Hochschule Esslingen
University of Applied Sciences
Kanalstrasse 33
73728 Esslingen am Neckar
Fachbereich Informationstechnik

2^{nd} Practical Semester

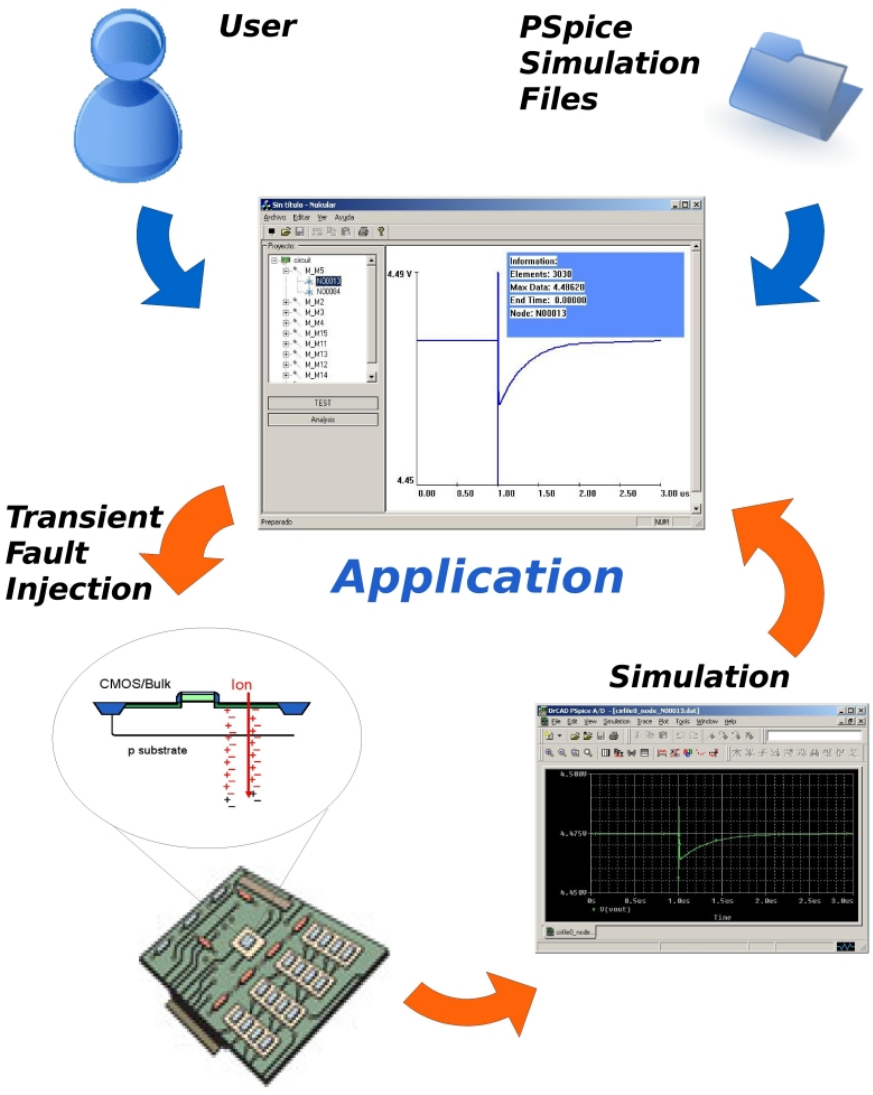
Injection of transient failures in analog CMOS structures

Felix Kunz
Matrikelnr.: 726577
Kommunikationstechnik
Semester NT8
Bellinostrasse 30/2
73732 Reutlingen

Profesor Eduardo Romero & Profesor Gabriela Peretti
depelectronica@yahoo.com.ar
Universidad Tecnológica Nacional
Facultad Regional de Villa María
Departamento Electrónico
Avenida Universidad 450
Villa Maria Crdoba, CP 5900
Argentina

Contents

Project Overview Diagram



The application which is created during the project automatically performs the injection, simulation and analysis of transient faults in circuits using PSpice.

Chapter 1

Introduction

The constant evolution of technologies which are employed to manufacture integrated circuits has lead to a drastic reduction of transistor dimensions. This process converges to a limit at which an invulnerability to errors, caused by external agents is very unlikely. These faults reduce the reliability of the considered circuits.

One of the most serious problems consists of the transient faults provocated by ionizing radiation: *Single-Event Effects* (SEE) are effects triggered in semiconductors as those are being transited by highly energetic and ionizing particles (e.g. protons, neutrons, alpha particles or other heavy ions) [1]. These phenomenons have been studied widely for aerospace environments in which effects like *Single Event Transient* (SET) or *Single Event Upset* (SEU) are very common and require special precautions in the design of aerospace electronics due to their geater exposure of radiation.

However, as the reduction concerning the size of the regarded active components continues, the previously described phenomenons start to occur at sea level principally due to the interaction of neutrons. Therefore the scientific community has started making great efforts to study the behavior of circuits exposed to such conditions.

The effects caused by radiation on digital circuits has been studied thoroughly unlike the effects on analog circuits. As the integration of mixed systems (digital and analog) to one solitary integrated circuit proceeds, the knowledge of the effects of radiation on the analog sections becomes more and more important due to its impact on the whole circuit.

The principal idea is to identify the sensitive nodes of the circuit and design strategies to make them insensible to radiation. One way could be to alter the technology to reach radiation hardness or to alter the circuit layout. Another possibility would be to develop circuits posessing a tolerance which allows them recovering online from induced faults. Both researches require the usage of models which should reproduce the regarded faults as exactly as possible at a reasonable computational complexity. An option to address this research is to inject pulses of electric current at each one of the sensitive nodes of the circuit (generally at the drains of the MOS transistors) and evaluate its behavior regarding injected stimulus of interest: continuous, sine waves of different frequencies, pulses, etc. Generally the pulses of current which approximate the effects generated by radiation most exactly are double exponential, having a form similar to the Gauss error distribution curve. Normally, the evaluation implies connecting those current sources in the schematic of the circuit and simulating its temporal behavior. This process requires the manual modification of the schematic before running the simulation and the individual analysis of each one of the gathered results. Usually, the widely known and accepted simulator SPICE is employed for such researches. A problem arises as the investigation suffers severely of the tedious process and the gathering of results.

The solution of this issue requires an automation of the process of the injection of currents, the simulation and capture of the information and (if possible) the analysis of these experiments. This can be achieved through the development of a special program which is capable of realizing the processing of the experiments. This project proposes the analysis, design and implementation of a system of injection and simulation of transient faults for analogic circuits and its application in the research of effects of radiation on semiconductors.

1.1 Objective

Design and implementation of a system which is capabe of injecting transient faults in analogic circuits. Design of a user-friendly interface which allows editing the characteristics of the pulses which are injected and of the nodes in which they are injected. Develop a program which runs the simulations automatically, stores the results and provides a basic analysis functionality.

1.2 Methodology

The simulator SPICE should be used to perform the simulations as the basic models which are to be analyzed are stored in the SPICE format and simulation models of transistors of different technologies are available for this simulator. The system should provide an interface which allows to edit the parameters of the injected pulses, specify which nodes to analyze and change the basic settings of the simulation. Moreover the system should be capable to accept different forms of pulses. The compability to the simulator has to be assured. The SPICE netlist should serve as basic source of information to the system. The nodes at which the faults have to be injected should be detected automatically and the generated simulation files (e.g. the new netlist) have to be stored. A new simulation should be generated and run for each selected node. Finally, results should be gathered and analyzed for special circuits. If possible, the system shall perform a basic analysis of the results to make it easier to categorize the results.

1.3 Available Infrastructure

The project is developed within the *Grupo de Investigación y Servicios en Electrónica y Control* at the *Facultad Regional Villa María de la Universidad Tecnológica Nacional.* The group has access to the computational resources and the tools necessary to develop the proposed project as well as scientific literature, bibliografy and adequate workspace.

Chapter 2

Theory

2.1 History

In 1962 a publication [2] mentioned for the first time, that terrestrial and cosmic rays would eventually cause *Single-Event Upsets* (SEUs) in microelectronics. Moreover the authors predicted that therefore the minimum volume of these semiconductor devices would be limited to 10 μm. In 1975, Binder *et al* [3] presented the first confirmation of cosmic-ray-induced upsets at the NSREC [1]. This paper reported four upsets within 17 years of satelite operation which had been observed in J-K-flip-flops in a satelite. A few years later, the described effect occurred for the first time in a terrestrial environment when Intel discovered a significant rate of errors in DRAMS as their size was increased from 16K to 64K. Soon, the cause of SEU could be identified as alpha-particle contamination of the package materials. A new package plant in which ceramic packages for semiconductors were produced had recently been build at a river downstream of an abandoned uranium mine which contaminated the water used for the manufacturing process [4]. In the late 1970 the size of satelite memories had increased vastly and the error rate in satelites reached a level of one error per day. Further studies showed that errors were also caused by indirect ionization caused by protons and neutrons. In space there are many more of these particles which are emitted by the sun or trapped in the radiation belts of the earth, than heavy ions. This discovery revealed the great importance of the matter to future development of aerospace electronics and lead to more intense studies on the topic. In 1979 the first report on “single-event latchup”, SEL was released which pointed out the particles’ potential of destruction [5]. By the year 1980, methods to harden integrated circuits were researched and applied throughout the decade. A few studies lead in the 1980s indicated the problematic of logical errors occurring in system core logic. In the 1990s the situation changed drastically as the number of manufacturers which produced radiation hardenend circuits decreased and more commercial electronics were used in in aerospace systems due to their increased functionality. This change led to great challenges in keeping the systems reliable. Moreover, the decreased dimensions and the greater speed at which semiconductors were operated led to an increase in susceptibility to SEE, even at terrestrial level. The prediction

[1]Nuclear and Space Radiation Effects Conference

for the 21st century is a further increase in sensibility to SEU in memories and core logic thus making it an important matter for the entire integrated circuit industry.

2.2 Physical Contemplation

The physical complexity of the matter is very great and still being researched. However, the main mechanisms leading to the errornous behavior have been revealed in the past and shall be presented briefly in the following section.

2.2.1 Single-Event Effects

Single-Event Effects (SEEs) are being described as the generic term for effects which are triggered by the impact or the transition of a semiconductor by a particle of an ionizing radiation [6]. SEEs can be categorized in soft and hard errors. A soft error is device specific and occurs through a bit-flip or a false transient signal which could cause erroneous behavior of the circuit. Soft errors are temporary and dissapear after a reset of the affected system. A hard error may even physically destroy the device or affect it permanently. The following effects are subcategories of SEE [7]:

Single-Event Upset

The SEU is a soft error and represents the most common SEE. The SEU leads to a byte-flip which affects the value of a memory cell or alters the workflow of a digital circuit. It does not destroy the semiconductor element. It is usually a single-bit upset which just alters one bit but may in case of higher particle energy also result in a multiple-bit upset. MBUs represent the smaller fraction of SEUs but has effects on the error correction of memory architectures. A complete system failure caused by a SEU is called a *Single-Event Functional Interrupt* (SEFI). A SEU leading to an SEFI could be e.g. a bit-flip of a bit storing the system state which may lead to an undefined state and unpredictable behavior.

Single-Event Latchup

Another error which can be caused by an SEE is the indirect setup of parasitic bipolar transistors in between the CMOS well and substrate. This induced high-current latch-up condition may result in disrupture. This condition can only be removed, if the circuit is powered-off. If the current is not limited by external circuit resistances, a SEL may even lead to destruction.

Single-Event Transient

A SET describes the induction of a fault transient which temporarily alters the signal level and may cause erroneous behavior. SETs are considered soft errors and do not directly harm the circuit physically.

2.2.2 Radiation

Ionizing radiation may release charge in a semiconductor by two basic mechanisms: The direct ionization caused by ions or secondary ionization caused by smaller particles such as neutrons or protons. Both of the mentioned methods are capable of causing malfunction of semiconductors [8].

An ion that travels through matter looses its kinetic energy due to interaction with the electrons of the material. The kinetic energy of the ion determines the distance it travels through the material before being stopped (this distance is called its *range*). Moreover, it frees electron-hole pairs in its trail through the material. The *linear energy transfer* (LET) is used to describe the energy loss per path unit of a particle which travels through a material. The unlimited linear energy equals $L_\infty = \frac{dE}{dl}$ [9]. The LET curve is a function of the ion's mass and energy as well as the material's density and is usually presented in units of $MeV cm^2/mg$.

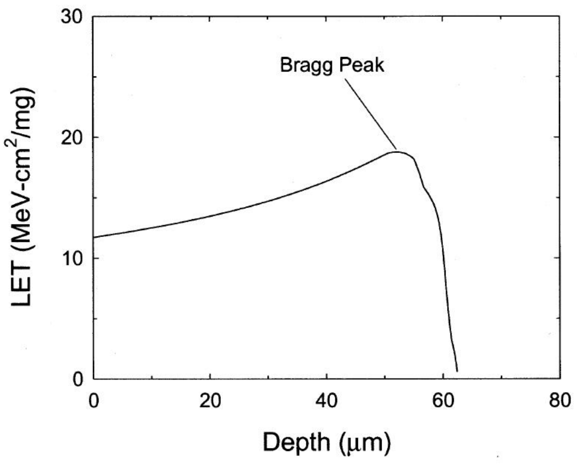


Figure 2.1: LET vs. depth curve for 210-MeV chlorine ions in silicon [8].

The LET-curve reaches its maximum as the ion slows down giving it more time to interact with the local electronic charge as shown in Figure 2.1. The higher the LET, the more energy is deposited in a given volume. The curves of the energetical interaction of different particles and materials differ highly. Moreover the curves also depend on the angle in which the particle traverses the material resulting in many possible scenarios regarding the ionization.

The occurrance of heavy ions like iron are limited to space environment and the terrestrial LET is usually smaller than $14 MeV cm^2/mg$. This LET is equal to the generation of a peak charge of $145 fC/\mu m$ and can easily disrupt most modern microenelcronic circuits which have a charge of 1-30fC stored in their nodes. Light particles usually do not produce enough charge to cause an SEU. However, these particles may cause upsets by secondary mechanisms. The radiation considered in the following paragraphs constitutes to the generation of SEE on terrestrial level.

Primary cosmic particles are considered to be left over from the Big Bang or Super Novaes (E $\gg$ 1 GeV) or emerge from solar events (E $<$ 1 GeV). Among those particles *high-energy cosmic ray protons and neutrons* cause a great part of all SEE and therefore present the primary cosmic source of soft errors in terrestrial environment. The occurrance of these particles is directly connected to the altitude and represents a much greater thread to electronics in aerospace environment than at terrestrial level (at an altitude of 10-20km the neutron flux is 100-350 times higher than at sea-level). The neutrons can interact with the material in two ways: An elastic interaction causes the affected nucleus to leave its lattice position in the material and thus causing the generated ion to produce electron-hole pairs in its trajectory. The inelastic interaction causes the absorption of the neutron leading to instability which causes the ejection of secondary ions from the nucleus [7].

As in the case of the contaminated packages which have been described previously, *alpha particles* present another cause of SEE on terrestrial level. Alpha particles (He^{2+} or ^2_4He)are composed of two protons and two neutrons identical to a helium nucleus and are highly ionizing (direct ionization). Due to their charge and mass, alpha particles are absorbed very easily and even the highest energy alpha particles have only a range smaller than $100\mu m$ in silicon [10] [7]. This is why the source of alpha radiation has to be very close to the semiconductor in order to produce SEEs.

2.2.3 Effects on Semiconductors

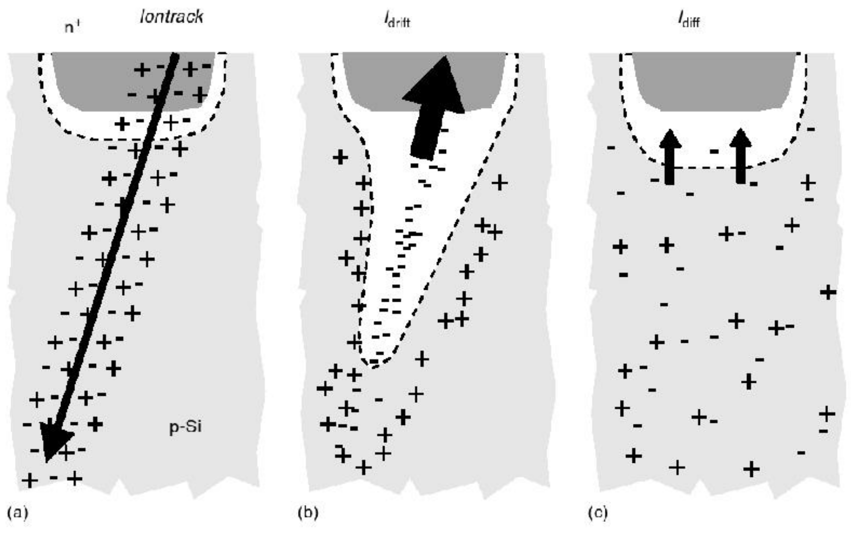


Figure 2.2: Charge generation and collection during ion strike in junction [7]

The most vulnerable structure to radiation effects is the reverse-biased junction. In the worst case, the junction is floating and a stored signal charge is being reduced by any charge injected through radiation. As electrons have a greater mobility compared to holes, the n^+/p junction is more sensitive to radiation events. Figure 2.2 shows the effects of an ion striking a reverse biased n^+/p

junction with a positive voltage connected to the n^+ node. Figure 2.2a shows the ion traversing the junction and leaving electron-hole pairs in its path. The carriers created by the strike are rapidly collected by the electric field and generate a large (current/voltage) transient in the node (figure 2.2b). This collection phase usually is completed within one nanosecond followed by a secondary phase of collection due to diffusion which is significantly slower (hundreds of nanoseconds) and less intense (as shown in figure 2.2c).

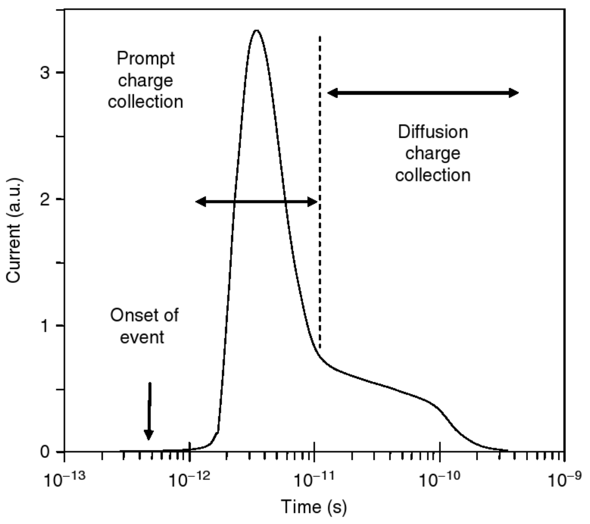
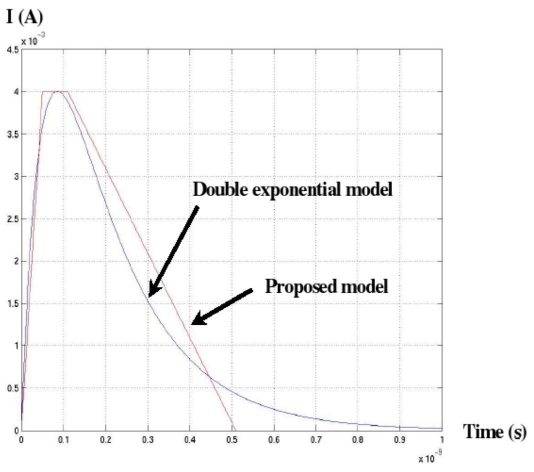


Figure 2.3: Current generated by ion strike [7]

The curve of the current resulting from an ion strike can bedividedd in two parts as shown in figure 2.3: The first peak indicates the first phase of electron drift-collection whereas the second gradient part results from the diffusion collection.



exponential

Figure 2.4: SET model as double exponential and "pulse" [11]

The impact on the circuit depends of its sensivity to the generated charge. The effect is difficult to model as the impact on the circuit depends on the pulse as well as the dynamic response of the circuit itself. The transient can be modeled as a double exponential injection current [12].

$$I(t) = \frac{Q}{\tau_1 - \tau_2}(e^{-t/\tau_1} - e^{-t/\tau_2})$$

τ_1 represents the collection time constant of the junction and τ_2 the constant for initially establishing the ion trajectory. The two constants depend on various factors related to the process and therefore technology generations. A double exponential model of the transient could lead to a very high computational complexity. Therefore the exacter double exponential model can be approached by a piecewise linear model. Figure 2.4 shows the two possibilities of modeling the transient.

Chapter 3

Basic Concept

At the moment all simulations have to be generated manually which consumes a huge amount of time. The goal of the project is to achieve a tool, which automatomizes the complete process of generating simulations, running the simulations employing common simulator software and saving the data in a format which allows further access to all relevant information. The primary goal of the project is to minimize the tedious manual effort giving more time to evaluate and study the vast amount of collected data. The tool shall provide an intuitive and user-friendly interface keeping the training period as small as possible. The tool is integrated smoothly into the employed tool chain and the format of all data currently used remains unchanged as shown in figure3.1.

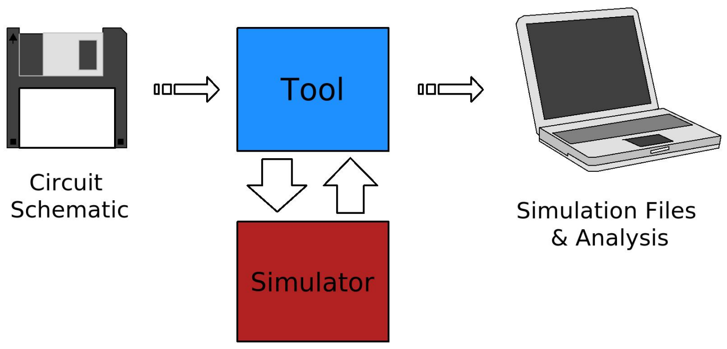


Figure 3.1: Basic concept

Moreover to tool shall be able to present the results in a adequate manner and present the most relevant information gathered. However, a more specific analysis of the data is not required as such programs are already available (the implementation of complex analysis algorithms would exceed the scope of this project).

Chapter 4

Realization

This chapter is divided in two parts to explain the project realization. The first section presents the ideas and mechanisms employed to interact with the simulation environment whereas the second part refers to the implementation and adaptation of the application.

4.1 Project Design

As the requirements of the project are not yet fully available and may change as more information of the subject itself becomes available a very exact elaboration of the conceptual design and project schedule are rather difficult. However, this challenge refers more to handling the simulation data e.g. automatic analysis than the automatic generation of the simulations.

4.1.1 The PSpice Simulator

One of the basic concepts of the project is using a common simulator to perform the simulation. Considering the possibilities of common simulators, the usage of the SPICE simulator seems the better choice for various reasons. Firstly, the circuit schematics, which shall be tested all exist in form of PSpice archives granting their usage without any complicated conversion. Moreover, PSpice takes commandline parameters which allow the simulation of a specific simulation file without interaction of a graphical user interface. Another advantage is that all files which are required to run a simulation exist in textbased format which makes it easier to edit and generate them.

PSpice control

A substantial concern of the project is the methodology of interacting with the simulator. An online research about internal interfaces and structures (e.g. the usage of dlls) of the Orcad PSpice simulator did not really lead to any results (which does not really surprise as PSpice is proprietary software). A general solution could be to control PSpice by sending windows messages such as keystrokes to the PSpice window. However, this solution is extremely depreciable due to its complexity and its high suceptability to errors. Fortunately, PSpice is

Parameter	Description
-bn	number of buffers
-bs	buffer size
-i	ini file name
-bf	flush interval
-@	command file name
-t	temp dir name
-c	command file name
-l	log file name
-p	goal function file
-e	exit program after simulating
-r	run/simulate the specified file
-o	output file generated
-d	name of the data file generated

Table 4.1: PSpice parameters [13]

able to take commandline parameters. Table 4.1 shows a summary of command line parameters accepted by PSpice.

The most relevant of those are the *-r* and the *-e* switches which allow running a specific simulation automatically and terminating the program when done. This presents a very secure and stable way to interact with PSpice. One disadvantage of this solution is that PSpice has to be loaded for every simulation which needs some additional time but can be disregarded as loading time is rather short compared to simulation time.

Simulation Input Files

Another very important subject concerns the way how PSpice handles the simulation resources. Some of the files (e.g. schematic or data files) include binary sections or are entirely binary which makes it much harder to deal with their content as dealing with text files. Fortunately PSpice also uses many text files as input.

Running a PSpice simulation basically requires three filetypes: A netlist (.net), circuit (.cir) and alias (.als) file. The following section contains a brief explanation of which information is stored in each one of these files.

The circuit file could be considered to be the "project" file of a specific simulation. All other files needed are referenced within the circuit file. The files containing circuit related data are included by using the `.INC` statement. Part model libraries are included with the `.lib` statement. Moreover, the circuit file contains the so called "*analysis directives*" specifying the kind of analysis and its timing parameters as well as which voltages and currents to analyze. The relevant analysis type for the simulations which have to be generated is the transient analysis. Its syntax is `.TRAN` [*start time*] [*stop time*] [*start after*] [*time step*]. The following figure shows an example of a cir file.

```
*Libraries:
.lib ``C: Tests modelo.lib''
.lib ``nom.lib''
Analysis directives:
.TRAN 0 3u 0 1ns
.PROBE
Netlist File:
.INC ''passbandfilter-PASA_BAJO.net``
Alias File:
.INC ''passbandfilter-PASA_BAJO.als``
.END
```

The netlist file is of special interest to the project, as it contains almost the complete circuit data such as its structure and the part parameters like resistances, capacities etc. PSpice determines how the parts of a circuit are connected by the utilization of nodes. A node is a point of reference connecting two or more elements. In other words nodes create the *links* between elements of a circuit. An entry in the netlist file always has the following form: [Partname] [connected nodes] [parameters]. If the parameters exceed the line, the following line begins with a `+` to indicate its affiliation to the previous entry. The number of nodes and parameters varies from part to part. Moreover, some of the parameters are optional and may be omitted. The following section shows an extraction of a netlist file:

```
* source PASSBANDFILTER
V_V12 N03938 0 2.5
V_V13 VIN 0 4.5
M_M5 N00013 N00013 N00084 N00084 PMOS
+ L=1.5u
+ W=20u
M_M4 N02156 N00013 N00084 N00084 PMOS
+ L=1.5u
+ W=20u
M_M3 N00016 N00184 0 0 NMOS
+ L=1.5u
+ W=4.5u
```

As shown in the previous netlist entries, each element begins with a character specifying its part type (e.g. M indicates a MOSFET transistor or V a voltage source) followed by a syntax containing its name (usually set automatically using the part character and a consecutive number).

The alias file contains a list of the alias names used in connection with the part identifier. These aliases are also used by the probe program to rely to nodes.

Simulation Output Files

Processing the results of the simulation requires a better understanding of the files generated by PSpice. The following files generated by PSpice upon simulating a circuit contain the complete data gathered.

The output file has the extension *.out* and is the log file created by PSpice. It contains detailed information about the included files and their processing as well as the generation of simulation results. Any errors will be logged in this file with a description. The output file is of little interest to the project as it does not contain the generated simulation data.

The probe file is an almost completely binary file which has the extension *.dat* and contains the complete gathered simulation data. Unfortunately, the format is proprietary and therefore an online search on its internal structure did not lead to any results. The structure of the probe file will be discussed in the implementation section of this chapter.

The CSDF file PSpice offers the option to store simulation data in a text file (*.csd*) instead of the binary probe file. The **C**ommon **S**imulation **D**ata **F**ormat represents a standard format for data gathered by simulation programs. The disadvantage of storing the data in text format lies in the filesize. Storing a 64bit double value as text occupies much more bytes (the actual number depends on the formatting of the string) than the direct 8byte storage.

4.1.2 Transient Fault Injection

One of the main tasks of the desired application is the automatic injection of transient faults. Such a injection is realized by connecting a current source to the relevant nodes. Thus a separate circuit has to generated for every single fault injection.

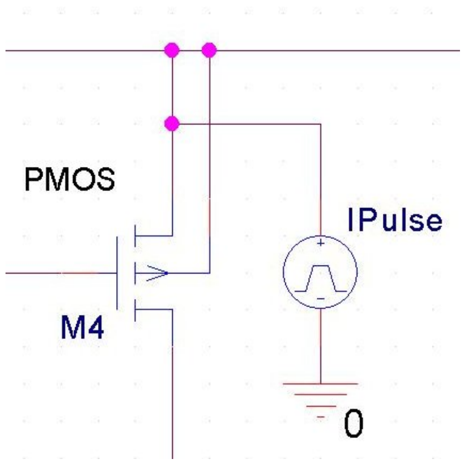


Figure 4.1: Insertion of current source in schematic

The injection of such a transient fault can be achieved by inserting a current source. As shown in figure 4.1 the source is connected to a specific node and ground.

Pulse Model

PSpice provides various power sources. A compromise between model precision and computational complexity is the utilization of a trapezoid shaped current pulse (as shown in figure 2.4. The netlist syntax for such a pulse source is:

```
I_IPULSE N00019 0 DC 0 AC 0
+PULSE 0 500u 0 250p 250p 5n 1
```

The syntax of the first line has the structure *Source name, 1st node, 2nd node, DC value (0 DC), AC value (0 AC)* the second line specifies the pulse using the syntax `+PULSE` *1st current value, 2nd current value, delay time, rise time, fall time, pulse width, period.* The pulse parameters have to be altered as the SEE causes different timing and current parameters. A graph of a basic pulse model used with the parameters shown above can be found in the appendix (figure A.1).

Injection

The injection of the sources can be accomplished by adding the pulse syntax described above to the netlist. The first node parameter allows to specify at which point to connect the source. The position within the netlist file does not matter.

4.2 Implementation

The complete project is realized in C++ using Microsoft Visual Studio 6.0. Moreover, the application uses the MFC (Microsoft Foundation Classes) framework (statically linked). The utilization of objects makes it easier to manage the project structure and interfaces.

4.2.1 Application Interface Type

SDI (single document interface) is the most suitable interface type for the tool and provides a structure which can be adapted to fit the project with reasonable effort. Taking into account the type of information which has to be displayed and the complexity of the application would be inappropriate for a dialog based application. A MDI (multiple document interface) application would be to complex for the application's requirements.

4.2.2 Object-Oriented Structure

As mentioned above, the methodology of object-oriented programming is utilized to structure the program, make it more manageable and prevent redundancy. Figure 4.2 shows the overview of the class diagram presenting the basic relations between the classes. To support a clear structure of the project the classes are implemented in separate source and header files. The (hierarchically seen) most significant class concerning the management of simulation data is the CSpiceSimulation class. Figure 4.2 shows only those classes which are directly connected to the model of simulation data and its presentation. Other classes such as those used to store or read simulation data are not included in the diagram.

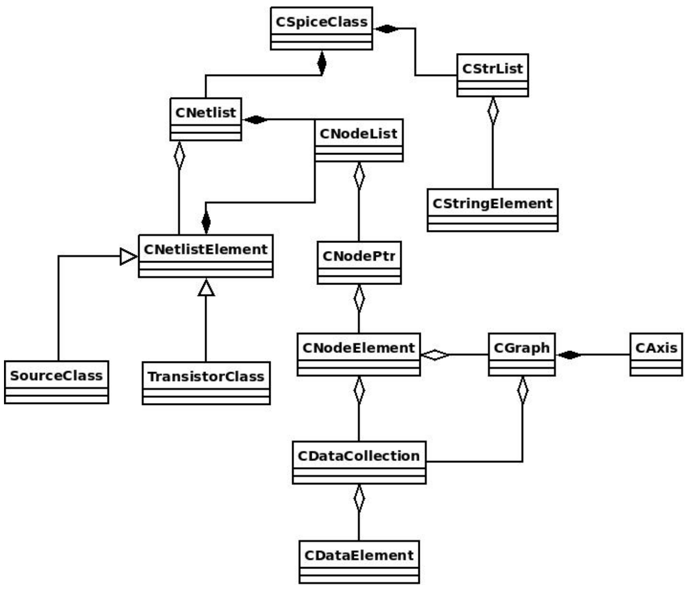


Figure 4.2: Main simulation model classes

4.2.3 Internal Circuit Model

A very important challenge of the project is how to handle to structure of the simulation. One mayor requirement is the flexibility of the model in terms of its size. Netlists may vary in size and contain many different part types therefor it is a good idea to follow the hierarchical structure of the netlist format.

Figure 4.3 shows the basic concept of the data model. The CSpiceSimulation class has access to virtually all information related with the simulation model that is relevant for the tasks of the program. Chained lists are utilized throughout the project to introduce more flexibility and prevent excessive usage of system resources. The CNetlist class implements such a chained list by managing all of the CNetlistElement objects. Each entry (e.g. resistor, transistor, source etc.) of a netlist file is mapped to such an object. Furthermore, each netlist element possesses a CNodeList object which implements another chained list which manages the nodes connected to the part. All of the classes provide interfaces to access the private content (such as get() and set() methods). The list classes also provide methods to manage their contents such as add, delete, insert or get items at a certain position and clear all of its contents. All elements of the lists are created on the heap using the `new` statement.

As shown in figure 4.3, the structure of the node list differs from the first one. The reason is related to the prevention of redundancy and the consistency of the model. As nodes may be connected to several parts, the CNodePtr class has been introduced to serve as a "connector" which allows the usage of unique node elements in various node lists. Moreover, all nodes are first stored in a master node list. The node lists of the netlist elements then point to the nodes stored

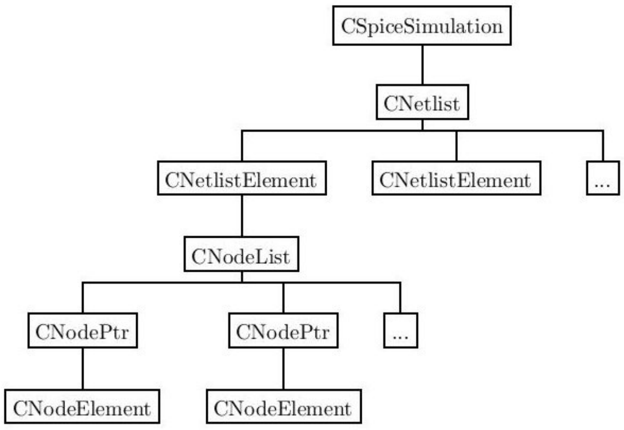


Figure 4.3: Model data structure

in the master node list. Another advantage is the easier removal of all node elements upon clearing an entire simulation model. Data such as the filenames, the timing parameters and data buffers are stored as private CStrings in this class. Moreover the class has an CNetlist element to store the circuit data and a CStrList element containing the test list.

4.2.4 Reading Circuit Data

The first task which has to be accomplished by the application is to read the circuit model file. The CSpiceSimulation provides a function called `readCirFile(...)` which takes a cir filename as parameter. This function retrieves the name of the alias and netlist file from the cir file. Moreover, it reads the transient section to get the start and stop time as well as the timestep and the time at which the simulator shall start to save data. After completing the previous tasks, it loads the alias data by calling the `loadAliasData()` member function which stores the alias file contents in a buffer. Finally, it calls the `loadFromFile(...)` function of its CNetlist object passing the name of the netlist as parameter. This function opens the netlist file, retrieves all parts of the circuit and passes them to the `addElement(...)` function.

Netlist Elements

At this point, the application has to detect the part type, as transistors and sources are of special interest. All other parts can be treated as default netlist element without any further processing. Those entries are stored without any further processing. However, all elements despite of what class they are have to be stored in one netlist. To solve this challenge, the methodology of polymorphism is applied as shown in figure 4.4.

The two derived classes implement the functionality to interpret the parameters of the specific netlist entry in their constructors. The `addElement(...)` function reads the first character of the netlist entry and then creates an ob-

Figure 4.4: Netlist elements

ject of the adequate class passing the netlist string to the constructor. The list itself treats the objects of the different classes alike. The classes follow the requirement to implement the same interface.

The CNetlistElement constructor offers a very generic methodology to handle netlist entries. It initializes the member variables and reads the part name of the entry. The item data is simply stored as a CString. This enables the class to handle all types of netlist entries but limits the capability of accessing specific parameters. The enum type used internally to describe the part type is "`Unknown`".

The SourceClass constructor implements specific procedure to read source parameters. Moreover it is able to identify the nodes to which the source is connected and has its own node list. The enum type assigned to the objects of this class is "`Source`".

The TransistorClass constructor offers the most significant and specialized functionality concerning the interpretation of netlist entries. It distinguishes between bipolar and mosfet transistors and identifies all of the nodes. Moreover, it is able to specify which node connects to which pin. Thy enum type internally assigned is "`Transistor`". As mosfets are of special interest to the project, the drain, gate or source pin switches of the nodes connected to a mosfet are set. The structure of a mosfet part in the netlist is:
M_[*part name*] [*drain node*] [*gate node*] [*source node*] ...

Up to this level, all of the netlist elements are created. However, the more specific elements presented above use the internal node lists. Handling nodes is a more complex task as one node may be connected to various circuit parts. To prevent multiple objects of the same node, a master node list which is a member of the CNetlist class is utilized. Nodes are added to the internal lists of the source or transistor elements by using the following two lines:

```
CNodeElement * element = netlistPtr->addMasterNode(s_node);
result = nodeList.addNode(element);
```

The variable netlistPtr is a pointer to the netlist containing the element. All of the CNodeElement objects are created by the `addMasterNode(CString s_node)` function of the CNetlist class. Before creating a new node, this function makes sure, that the node has not been created previously. If the node has not been created so far, the function creates a new CNodeElement, adds it to the master list and returns a pointer to the element. It the node has already been created, it simply returns a pointer to the corresponding CNodeElement stored in the master list. In any event, the function will return a pointer to a CNodeElement presenting the specified node. The next step is to add the node to the local CNodeList object which is accomplished by calling its `addNode(CNodeElement element)` function. The CNodeList objects manage lists of CNodePtr objects which serve as connectors to the real CNodeElement objects.

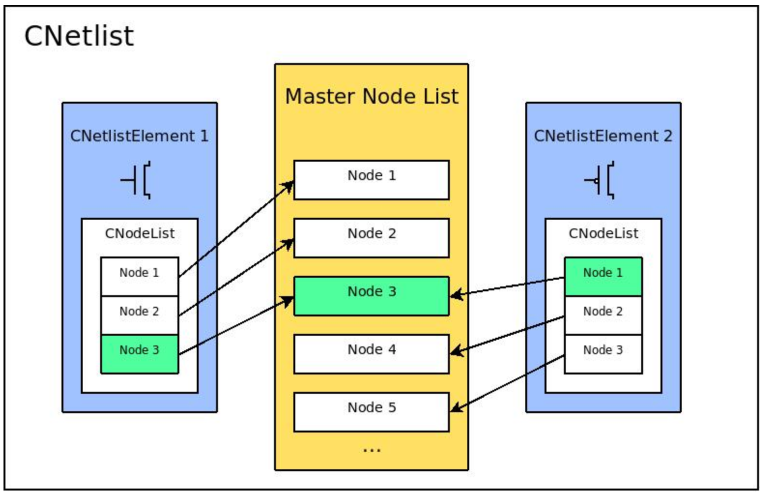


Figure 4.5: Master Node List

A simplified relation between the node lists of the netlist elements and the master node list is shown in figure 4.5. Node 3 of the master node list is a member of both of the node lists belonging to the elements. The function `addNode(...)` of the CNodeList class is used by the master node list to actually create the new CNodeElement objects. As there are two possible ways to add a node to a CNodeList, the concept of method overloading is used at this point. The first implementation of the function takes a `CString` as parameter and creates a new CNodePtr and CNodeElement object on the heap if a node with the same name is not already in the list (in this case the function returns 0). Then it connects the CNodeElement object to the CNodePtr (through the `setNode(CNodeElement)` function of the CNodePtr class) and adds the CNodePtr object to its list. The second implementation takes a `CNodeElement *` as argument. After ensuring, that this object is not yet in the list, it creates a CNodePtr object and links the already existing CNodeElement as explained above.

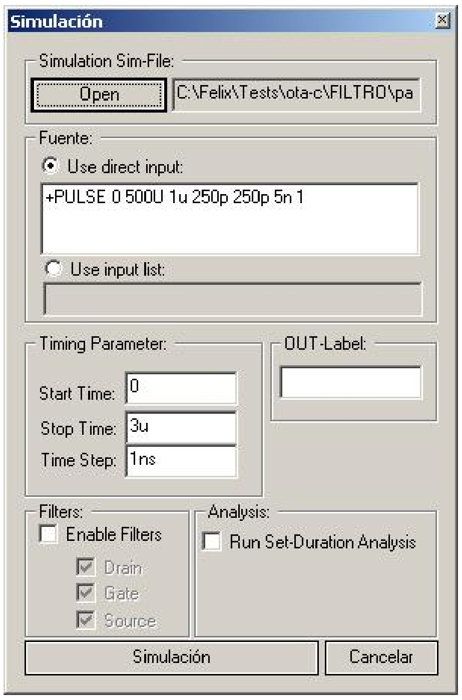
As it does not make any sense to connect a transient fault source to the ground node, the constructor of the CNodeElement automatically detects if the node is the ground node and sets the isGnd flag.

This concludes the first task of the application of loading PSpice data from simulation files and their storage in an internal model for further processing.

4.2.5 Simulation Settings

After loading all available data, the next step is to specify the simulation parameters. Figure 4.6 shows the simulation dialog which is displayed when starting a new simulation. The first step the user has take is to select a cir file by clicking the *Open* button. This action causes the application to read all necessary

PSpice files affiliated with the selected circuit and construct the internal model as described in the previous section.



Simulación

Simulation Sim-File: C:\Felix\Tests\ota-c\FILTRO\pa

Fuente:
● Use direct input: +PULSE 0 500U 1u 250p 250p 5n 1
○ Use input list:

Timing Parameter:
Start Time: 0
Stop Time: 3u
Time Step: 1ns

OUT-Label:

Filters:
☐ Enable Filters
☑ Drain
☑ Gate
☑ Source

Analysis:
☐ Run Set-Duration Analysis

Figure 4.6: Simulation Settings

The dialog allows the user to change the source to be injected as well as its parameters. Alternatively the program can read a source list file to run several simulations with different source parameters. Each line in the source file is expected to have a `name` and a `source` parameter as shown in the following example:

```
name="falla+500uA" source="+PULSE 0 500U 1u 250p 250p 5n 1"
name="falla-500uA" source="+PULSE 0 -500U 1u 250p 250p 5n 1"
name="falla+250uA" source="+PULSE 0 250U 1u 250p 250p 5n 1"
name="falla-250uA" source="+PULSE 0 -250U 1u 250p 250p 5n 1"
```

Using such input files makes it much easier to generate a large amount of simulations using different transient faults. The different simulations are stored in folders containing the name of the source to organize the collected data. Moreover, the timing parameters read from the cir file can be edited in the dialog window using common PSpice values.

A very important option is the usage of filters: The user can specify explicitly at which pins of the transistors the transient faults will be injected. Usually the only pins at which the sources have to be connected are the drain pins. Connecting the source to all of the pins would unnecessarily increase simulation time and the data volume. The analysis options are used to automatically perform data analysis after launching the simulations and will be explained later.

4.2.6 Generating Simulation Files

The process of creating all of the archives necessary to run the transient fault simulations is started after the user has completed the simulation setup. At first, all of the setup options are updated using the CSpiceSimulation interface. Then the function `createTests()` of the CSpiceSimulation class is called to create the simulation files. This function creates a new directory in which all of the generated files are stored and whose unique name contains the simulation name (if available) and a timestamp (of the format YYYY-MM-DD_HH-MM-SS). In the next step, the function calls the `generateNodeList()` function of its CNetlist object which returns a complete list of all the transistor nodes at which the source has to be injected (respecting the filter rules) and finally, generates an writes a cir, net and als file for each injection numbered subsequently. This is also the point at which the source element is added to the netlist file. Another important action performed upon the generation of the simulation files is setting the path string variable *m_testFile* of each of the selected nodes.

4.2.7 Running the Simulations

As the execution of all simulations may be a time consuming process, the dialog window shown in figure 4.8 is displayed to show the simulation progress. Moreover, a spinning sign and a progress bar are displayed in the dialog window to indicate that the simulation is running. The realization of the two simultaneous functions requires the utilization of tasks. Those tasks are started immediately after creating the dialog in the `OnInitDialog()` function by using the `AfxBeginThread(...)` procedure. The first task is implemented in the function `SimulationThread(...)` and takes a pointer to the dialog as LPVOID argument. Figure 4.7 shows the flowchart of the thread. After setting the display status the thread checks whether a copy of PSpice is already running. In this case the thread is aborted with an error message which asks the user to shut down all running instances of PSpice. This is a safety measure to prevent malfunction. In the next step the previously generated testlist is fetched from the CSpiceSimulation which contains all of the cir files which shall be simulated using PSpice. This is done through the following syntax:

```
s_cirFile = testlist.getAt(i)->getString();
sprintf(CommandLine,
   ‘‘[PSpice path]pspice.exe -r -e \ ‘‘%s\’’’’, s_cirFile);

::CreateProcess(NULL, CommandLine, NULL, NULL, FALSE,
    HIGH_PRIORITY_CLASS, NULL, NULL, &si, &pi);
::WaitForSingleObject( pi.hProcess, INFINITE);

CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
```

In order to pass the cir file which shall be simulated, the filename is retrieved from the test list. Then the command string is assembled using the `sprintf` function and used to invoke the PSpice process through the `CreateProcess(...)` function. The `CloseHandle(...)` function which is called at last waits until the created process is terminated (which is done automatically after completion of the simulation) and releases the handle. After completing each one of the simulations the progress information is updated. To manage the access to the same (dialog) resource, a CMutex member variable of the progress dialog is used. After all simulations are completed, the threads are terminated and the dialog closed.

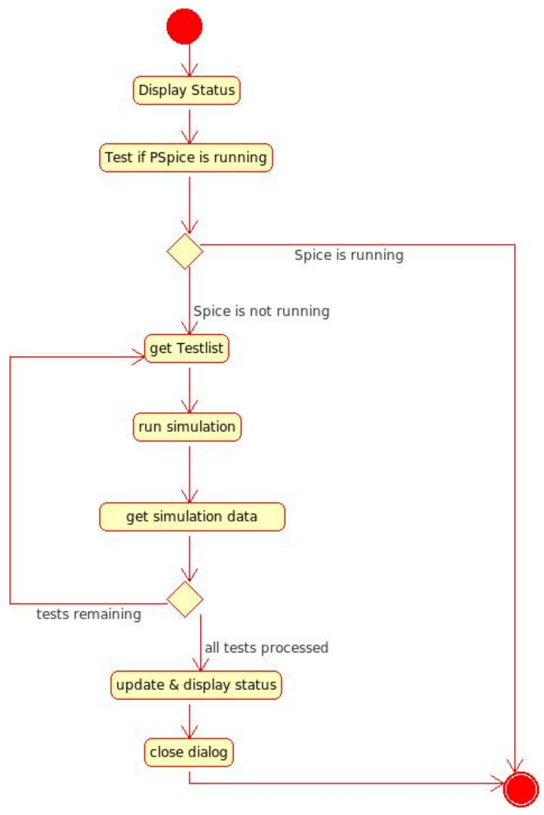
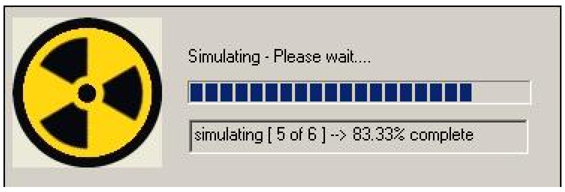


Figure 4.7: Simulation Thread

The second task is implemented in the `StatusThread(...)` function. It displays the spinning icon shown in figure 4.8. Due to the icon's symmetry the usage of a series of four bitmaps of which each one is rotated 30 in relation to the previous bitmap is adequate to create the impression of a smooth animation. All of the bitmaps are stored as project resource and are loaded as CBitmaps using the `LoadBitmap(...)` function which takes the resource ID as parameter.


Simulating - Please wait....

simulating [5 of 6] --> 83.33% complete

Figure 4.8: Progress Dialog

Figure 4.9 shows the flowchart of the status thread. At first the thread loads the four images. Then it enters a while loop which checks whether the simulation is still running, displays the next bitmap, updates the dialog and sleeps for 100ms.

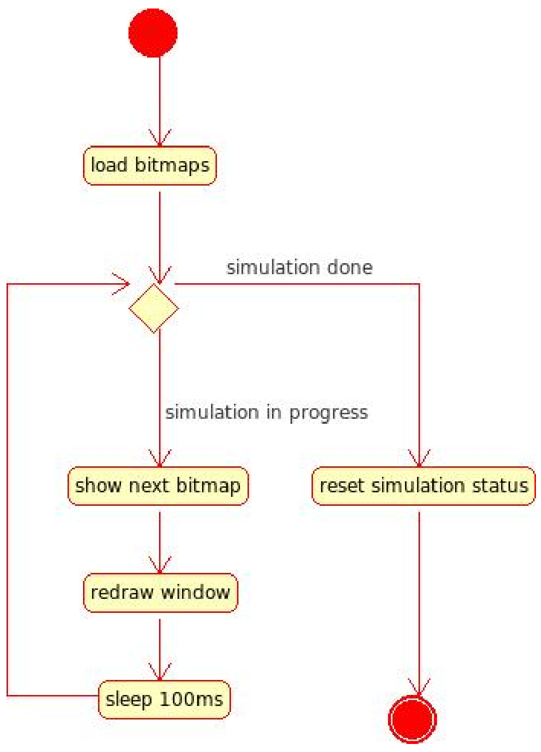


Figure 4.9: Status Thread

PSpice automatically stores the generated data files in the direcory which contains the input files for the simulation. Moreover, the data files have the same name as the corresponding cir file followed by the *.dat* extension.

4.2.8 Simulation Data

As described in chapter 4.1.1 PSpice is able to store the simulation data in text and binary format. The great disadvantage of the text files is the inefficient usage of memory which leads to huge file sizes (depending on the simulation settings) compared to the binary files. PSpice also offers a second possibility to export simulation data in text format. This is especially useful when the exported data shall be copied to Excel sheets. Moreover it allows the data extraction of a specific graph. This method works quite well for smaller simulations (in terms of data volume). Atomizing the procedure of opening a probe file, selecting the desired variable, copying and finally pasting the data to a text file can be achieved by using a macro (*.cmd*) file using the commands presented in figure 4.10.

```
File Open
(dat filename)
OK
Trace Add
(trace name e.g.
V(TRANS1))
OK
Trace Select
Plot 1
Y_Axis 1
V(TRANS1)
Edit Copy
File Close
(dat filename)
OK
```

Figure 4.10: PSpice macro file

If such a file is passed to PSpice as a command line parameter, PSpice automatically executes all of the instructions contained. After running this "macro", the application simply creates a new text file and uses the function presented in listing 4.1 to paste the data stored in the clip-board to a new file. The HWND parameter passed to the function has to be a CWnd handle to the application window itself.

```
int ClipboardToFile(CString s_file, HWND handle) {
    char * buffer;
    if(OpenClipboard(handle))
    {
        CFile dataFile;
        CString s_data;
        buffer = (char*)GetClipboardData(CF_TEXT);
        dataFile.Open(s_file, CFile::modeCreate
                | CFile::modeWrite, NULL);
        s_data = buffer;
        dataFile.Write(s_data, s_data.GetLength());
        dataFile.Close();
        CloseClipboard();
        return 1;
    } else {
    return 0;
    }
}
```

Listing 4.1: Accessing the clipboard

Although this solution works it is slow and has the same disadvantage as the using a CSDF file concerning filesize as the data is stored in text format. The most efficient solution is to access the PSpice probe files.

The Probe Format

The utilization of probe files is the most challenging approach due to the lack of information about its file structure. An initial online investigation as well as a search in the PSpice help did not reveal any information about how data is stored in the probe file. A first glance at the file using a common text editor revealed its semi-binary nature: The file is divided in a block containing structural information in text format and a second significantly larger block of binary information which contains the timing and trace data.

```
I$i=0x00000000H=0x0000008Bh=0x00000000N=0x000001B6n
=0x00000000S=0x00000351s=0x00000000A=0x000003AAD=0x00000000M=0
x00000000H+AllCol=1AnaCols=22Analysis=TransientAnalysisAnaRows
=0x0000030FCirName=** circuit file for profile:biasCirSub=
Complex=1DataType=0Date=10/08/07DigCols=0DigRows=0x00000000
EndSweep=0.2Param1=TimeParam2=ProgID=61413RevNo=9.0StartSweep=
5e-009SweepMode=4Temp=27Time=15:45:27NTime V(V2:+);V(R1:1);
V(N00750);V1(R1);V1(V2) V(Q3:b);V(R1:2);V(N00179);VB(Q3);
V2(R1) V(TRANS1);V(Q3:c);V(R2:1);V(R4:1);VC(Q3);V1(R2);
V1(R4) V(R3:2);V(R2:2);V(V1:+);V(N00478);V2(R2);V2(R3);
V1(V1) V(vout);V(R3:1);V(Q4:c);VC(Q4);V1(R3)V(R4:2);V(Q4:b);
V(N01977);VB(Q4);V2(R4) I(R1) I(R2) I(R3) I(R4) I(V1) I(V2)
IC(Q3) IB(Q3) IE(Q3) IS(Q3) IC(Q4) IB(Q4) IE(Q4) IS(Q4)
V(Q3:e);V(V2:-);V(V1:-);V(Q4:e);V(0)
```

Figure 4.11: Text block of a probe file

A closer look at the data shown in figure 4.2.8 reveals much information about the simulation. The first parameter which is of great importance is called `AnaCols`. Comparing different probe files showed that its value stands for the number of traces plus one. As the parameter name indicates the usage of "columns" the increase by one probably arises from the addition of the time column. A look at the last part of the block which lists all of the trace names may be confusing as their number exceeds the number of traces suggested by the `AnaCols` parameter by far. However, many of those trace names are aliases. All of the trace parameters are separated by a character. Each new "real" trace is separated from the last trace tag by a `0x00` byte. The aliases of a trace are separated by a semicolon. Counting the "real" traces shows that their number corresponds to `AnaCols` - 1. This information is substantial to interpreting and reading the .dat file.

The structure of the second part of the probe file is far more complicated due to its binary format. The precision of the timing values produced by PSpice indicates *double* values. Searching a certain time value extracted from a probe file employing the copy method previously described shows, that the time values are indeed stored as double or 64Bit values (this was accomplished by using the program *KHexEditor*). However, the first time byte is located at position 10 of the binary part. The previous bytes form a header of a complete frame including a time and a data block. The block of the "time"-bytes always has a length $10 * 8bytes = 80bytes$. Unlike the time values, the data is stored as *32bit float* values. The data is arranged in packages of 10 subsequent values for each trace. The position of the values within this $10 * 4bytes = 40bytes$ block corresponds to the time value at the same position within the time block. Like the time block, the data block has a 9 byte header. Comparing different probe files leads to the conclusion, that the headers contain check sums indicating the size of the blocks. Table 4.2 shows the formulas employed to calculate the header sums.

Frame	*Equation*
Time	$total - framesum = 98 + (cols * 40)$
	$partial - framesum = 88$
Data	$total - framesum = 98 + (cols * 40)$
	$partial - framesum = 8 + (cols * 40)$

Table 4.2: Calculation of frame headers

The header of the time frames has the structure `0x53 [4 bytes` ***`total-frame sum`***`] [4 bytes` ***`partial-frame sum`***`]`. The structure of the inner data frame is `0x41 [4 bytes` ***`total-frame sum`***`] [4 bytes` ***`partial-frame sum`***`]`.

Offset	*Data*
0000000496	00 **53 8a 00 00 00 58 00 00 00** f1 68 e3 88 b5 f8
0000000512	e4 3e f1 68 e3 88 b5 f8 f4 3e b8 7a 4e 7a df f8
0000000528	04 3f 9b 03 04 73 f4 f8 14 3f 0c c8 5e ef fe f8
0000000544	24 3f 45 2a 8c 2d 04 f9 34 3f 62 db a2 cc 06 f9
0000000560	44 3f be cf 0d d3 4d 62 50 3f 31 38 9f c0 ca 77
0000000576	50 3f 16 09 c2 9b c4 a2 50 3f

Table 4.3: Binary time block

An exemplary time block is shown in table 4.2.8. The highlighted bytes represent a time frame header followed by the 80byte block containing 10 time values. The complete functionality of reading a probe file is implemented in the CDatReader class which is able to assemble a CDataCollection object and store all of the trace and the alias names in dynamic arrays handling all of the necessary information stored in probe files.

4.2.9 Simulation Format

Although the simulation data is stored in the probe files generated by PSpice it may be necessary to open the simulation at another time using the application's analysis functions or graphical interface. Furthermore, it may be necessary to revise the settings of the simulation to detect errors. The second task can be accomplished by reading the input files (netlist etc.) using a text editor but is inconvenient and may be difficult for someone who does not have knowledge of the syntax used in those files.

To solve these problems, the internal model of the simulation has to be saved in a file so that it can be restored to the same state as after running the simulation. Regarding the second requirement described above, the syntax and structure has to be simple and organized in a way which permits its editing using a text editor. The choice of the syntax is XML as it meets all of the requirements. Listing 4.2 shows the initial part of a XML file generated by the application. The `<Source>...</Source>` section contains all of the input files used to generate the simulation. The simulation parameters such as the timing parameters and the parameters of the injected source are stored in the `<Parameter>...</Parameter>` section.

```
<?xml version="1.0" encoding="UTF-8"?>
<Simulation name="falla+500uA">
    <Sources>
        <Alias filename="C:\passbandfilter.als"/>
        <Cir filename="C:\passbandfilter.sim.cir"/>
        <Netlist filename="C:\passbandfilter.net"/>
    </Sources>
    <Parameter>
        <Start time="0"/>
        <Stop time="3u"/>
        <Timestep time="1ns"/>
        <Source param="+PULSE 0 500U 200n 250p 250p 5n 1"/>
    </Parameter>
```

Listing 4.2: Simulation settings in xml-format

The second part of the xml-file contains the model data used internally. The easiest way to map the model to the xml-format is following its hierarchical structure. The `<Netlist>...</Netlist>` section includes all of the model elements. Each CNetlistElement object is mapped as an `<NetlistElement>` entry which has a `<Raw..\>` element and a nodelist stored in a `<Nodes>...</Nodes>` section. All of the nodes connected to an element are listed in this section of `<Node>..</Node>` elements. Listing 4.3 shows the structure of the second part of the xml-file:

```
<Netlist>
    <NetlistElement name="M_M5" type="Transistor">
        <Raw data="M_M5   N00013 N00013 [.....]"/>
        <Nodes>
            <Node name="N00013">
                <Test filename="[...]cir_node_N00013.dat"/>
            </Node>
            <Node name="N00084">
                <Test filename=""/>
            </Node>
        </Nodes>
    </NetlistElement>
    .
    .
    .
</Netlist>
```

Listing 4.3: Model in xml-format

The functions to write and read the xml-files are implemented in the class called CXMLreader. Due to the hierarchical xml structure, the model can be easily read from a file using a state machine. The source code of the functions used to read and write such a file are listed in the appendix.

4.2.10 Displaying Results

As the simulation results are stored in various probe files, previewing the simulation results manually may be a time-consuming task. Therefore the implementation of a (simple) graphical interface which allows the user to preview the traces has been considered part of this project. Two fields are necessary to display the results: The first one to show the circuit structure and enable the user to select a specific node and the second one to show the graph of the traces stored in the probe file connected to the node. Therefore the main (dialog style) window of the application is "split" into two parts. The left part appears as a bar containing a CTreeDlg item to display the circuit structure and the right part presented by a CStatic item. In order to arrange those items properly especially after and during resizing the window, the `OnPaint` handler of the window has been altered to automatically fit those elements to the window size.

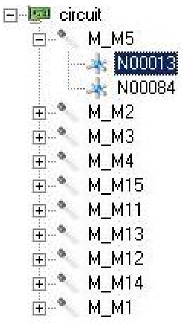


Figure 4.12: Circuit tree

The circuit tree is generated directly after terminating the PSpice simulation. Figure 4.12 shows the circuit tree of a completed simulation containing all circuit elements and nodes. A left-click on one of the nodes triggers a event updating a second list under the tree which lists all of the traces available in the node's probe file. Selecting one of the traces causes the program to display the trace on the right side of the application window.

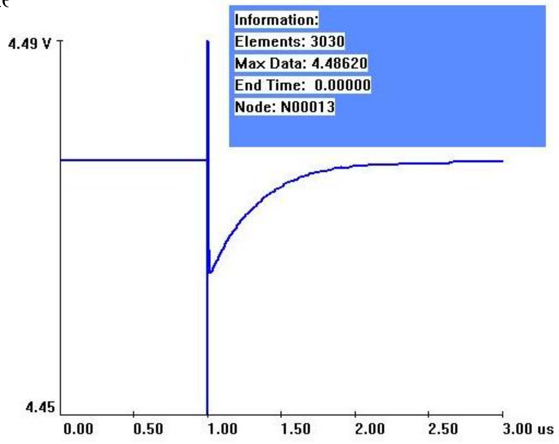

Information:
Elements: 3030
Max Data: 4.48620
End Time: 0.00000
Node: N00013

Figure 4.13: Trace graph

The whole process of handling the graphical display of a trace is performend by the CGraph and the CAxis class. The application automatically creates a CGraph object and "connects" a handle of the CStatic item on the right side of the window. Hence, the print functions of CGraph use the device context of the CStatic object to perform its graphical output. Selecting a trace in the lower trace list also "connects" the CDataCollection object containing the trace data to the CGraph object by passing its pointer. The first step performed to display the data is an internal call of the `Autoscale()` function which initializes two CAxis objects and scales them using the `scale()` function of the CAxis class. Moreover it sets the spacing of the labels, the data range, origin and data scaling. Then it draws the two axis and the data points. Those points are connected by lines in case that the number of pixels of the x-axis exceeds the number of data samples available (which should never be the case but might happen as PSpice stops collecting data in case of malfunction). Furthermore, a small box containing some basic information as the node name and the maximum and minimum value of the trace is displayed in the upper right corner of the graph. The autoscale function is capable to detect the size of the device context and automatically scale the graph in relation to the window size.

4.2.11 Data Analysis

A very interesting task of the application is the automatic analysis of data. The functions to perform analysis are stored in the file *AnalysisFunctions.cpp*. So far only one type of analysis has been implemented: The *Set-Duration Analysis*. However, as other types become relevant they can be easily added to this file. There are two ways to carry out a *Set-Duration Analysis*. The first is to enable the corresponding check-box in the "simulation window" (used to set the parameters of the simulation). This will run the analysis for all of the relevant nodes a :un the analysi

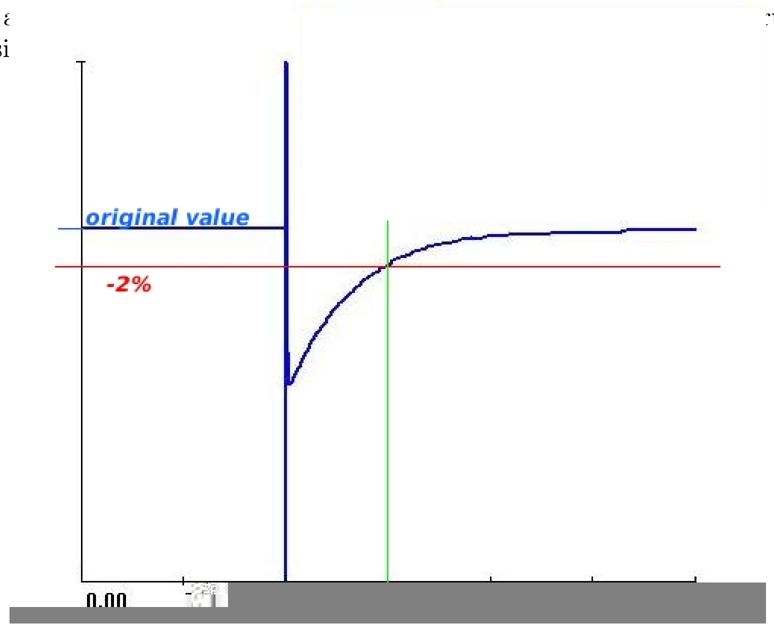
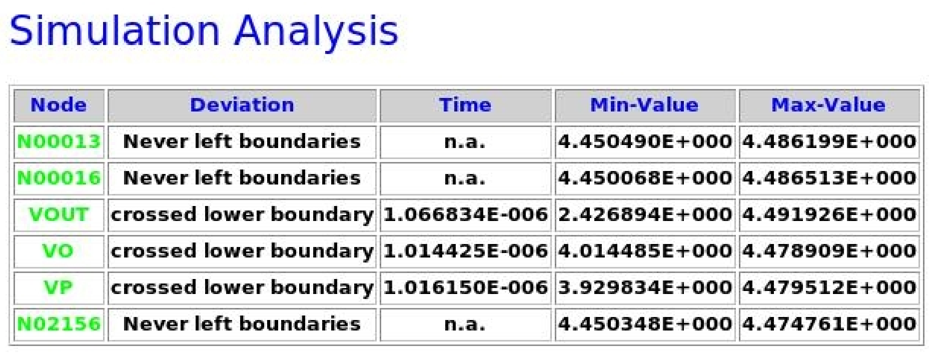


Figure 4.14: Set-Duration Analysis

The implemented analysis detects the last time at which the trace of the output voltage enters the boundaries at ±2% of the original voltage. Figure 4.14 shows the principle of analysis. The last time the trace passes (here: the lower) boundary is recorded. This method makes it possible to calculate the duration of the transient fault. The results of the analysis are stored in html-format. This helps to present the data well arranged and allows to copy the tables containing to Excel without loosing their format.


Simulation Analysis

Node	Deviation	Time	Min-Value	Max-Value
N00013	Never left boundaries	n.a.	4.450490E+000	4.486199E+000
N00016	Never left boundaries	n.a.	4.450068E+000	4.486513E+000
VOUT	crossed lower boundary	1.066834E-006	2.426894E+000	4.491926E+000
VO	crossed lower boundary	1.014425E-006	4.014485E+000	4.478909E+000
VP	crossed lower boundary	1.016150E-006	3.929834E+000	4.479512E+000
N02156	Never left boundaries	n.a.	4.450348E+000	4.474761E+000

Figure 4.15: Analysis html-file

The output file shown in figure 4.15 also provides information wether the trace has last crossed the upper or lower boundary, has never left, or has never returned to the area between the boundaries. Moreover, it lists the minimal and maximal value of each trace.

This concludes the part of the implementation. Further information such as source code or UML diagrams can be found in the appendix.

Chapter 5

Discussion of results and outlook

Currently, the application is fully capable of automizing the injection and simulation process. A further improvement may consist of adding special analysis functions to adapt the program to meet more specific research requirements. The data gathered up to this point implies that the effects of transient faults on output-signals are more severe than previously anticipated. The program may provide considerable help to detect the most susceptible nodes of a circuit which have to be hardened to transient faults. Currently, further circuits are tested using the program and search patterns which can be used to implement automatic analysis are developed. The schematics of the circuits which are currently tested are shown in the appendix section A.The group plans to release the first results of the research at one of the upcoming international congresses.

The group also leads further investigations concerning the matter such as modeling the physical events occurring during the SEEs or the thermodynamical and mechanical behavior of the semiconductor during a SEE.

Bibliography

[1] http://de.wikipedia.org/wiki/single_event_effect.

[2] J.T. Wallmark and S.M. Marcus. Minimum size and maximum packing density of nonredundant semiconductor devices. *Proc. IRE*, vol 50:pp. 286–298, 1962.

[3] D. Binder, E.C. Smith, and A.B. Holman. Satelite anomalities from galactic cosmic rays. *IEEE Trans. Nucl. Sci.*, vol 22:pp. 2675–2680, Dec 1975.

[4] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russel, W. Y. Yang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. G. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus. Ibm experiments in soft fails in computer electronics (1978-1994). *IBM J. Res. Develop.*, vol. 40(no. 1):pp. 3–18, 1996.

[5] W. A. Kolasinski, J. B. Blake, J. K. Anthony, W. E. Price, and E. C. Smith. Simulation of cosmic-ray induced soft errors and latchup in integrated-circuit computer memories. *IEEE Trans. Nucl. Sci.*, vol. 26:pp. 5087–5091, Dec. 1979.

[6] http://radhome.gsfc.nasa.gov/radhome/nat_space_rad_tech.htm.

[7] Robert Baumann. *Handbook of semiconductor manufacturing tecnology*, chapter chapter 31, pages 31–1 to 31–23. Taylor & Francis Group, LCC, second edition, 2007.

[8] L. W. Massengill P. E. Dodd. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Trans. Nucl. Sci.*, vol. 50(no. 3), June 2003.

[9] http://www.euronuclear.org/info/encyclopedia/l/linearenergytransfer.htm.

[10] http://en.wikipedia.org/wiki/alpha_particles.

[11] A. Ammari, L. Anghel, R. Leveugle, C. Lazzari, and R. Reis. Set fault injection methods in analog circuits: Case study. In *8th IEEE Latin American Test Workshop*, 2007.

[12] G. C. Messenger. Collection of charge on junction nodes from ion tracks. *IEEE Trans. Nucl. Sci.*, pages pp. 20024–2031, 1982.

[13] http://www.orcad.com/documents/community.faqs/pspice/020271.aspx.

Appendix A

Schematics

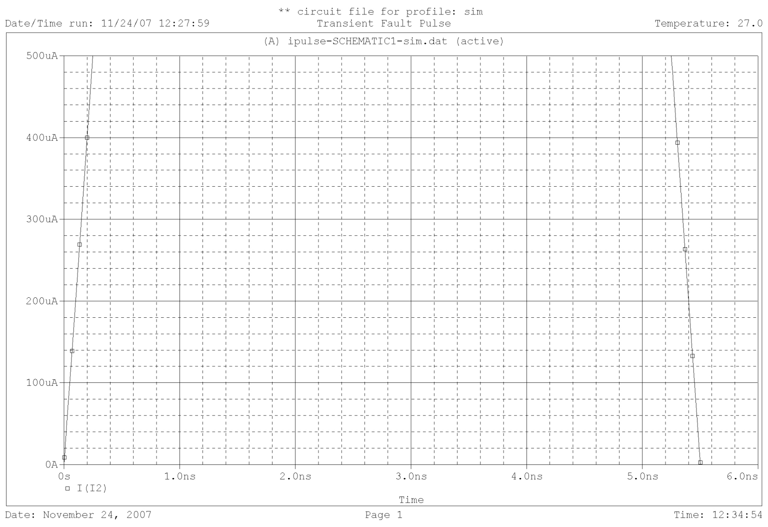


Figure A.1: Transient fault model used in PSpice

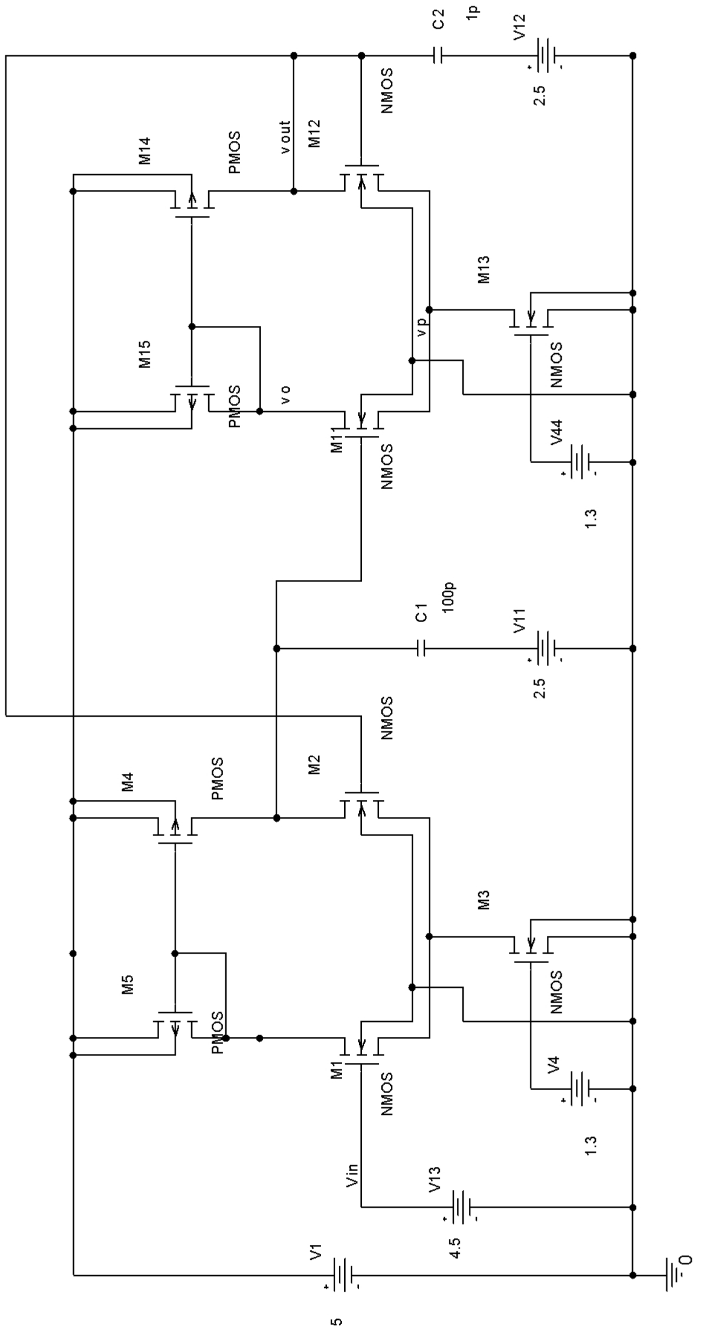


Figure A.2: Passband-filter

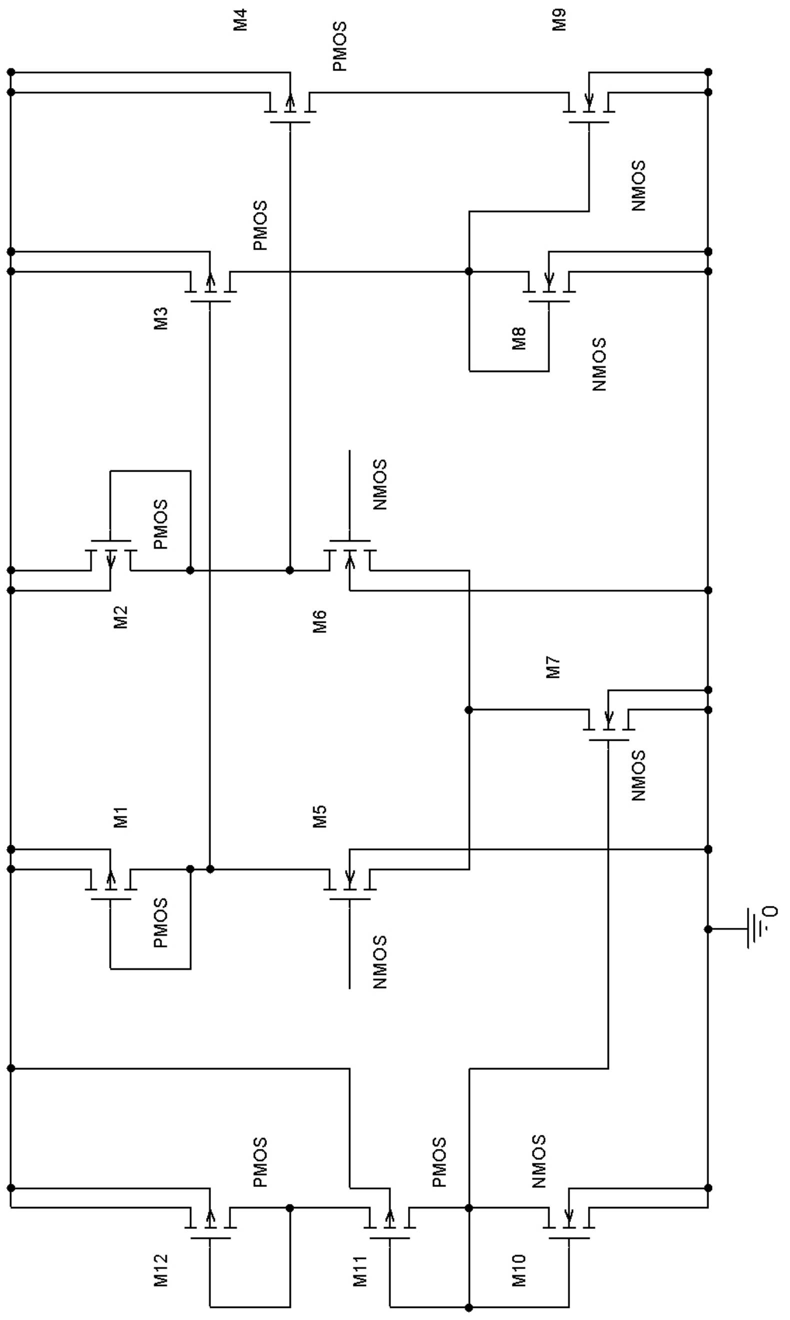


Figure A.3: Operational Amplifier A

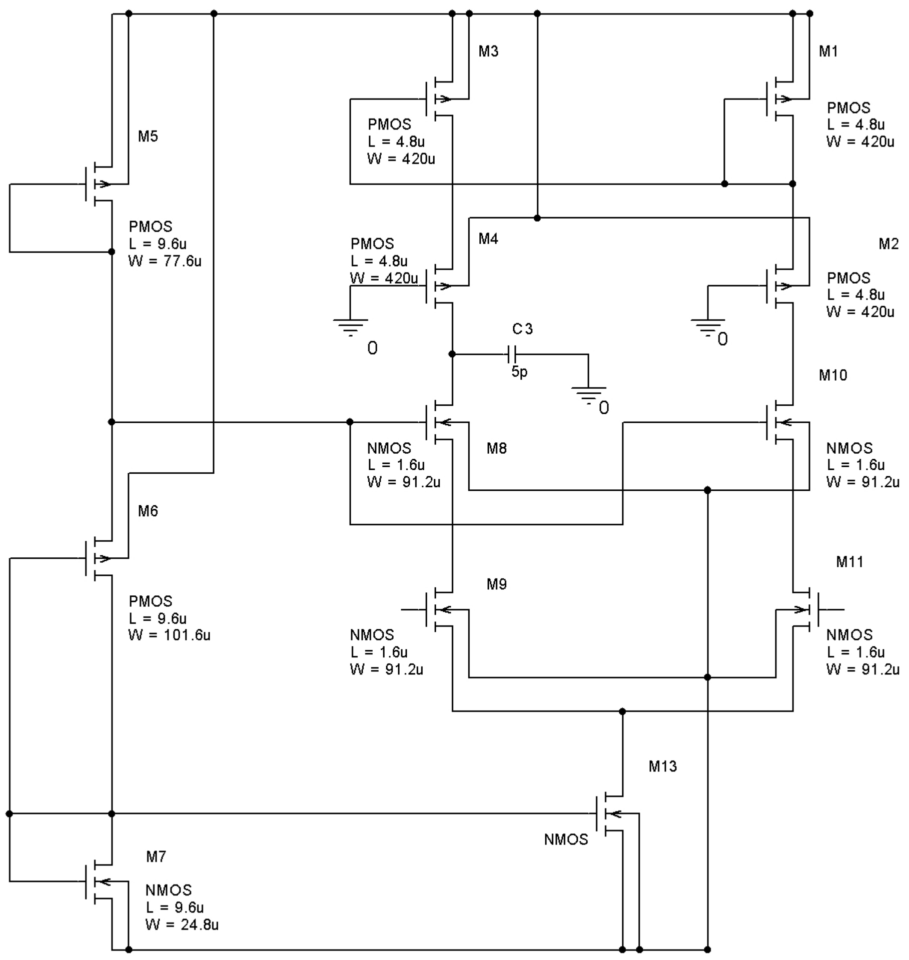
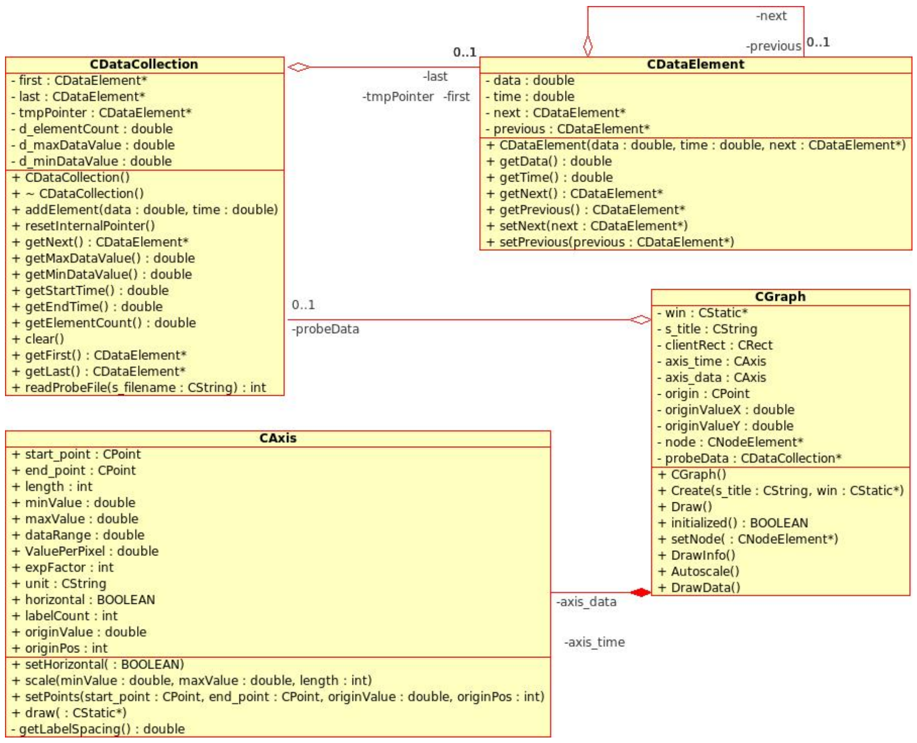


Figure A.4: Operational Amplifier B

Appendix B

Diagrams

Due to the complexity the whole model can not be presented on one page. However, selected class diagrams of certain functional context are shown in this secton. The following class diagrams were created using the UML-tool *Umbrello* for Linux.



-next
-previous
0..1
0..1
-last
-tmpPointer
-first
CDataCollection
- first : CDataElement*
- last : CDataElement*
- tmpPointer : CDataElement*
- d_elementCount : double
- d_maxDataValue : double
- d_minDataValue : double
+ CDataCollection()
+ ~ CDataCollection()
+ addElement(data : double, time : double)
+ resetInternalPointer()
+ getNext() : CDataElement*
+ getMaxDataValue() : double
+ getMinDataValue() : double
+ getStartTime() : double
+ getEndTime() : double
+ getElementCount() : double
+ clear()
+ getFirst() : CDataElement*
+ getLast() : CDataElement*
+ readProbeFile(s_filename : CString) : int
CDataElement
- data : double
- time : double
- next : CDataElement*
- previous : CDataElement*
+ CDataElement(data : double, time : double, next : CDataElement*)
+ getData() : double
+ getTime() : double
+ getNext() : CDataElement*
+ getPrevious() : CDataElement*
+ setNext(next : CDataElement*)
+ setPrevious(previous : CDataElement*)
0..1
-probeData
CGraph
- win : CStatic*
- s_title : CString
- clientRect : CRect
- axis_time : CAxis
- axis_data : CAxis
- origin : CPoint
- originValueX : double
- originValueY : double
- node : CNodeElement*
- probeData : CDataCollection*
+ CGraph()
+ Create(s_title : CString, win : CStatic*)
+ Draw()
+ initialized() : BOOLEAN
+ setNode(: CNodeElement*)
+ DrawInfo()
+ Autoscale()
+ DrawData()
CAxis
+ start_point : CPoint
+ end_point : CPoint
+ length : int
+ minValue : double
+ maxValue : double
+ dataRange : double
+ ValuePerPixel : double
+ expFactor : int
+ unit : CString
+ horizontal : BOOLEAN
+ labelCount : int
+ originValue : double
+ originPos : int
+ setHorizontal(: BOOLEAN)
+ scale(minValue : double, maxValue : double, length : int)
+ setPoints(start_point : CPoint, end_point : CPoint, originValue : double, originPos : int)
+ draw(: CStatic*)
- getLabelSpacing() : double
-axis_data
-axis_time

Figure B.1: Handling simulation data

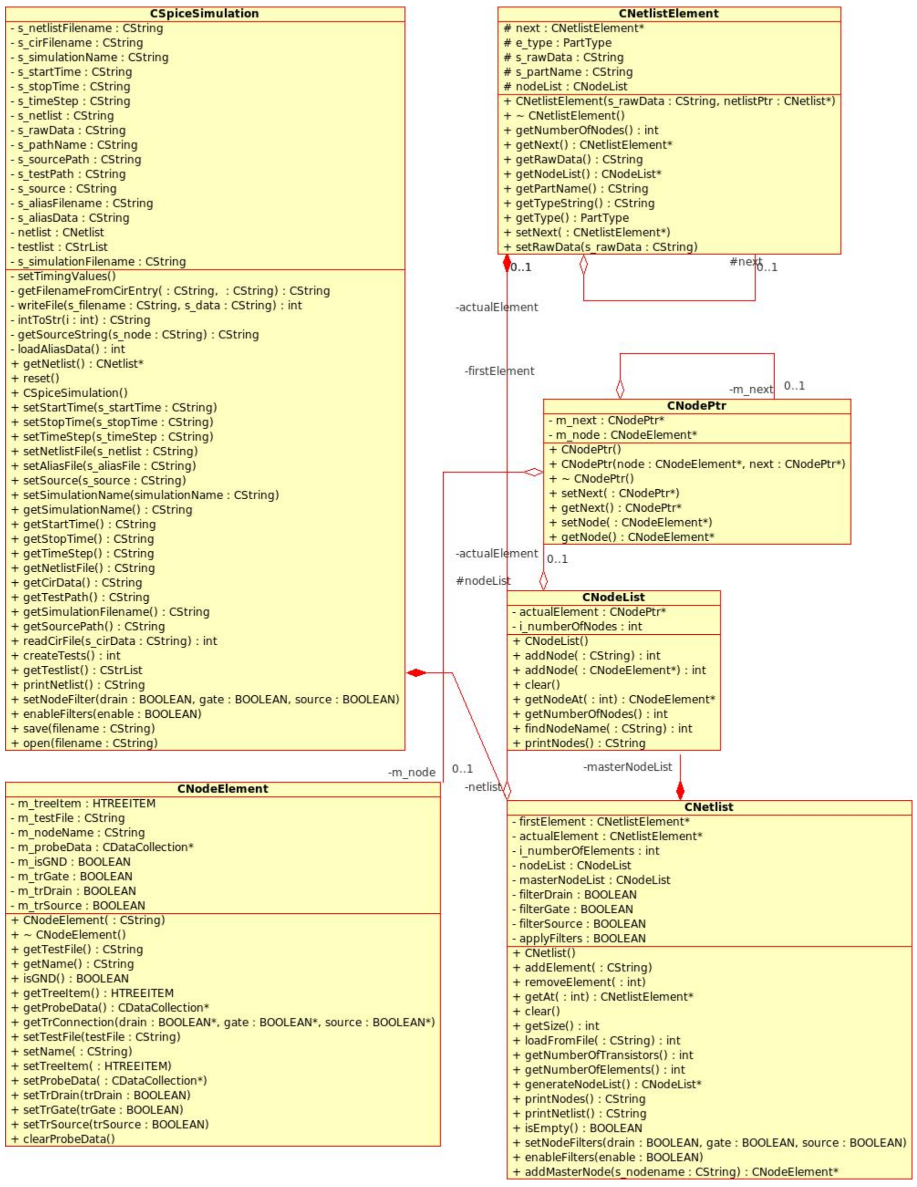

CSpiceSimulation
- s_netlistFilename : CString
- s_cirFilename : CString
- s_simulationName : CString
- s_startTime : CString
- s_stopTime : CString
- s_timeStep : CString
- s_netlist : CString
- s_rawData : CString
- s_pathName : CString
- s_sourcePath : CString
- s_testPath : CString
- s_source : CString
- s_aliasFilename : CString
- s_aliasData : CString
- netlist : CNetlist
- testlist : CStrList
- s_simulationFilename : CString
- setTimingValues()
- getFilenameFromCirEntry(: CString, : CString) : CString
- writeFile(s_filename : CString, s_data : CString) : int
- intToStr(i : int) : CString
- getSourceString(s_node : CString) : CString
- loadAliasData() : int
+ getNetlist() : CNetlist*
+ reset()
+ CSpiceSimulation()
+ setStartTime(s_startTime : CString)
+ setStopTime(s_stopTime : CString)
+ setTimeStep(s_timeStep : CString)
+ setNetlistFile(s_netlist : CString)
+ setAliasFile(s_aliasFile : CString)
+ setSource(s_source : CString)
+ setSimulationName(simulationName : CString)
+ getSimulationName() : CString
+ getStartTime() : CString
+ getStopTime() : CString
+ getTimeStep() : CString
+ getNetlistFile() : CString
+ getCirData() : CString
+ getTestPath() : CString
+ getSimulationFilename() : CString
+ getSourcePath() : CString
+ readCirFile(s_cirData : CString) : int
+ createTests() : int
+ getTestlist() : CStrList
+ printNetlist() : CString
+ setNodeFilter(drain : BOOLEAN, gate : BOOLEAN, source : BOOLEAN)
+ enableFilters(enable : BOOLEAN)
+ save(filename : CString)
+ open(filename : CString)
CNetlistElement
# next : CNetlistElement*
# e_type : PartType
# s_rawData : CString
# s_partName : CString
# nodeList : CNodeList
+ CNetlistElement(s_rawData : CString, netlistPtr : CNetlist*)
+ ~ CNetlistElement()
+ getNumberOfNodes() : int
+ getNext() : CNetlistElement*
+ getRawData() : CString
+ getNodeList() : CNodeList*
+ getPartName() : CString
+ getTypeString() : CString
+ getType() : PartType
+ setNext(: CNetlistElement*)
+ setRawData(s_rawData : CString)
0..1
#next
0..1
-actualElement
-firstElement
-m_next
0..1
CNodePtr
- m_next : CNodePtr*
- m_node : CNodeElement*
+ CNodePtr()
+ CNodePtr(node : CNodeElement*, next : CNodePtr*)
+ ~ CNodePtr()
+ setNext(: CNodePtr*)
+ getNext() : CNodePtr*
+ setNode(: CNodeElement*)
+ getNode() : CNodeElement*
-actualElement
0..1
#nodeList
CNodeList
- actualElement : CNodePtr*
- i_numberOfNodes : int
+ CNodeList()
+ addNode(: CString) : int
+ addNode(: CNodeElement*) : int
+ clear()
+ getNodeAt(: int) : CNodeElement*
+ getNumberOfNodes() : int
+ findNodeName(: CString) : int
+ printNodes() : CString
-m_node
0..1
-netlist
-masterNodeList
CNodeElement
- m_treeItem : HTREEITEM
- m_testFile : CString
- m_nodeName : CString
- m_probeData : CDataCollection*
- m_isGND : BOOLEAN
- m_trGate : BOOLEAN
- m_trDrain : BOOLEAN
- m_trSource : BOOLEAN
+ CNodeElement(: CString)
+ ~ CNodeElement()
+ getTestFile() : CString
+ getName() : CString
+ isGND() : BOOLEAN
+ getTreeItem() : HTREEITEM
+ getProbeData() : CDataCollection*
+ getTrConnection(drain : BOOLEAN*, gate : BOOLEAN*, source : BOOLEAN*)
+ setTestFile(testFile : CString)
+ setName(: CString)
+ setTreeItem(: HTREEITEM)
+ setProbeData(: CDataCollection*)
+ setTrDrain(trDrain : BOOLEAN)
+ setTrGate(trGate : BOOLEAN)
+ setTrSource(trSource : BOOLEAN)
+ clearProbeData()
CNetlist
- firstElement : CNetlistElement*
- actualElement : CNetlistElement*
- i_numberOfElements : int
- nodeList : CNodeList
- masterNodeList : CNodeList
- filterDrain : BOOLEAN
- filterGate : BOOLEAN
- filterSource : BOOLEAN
- applyFilters : BOOLEAN
+ CNetlist()
+ addElement(: CString)
+ removeElement(: int)
+ getAt(: int) : CNetlistElement*
+ clear()
+ getSize() : int
+ loadFromFile(: CString) : int
+ getNumberOfTransistors() : int
+ getNumberOfElements() : int
+ generateNodeList() : CNodeList*
+ printNodes() : CString
+ printNetlist() : CString
+ isEmpty() : BOOLEAN
+ setNodeFilters(drain : BOOLEAN, gate : BOOLEAN, source : BOOLEAN)
+ enableFilters(enable : BOOLEAN)
+ addMasterNode(s_nodename : CString) : CNodeElement*

Figure B.2: Class diagram of data model

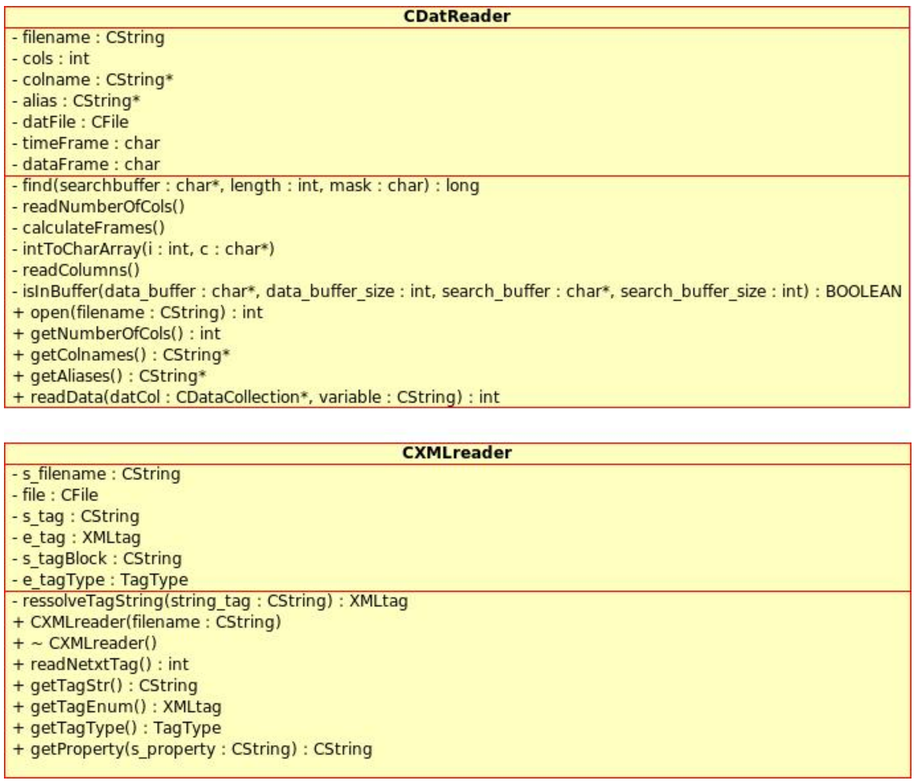

CDatReader
- filename : CString
- cols : int
- colname : CString*
- alias : CString*
- datFile : CFile
- timeFrame : char
- dataFrame : char
- find(searchbuffer : char*, length : int, mask : char) : long
- readNumberOfCols()
- calculateFrames()
- intToCharArray(i : int, c : char*)
- readColumns()
- isInBuffer(data_buffer : char*, data_buffer_size : int, search_buffer : char*, search_buffer_size : int) : BOOLEAN
+ open(filename : CString) : int
+ getNumberOfCols() : int
+ getColnames() : CString*
+ getAliases() : CString*
+ readData(datCol : CDataCollection*, variable : CString) : int
CXMLreader
- s_filename : CString
- file : CFile
- s_tag : CString
- e_tag : XMLtag
- s_tagBlock : CString
- e_tagType : TagType
- ressolveTagString(string_tag : CString) : XMLtag
+ CXMLreader(filename : CString)
+ ~ CXMLreader()
+ readNetxtTag() : int
+ getTagStr() : CString
+ getTagEnum() : XMLtag
+ getTagType() : TagType
+ getProperty(s_property : CString) : CString

Figure B.3: I/O classes implementing adaptors

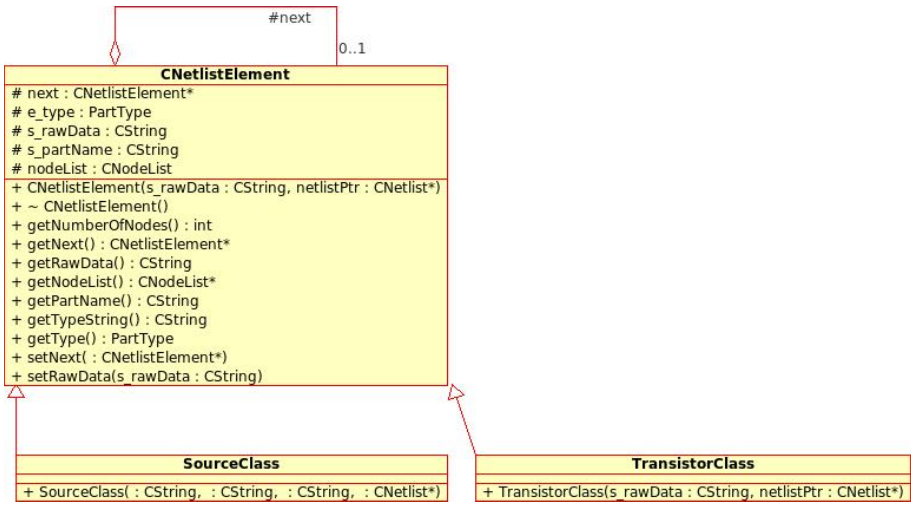

#next
0..1
CNetlistElement
# next : CNetlistElement*
# e_type : PartType
# s_rawData : CString
# s_partName : CString
# nodeList : CNodeList
+ CNetlistElement(s_rawData : CString, netlistPtr : CNetlist*)
+ ~ CNetlistElement()
+ getNumberOfNodes() : int
+ getNext() : CNetlistElement*
+ getRawData() : CString
+ getNodeList() : CNodeList*
+ getPartName() : CString
+ getTypeString() : CString
+ getType() : PartType
+ setNext(: CNetlistElement*)
+ setRawData(s_rawData : CString)
SourceClass
+ SourceClass(: CString, : CString, : CString, : CNetlist*)
TransistorClass
+ TransistorClass(s_rawData : CString, netlistPtr : CNetlist*)

Figure B.4: Polymorphy of netlist elements

Appendix C

Source Code

As the complete source code exceeds 6000 lines, it is not presented entirely in this section of the appendix. Instead, some of the most important functions are listed to show the solution of specific tasks.

C.1 Creating the circuit model

Listing C.1: CSpiceSimulation::readCirFile(...) function

```
/* DEFINITION:
   The readCirFile(...) Function reads the main values needed
   for the simulation model from a specified .cir file.
   It finds all further files needed for the simulation and
   stores the data in the simulation model.
   RETURN VALUE:
   Returns 0 if unsuccessful, otherwise the loadFromFile() result.
  */

int CSpiceSimulation::readCirFile(CString s_cirFile) {
   CFile cirFile;
   CFileException exception;
   s_cirFilename = s_cirFile;

   if(!cirFile.Open(s_cirFile, CFile::modeRead, &exception)) {
      return 0;
   } else {
      char c_buf[2];
      int i_start;
      int i_startSearch = 0;
      s_rawData = "";
      while(cirFile.GetPosition() < cirFile.GetLength()) {
         cirFile.Read(c_buf, 1);
         s_rawData = s_rawData + c_buf[0];
      }
      //Use CString Find function to retrieve directives:
      // [.INC]
      i_startSearch = s_rawData.Find("*Netlist", 0);
      if((i_start = s_rawData.Find(".INC ", i_startSearch)) > 0) {
         //find end-of-line
         i_start += 6;
         s_netlist = "";
         while(s_rawData.GetAt(i_start) != '\"') {
             s_netlist = s_netlist + s_rawData.GetAt(i_start);
             i_start++;
         }
      }
      //get Alias File:
      i_startSearch = s_rawData.Find("*Alias", 0);
      if((i_start = s_rawData.Find(".INC ", i_startSearch)) > 0) {
         //find end-of-line
         i_start += 6;
         s_aliasFilename = "";
         while(s_rawData.GetAt(i_start) != '\"') {
             s_aliasFilename += s_rawData.GetAt(i_start);
             i_start++;
         }
      }

      // [.TRAN]
      if((i_start = s_rawData.Find(".TRAN ")) > 0) {
         //find end-of-line
         i_start += 5;
         while(s_rawData.GetAt(i_start) == ' ') {i_start++;}
         //1st entry -> start time
         s_startTime = "";
         while(s_rawData.GetAt(i_start) != ' ') {
             s_startTime += s_rawData.GetAt(i_start);
             i_start++;
         }
         while(s_rawData.GetAt(i_start) == ' ') {i_start++;}
         //2nd entry -> stop time
         s_stopTime = "";
         while(s_rawData.GetAt(i_start) != ' ') {
             s_stopTime += s_rawData.GetAt(i_start);
             i_start++;
         }
         while(s_rawData.GetAt(i_start) == ' ') {i_start++;}
         //3rd entry -> ?
         while(s_rawData.GetAt(i_start) != ' ') {
             i_start++;
         }
         while(s_rawData.GetAt(i_start) == ' ') {i_start++;}
         //4th entry -> time step
         s_timeStep = "";
         while(s_rawData.GetAt(i_start) != ' ') {
             s_timeStep += s_rawData.GetAt(i_start);
             i_start++;
         }
      }

      cirFile.Close();
      //read Netlist:
      s_netlist = getFilenameFromCirEntry(s_cirFile, s_netlist);
      s_aliasFilename = getFilenameFromCirEntry(s_cirFile,
         s_aliasFilename);
      if(!loadAliasData()) {
         return 0;
      } else {
         s_netlistFilename = s_netlist;
         return netlist.loadFromFile(s_netlist);
      }
   }
}
```

Listing C.2: CNetlist::loadFromFile(...) function

```
/*
   Description:
   The loadFromFile(...) function opens the file specified
   by the parameter and constructs the netlist. It reads
   and adds all elements and nodes. After this operation
   the netlist should be set up completely and contain all
   required information.
*/

int CNetlist::loadFromFile(CString s_filename) {
   //Open netlist and load contents
   //function for Windows OS
   int i_numberOfLines = 0;
   CFile cf_netlist;
   CFileException fileError;
   //MessageBox(hWnd, "init",NULL, MB_OK);
   if(!cf_netlist.Open(s_filename, CFile::modeRead, &fileError)) {
      return 0;
   } else {
      //read content:
      long l_pos = 0;
      char c_buf[2];
      CString line = "";
      while(cf_netlist.Read(c_buf, 1)) {
         //get lines
         //lf+cr -> 0D 0A Hex
         switch(c_buf[0]) {
         case (char) 10:
            //backup pos:
            l_pos = cf_netlist.GetPosition();
            //peek if next line begins with a '+'
            if(cf_netlist.Read(c_buf, 1) && (c_buf[0] == '+')) {
               //line not complete!
               line += (char) 13;
               line += (char) 10;
            } else {
               //ignore comments:
               if(line.GetAt(0) != '*') {
                  addElement(line);
               }
               line = "";
               i_numberOfLines++;
            }
            //go back to pos!
            cf_netlist.Seek(l_pos, 0);
            break;
         case (char) 13:
            //ignore!
            break;
         default:
            //process:
            line += c_buf[0];
            break;
         }
      }
      //close
      cf_netlist.Close();
      //file operation succesful
      return i_numberOfLines;
   }
}
```

Listing C.3: CNetlist::addElement(...) function

```
/*
   Definition:
   The addElement function determines the type of the
   item and appends a new TransistorClass or CNetlistElement
   object to the list.
*/

void CNetlist::addElement(CString s_rawData) {
   CNetlistElement * element;
   //distinguish between Types
   switch(s_rawData.GetAt(0)) {
      case 'Q':
         //BJT Transistor:
         element = new TransistorClass(s_rawData, this);
         break;
      case 'M':
         //Mosfet Transistor:
         element = new TransistorClass(s_rawData, this);
         break;
      default:
         element = new CNetlistElement(s_rawData, this);
         break;
   }
   //first item?
   if(i_numberOfElements == 0) {
      //link the item to firstElement
      firstElement = element;

      actualElement = element;
   } else   {
      //set pointer to last element:
      element->setNext(actualElement);
      //update actualElement
      actualElement = element;
   }
   i_numberOfElements++;
}
```

Listing C.4: CNodeList::addNode(...) function overloading

```
/*
   Definition:
   There are two ways to add a new node to a CNodeList:
   The first addNode(...) function takes a CString with
   the name of the node which should be created and
   creates a new CNodeElement object, as well as a new
   CNodePtr object. The second way takes an existing
   CNodeElement and links it to the list by creating a
   new CNodePtr object whicht points to the existing
   node. A special mecanism prevents any node to appear
   in the list more than once.

   Return Value:
   1: function successful
   0: function aborted - node already in list
*/

//1st posibility: Add new Node
int CNodeList::addNode(CString s_name) {
   //check, if name already in list
   if(findNodeName(s_name) > 0) {
        //Node already in list
        return 0;
    } else {
        //Create new node
        CNodeElement * node;
        node = new CNodeElement(s_name);
        //Create list element
        CNodePtr * element;
        element = new CNodePtr();

        element->setNode(node);

        //Add the new element
        if(i_numberOfNodes == 0) {
            actualElement = element;
        } else {
            element->setNext(actualElement);
            actualElement = element;
        }
        i_numberOfNodes++;
        //new node created
        return 1;
    }
}

//Overloading: As existing Node:
int CNodeList::addNode(CNodeElement * nodeElement) {
    //check, if name already in list
    if(findNodeName(nodeElement->getName()) > 0) {
        //Node already in list
        return 0;
    } else {
        //Create new element!
        CNodePtr * element;
        element = new CNodePtr();
        element->setNode(nodeElement);

        //Add the new element
        if(i_numberOfNodes == 0) {
            actualElement = element;
        } else {
            element->setNext(actualElement);
            actualElement = element;
        }
        i_numberOfNodes++;
        //new node created
        return 1;
    }
}
```

Listing C.5: CNetlist::addMasterNode(...) function

```
/*
    DEFINITION:
    The addMasterNode(...) function is a wrapper function
    which handles the masterNodeList. It tries to create a
    new CNodeElement object and add it to the masterNodeList.
    If the node already exists, it retrieves its reference.
    Ether way, it returns a reference of the real CNodeElement
    object to be used in other node lists.

    RETURN VALUE:
    Returns the reference of the real object in the masterNodeList.
```

```
   Type: CNodeElement *

 */

CNodeElement * CNetlist::addMasterNode(CString s_nodename) {
   //does the node already exist?
   if(masterNodeList.findNodeName(s_nodename) > 0) {
      //Node already exists -> return pointer:
      for(int i = 0; i < masterNodeList.getNumberOfNodes(); i++) {
         if(masterNodeList.getNodeAt(i)->getName() == s_nodename)
             return masterNodeList.getNodeAt(i);
      }
      //worst case: internal error
      return NULL;
   } else {
      //Node doesn't exist yet so create it!
      masterNodeList.addNode(s_nodename);
      //and return its pointer:
      return masterNodeList.getNodeAt(masterNodeList.
          getNumberOfNodes() - 1);
   }
}
```

C.2 Running PSpice

Listing C.6: Simulation Thread

```
UINT SimulationThread(LPVOID pParam) {
   //dialog handle
   CDialogProgress * dlg = (CDialogProgress *) pParam;
   //window handle
   CWnd * spiceWin = NULL;
   PROCESS_INFORMATION pi = {0};
   STARTUPINFO si = {sizeof(si)};
   si.wShowWindow = SW_NORMAL;
   char CommandLine[200];
   float f_percent = 0;
   char c_percent[30];
   CString s_percent = "";
   CNodeList * transistor_nodelist;

   //set init text
   dlg->m_mutex.Lock();
   dlg->m_ctl_percent.SetWindowText("Simulation started...");
   dlg->m_mutex.Unlock();

   // (1) Make sure that PSpice is not running:

   //Connect Windows handle:
   if (spiceWin = CWnd::FindWindow(NULL, "PSpice A/D Basics ")){
      dlg->m_terminateThread = 1;
      dlg->MessageBox("Please close PSpice before starting the 
          simulation!", "Error", MB_ICONEXCLAMATION);
   } else {
      int n;
      int i = 0;
      CString s_cirFile = "";
      CString s_datFile = "";
      CString s_storageFile = "";
      //char buffer[200];
      int i_pathEnd = 0;

```

```
        CStrList testlist = g_simulation.getTestlist();//dlg->m_simulation->getTestlist();

        transistor_nodelist = g_simulation.getNetlist()->generateNodeList();

        for(i = 0; i < testlist.getSize(); i++) {

            s_cirFile = testlist.getAt(i)->getString();
            n = sprintf(CommandLine, "C:\\Program Files\\OrCAD\\PSpice\\pspice.exe -r -e \"%s\"", s_cirFile);

            //Run at high priority to accelerate simulation:
            ::CreateProcess(NULL, CommandLine, NULL, NULL, FALSE, HIGH_PRIORITY_CLASS, NULL, NULL, &si, &pi);
            ::WaitForSingleObject( pi.hProcess, INFINITE);

            CloseHandle(pi.hProcess);
            CloseHandle(pi.hThread);
            //PSpice terminated...

            //Get dat file:
            s_datFile = s_cirFile;
            s_datFile.Replace(".cir", ".dat");

            //Get and save the simulation data:

            //IMPORTANT: SET NODE TESTFILE!!!
            transistor_nodelist->getNodeAt(i)->setTestFile(s_datFile); //s_storageFile);

            //UpdateProgressbar:
            dlg->m_mutex.Lock(INFINITE);
            dlg->m_progressBar.SetRange(0, (short) testlist.getSize());
            dlg->m_progressBar.SetPos(i+1);

            //calculate & display percentage:
            f_percent = ((float) (i+1) / (float) testlist.getSize()) * 100;
            sprintf(c_percent, "simulating [ %i of %i ] --> %.2f", (i+1), testlist.getSize(), f_percent);
            s_percent = c_percent;
            s_percent += "% complete";
            dlg->m_ctl_percent.SetWindowText(s_percent);
            dlg->RedrawWindow(NULL, NULL, RDW_UPDATENOW | RDW_ERASE);
            dlg->m_mutex.Unlock();
        }
    }
    dlg->m_mutex.Lock(INFINITE);
        dlg->m_terminateThread = 1;
    dlg->m_mutex.Unlock();

    while(dlg->m_terminateThread == 1) {
        Sleep(100);
    }
    Sleep(1000);
    g_simulation.save(g_simulation.getTestPath() + "\\simulation.xml");
    dlg->CloseDialog();
    return 0;
}
```

C.3 Reading Simulation Data

Listing C.7: CDatReader::readData(...) function

```
int CDatReader::readData(CDataCollection * dataCollection, CString
    variable) {
  int i = 0;
  int ii = 0;
  const int data_buffer_size = 15;
  int selected_var = 0;
  CFile datFile;
  char data_buffer[data_buffer_size];
  char buffer[1];
  //state machine:
  ReaderState state_machine;
  double d_time_buffer[10];
  float f_data_buffer[10];
  long fileptr = 0;
  char dump[5];



  //init data window:
  for(i = 0; i < data_buffer_size; i++) data_buffer[i] = (char) 0
      x00;
  i = 0;

  //select variable
  while((i < cols) && (colname[i] != variable)) i++;
  if(i == cols) return -1;
  selected_var = i;

  //open file
  if(datFile.Open(filename, CFile::modeRead, NULL) != NULL) {
    //start reading data:
    state_machine = DUMP_DATA;
    //read to eof:
    while(datFile.Read(buffer, 1)) {
      fileptr++;
      //shift+add data_buffer:
      for(i = 0; i < (data_buffer_size - 1); i++) data_buffer[i]
          = data_buffer[i+1];
      data_buffer[data_buffer_size-1] = buffer[0];
      //state machine:
      switch(state_machine) {
      case DUMP_DATA:
        //do nothing but wait for first frame:
        if(isInBuffer(data_buffer, data_buffer_size, timeFrame,
            8)) state_machine = READ_TIME_FRAME;
        if(isInBuffer(data_buffer, data_buffer_size, dataFrame,
            8)) state_machine = READ_DATA_FRAME;


        break;
      case READ_DATA_FRAME:
        //skip blocks of unwanted data and keep last one
        for(i = 0; i < 10; i++) {
          for(ii = 0; ii < cols; ii++) {
            if(ii == selected_var) {
              datFile.Read(&f_data_buffer[i], 4);
            } else {
              datFile.Read(dump, 4);
```

```
                    }
                }
            }
            //now add data to CDataCollection:
            for(i = 0; i < 10; i++) {
                dataCollection->addElement(f_data_buffer[i],
                     d_time_buffer[i]);
            }

            state_machine = DUMP_DATA;
            break;

        case READ_TIME_FRAME:
            //read 10x 4byte integer:
            for(i = 0; i < 10; i++) {
                datFile.Read(&d_time_buffer[i], 8);
                fileptr+=8;
            }
            state_machine = DUMP_DATA;
            break;
        }
    }
    return 1;
} else {
    return -1;
}
}
```

C.4 Saving the Simulation Model

Listing C.8: CXMLreader::readNextTag(...) function

```
/* DEFINITION:
   The readNextTag function reads the next tag in the xml file.
   When the file pointer has reached the end of the file it returns
       -1
   otherwise the current node string is stored in the reader.

*/

int CXMLreader::readNetxtTag() {
    s_tagBlock = "";
    s_tag = "";
    int pos = 1;

    char buffer[2];
    buffer[0] = (char) 0;
    while(buffer[0] != '<') {
        //return -1 if eof
        if(file.Read(buffer, 1) == 0) return -1;
    }

    //now get the tag:
    while(buffer[0] != '>') {
        s_tagBlock += buffer[0];
        //retrun -1 if eof
        if(file.Read(buffer, 1) == 0) return -1;
    }
    s_tagBlock += ">";
    //now we have the complete tag!
    //closing tag?
    if(s_tagBlock.GetAt(pos) == '/') {
        e_tagType = close_tag;
        pos++;
    } else {
        //open or single?
        if(s_tagBlock.GetAt(s_tagBlock.GetLength() - 2) == '/') {
           e_tagType = single_tag;
        } else {
           e_tagType = open_tag;
        }
    }


    //so let's get the tag name:
    while((s_tagBlock.GetAt(pos) != ' ') &&
           (s_tagBlock.GetAt(pos) != '/') &&
           (s_tagBlock.GetAt(pos) != '>') &&
           (pos < s_tagBlock.GetLength()))
    {
        s_tag += s_tagBlock.GetAt(pos++);
    }


    //last not least: ressolve the tag:
    e_tag = ressolveTagString(s_tag);

    return 0;
}
```

Listing C.9: CXMLreader::getProperty(...) function

```
/*
   DEFINITION:
   The getProperty function retrieves a specified property from
   the current tag. The property is being returned as a CString.
   Properties should always have the format:
   property="value"
*/

CString CXMLreader::getProperty(CString s_property) {
   CString s_value = "";
   int pos = 0;
   //is the property available?
   s_property += "=";
   pos = s_tagBlock.Find(s_property);
   if(pos == -1) return "";
   //copy the property:
   pos += s_property.GetLength() + 1;
   while(s_tagBlock.GetAt(pos) != '\"') s_value += s_tagBlock.GetAt
       (pos++);
   return s_value;
}
```

Listing C.10: CXMLreader::ressolveTagString(...) function

```
/*
   DEFINITION:
   The ressolveTagString is used internally to map CString tags
   to the XMLtag enumerated format.
*/


XMLtag CXMLreader::ressolveTagString(CString string_tag) {

   XMLtag enum_tag;
   //ressolve tag table
   if (string_tag == "Simulation")         enum_tag = xmlSimulation;
   else if (string_tag == "Sources")       enum_tag = xmlSources;
   else if (string_tag == "Alias")         enum_tag = xmlAlias;
   else if (string_tag == "Cir")       enum_tag = xmlCir;
   else if (string_tag == "Net")       enum_tag = xmlNetlist;
   else if (string_tag == "Start")         enum_tag = xmlStart;
   else if (string_tag == "Stop")          enum_tag = xmlStop;
   else if (string_tag == "Timestep")      enum_tag = xmlTimestep;
   else if (string_tag == "Source")    enum_tag = xmlSource;
   else if (string_tag == "Parameter")     enum_tag = xmlParameter;
   else if (string_tag == "Raw")       enum_tag = xmlRaw;
   else if (string_tag == "NetlistElement")  enum_tag =
       xmlNetlistElement;
   else if (string_tag == "Node")          enum_tag = xmlNode;
   else if (string_tag == "Nodes")         enum_tag = xmlNodes;
   else if (string_tag == "Test")          enum_tag = xmlTest;
   else                    enum_tag = xmlUnknown;

   return enum_tag;
}
```