Micromechanical Systems"*, 6th International Journal of Theoretical Electrotechnics, Thessaloniki, Greece, Sep., 1996, pp. 141-148.

Litovski, V., Mrčarica, Ž., Ilić, T., *"Simulation Of Non-Linear Magnetic Circuits Modelled Using Artificial Neural Network"*, Journal Simulation Practice and Theory, Vol. 5, 1997, pp. 553-570.

Maksimović, D., Litovski, V., *"Logic Simulation Based Method For Digital CMOS VLSI Circuits Power Estimation "*, 7th International J. Of Theoretical Electrotechnics, Join to ISTET '97, Cottbus, 1999, pp. 40-43.

Maksimović, D., Litovski, V., "*Tuning The Logic Simulator For Timing Analysis*", Electronics Letters, Vol. 35, No. 10, 13th May, 1999, pp. 800-802.

Mrčarica, Ž., Ilić, T., Litovski, V., "*Time-Domain Analysis Of Nolinear Switched Networks With Internally Controlled Switches*", IEEE Transactions on Circuits and Systems I, Vol. 46, No. 3, March, 1999, pp. 373-378.

Jakovljević, M., Mrčarica, Ž., Fotiu, P., Detter, H., Litovski, V., "*Transient Electro-Thermal Simulation Of Microsystems With Space-Continuous Thermal Models In Analogue Behavioural Simulator*", Microelectronics and Reliability, Vol. 40, No. 3, March 2000, pp. 507-516.

Ilić, T., Litovski, V.**,** Litovski, I., Stojilković, S., "*Computationally Efficient Large-Change Statical Analysis Of Linear Electronic Circuits*", Microelectronics and Reliability, Vol. 40, 2000, pp. 1023-38.

Litovski, V., “*New Methods in Modeling and Simulation of Electronic Circuits and Systems*”, Scientific Review, No. 29-30, 2001-2002, pp.189-207.

Maksimović, D., Litovski, V., "*Logic simulation methods for longest path delay estimation*", IEE Proceedings Computers and Digital Techniques, Vol. 149, No. 2, 2002, pp. 53-59.

Savić, M., Litovski, V.**,** "*Behavioral object-oriented frequency domain electronic simulation*", Simulation News Europe, No. 38/39, December 2003, pp. 38-39.

Litovski, V. B., Andrejević, M., Petković, P., Damper, R., "*ANN Application to Modelling of the D/A and A/D Interface for Mixed-Mode Behavioural Simulation*", Journal of Circuits, Systems and Computers, Accepted for publication (January 2004).

Andrejević, M., Litovski, V.**,** "*Electronic Modelling Using ANNs For Analogue and Mixed-Mode Behavioural Simulation*", Journal of Automatic Control, ETF Belgrade, accepted for publication

Hardware description languages

Foreign conferences

Damnjanović, M., Litovski, V., *"CAD Systems Based On VHDL Description"*, 2nd Serbian Conf. on Microelectronics and Optoelectronics, MIOPEL '93, Niš, 26-28 October, 1993, pp. 443-448.

Dimić, Ž., Damnjanović, M., Glozić, D., Litovski, V., *"VHDL Use For Alecsis Hybrid Simulation"*, Annual School - 17th International Spring seminar on Semiconductor and Hybrid Technologies 1994-1995, Sozopol, 1995, pp. 228-241.

Damnjanović, M., Dimić, Ž., Litovski, V., Glozić, D., *"Hardware Description Language For Alecsis Simulator"*, 20th Int. Conf. on Microelectronics, MIEL '95, Vol. 2, Niš, 12-14 Sep., 1995, pp. 525-528.

Mrčarica, Ž., Litovski, V., Delić, N., Detter, H., *"Modelling Of Micromechanical Devices Using Hardware Description Language"*, 5th Int. Conf. and Exhibition on Micro, Electro, Opto, Mechanical Systems and Components, Microsystem Technologies '96, Potsdam, 17-19 September, 1996, pp. 293-298.

Mrčarica, Ž., Litovski, V., Jakovljević, M., Detter, H., *"Hierarchical Modelling Of Microsystems In An Object-Oriented HDL"*, 21st Int. Con. on Microelectronics, MIEL '97, Vol. 2, Niš, Sep., 1997, pp. 475-478.

Dimić, Ž., Damnjanović, M., Litovski, V., *"VHDL - Alec++ Cosimulation"*, 21st Int. Conf. on Microelectronics, MIEL '97, Vol. 2, Niš, Sep., 1997, pp. 725-728.

Mrčarica, Ž., Maksimović, D., Tasić, A., and Litovski, V., "*Some features of analogue and Mixed-Signal HDL AleC++*", Proceedings of SSSS2000, Niš, September 2000, pp. 37-42.

Maksimović, D., Litovski, V.**,** "*Timing Simulation With VHDL Simulators*", Proceedings of MIEL'02, Niš, 2002, pp. 655-658.

Litovski, V.**,** Damnjanović, M., Andjelković,B., "*ALECSIS/VHDL–AMS Mixed Language Simulation*", Proc. of The 11th Int. Scientific and Appl. Sci. Conf. Electronic ET2002, Sozopol, Sep., 2002. Vol. 1, pp. 25-30

Foreign papers

Mrčarica, Ž., Litovski, V., Detter, H., *"Modelling And Simulation Of Microsystems Using Hardware Description Language"*, Research Journal on Microsystem Technologies, Vol. 3, No. 2, Feb., 1997, pp. 80-85.

Litovski, V., Dimić, Ž., Damnjanović, M., Mrčarica, Ž.*, "Electronic Circuit Simulation In A Mixed-Language Environment*", Miroelectronics Journal, Vol. 29, No. 8, 1998, pp. 553-558.

Litovski, V.**,** Maksimović, D., and Mrčarica, Ž., "*Mixed-signal modelling with AleC++: Specific features of the HDL*", Simulation Practice and Theory, Vol. 8, 2001, pp. 433-449.

Damnjanović, M., Anđelković, B., Litovski, V., "*VHDL-AMS Compiler for Alecsis Simulation Environment*", Electronics, Vol. 6, No. 2, December 2002, pp. 6-11.