Table of Contents

1.ABSTRACT

This document tents to be the user guide and reference for program CASPER.

2. DEFINITIONS

CASPER	**C**ode for **A**uxiliary **S**ystems and **PER**formance
YACC	**Y**et **A**nother **C**ompiler **C**ompiler

3. SCOPE

The present document is focused to program CASPER developers, physical modellers. The program features described are the most advanced coded so far. Future developments will be based on this program version. It is also the basis for people in charge to create new components libraries for new modelling areas. Finally is also the reference for those users that are interested uniquely in the simulation of existing physical components.

4.INTRODUCTION

This is an in house and multi-purpose tool in multiplatform philosophy. Its libraries have been programmed in C++.

5.CASPER LIBRARIES

5.1Object oriented modelling

Object-Oriented modelling is a powerful and intuitive paradigm for building models that can outlive the future changes of the growth and ageing of any dynamic system. It provides the modeller with powerful features as *encapsulation*, *inheritance* and *polymorphism*.

Modular development allows a system to be modelled bottom-up. Basic library components can be combined to create complex components by combining two methods:

- Extension by inheritance from existing components
- Instantiation and connection of existing components

Moreover, unlike some other object-oriented programming languages, C++ makes a clear distinction between a class, which is a user-defined type, and an object, which is an instance thereof.

5.1.1Encapsulation

With a conventional object-oriented language such as C++, the public interface is the data and methods declared public. A *class* provides a distinct separation between its internal implementation (the part is more likely to change) and its public interface. In CASPER a component's public interface consists of its ports, construction parameters and data. These elements are unique and are visible from outside during modelling, reinforcing encapsulation and favouring reuse the component.

5.1.2Inheritance

Inheritance enables a derivate class to reuse the functionality and interface of its base class. The advantages of reuse are enormous and it gives CASPER a tremendous power. This allows the creation of libraries based on parent components with a linear rather than geometric order or complexity. A new component based on another parent will include all its data and behaviours. C++ provides multiple inheritance, i.e. a component can inherit data and behaviour from one or many components that have previously been designed and tested.

5.1.3Polymorphism

Polymorphism is the capability of different objects to react in an individual manner to the same message and it is the most powerful tool in object-oriented programming. Polymorphism in object-oriented programming means that the interpretation of a message (an action) depends on the object.

5.2Organization in libraries

Libraries are a mechanism used to organise design information. They are a natural way to group components, functions, etc that are related to a discipline. Libraries are the basic elements around which CASPER manipulates the information. These libraries must be programmed in C++ and their basic purpose is to store a set of related elements.

As well as keeping work organized, working with libraries allows the modelling environment to provide other services, such as:

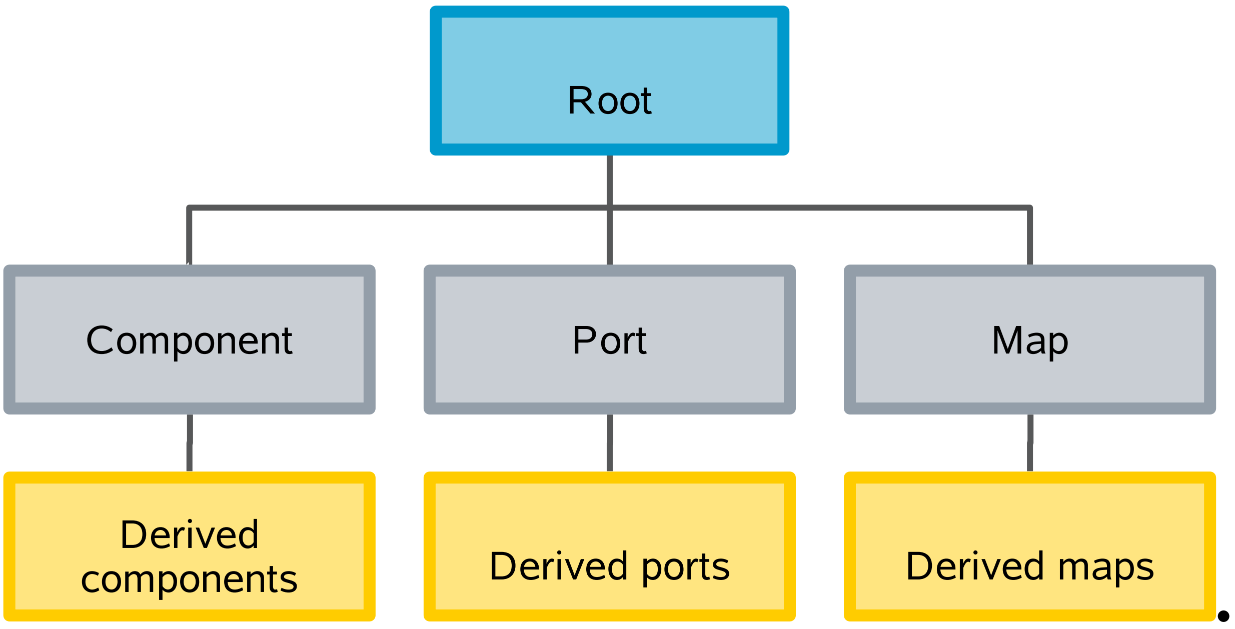
- Checking for obsolete units (components, ports, etc) that should be recompiled
- Encapsulation of elements, allowing different libraries to contain elements with the same name

A CASPER library can contain the following type of elements:

- Components
- Ports types
- Extern functions
- Global variables and constants
- Global enumeration types
- Directives for using other libraries (`#include` statements)

5.3General Library

General Library is the base library, i.e. all the new defined-user libraries must be based on General Library. This contains all the elements (functions and variables) that are used for other derived libraries. The GL scheme is the following:



5.4How to build new libraries

CASPER allows the user manipulate the libraries in two possible ways:

- Create a new library. The user creates all the required elements for the new library.
- Use an existing library. In this case the library is already created and is available to be used. An example of an existing library is the GENERAL library.

We will explain the first point in this chapter.

Previously we have seen how to create new components and ports. Usually we need several components for the same system. For example, in an electronic system, we should create a different component for each element: *capacitor, resistor, transistor,* etc. They must be included within a same library.

Firstly a new file called *Interface.cpp* is required. Once we have created all the components we must include into this file all the header files (using `#include` statements) and the component's names as follows:

```
//GLOBAL LIBRARY INCLUDE
#include <GENERAL.h>
#include <INTERFACE.h>

//component include
#include <component_1.h>
#include <component_2.h>
…
#include <component_n.h>
```

```
LIBRARY(name_of_library, "library_description")
      component(name_of_library, name_of_Component_1)
      component(name_of_library, name_of_Component_2)
      …
      component(name_of_library, name_of_Component_n)
END_LIBRARY
```

For example, for our AIRFLOW library we could write the following *Interface.cpp*:

```
//GLOBAL LIBRARY INCLUDE
#include <GENERAL.h>
#include <INTERFACE.h>

//component include
#include <tank.h>
#include <pipe.h>

LIBRARY(AIRFLOW, "AIRFLOW system library")
      component(AIRFLOW, Tank)
      component(AIRFLOW, Pipe)
END_LIBRARY
```

It is very convenient to create a new file which contains all `#include` statements. This file will be very useful if we decide to use this library in the future for other libraries. Normally, you should call this file either *AIRFLOW.h* or *Airflow.h* (*name_of_library.h*) and must contain:

```
//component include
#include <tank.h>
#include <pipe.h>
```

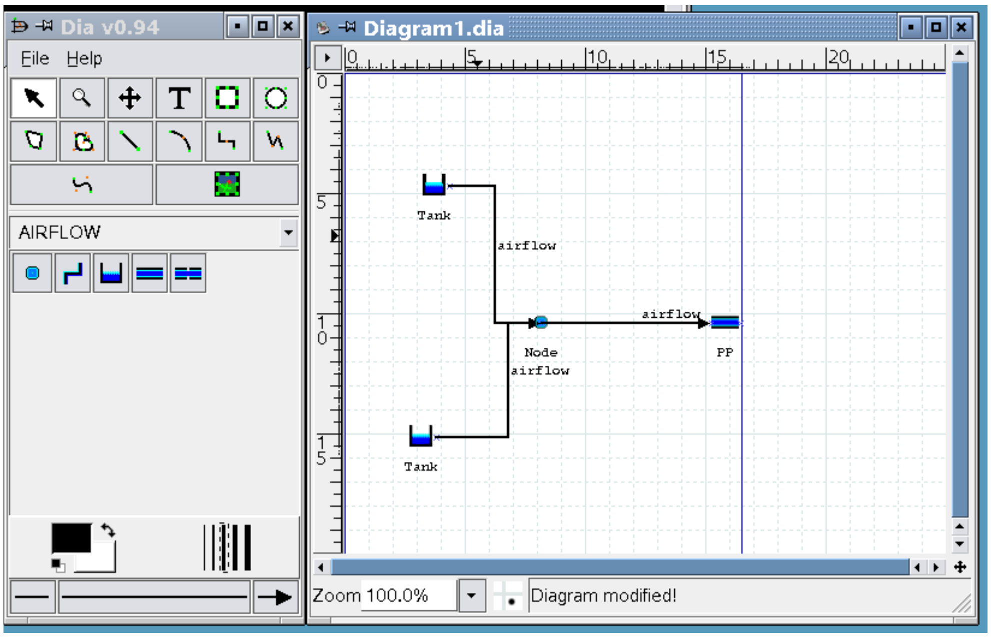
Well, from this point we should be able to compile our library.

6.INTERFACE WITH dia PROGRAM

6.1CASPER and dia

Once we have created our library we can use it to model our system using its components. Previously we said that components are connected between them through ports. Well, `Dia` is the tool that we use to create a diagram to show how our system works. For that, we must connect several components using *links*. *Link* is a special component which is defined in `Dia` by default.

It is very easy modelling using `Dia`. You only have to link your components conveniently. In the following example we can see a single example of an air system:



6.1.1How to load our library

`Dia` needs to know where our library is. For this, we must install our library in the path indicated by `DIA_LIB_PATH` variable. In the other hand, we have to create a new *sheet* file for our library where we must indicate which components of the library we want to load and also where are their icons. This file is a *xml* file and must be in the path indicated by `DIA_SHEET_PATH` variable.

The *sheet* file must be called *`name_of_library`*`.sheet` and must contain the following:

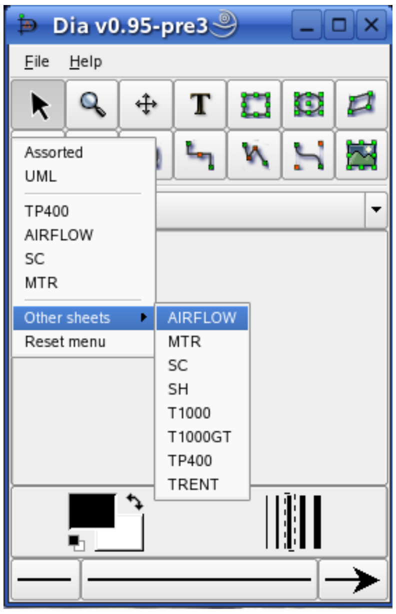
```
<?xml version="1.0" encoding="UTF-8"?>
<sheet xmlns="http://www.lysator.liu.se/~alla/dia/dia-sheet-ns">
  <name>name_of_library</name>
  <description>description_of_library</description>
  <contents>
    <object name="Link">
      <description>Link</description>
      <icon>Link.xpm</icon>
    </object>
    <object name="name_of_library - Component1">
      <description>description_of_Component1</description>
      <icon>Component1_icon</icon>
    </object>
    <object name="name_of_library - Component2">
      <description>description_of_Component2</description>
      <icon>Component2_icon</icon>
    </object>
    …
    <object name="name_of_library - Component_n">
      <description>description_of_Component_n</description>
      <icon>Component_n_icon</icon>
    </object>
   </contents>
```

```
</sheet>
```

For example, for the AIRFLOW library the *AIRFLOW.sheet* file should be:

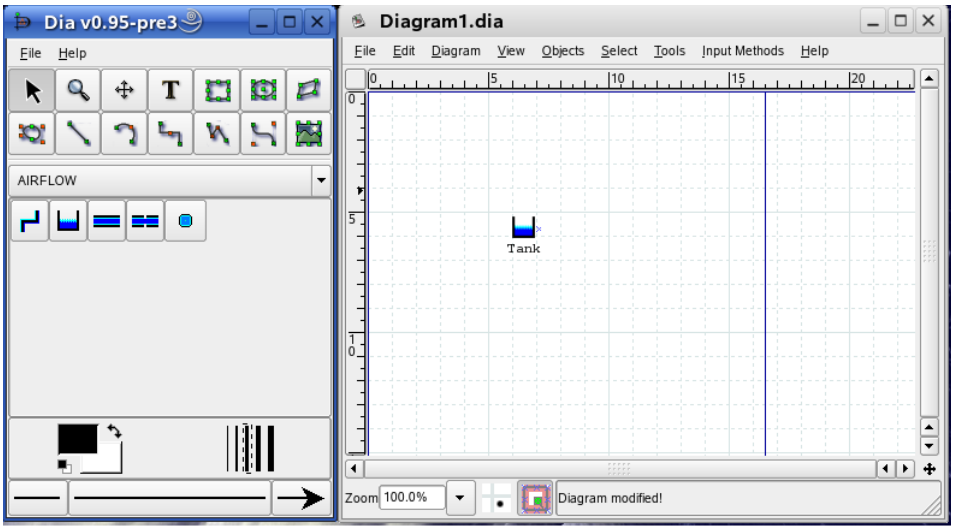
```
<?xml version="1.0" encoding="UTF-8"?>
<sheet xmlns="http://www.lysator.liu.se/~alla/dia/dia-sheet-ns">
  <name>AIRFLOW</name>
  <description>Cp constant airflow library</description>
  <contents>
    <object name="Link">
      <description>Link</description>
       <icon>Link.xpm</icon>
    </object>
    <object name="AIRFLOW - Tank">
      <description>Tank</description>
       <icon>AIRFLOW/Tank.xpm</icon>
    </object>
    <object name="AIRFLOW - PP">
      <description>Pipe</description>
       <icon>AIRFLOW/PP.xpm</icon>
    </object>
    <object name="AIRFLOW - PL">
      <description>P_loss</description>
       <icon>AIRFLOW/PL.xpm</icon>
    </object>
    <object name="AIRFLOW - Node">
      <description>Airflow Node</description>
       <icon>AIRFLOW/Node.xpm</icon>
    </object>
   </contents>
</sheet>
```

Once we have created the *sheet* file, we will be able to load our library using `Dia`. You can start `Dia` by going into the Applications on the Main menu and clicking in the `Dia` icon or you can type `dia` in your shell. In the main menu you should see our library:



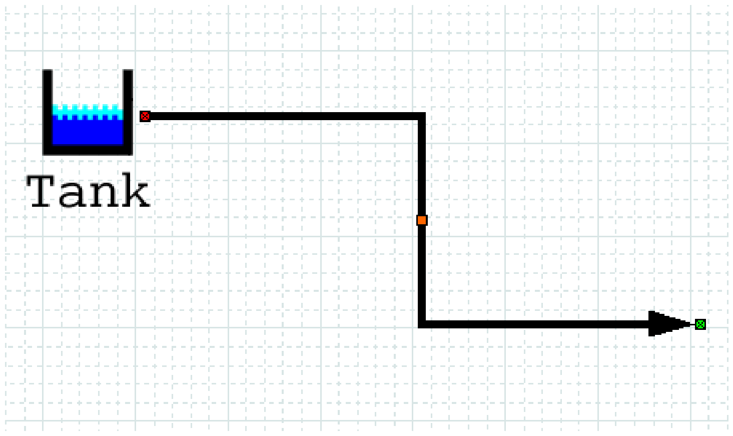
6.1.2Adding components

To add component to the canvas, click on an component in the toolbox and click on the canvas. The selected component will appear (see the following figure). The object can be manipulated by clicking and dragging on the corner buttons.



6.1.3Connecting components

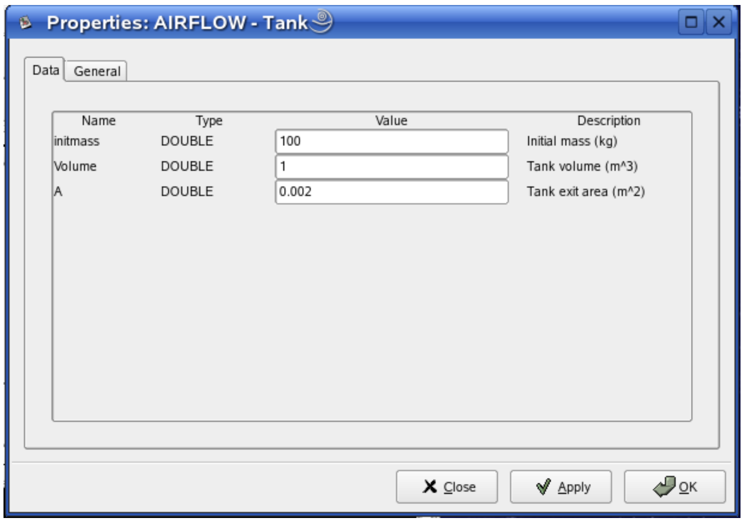
The powerful feature of `Dia` is its ability of creating connections between components. For this, click on *Link* object and then drag on one link point up to one component port. If they are linked the joint point will appear in red colour.



Note that in the link appears an arrow indicating the direction of the information.

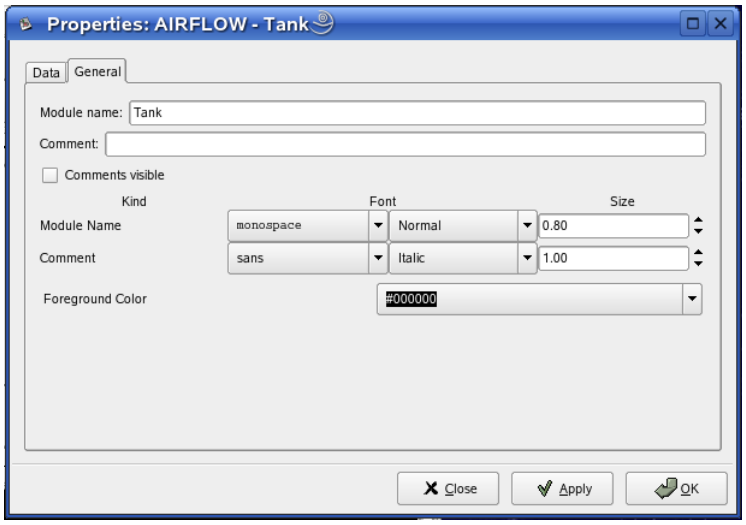
6.3.4 Editing components

In the canvas you can double-click on your component and to view its properties. A new window will appear, which contains all data of the component (that were introduced by *AddData* functions within the component constructor). All of them are input of our system and therefore you can give a value for them. These data have a value by default. If you want to modify someone of them, you must introduce the value into the corresponding field.



Name	Type	Value	Description
initmass	DOUBLE	100	Initial mass (kg)
Volume	DOUBLE	1	Tank volume (m^3)
A	DOUBLE	0.002	Tank exit area (m^2)

In the General tab you can change the name and other properties (font, size, colour, etc.) of the component:



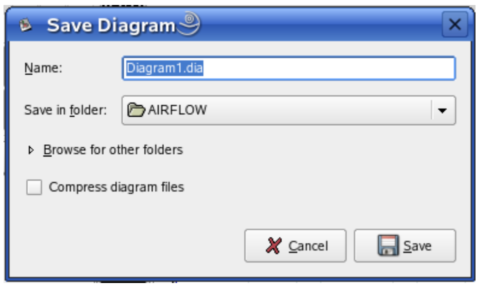
Module name: Tank
Comment:
Comments visible

Kind	Font		Size
Module Name	monospace	Normal	0.80
Comment	sans	Italic	1.00

Foreground Color #000000

6.3.5Saving the model

When we have finished our model, we should save it into the analysis folder. By default, the file's extension is *.dia* and we can save in a compress or not compress format. `Dia` generates a xml file, which can be understood by CASPER.



Name: Diagram1.dia
Save in folder: AIRFLOW
Compress diagram files

7.CASPER KERNEL

7.1Basis

7.2Input syntax

CASPER needs an input file (with *.cmp* extension) where we have to specify all the closure equations in the model and the unknowns. We also can give initial values for input data. This file is also called COMPO.

7.2.1Loading the model

Once we have generated our model by `Dia`, we must specify the *dia* file within COMPO using the following syntax:

```
MODEL="name_of_model.dia"
```

7.2.2Setting unknowns

Now we have to set which our unknowns are. We only can use variables that we have added by *AddUnknown* and the port variables. The rest can not be used in COMPO like unknowns. The syntax is the following:

```
UNKNOWNS = n {
1      unknown_1   initial_value_1
2      unknown_2   initial_value_2
…
n      unknown_n   initial_value_n
}
```

where *n* is the number of unknowns.

7.2.3Setting limits

We can set limits for our variables (unknowns or data). We can specify if a variable must be greater (`GE`) or lower (`LE`) than a given limit:

```
LIMITS = number_of_limits {
1      variable_1 [LT|GT|LE|GE|EQ|NE] limit_1
2      variable_2 [LT|GT|LE|GE|EQ|NE] limit_2
…
n      variable_n [LT|GT|LE|GE|EQ|NE] limit_n
}
```

It is not necessary to set limits if they are not required.

- `LE` or `GE`: lower or equal, greater or equal than a given value, respectively.
- `LT` or `GT`: lower, greater than a given value, respectively.
- `EQ`: equal to a value.
- `NE`: not equal to a value.

7.2.4Setting equations

In this part we set all the closure equations. Obiously, the number of these must be equal to the number of unknowns and they must be independent, or else the system should be singular and it could not be solved.

The syntax is as follows:

```
EQUATIONS = {
1      [s|c] equation_1
2      [s|c] equation_2
…
n      [s|c] equation_n
}
```

where we use `s` (single equation) if the equation is a single function or `c` (complex equation) if there are several expresions for the same function.

7.2.5Setting data

How we have said we can set data values into COMPO file. That also was possible in `Dia` openning the data tab in the component properties. However, if you want to modify some value you can do it in COMPO file. For that, you must specify the data name and then its value.

```
Tank.initmass=20
```

In the other hand, CASPER allows to define new variables into COMPO file. For example, you could write within COMPO file:

```
MASS=20
```

or even

```
MASS=Tank.initmass
```

These values can be used within the closure equations.

7.2.6Analysis keywords

CASPER allows three different analysis: DEBUG, STEADY and TRANSIENT.

- DEBUG: It is a steady analysis. However, this type allows you to view every step of the analysis. This option is very useful when CASPER is not able to found a convergent solution:

  ```
  DEBUG()
  ```

- STEADY: it is the same DEBUG analysis, but now you only can view the convergent solution:

  ```
  STEADY()
  ```

- TRANSIENT: for transient analysis. We must indicate at which integration should stop an the interval to produce output results of simulation:

  ```
  TRANSIENT(to, tf, delta_t)
  ```

  For example, if we want to analyze in transient our problem from 0 up to 30 seconds, using an interval of 1 second, the syntax should be:

  ```
  TRANSIENT(0, 30, 1)
  ```

7.2.7Output file

CASPER creates a file in binnary code that contains the data results from the analysis. We must specify the name of this file using the following syntax:

```
REPORT=”file.bin”
```

7.2.8Other keywords

- `exit:` stops the program.

7.2.9Example

Now we will see an example of COMPO file. How we have seen, the closure equations can not be used like equations within the *Continuous* block because the global equation system that results should be singular.

For the generation of COMPO file we must know our model from beginning to end. Now, we will create an electric library called ELEC and we will show you how to write the COMPO file for our model. It will have three components and one port type.

a) Creation of components: *battery, resistor, inductor*

File: `battery.h`

```
#ifndef _BATTERY_H_
#define _BATTERY_H_
#include <GENERAL.h>
#include "elec_ports.h"

class Battery : public Component {

      public:
            Battery();
            elec_p e1;
            elec_p e2;

      protected:
            double Vb;

      public:
            void Continuous();
            void Discrete(){}

};

#endif
```

File: `battery.cpp`

```
#include "battery.h"
```

```
Battery::Battery() {

      //Port definitions
      AddPort(&e1, "e1", "Electrical point 1", INPUT);
      AddPort(&e2, "e2", "Electrical point 2", OUTPUT);

      //Data definitions
      AddData(&Vb, "Vb", DOUBLE, "V", "Voltage");
      Vb = 0;
}

void Battery::Continuous() {

      e2.I = e1.I;
      e2.V = e1.V + Vb;

}
```

File: `resitor.h`

```
#ifndef _RESISTOR_H_
#define _RESISTOR_H_
#include <GENERAL.h>
#include "elec_ports.h"

class Resistor : public Component {

      public:
            Resistor();
            elec_p  e1;
            elec_p  e2;

      protected:
            double R;   //Resistance (ohm)

      public:
            void Continuous();
            void Discrete(){}

};

#endif
```

File: `resistor.cpp`

```
#include "resistor.h"


Resistor::Resistor() {

      //Port definitions
      AddPort(&e1, "e1", "Electrical point 1", INPUT);
      AddPort(&e2, "e2", "Electrical point 2", OUTPUT);
```

```
        //Data definitions
        AddData(&R, "R", DOUBLE, "ohm", "Resistance");
        R = 0;
}

void Resistor::Continuous() {

        e2.I = e1.I;
        e2.V = e1.V - e1.I*R;

}
```

File: `inductor.h`

```
#ifndef _INDUCTOR_H_
#define _INDUCTOR_H_
#include <GENERAL.h>
#include "elec_ports.h"

class Inductor : public Component {

        public:
                Inductor();
                elec_p e1;
                elec_p e2;

        protected:
                double L;
                double Vind;
                double Il;
                double dIdt;

        public:
                void Continuous();
                void Discrete(){}

};

#endif
```

File: `inductor.cpp`

```
#include "inductor.h"

Inductor::Inductor() {

        //Port definitions
        AddPort(&e1, "e1", "Electrical point 1", INPUT);
        AddPort(&e2, "e2", "Electrical point 2", OUTPUT);

        //Data definitions
        AddData(&L, "L", DOUBLE, "--", "Inductance");
        L = 0;
        //Unknown definitions
```

```
	AddUnknown(&Vind, "Vind", "V", "Inductor voltage");
	AddUnknown(&dIdt, "dIdt", "V", "Inductor voltage");
	dIdt=0;

	//Derivate definitions
	AddDerivate(&dIdt, &Il, "dIdt", "Il", "A", "Derivate of Il");

}

void Inductor::Continuous() {

	dIdt = Vind/L;

	e2.I = e1.I;
	e2.V = e1.V - Vind;

}
```

b) Creation of ports: *elec_p*

File: `elec_ports.h`

```
#ifndef _ELECPORTS_H_
#define _ELECPORTS_H_
#include <GENERAL.h>


class elec_p : public Port {

	public:
		elec_p();
		double V;
		double I;
	public:
		void Continuous();
};

class Node : public Component {

	public:
		Node();
		elec_p node;
		void Continuous(){}
		void Discrete(){}

};

#endif
```

File: `elec_ports.cpp`

```
#include "elec_ports.h"

elec_p::elec_p() {
      AddInformation(&V, "V", DOUBLE, "V", "Electric potential");
      V = 10;
      AddInformation(&I, "I", DOUBLE, "A", "Intensity");
      I = 0;

}

void elec_p::Continuous() {

      double I_sum = 0;

      for(int i=0; i<ENTRY_SIZE; i++) {
            if(!i) {
                  I = GET_ENTRY_VAL(I, i);
                  V = GET_ENTRY_VAL(V, i);
            }
            else
                  I += GET_ENTRY_VAL(I, i);
      }

      for(int i=0; i<EXIT_SIZE; i++) {
            SET_EXIT_VAL(V, i, V);
            if(!i)
                  I_sum = I;
            if(i!=(EXIT_SIZE-1))
                  I_sum -= GET_EXIT_VAL(I, i);
            else
                  SET_EXIT_VAL(I, i, I_sum);
      }

}

Node::Node() {

      AddPort(&node, "node", "Electric node", NODE);

}
```

c) Setting the library

File: `Interface.cpp`

```
//GENERAL INCLUDE
#include <GENERAL.h>
#include <INTERFACE.h>

//GLOBAL LIBRARY INCLUDE
#include "resistor.h"
#include "elec_ports.h"
#include "inductor.h"
#include "battery.h"

LIBRARY(ELEC, "ELECTRICAL Library")
      component(ELEC, Battery)
      component(ELEC, Resistor)
      component(ELEC, Inductor)
      component(ELEC, Node)
END_LIBRARY
```

d) Compiling the library

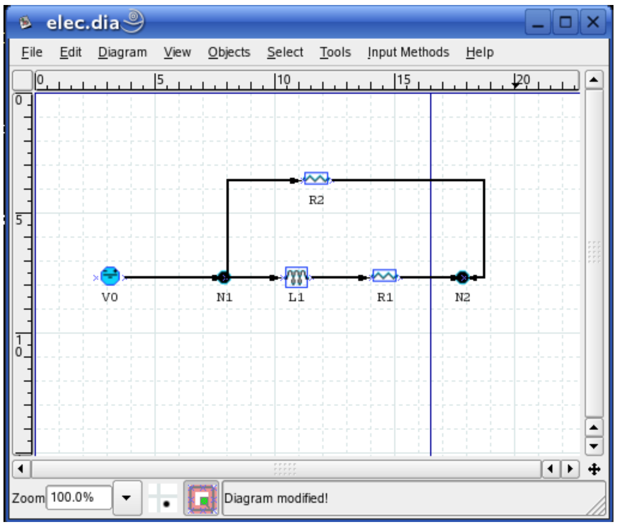
e) Creation of *sheet* file

File: `ELEC.sheet`

```
<?xml version="1.0" encoding="UTF-8"?>
<sheet xmlns="http://www.lysator.liu.se/~alla/dia/dia-sheet-ns">
  <name>ELEC</name>
  <description>ELECTRIC library</description>
  <contents>
    <object name="ELEC - Link">
      <description>Link</description>
      <icon>Link.xpm</icon>
    </object>
    <object name="ELEC - Battery">
      <description>ELEC Battery</description>
        <icon>ELEC/battery.xpm</icon>
     </object>
    <object name="ELEC - Resistor">
      <description>ELEC Resistor</description>
        <icon>ELEC/resistor.xpm</icon>
    </object>
    <object name="ELEC - Inductor">
      <description>ELEC Inductor</description>
        <icon>ELEC/inductor.xpm</icon>
    </object>
    <object name="ELEC - Node">
      <description>ELEC Node</description>
        <icon>ELEC/node.xpm</icon>
    </object>
  </contents>
</sheet>
```

f) Modelling with dia

File: `elec.dia`


elec.dia
File Edit Diagram View Objects Select Tools Input Methods Help
R2
V0
N1
L1
R1
N2
Zoom 100.0%
Diagram modified!

Remember that you have to introduce the data values double-clicking on every component.

g) Creation of COMPO file. Now we must decide which the boundary conditions are. Viewing our model, the BC should be: V0.e1.V and V0.e1.I. In function of these, we have to write the closure equations. Thus our COMPO file should be as follows:

```
MODEL="elec.dia"

UNKNOWNS = 4  {
1      V0.e1.V           0
2      V0.e1.I           0
3      N1.I              0
4      L1.Vind           0
}

LIMITS = 0  {
}

EQUATIONS {
1      s (R2.e2.V-R1.e2.V)/100
2      s (N2.V-V0.e1.V)/100
3      s (V0.e1.V-TIERRA)/100
4      s (L1.dIdt)/10
}

TIERRA = 0

REPORT="elec05.bin"
STEADY()

V0.Vb  = 20
L1.Il  = L1.e1.I

EQUATIONS {
4      s (L1.Il - L1.e1.I)/1
}

TRANSIENT(0,1,0.1)
```

Note that the equation number 4 has been modified in transient mode.