

---

# Design and implementation of a technology file interface for SPACE

---

*Xander Burgerhout*

*Circuits and Systems group  
Department of Electrical Engineering  
Delft University of Technology  
Mekelweg 4, 2628 CD Delft  
The Netherlands*

Delft, January 21, 2001

Technical Report no. : N&S-A-2003  
Mentor : dr. ir. N. P. van der Meijs



Delft University of Technology

---

# Preface

This report and the application it describes is the result of a graduation project. This graduation project has been performed for (and at) the Circuits and Systems group which is part of the faculty of Electrical Engineering at Delft University of Technology.

Two important parts of this report are the design and implementation of the architecture and a sort of a reference guide to the configuration file language. Readers interested in the architecture of the application should read at least Chapter 3. Those more interested in the configuration file language are referred to Chapter 5.

Finally, I would like to thank dr. ir. Nick van der Meijs for his guidance, opinions and advice during the project.

Delft, January 2001

X.B.

---

# Contents

<b>Preface</b>	<b>i</b>
<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>7</b>
<b>2 Application requirements</b>	<b>9</b>
2.1 Application environment constraints . . . . .	9
2.1.1 Platform . . . . .	9
2.1.2 Integration . . . . .	9
2.2 Functional demands . . . . .	10
2.2.1 Technology file generation . . . . .	10
2.2.2 Flexibility . . . . .	10
2.3 Documentation demands . . . . .	10
2.4 Additional wishes . . . . .	10
<b>3 Design and implementation</b>	<b>11</b>
3.1 Application architecture . . . . .	11
3.1.1 The problem . . . . .	11
3.1.2 The solution . . . . .	12
3.1.3 General overview . . . . .	13
3.2 User interface architecture . . . . .	14
3.2.1 The framework . . . . .	14
3.2.2 The generated user interface . . . . .	15
3.3 Design methodology and conventions . . . . .	16
3.4 The parser . . . . .	16
3.5 Components and the component tree . . . . .	18
3.5.1 Structure . . . . .	18
3.5.2 CComponent and CComponentTree abilities . . . . .	18
3.5.3 Component tree creation . . . . .	20
3.5.4 Iterating through the component tree . . . . .	20
3.6 The user interface builder . . . . .	20
3.6.1 The CGUIBuilderVisitor . . . . .	21
3.6.2 The CGUITree . . . . .	22
3.7 Data connections . . . . .	23
3.8 Technology file generation . . . . .	24
3.8.1 File generation datastructures . . . . .	25
3.8.2 The generation process . . . . .	26

## CONTENTS

---

3.8.3	Integration with the SPACE process tree . . . . .	26
3.9	Serialization . . . . .	27
3.9.1	Serialization method . . . . .	27
3.9.2	Compatibility issues . . . . .	27
3.9.3	Implementation . . . . .	28
3.10	The application . . . . .	28
3.11	Recommendations for future features . . . . .	29
3.12	Graphical overview . . . . .	30
<b>4</b>	<b>User interface elements</b>	<b>33</b>
4.1	Choosing a user interface toolkit . . . . .	33
4.2	Qt native user interface elements . . . . .	34
4.2.1	Labels and edits . . . . .	34
4.2.2	Dropdowns and comboboxes . . . . .	34
4.2.3	Listview . . . . .	35
4.2.4	The color select dialog box . . . . .	35
4.3	Custom user interface elements . . . . .	36
4.3.1	The scrollframe widget . . . . .	36
4.3.2	The section widget . . . . .	36
4.3.3	The spreadsheetview widget . . . . .	37
4.3.4	The condition list dialog . . . . .	39
4.3.5	The dali colorpicker dialog . . . . .	40
<b>5</b>	<b>The configuration file language</b>	<b>43</b>
5.1	Language design and overview . . . . .	43
5.1.1	Structuring of components . . . . .	43
5.1.2	Component types . . . . .	45
5.1.3	Properties . . . . .	45
5.1.4	Generators . . . . .	46
5.1.5	Macro definitions . . . . .	46
5.1.6	Reducing indentation . . . . .	48
5.1.7	Data connections . . . . .	50
5.2	Language overview for user interface elements . . . . .	50
5.2.1	Components in a dropdown or combobox context . . . . .	51
5.2.2	Components in a spreadsheet context . . . . .	52
5.2.3	Components in a paramlist context . . . . .	52
5.2.4	Components in a section context . . . . .	54
5.2.5	Components in a scrollframe context . . . . .	54
5.2.6	Components in a tabpage context . . . . .	55
5.2.7	Overview of combinations . . . . .	55
5.3	Language overview for generators . . . . .	57
5.3.1	Generator components . . . . .	58
5.3.2	Value mappings . . . . .	58
5.3.3	Literal text . . . . .	59
5.3.4	Identifier substitution . . . . .	59
5.3.5	Number addition and subtraction . . . . .	60
5.3.6	Loops . . . . .	60
5.3.7	Conditionals . . . . .	61
5.4	Recommended language extensions . . . . .	62
5.4.1	Language extensions . . . . .	62

## CONTENTS

---

5.4.2	New types and properties . . . . .	62
<b>6</b>	<b>Conclusion</b>	<b>65</b>
	<b>Bibliography</b>	<b>67</b>
<b>A</b>	<b>Development environment</b>	<b>69</b>
A.1	Platform and compiler . . . . .	69
A.2	Makefile generation . . . . .	69
A.3	Doxygen . . . . .	70
A.3.1	Possible output formats . . . . .	70
A.3.2	Doxygen options . . . . .	70
A.3.3	Collaboration diagrams . . . . .	70
A.4	Qt Designer . . . . .	71
<b>B</b>	<b>Testing and debugging the application</b>	<b>73</b>
B.1	The test – debug – develop cycle . . . . .	73
B.2	Debug techniques used . . . . .	74
B.2.1	Unexpected behaviour . . . . .	74
B.2.2	Locating segmentation faults . . . . .	75
B.2.3	Signal/slot debugging . . . . .	75
<b>C</b>	<b>About the CD-ROM</b>	<b>77</b>
<b>D</b>	<b>A sample configuration file</b>	<b>79</b>

## CONTENTS

---



# Abstract

The current version of the layout-to-circuit extractor SPACE does not have a user friendly interface to add new process descriptions or change existing descriptions. This is currently a specialized job.

This report describes the design and implementation of a user-friendly tool that can be used to add and manage the technology files that describe a process.

Flexibility is an important requirement. It must be possible to easily add new technology files to the interface. Another important requirement is platform independence. It must be possible to use the platform on many different flavors of Unix.

To meet the demand of flexibility the application uses a configuration file in which the user interface and the format of the technology files is specified. The developed application uses the information in the configuration file to create a user interface and to generate the required technology files.

A parser has been created that can read the configuration file and build a graphical user interface from the information it encounters. The values entered by the user into this user interface are then used by the generators specified in the configuration file to generate the requested technology files.

The Qt toolkit from TrollTech ensures platform independence of the graphical user interface. This popular toolkit is available for many types of Unix, Linux, Windows and even for embedded environments.



# List of Figures

3.1	General view of how the application performs its task . . . . .	13
3.2	Partitioning and subpartitioning. The tabpages represent the first level of partitioning. The sections (the blue horizontal bars) provide the subpartitioning. . . . .	16
3.3	The path from configuration file to component tree and generator data structures. On the right side the files containing the implementation. . . . .	17
3.4	CComponentTree collaboration graph. . . . .	18
3.5	Partial collaboration diagram for class CGUIBuilderVisitor . . .	23
3.6	Data connection collaboration diagram. . . . .	23
3.7	Example of Qt's signal/slot mechanism. . . . .	24
3.8	Overview of the file generation process. . . . .	25
3.9	The technology file generation dialog box. . . . .	26
3.10	The application in action. . . . .	29
3.11	Complete architecture overview . . . . .	30
3.12	Class diagram as reverse engineered by Rational Rose 2000 . . .	31
4.1	Two label widgets and one edit widget. . . . .	34
4.2	A combobox widget. . . . .	35
4.3	A listview widget used to display files. . . . .	35
4.4	The Qt color selection dialog box. . . . .	35
4.5	A few section widgets. . . . .	36
4.6	The spreadsheetview widget. . . . .	38
4.7	CSpreadSheetView collaboration diagram. . . . .	38
4.8	CSpreadSheetView class diagram. . . . .	39
4.9	The conditionlist editor dialog box. . . . .	40
4.10	The dali colorpicker dialog box. . . . .	41
5.1	Structuring example . . . . .	44
5.2	Dropdown example result. . . . .	52
5.3	Spreadsheet example result. . . . .	53
5.4	Parameter list example result. . . . .	53
5.5	Section example result. . . . .	54
5.6	Scrollframe example result. . . . .	55
B.1	The test – debug – develop cycle . . . . .	74



# List of Tables

3.1	Partial list of methods provided by the CProcessManager class. .	15
3.2	Visitors and their purpose . . . . .	21
3.3	CGUIBuilderVisitor build methods. . . . .	22
3.4	Possible data targets. . . . .	24
3.5	Methods provided by the CGenerators class . . . . .	26
3.6	Important methods provided by the CSerializerVisitor class . . .	28
5.1	Classification of components. . . . .	51
5.2	Combinations of components and properties in a context. . . . .	56
5.3	Supported fields: . . . . .	60



# Chapter 1

## Introduction

A layout-to-circuit extractor like SPACE<sup>1</sup> must (of course) be able to handle all kinds of processes. In SPACE, each of these processes is described by a set of *technology files*. The contents of these files vary from simple layer specification to complex capacitance tables.

Until now, the technology files were maintained by hand. This means that for each process all the technology files had to be entered manually, a time-consuming and error-prone task. Also, the user must be familiar with the file format of each of the technology files, which makes process entry and maintenance a specialized task.

*SPOCK* is to alleviate the problems described above by presenting the user with a graphical user interface in which the required data can be entered.

The main goal of this thesis is to identify the problems encountered during the design and implementation of SPOCK and to describe the solutions that were used to solve these problems.

Chapter 2 presents the demanded functionality of the application as well as some imposed constraints. Chapter 3 discusses and explains the final design. The user interface plays an important role in this application. Some standard and custom user interface elements are therefore presented in Chapter 4. Chapter 5 describes the configuration file language. As will become clear later, the configuration file describes the technology files supported by the application. The final chapter will contain some concluding remarks.

---

<sup>1</sup>*SPACE* is the layout-to-circuit extractor developed at the Circuits and Systems group of the Electrical Engineering faculty. SPACE has been quite successful so far. It is used by several companies and universities, including Agilent (Hewlett Packard) and Level One (Intel).





## Chapter 2

# Application requirements

As a first step in the design process, the demands and constraints for the application must be charted. In this chapter the most important demands and constraints are given.

Firstly, the constraints resulting from the environment in which the application must function are enumerated in Section 2.1). Secondly, the functional demands are listed in Section 2.2. Documentation is also important: demands related to documentation can be found in Section 2.3. Some general wishes are mentioned in Section 2.4.

## 2.1 Application environment constraints

### 2.1.1 Platform

It must be possible to compile the application on various platforms since SPACE can also be compiled on various platforms. The following platforms must be supported:

- Solaris
- HP UX
- Linux

Any input and/or output files that are read or written by the application must also be platform independent. This means for example, that it must be possible to read a file saved with the Solaris version into the Linux version and vice-versa.

### 2.1.2 Integration

The application must integrate correctly with the rest of the SPACE package. This mainly means that the application should honor the directory structure and the environment settings used by SPACE.

## 2.2 Functional demands

### 2.2.1 Technology file generation

It should be possible to generate technology files with a plain-text file format. There must also be a mechanism that will allow the addition (or removal) of technology file formats. As a proof of concept the application must be able to generate the following subset of technology files:

- *maskdata*, which defines the layers present in the process and the colors used to represent them in the programs and their output.
- *space.xxx.s*, the element definition files used by SPACE.
- *space.xxx.p*, the parameter files used by space.
- *bmlist.gds*, which provides a mapping between the GDS layer format and the format used by SPACE.
- *xspicerc*, a control file that specifies which models are used for the devices.

### 2.2.2 Flexibility

Flexibility is very important. Recompilations as a result of adding, changing or removing a technology file format should be kept to a minimum to decrease the maintenance cost of the application.

## 2.3 Documentation demands

The following documentation must be created:

- Source code documentation
- Installation manual
- Programmers manual
- Maintainers manual

## 2.4 Additional wishes

The user interface must be consequent and consistent. This means the user interface should conform to the currently accepted standard.

A “help” facility would be welcomed. It should at least be possible to show “tooltips” (popup hints).

If possible, the application will be implemented in C++.

## Chapter 3

# Design and implementation

Designing and implementing is often an iterative process. No matter how detailed your design is, chances are that you will still encounter problems during implementation that will require a design change. In this chapter, design and implementation issues are therefore presented together.

The main goal of this chapter is to present the entire design and the related implementation issues and choices.

In the first section (Section 3.1) of this chapter the architecture of the application will be presented. Next, the architecture of the user interface will be explained. This will not include the user interface elements (see Chapter 4 for that) but will be about the organization of the user interface. The main parts of the design will be explained in detail in Section 3.4 to Section 3.9.

Before the chapter is concluded some recommendations for future extensions are given in Section 3.11. A graphical overview of the implemented design can be found in the final section of this chapter.

### 3.1 Application architecture

First, the problems related to a traditional architecture are discussed. Next, a solution to these problems is presented.

#### 3.1.1 The problem

The main function of the application can be described quite easily. The user enters data into the application in a user-friendly way. The application then takes this data and generates the more complex files needed by SPACE. This means the user interface and the data are strongly related.

The traditional way to implement this kind of application is to use a rapid application development (RAD) tool like X-designer<sup>1</sup> or Qt-designer<sup>2</sup>. The user interface can visually be created with these applications.

---

<sup>1</sup>X-designer allows a programmer to visually create a user interface based on Motif. More information on X-designer can be found on <http://www.ist.co.uk>.

<sup>2</sup>Qt-designer allows a programmer to visually create a user interface based on the Qt toolkit. More information can be found on the Qt-designer section of TrollTech [Tro] <http://www.trolltech.com/products/qt/designer/index.html>.

As was mentioned before, flexibility is a very strong demand. Recompilations resulting from the actions listed below should be kept to a minimum:

- make changes to existing files
- add new files to the generation process
- remove files from the generation process

This requirement has some consequences though. In these cases, using a tool like X-designer or Qt-designer has the following disadvantages:

- A change in the file format of one of the technology files to be generated requires changes to the user interface. This means that source code has to be changed manually or be regenerated when using tools like X-Designer or Qt-Designer.
- A recompilation is always necessary. Even if the user interface can be left unchanged, at least the generating algorithm must be updated.
- User interface and file generation information is distributed over many files, making maintenance non-trivial.

Recompilation as a result of a change in the format of one of the files to be generated would dramatically increase the maintenance costs required by the application, so a solution should be found for the problems mentioned above.

The second problem mentioned in the list above can be avoided by using a file which specifies the generation algorithm. This is useful in the case that only the format of a technology file changes. It hardly reduces the need for a recompilation however. A change in the format of a technology file (new features e.g.) is far more likely to occur than a change in the generating algorithm. It also does not solve the problem of adding files to the generation process.

RAD tools like X-designer and Qt-designer speed up the development of the user interface and introduce a certain flexibility. However, in this case they do not introduce enough flexibility, because a recompilation is still very likely to be needed in the case of adding, removing or changing technology file formats. The approach presented in the next section does not have these disadvantages.

### 3.1.2 The solution

To solve the problems mentioned in the previous section we need a different approach. Instead of using a design tool like X-designer or Qt-designer, we can use a configuration file.

The configuration file will describe the user interface, the technology file format descriptions and the relationship between them. This configuration file will then be parsed by the application. This approach has the advantages we are looking for:

- Adding, removing or changing a technology file format no longer results in the need for a recompilation of the application. Only the configuration file needs to be updated.
- The user interface and technology file format descriptions are contained in one file. This means that maintenance is also limited to one file.

The next (sub)section will give a general overview of the resulting architecture.

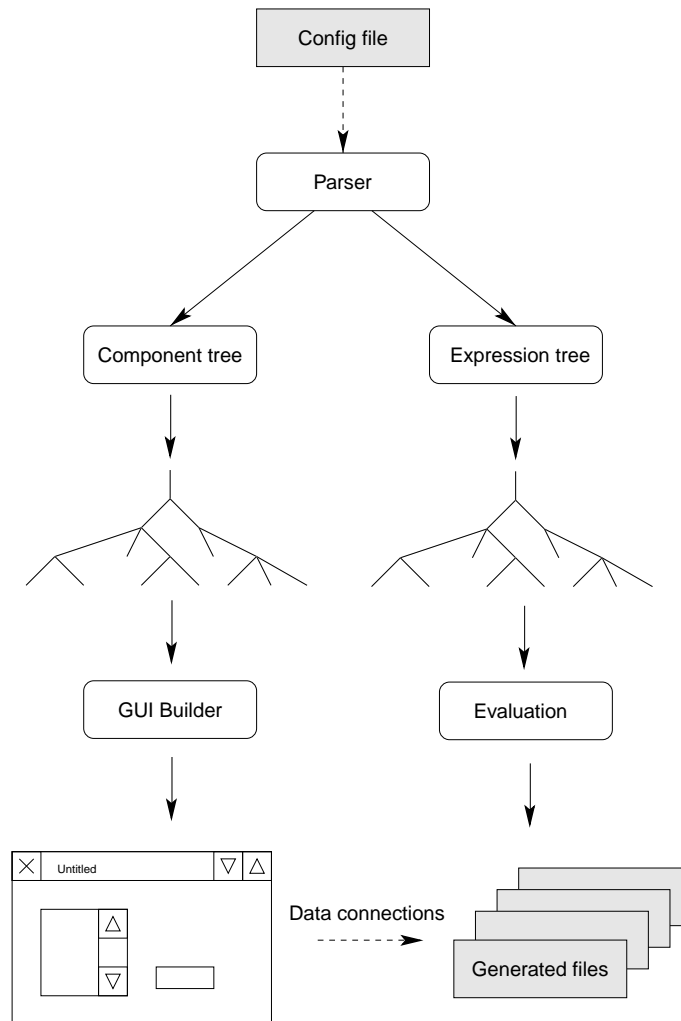


Figure 3.1: General view of how the application performs its task

### 3.1.3 General overview

The solution as presented in Section 3.1.2 leads to the architecture presented in Figure 3.1. The main function of the application — generating files based on some configuration and the users input — is quite clear in this picture. The entire process can be summarized as follows:

- Read and parse the configuration file containing a user interface description and technology file format specifications.
- Build the the technology file generators and build the user interface using a “component tree”.
- Use the generators and the data entered in the user interface to generate the required files.

First, the configuration file is read. The configuration file contains a description of the user interface as well as information on how to generate the required technology files. The user interface description actually specifies *what* data the user must enter and also *how* the user must do this. The technology file generator specification contains literal pieces of files as well as references to elements in the user interface that will be substituted with entered values. For more information on the format of this file see Chapter 5.

To efficiently build the user interface, a component tree is first created. This component tree is then used to build the actual Qt widgets<sup>3</sup>. A precise description of this process can be found in section 3.6.

The last step is file generation. To be able to generate the technology files, data is collected from the user interface and fed into the file generators, resulting in the required files.

## 3.2 User interface architecture

The user interface can be separated into two parts. The first part handles opening and closing files and file generation requests. It also provides an interface for the second part. This first part will be called the “framework”.

The second part of the user interface is the part of the user interface that is described in the configuration file. This part specifies what data must be entered (and how) to be able to generate a certain file. This second part will be called the generated user interface.

### 3.2.1 The framework

The framework used is just an implementation of the well-known multiple document interface (MDI). The framework provides the actions the user expects to find in a multiple document interface:

- Creating a new file
- Opening (loading) of a file
- Saving of a file
- Closing of a file
- Switching between open files

If we translate this abstraction to our case, the “file” is a process description. Switching between files means that the user would like to start editing on another process. The actions provided are grouped together in the “File” menu, as is common practice. The interested reader is referred to an excellent book on user interface standards by Fowler and Stanwick [FS98].

In the implementation, the multiple document interface can mainly be found in three classes: `CAppMainWindow`, `CProcessManager` and `CProcess`. The `CAppMainWindow` class receives the signals given by the user whenever he selects one of the actions mentioned before. The `CProcessManager` class methods are

---

<sup>3</sup>Widgets are the basic elements a user interface consists of, e.g. buttons, listboxes, checkboxes, etc.

then called to perform the actions. This usually involves an operation on a `CProcess` class instance. All instances of the `CProcess` class are “registered” and managed by the `CProcessManager` class. The most important methods provided by the `CProcessManager` class are listed in Table 3.1. The `CProcess` class will be discussed again in Section 3.8, since it is also part of the file generation process.

### 3.2.2 The generated user interface

The part of the user interface that is specified in the configuration file is open to change. However, we must specify how this part of the user interface integrates with the framework.

The most straightforward approach is to put all the user interface elements specified inside the client area (the application window minus the menus, toolbar and status bar). However, this is not a very practical approach. The amount of data that must be entered and thus the number of input fields in the user interface is large. This results in a too big and thus obscure scrollable area.

A solution is to make it possible to “partition” the user interface. Each of these parts can correspond to a file, or a certain aspect of the process description. For example, a part can contain all information about the layers in a process.

These “parts” can still contain much data that must be entered. The element definition file is good example of this. It would be better if we partition each part even further. An effective user interface must thus be able to handle partitions with sub-partitions.

The big question is now how we can achieve this. The first rough partitioning that corresponds to the files can be implemented using “tabpages”. Each tabpage will correspond to a file or a general aspect of the process description. The elements on the tabpage will describe the data that must be entered.

As mentioned before, it would be profitable if we could further partition a tabpage. Some files just will not fit on a tabpage. For this purpose, a custom widget named the “section” widget was designed. It is described in detail in Section 4.3.2. With this widget, a tabpage can be divided into multiple sections,

Table 3.1: Partial list of methods provided by the `CProcessManager` class.

Method	Description
<code>currentProcess()</code>	Gives a pointer to the process currently being edited by the user.
<code>newProcess()</code>	Creates a new process and makes it the current process.
<code>activateProcess()</code>	Switches to a new process.
<code>removeProcess()</code>	Closes and removes the process from the process manager.
<code>saveProcess()</code>	Saves a process to disk.
<code>loadProcess()</code>	Loads a process from disk.
<code>generateFile()</code>	Generates the requested technology file for this process.

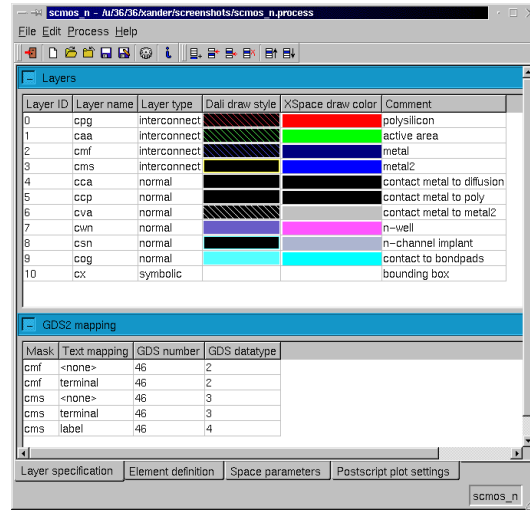


Figure 3.2: Partitioning and subpartitioning. The tabpages represent the first level of partitioning. The sections (the blue horizontal bars) provide the subpartitioning.

that can be shown either expanded or collapsed. A collapsed section hides a part of its interface. This mechanism allows for a much clearer arrangement of the user interface. Figure 3.2 shows the tabpages and sections in action.

### 3.3 Design methodology and conventions

Before each part of the design is discussed some remarks about the design methodology and the used conventions are in place.

The *Design Patterns* as described by Erich Gamma et al [GHJV95] are used in the discussion of several parts of the design. These design patterns provide a common vocabulary for computer program designers and therefore references to these patterns will occur throughout the text.

Doxygen was used to generate the source code documentation. It is also capable of generating so-called collaboration graphs, although the term collaboration graph can be a bit misleading. The collaboration graphs generated by **doxygen** are a mix of what are called class diagrams and collaboration diagrams in some well-known notations like Booch or UML.

In several places the collaboration graphs generated by **doxygen**[vH] are used. Doxygen is a source code documentation generator. The term

### 3.4 The parser

To be able to read and interpret the configuration file, a parser has been created. The parser consists of three parts (as shown in Figure 3.3):



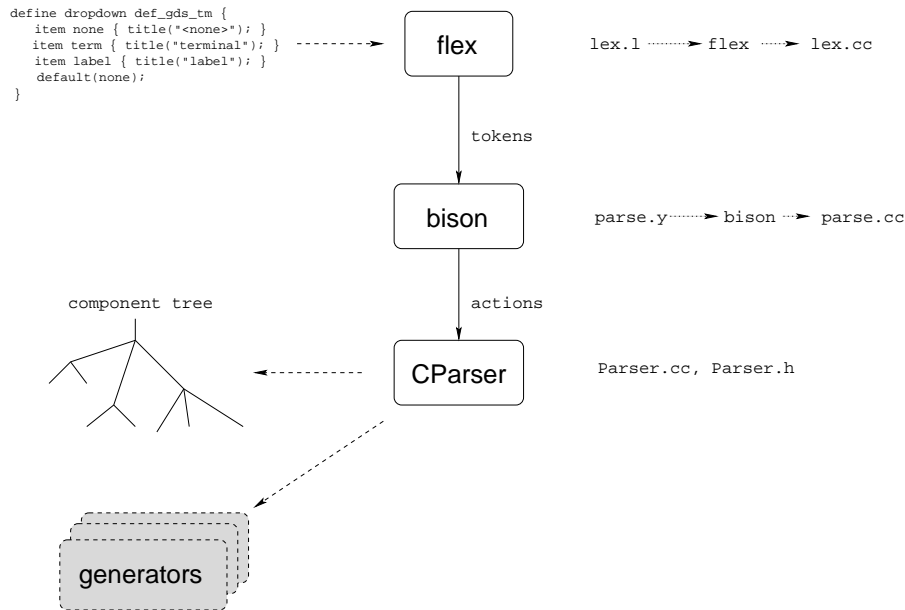


Figure 3.3: The path from configuration file to component tree and generator data structures. On the right side the files containing the implementation.

1. a lexical analyzer
2. a grammar parser
3. a class that provides methods for the parser to build the component tree and the generators.

The lexical analyzer scans the configuration file for keywords and other special language elements like string delimiters and parentheses. The lexical analyzer used is generated by *flex*<sup>4</sup>. Recognized keywords and elements are called *tokens*.

The lexical analyzer is used by the grammar parser. This grammar parser can recognize and analyze a series of *tokens*. It can then perform actions related to the sequence of tokens it encountered. The grammar parser is generated with *bison*<sup>5</sup>. For more information about the language used in the configuration file, please refer to Chapter 5.

The actions that the parser performs are the methods provided by the parser class, **CParser**. The **CParser** class builds the component tree and the generators, which are described in the following sections. The **CParser** class also extends the functionality of the parser generated by bison:

- Disambiguation of identifiers specified in the configuration file. If ambiguities arise, these are reported.

<sup>4</sup>flex is a lexical analyzer based on and compatible with lex.

<sup>5</sup>bison is a parser based on and compatible with the well-known grammar parser yacc.

- The `CParser` class also acts as a *builder*<sup>6</sup> for the component trees and generators.
- The `CParser` class is implemented as a *singleton*<sup>7</sup>. This ensures that only one instance of the parser will be created and that this instance will be easily accessible.
- Errors in the configuration file are reported in a dialog box instead of on the console. The line number of the line(s) containing the error(s) are also reported.

## 3.5 Components and the component tree

The parser described in the previous section creates a tree, as is illustrated in Figure 3.1 and Figure 3.3. This tree is called the component tree and consists of instances of the `CComponent` class.

In the following sections, the structure and abilities of the component tree will be explained.

### 3.5.1 Structure

The tree is based on the *composite* pattern, as described in Gamma et al [GHJV95]. The difference between the composite pattern and the structure used here is the lack of the “leaf” nodes. Only the “composite” nodes are used in the component tree. The reasons for this are simple:

- The number of nodes in the component tree is relatively small. The argument of efficient memory usage by using separate classes for leaves and composite nodes is a weak one in this case.
- The complexity decreases, thereby simplifying implementation.

Root components are tied together in the `CComponentTree` class. The collaboration graph for the `CComponentTree` is shown in Figure 3.4

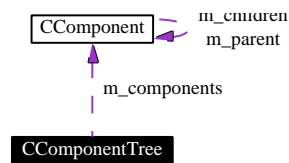


Figure 3.4: `CComponentTree` collaboration graph.

As can be seen in Figure 3.4 the `CComponentTree` contains references to the `CComponents` that are root components. The `CComponent` class contains references to `CComponents` that are either the parent component (`m_parent`) or child components (`m_children`).

<sup>6</sup>A simplified version of the *builder* pattern described by Gamma et al [GHJV95] is used here.

<sup>7</sup>The *singleton* pattern as described by Gamma et al [GHJV95].

### 3.5.2 CComponent and CComponentTree abilities

The `CComponent` class provides the following functionality for its clients:

- Name and context methods
- Property support
- Simple type support
- Parent/child access

**Name and context methods** control access to the components name. Components have names that function as an identifier. The names of the parent components concatenated with dots form the “context” in which the identifier lives. For example, if `A` is a parent of `B` and `B` is the parent of `C` then `A.B` is the context of `C` and `A.B.C` is the “full-context name” of component `C`.

Access to names and contexts is provided by the `setName()`, `getName()` and `getFullContextName()` methods.

**Property support** is something not provided by C++. C++ requires explicit set and get methods for class members. Properties would be very useful for the `CComponent` class. Many different types of components will be described by the `CComponent` class and properties would make this easier. Inheritance is also an option, but the number of classes required would be too large to implement in the given time.

Property support is therefore added to the `CComponent` class by providing methods like `setProperty()` and `getProperty()`. The properties are stored in the STL<sup>8</sup> container *map*. A property can have *multiple* values, because the values are stored as a vector of strings. For more implementation specifics, please refer to the source code documentation [Bur01].

**Simple type support** must be present to be able to differentiate between the component types. Type support is very crude, only methods like `setType()` and `getType()` are implemented. There is currently no support for type checking and type conversion.

**Parent/child access** and modification is also provided. The available methods include:

- `getParent()` retrieves a pointer to the parent component
- `add()` adds a child to this component
- `removeChild()` removes a child from the list of children
- `getChild()` can retrieve a pointer to a child component
- `numChildren()` and `childPos()` provide array-like access to the children.

Some of these methods have multiple implementations accepting different arguments, increasing the flexibility of this much-used class.

---

<sup>8</sup>STL is the Standard Template Library, which is part of the C++ standard. The STL provides containers, algorithms, strings, streams and many other utility classes.

### 3.5.3 Component tree creation

A prototype tree is created by the `CParser` class described in Section 3.4. The prototype tree is copied for every process that is created with “File → new” or loaded into the application. The reasons for this will be explained further in Section 3.6, where the user interface builder is described.

### 3.5.4 Iterating through the component tree

Iterating through the tree is necessary for building the user interface, searching components, printing debug information about the components, etc. Walking through the tree is such a common action that the interface should be open to extension.

The *visitor* pattern is applied to make the interface flexible. The visitor pattern allows us to easily apply an operation to the component tree. Some of the advantages of using the visitor pattern:

- The `CComponent` class is not polluted by methods and attributes related to a certain operation on the entire tree. This related behaviour is placed in the visitor class instead.
- The visitor pattern makes it easy to define new operations on the tree. Adding a new visitor is all that is required.
- Visitors can also accumulate state as they visit each component in the component tree. This is done for example in the `FindComponentVisitor`, which counts the number of hits.

The visitor pattern is described in Gamma et al. [GHJV95].

#### Available visitors

The available and by `CComponentTree` accepted visitors are listed in Table 3.2. The implementation of the `CFindComponentVisitor`, `CFindPropertyVisitor` and `CDumpComponentTreeVisitor` is straightforward and the interested reader is referred to the source code documentation.

`CSerializerVisitor` and `CGUIBuilderVisitor` are not so straightforward. The `CGUIBuilderVisitor` class is described in detail in Section 3.6. The `CSerializerVisitor` and serialization<sup>9</sup> in general is discussed in Section 3.9.

## 3.6 The user interface builder

So, after parsing the configuration file the component tree is created and ready to be processed further. The user interface builder creates the actual widgets (the basic user interface elements) from the component tree by visiting the tree with the `CGUIBuilderVisitor`.

The creation of the user interface is thus a two step process:

1. The component tree with the structure of the user interface is created from the configuration file.

---

<sup>9</sup>Serialization is the term used for the process of saving and loading application data.

Table 3.2: Visitors and their purpose

Visitor	Description
<code>CFindComponentVisitor</code>	Retrieves all components with a given name.
<code>CFindPropertyVisitor</code>	Retrieves all components that have the specified property. The value specified is compared with the value of the property considered. The comparison criterion can be specified as either equal or unequal. This could be extended with larger/smaller criteria in future versions.
<code>CDumpComponentTreeVisitor</code>	This visitor prints the (indented) tree to the console. This visitor is provided for debugging purposes.
<code>CGUIBuilderVisitor</code>	The GUI build process is described in detail in Section 3.6.
<code>CSerializerVisitor</code>	Provides means to load or save the information entered by the user and stored in this tree. This is discussed in Section 3.9

2. The `CGUIBuilderVisitor` visits the component tree and creates a `CGUITree` object that contains the created widgets and provides methods to access them.

One might wonder why the component tree does not contain the widgets and why the `CGUIBuilderVisitor` is needed. After all, all the information related to the user interface is already “in there”. The reasons behind this are actually quite straightforward:

- The hierarchy Qt uses for its widgets is different from the one used by the component tree. The Qt tree is very fine-grained compared with the component tree and incorporates layout information for the widgets.
- If we were to include the creation and management of the Qt widgets in the component tree, the `CComponent` class would get “bloated”. The `CComponent` class would serve two different purposes (`CComponent` parent/child management and Qt widget management), which is usually bad software engineering.

### 3.6.1 The `CGUIBuilderVisitor`

The `CGUIBuilderVisitor` first determines the type of the component being visited. It then calls the method needed to build that type of component. These (private) methods are listed in Table 3.3.

The widgets created are mapped to the components whenever this is desired. This mapping allows us to access the value a user enters in the widget using the `CGUITree`. The mapping is described in detail in the next section.

Table 3.3: CGUIBuilderVisitor build methods.

Method	Description
<code>buildTabPage()</code>	Creates a tabpage. A tabpage usually contains only a scrollframe.
<code>buildScrollFrame()</code>	Creates a scrollframe. A scrollframe can contain multiple widgets. If they do not fit inside the frame, scrollbars are displayed.
<code>buildSection()</code>	Creates a section. A section can contain multiple widgets, including nested sections.
<code>buildParamList()</code>	Creates an empty parameter list.
<code>buildSpreadSheet()</code>	Creates an empty spreadsheet.
<code>buildSpreadSheetColumn()</code>	Adds a column to the current spreadsheet.
<code>buildComboBox()</code>	Creates combobox and dropdown widgets.
<code>buildLineEdit()</code>	Creates a line edit.

### 3.6.2 The CGUITree

The `CGUITree` created by the `CGUIBuilderVisitor` provides a mapping between a component tree and the Qt widgets related to that tree. The methods provided reflect this.

**The mapping methods** allow creating a mapping and retrieval of values from the mapping. To create a mapping between a component and a widget `makeMapping()` can be used. To retrieve either a component given a widget or a widget given a component, `getComponent()` or `getWidget()` can be used. As a special case, `getTabPages()` retrieves all the tabpages in the gui tree.

**Value methods** are also implemented. These consist of get/set methods for normal components and get/set methods for spreadsheets. The spreadsheet methods take the column component as an extra argument. The methods are named `getValues()`, `setValues()`, `getSpreadSheetValues()` and `setSpreadSheetValues()`.

**Other methods** provided include methods for data connections, which will be discussed in Section 3.7. `findComponent()` is implemented as a convenience function. It calls the associated component tree's `findComponent()` method. Also implemented are some slot<sup>10</sup> methods needed by the spreadsheets. More about this in Section 4.3.3 where the spreadsheet widget is explained.

A partial collaboration diagram showing the classes discussed is presented in Figure 3.5.

<sup>10</sup>slots are part of Qt's signal/slot mechanism which allow sending and receiving signals to and from objects.

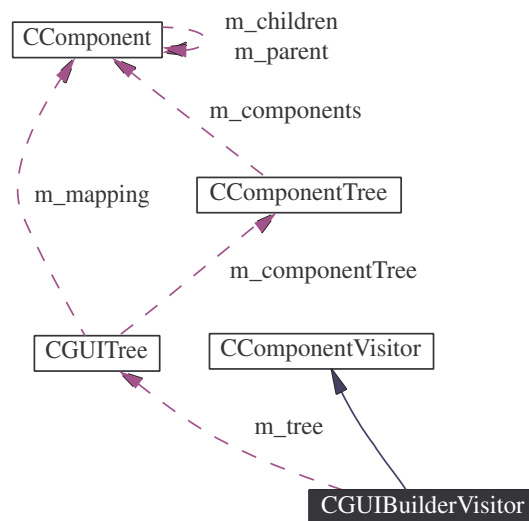


Figure 3.5: Partial collaboration diagram for class CGUIBuilderVisitor

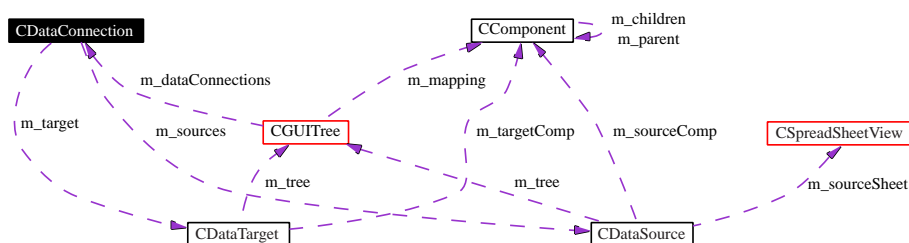


Figure 3.6: Data connection collaboration diagram.

### 3.7 Data connections

An important feature that partly determines the success of the chosen approach is the ability to use the data entered by the user in other fields. For example, in some part of the user interface the user must specify which layers exist in the application. In some other part, the user must map a layer to a GDS2 number. It would be convenient if the user could choose from the defined layers. This not only reduces the chance of (typing) errors, it also makes the user interface more comfortable to work with.

This dynamic behaviour is implemented by the data connection classes. A data connection consists of one or more data sources, a single data target and a data connection object that is responsible for the actual synchronization of values between the sources and the target. This is depicted in Figure 3.6.

The implementation relies heavily upon the Qt signal/slot mechanism. This mechanism implements the *observer* pattern [GHJV95]. The dynamics of this construction are expressed in Figure 3.7. The `CDataSource` object “detects” a change in the value it watches. It then signals the `CDataConnection` ob-

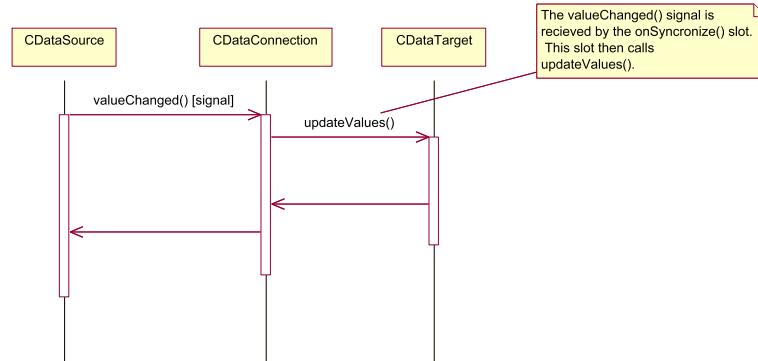


Figure 3.7: Example of Qt's signal/slot mechanism.

Table 3.4: Possible data targets.

Derived class name	Description
CColumnDataTarget	The data target is a column in a spreadsheet.
CComboDataTarget	The data target is a combobox like component
CConditionListDataTarget	The data target is a condition list.

ject that the target needs updating. The `onSynchronize()` method then calls `updateValues()` for the correct `CDataTarget` object.

Currently, there is only one type of data source supported. This is the column of a (custom) spreadsheet widget. The constructor checks if the source component specified is a column in a spreadsheet. The `getValues()` member is tailored to retrieve the values from a spreadsheet column.

Support for different types of data sources can easily be added. The methods involved are declared virtual. This means a new `CDataSource` derived class supporting the new type of source can easily be added.

The `CDataTarget` class already has some derived classes. They are listed (and explained) in Table 3.4.

## 3.8 Technology file generation

The file generation architecture strongly resembles the architecture used to build the user interface. Figure 3.8 presents this architecture. If we compare this process with the one that creates a user interface we can see a lot of similarities. Both start from a definition given in the configuration file. This definition is then parsed and a component tree is created.

The datastructures used for the file generation process are discussed next, followed by a description of the generation process itself.



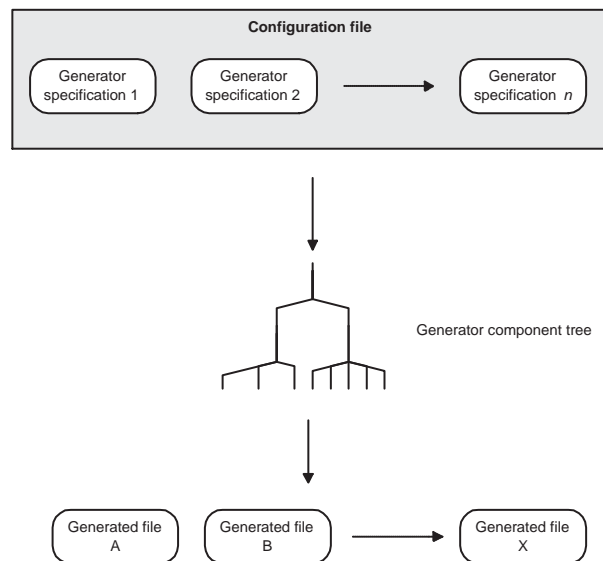


Figure 3.8: Overview of the file generation process.

### 3.8.1 File generation datastructures

As mentioned before, the format specifications defined in the configuration file are parsed and put into an expression tree. Like the component tree, this tree structure uses the `CComponent` class for its nodes. However, the tree interpretation is different. The nodes of the component tree represent a part of a hierarchy, while the nodes of the expression tree represent an action that should be performed (possibly involving the nodes' children).

For the file generator, the nodes in the tree can all be evaluated to plain text. The tree structure represents the more elaborate language constructs, like the `foreach` loops and `if` constructions. These constructs can affect the eventual result. More information on these language constructions can be found in Chapter 5.

The root components of the tree represent the generators. The class used for the root components is the `CGeneratorComp` component. This is a `CComponent` derived class. The added functionality lies in the support of value maps.

A value map allows the “translation” of dropdown item text to the text that should be generated instead. This mapping applies to all instances of the dropdown in that tree. In a future version of the application this behaviour could be extended to include other types of substitution as well.

The generators are collected in the `CGenerators` class. Currently, the interface only provides methods for access to the generators. They are listed in Table 3.5.

The `CGenerators` class could be extended with additional methods if the need for them arises.

Table 3.5: Methods provided by the CGenerators class

Method	Description
<code>addGeneratorComp()</code>	Adds a generator to the list of generators.
<code>numGenerators()</code>	Returns the number of generators.
<code>getGenerator()</code>	Retrieves the desired generator.

### 3.8.2 The generation process

Generating the files starts with the user bringing up the generation dialog box. As can be seen in Figure 3.9 each file can be generated separately. The result can be saved into a specified directory or into the SPACE process tree directly (if the user has the required rights). This is explained in detail in Section 3.8.3. Before the files are actually written to disk they can be inspected and edited if the user checks the “Edit and inspect results” option.

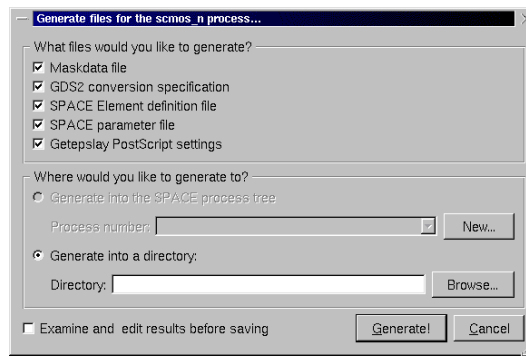


Figure 3.9: The technology file generation dialog box.

After all the required information is gathered, the `generateFile()` method from the `CProcess` class is called for each file that must be generated. This method evaluates the file’s generator tree to the file’s contents. The evaluation of the nodes is done by some private helper methods included in the `CProcess` class. They include the generation of literals, identifiers, numbers, binary operators like addition and subtraction, `if` constructions and `foreach` constructions. It is also possible to request some information from the application. These include process name and description and time and date of generation.

To allow nested `foreach` loops a dedicated class was written that contains the state of an iteration. The class is named `CIteratedState` and instances of this class are put into a stack by the `CProcess` class as necessary.

### 3.8.3 Integration with the SPACE process tree

As was mentioned in the previous section, it is also possible to generate the result directly into the SPACE process tree.

SPACE keeps the technology files for each process in a separate directory. The name of the directory is also the name of the process. The process direc-

tories all reside inside a directory that also contains the `processlist` file. This file contains a list of the processes mapped to a certain number. Changing the number of a process is potentially dangerous and should be avoided.

To generate a process into the SPACE process tree, an update of the `processlist` file is required. The user must either select the correct number or enter a new number for the process to be generated. To aid the user, the comments after each mapping are read and displayed together with the numbers and names of the processes in the `processlist` file.

If the user proceeds with the generation process, the `processlist` file is rewritten to disk. The entry of the generated process is added or replaced (depending on the users' choice for the process number). The comment after the new entry will contain the process description.

## 3.9 Serialization

With serialization we mean storing or retrieving the data describing a process to or from disk. The flexibility of the application makes this quite difficult however, as will be explained in the following sections.

### 3.9.1 Serialization method

The data that must be serialized is entered by the user in the user interface. The user interface is generated from the configuration file. This means that the method used to serialize the data must somehow include this information.

The obvious solution is to bundle the data and the user interface description present in the configuration file. However, this introduces a problem that is unacceptable. The whole idea of the chosen approach was to introduce flexibility. If we bundle the data and the user interface description, changes made to the configuration file are useless. The old configuration file that was bundled with the data will be used instead, thus effectively eliminating the introduced flexibility.

To prevent the problem described above, the data is stored as key-value pairs. The key is the full context name of the component associated with the data. The value is the actual value in the user interface. This ensures that every value can be restored, even if additions have been made to the configuration file.

Unfortunately, this approach has its defects as well. Defects that can largely be overcome though.

### 3.9.2 Compatibility issues

The proposed method of serializing the data works as long as the full context names of the components are still the same as in a newer configuration file. Since the full context name also contains information about how the user interface is arranged, moving parts of the user interface can be disastrous.

For example, in version 1 of the configuration file we have the component `A.B.C.D`. In version 2 we would like to have `C.D` somewhere else. Let us assume we want to move it to `A.E`, resulting in `A.E.C.D`. It is clear that the old version 1

Table 3.6: Important methods provided by the CSerializerVisitor class

Method	Description
<code>readConversion()</code>	While reading a file, converts an identifier-value pair from an old version to a new version. Currently does nothing.
<code>writeConversion()</code>	While writing a file, converts an identifier-value pair from this version to an old version. Currently does nothing.
<code>guessCorrectComponent()</code>	Tries to resolve relocated components.
<code>doFileInit()</code>	Reads or writes the file header. The header contains the process name and description.

files cannot be read by the version 2 configuration file, although no real changes in the data have occurred.

To solve this problem, a disambiguation method has been implemented. If the full context name cannot be resolved, context is added to the components name until a single match is found. Continuing the previous example, this would mean the search would start with D. If D is not unique, C is added to the context, resulting in C.D. In this case, A.E.C.D is the unique component found.

Renaming components between versions is another issue. Renaming can be solved by adding methods that can translate old version names to new version names. This means editing source. As a result, renaming components is an action that causes a major version change. This is acceptable, since renaming is mostly done because of the esthetics of the configuration file.

Adding and removing components is no problem. Adding a component means the component will be set to the default since it is not present in the old version. Removing a component means that the component cannot be found if an old version file is loaded. In this case, nothing happens.

### 3.9.3 Implementation

Because values are stored by their component name, a *visitor* can be used here. The visitor is called `CSerializerVisitor`. The most important methods are listed in Table 3.6.

## 3.10 The application

A screenshot of the application in action is shown in Figure 3.10.

As was already mentioned in Section 3.8.3, the technology files can be generated directly into the SPACE process tree. This is just one aspect of the relationship between the SPACE and the application.

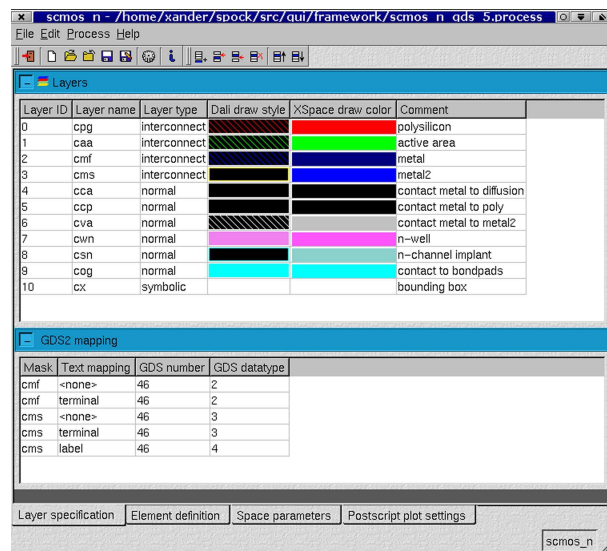


Figure 3.10: The application in action.

### Place of the configuration file

The configuration file should be present either in the directory the application was started from or in `$ICDPATH\lib\spock`. The filename should be `spock.uis`. If the file is not found in either location the application will exit. `$ICDPATH` is an environment variable that should point to the SPACE installation location.

### Existing technology files

Unfortunately, existing technology files cannot be read by the application. Only the processes entered in the application and saved in the applications' own file format can be read.

## 3.11 Recommendations for future features

Although the application is already quite complete, there is always room for improvement. Some recommendations were already mentioned somewhere in this chapter. These recommendations are included in the list below.

- **Components and the component tree**  
Better type support. Type checking and type conversion routines could be implemented.
- **Data connections**  
Support for more types of data sources and targets could be implemented.
- **Generator value maps**  
The value substitution method used by the generators only supports substitution of dropdown and combobox values defined in the configuration

file. This mechanism could be extended to include arbitrary substitution of values.

- **Framework enhancements**

The framework user interface could be extended to include cut, copy and paste operations. An undo/redo feature would also be a useful addition.

Recommendations for the configuration file language can be found in Chapter 5.

## 3.12 Graphical overview

In this last section a graphical overview of the completed and implemented design is presented.

Figure 3.11 contains a complete high-level overview of the design. Clearly visible are the similarities between the GUI creation part of the application and the file generating part of the application.

Figure 3.12 contains the class diagram as reverse engineered by Rational Rose (a graphical design tool). Associations due to connected signals and slots are not shown here, since Rational Rose cannot handle these special Qt constructs.

The central role of the `CGUITree` and `CComponent` tree class is quite clear.

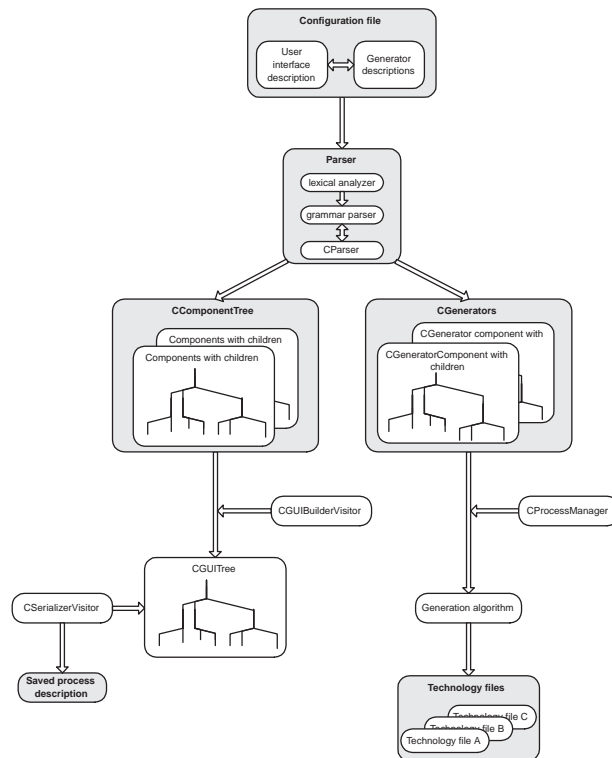


Figure 3.11: Complete architecture overview

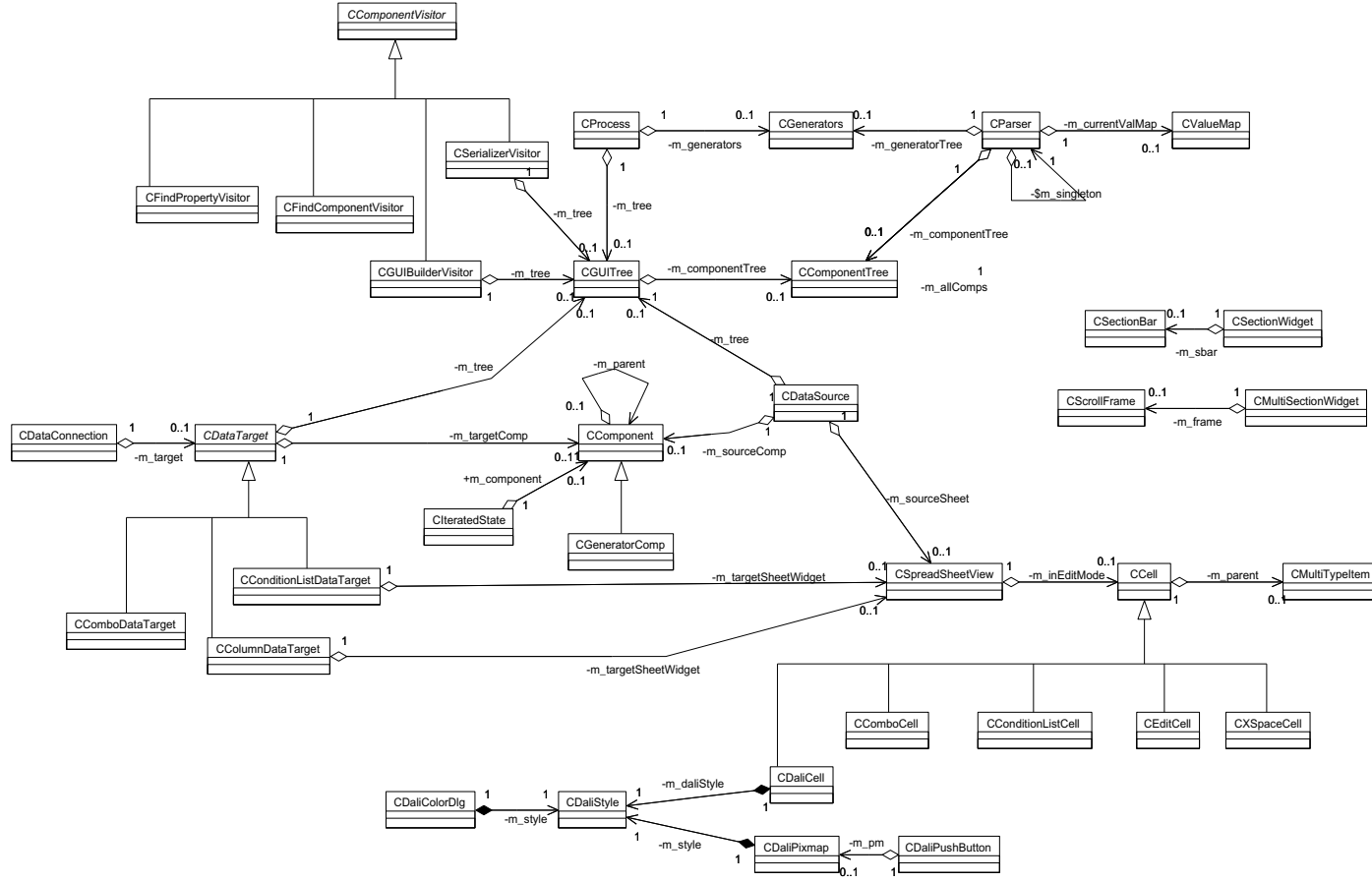


Figure 3.12: Class diagram as reverse engineered by Rational Rose 2000





## Chapter 4

# User interface elements

C++ does not have a default user interface library. Therefore, the user interface architecture presented in Section 3.2 needs to be implemented with a special toolkit. This toolkit will supply a set basic elements that can be extended with custom, specialized elements.

The main goal of this chapter is to present the user interface elements the application uses. These are the user interface elements provided by the chosen toolkit as well as some custom elements.

The choice for a specific toolkit is made in Section 4.1. After that decision has been made, the user interface elements supported by that toolkit will be described briefly. The custom user interface elements will then be presented in Section 4.3.

### 4.1 Choosing a user interface toolkit

There are several widget toolkits available for all flavors of unix operating systems. The most popular toolkits that interface directly with C/C++ are listed below:

- Motif or Lesstiff
- The Athena widget library
- GTK+
- Qt

**Motif (or Lesstiff)** is well known. Motif provides a rich set of user interface elements and is quite powerful. The main disadvantage of Motif is the old C-style function based interface.

**The Athena widget library** is also function based. The Athena widget set is very easy to learn but (as a consequence perhaps) not as rich as for example Motif.

**GTK+** is used in GNOME. It is a very complete toolkit. Contrary to Motif and Athena, this toolkit is object oriented. It is also a portable toolkit, although it has many dependencies that can make compilation difficult.

**Qt** is the toolkit on which the KDE window manager is based. Qt [Tro] is also very complete and object oriented. It was designed with portability in mind and is available for all Unix platforms as well as for Windows. There is even a version for embedded devices. It also provides an extension to C++ with the signal/slot mechanism.

The choice seems to be between Qt and GTK+. Qt was chosen because of its portability and the signal/slot mechanism.

## 4.2 Qt native user interface elements

Qt's complete list of user interface elements is too large to describe here. The elements that will be described next are the ones that are supported by the configuration file language as well as those that are used for the custom widgets. Readers that are already familiar with these widgets can skip to Section 4.3.

### 4.2.1 Labels and edits

Labels and editable widgets are heavily used. They are implemented by the `QLabel` and `QLineEdit` classes.

A label is a frame containing text. This text cannot be changed by the user. Contrary to this, the edit widget *can* be changed by the user. The main use for this type of widget is to retrieve numbers or text from the user.

Figure 4.1 shows these two widgets in action.

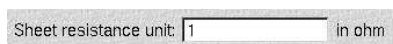


Figure 4.1: Two label widgets and one edit widget.

### 4.2.2 Dropdowns and comboboxes

Dropdowns can be used if the user can choose a value from a list of known values. A combobox should be used if it must also be possible for the user to type in a value not found in the list.

For example, suppose the user must enter a month name. In this case, a dropdown should be used containing the twelve months of the year. Now suppose the user must enter a word he would like to search for in a document. It is common practice to also let the user choose from the last few search terms. In this case the combobox contains the last few search terms but it is still possible to enter a new search term.

Figure 4.2 shows a combobox. The widgets are implemented by the `QComboBox` class. Dropdown behaviour can be achieved by making the `QComboBox` read only.

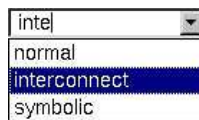


Figure 4.2: A combobox widget.

### 4.2.3 Listview

The listview widget is often used to display files and their properties. A screenshot of listview is depicted in Figure 4.3. Listviews usually have multiple

Name	Size	Type	Modified
figures		File Folder	29-12-2000 16:10
abstract.aux	1 KB	AUX File	3-1-2001 11:21
abstract.tex	1 KB	TEX File	7-11-2000 8:32
background.aux	1 KB	AUX File	3-1-2001 11:21
background.tex	1 KB	TEX File	6-11-2000 14:32
biblio	1 KB	File	22-11-2000 8:33
biblio.bak	1 KB	BAK File	22-11-2000 8:23
biblio.bib	1 KB	BIB File	27-12-2000 13:53
biblio.bib.bak	1 KB	BAK File	14-12-2000 19:50
cdrom.aux	2 KB	AUX File	3-1-2001 11:21
cdrom.tex	1 KB	TEX File	6-11-2000 14:43
conclusion.aux	1 KB	AUX File	3-1-2001 11:21

Figure 4.3: A listview widget used to display files.

columns. Clicking a row selects the entire row and editing is limited to the first column, if editing is allowed at all.

### 4.2.4 The color select dialog box

Color selection is also directly provided by Qt. The `QColorDialog` class implements a color selection dialog that can be called and used with one line of code. A screenshot is shown in Figure 4.4.

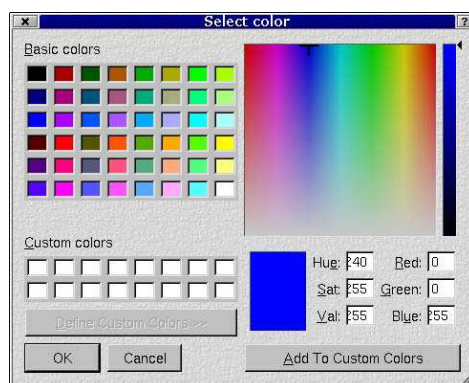


Figure 4.4: The Qt color selection dialog box.

## 4.3 Custom user interface elements

Not a single toolkit provides the specialized widgets a programmer always seems to need. The reason for this is simple: toolkits are designed to be general. However, extending the toolkit should be easy and with Qt it really is.

The widgets created can be split into two categories:

1. Organizational widgets.  
These widgets alter the presentation and layout of the widgets they contain. These are the scrollframe and the section widget.
2. Data entry widgets.  
These widgets provide a new way to enter data. These are the spreadsheet widget, the conditionlist dialog and the dali colorpicker dialog.

### 4.3.1 The scrollframe widget

The scrollframe widget is used when a part of the user interface becomes too large to display on the screen. If this happens, the scrollframe widget will show horizontal and/or vertical scrollbars that can be used to “scroll” the user interface to show the hidden parts.

The best place for them is inside the tabpages. If the widgets will not fit in the space the tabpage has available, the scrollbars will be shown.

### 4.3.2 The section widget

The section widget is a widget that can show or hide its contents on the user's request. It is especially useful if used in combination with other section widgets. Section widgets can be nested if desired. It is recommended to make them part of a scrollframe, as expanding all the widgets will require a lot of space.

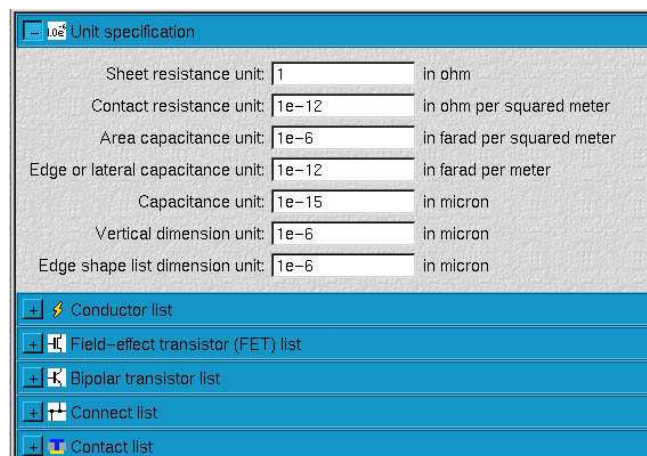


Figure 4.5: A few section widgets.

The section widget is demonstrated in Figure 4.5. The section widget is a blue bar with a button and a label. It is also possible to add an icon between the

button and the label. If a section is expanded the button is shown pressed and it will contain a – sign, indicating it can be collapsed. If a section is collapsed the button is shown depressed and it will contain a + sign, indicating it can be expanded.

The section widget has been created to add an extra level of partitioning to the user interface. This has been discussed in detail in Section 3.2.2.

### 4.3.3 The spreadsheetview widget

The spreadsheetview widget is a hybrid of a spreadsheet and a listview, extended with behaviour to make the widget more generally applicable.

The spreadsheetview visually resembles the listview. However, the default listview behaviour is not adequate for our purposes. The main deficiencies are listed below:

- Editing a cell can only be done with a simple in-place text edit widget.
- Only the cells in the first column can be edited.
- Only cells containing text are allowed.
- The selection and focus behaviour is row oriented and not cell oriented.

What we would like to see is a listview in which each cell can be edited. The way in which a cell can be edited should not be restricted to entering text. It should be possible to use comboboxes or even dialog boxes to enter a value in a cell. For example, to specify a layer's color, a cell that can display the current selected color is necessary. Editing this cell should somehow allow the user to pick any of the available colors.

The best way to implement this in a flexible way is by using an abstract `CCell` class. Deriving from this class allows the implementation of specialized behaviour. There are already five `CCell` derived classes implementing special cell behaviour. They are listed below:

- `CEditCell` implements a regular text cell that can be edited with an in-place line edit.
- `CComboCell` implements a combobox and dropdown cell. The normal display is plain text. The edit mode brings up an in-place dropdown or combobox.
- `CColorCell` shows a color. The edit mode brings up the default Qt color select dialog box.
- `CDaliCell` shows colors and fills as used in dali. The edit mode brings up the dali colorpicker dialog (which will be described later in this chapter).
- `CConditionListCell` shows a conditionlist in the format used in the SPACE element definition files. The edit mode brings up the conditionlist editor (which will be described later in this chapter).

The spreadsheetview widget is shown in Figure 4.6.

Layer ID	Layer name	Layer type	Dail draw style	XSpace draw color	Comment
0	cpg	interconnect		red	polysilicon
1	caa	interconnect		green	active area
2	cmf	interconnect		blue	metal
3	cms	interconnect		blue	metal2
4	cca	normal			contact metal to diffusion
5	ccp	normal			contact metal to poly
6	cva	normal			contact metal to metal2
7	cwn	normal			n-well
8	csn	normal			n-channel implant
9	cog	normal			contact to bondpads
10	cx	symbolic			bounding box

Figure 4.6: The spreadsheetview widget.

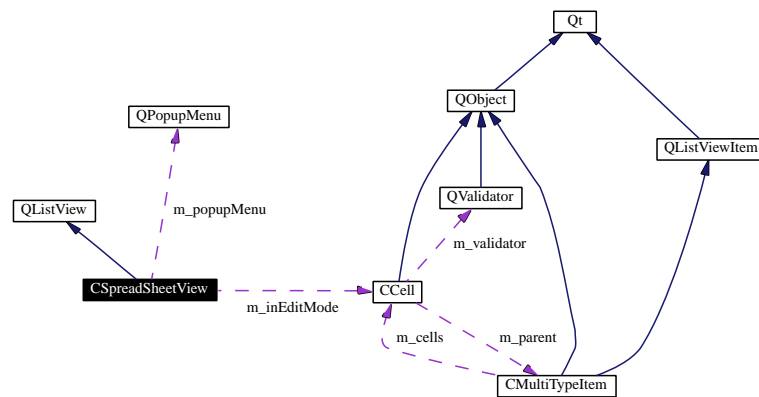


Figure 4.7: CSpreadSheetView collaboration diagram.

### Implementation

The spreadsheetview widget is implemented in the `CSpreadSheetView` class. This class is derived from the Qt `QListView` class. Also used in the implementation is the custom `CMultiTypeItem` class which is derived from the `QListViewItem` class. Finally, the `CCell` class specifies the interface for the derived classes and provides some basic functionality. The relationship between the classes is shown in Figure 4.7.

The collaboration diagram clearly shows that a `CSpreadSheetView` is a specialization of the `QListView` class. The `CMultiTypeItem` class is a specialization of the `QListViewItem` class and represents a row in the spreadsheetview. The `CMultiTypeItem` itself contains `CCell` objects, representing the columns in that row.

Figure 4.8 shows the class diagram. A `CSpreadSheetView` class contains rows, implemented by the `CMultiTypeItem` class. Each row consist of a number of `CCell` derived objects.

After the creation of a `CSpreadSheetView` object, columns can be added using the `addColumn()` method. Rows can be added using the `addNewRow()` method. The creation of the `CCell` objects is left to the client. If a cell object needs to be created the `CSpreadSheetView` objects emits a `pleaseCreateCell()` signal. The signal communicates the `CMultiTypeItem` (the row) and the column that

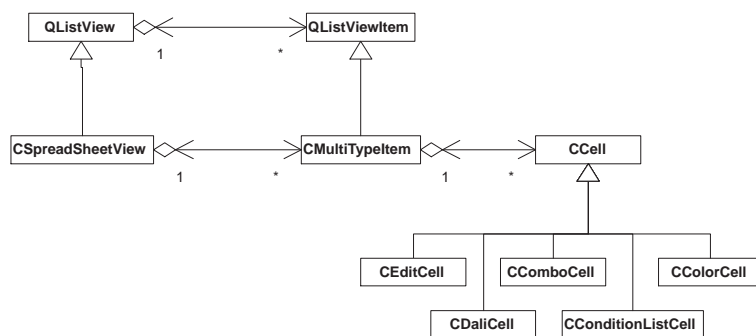


Figure 4.8: CSpreadSheetView class diagram.

needs a **CCell** (derived) object. If the **CCell** object is not created after the signal (either because the signal was not connected to a slot or the slot could not create a object) a default **CCell** will be created by the **CSpreadSheetView** itself.

This mechanism allows the client to provide its own **CCell** derived objects at any column/row combination

Starting and ending the edit mode of a cell is handled by the **CSpreadSheetView** class, calling the correct **CCell** object methods as needed. The latest version of Qt now includes a spreadsheet widget (which was absent during the development of the **CSpreadSheetView** widget). It uses the same mechanism of cell abstraction and delayed creation. It also uses start/end edit methods. Perhaps a future version of the **CSpreadSheetView** widget should be based on this new Qt widget instead of on the **QListView** widget.

#### 4.3.4 The condition list dialog

Condition lists are used often in the SPACE element definition files. A condition list specifies how the presence of a particular element depends on the presence or absence of the different masks [vdMvGBE99]. Basically, it is a boolean combination of mask names. A space means the AND operation should be applied. A | character corresponds to an OR operation and an exclamation mark (!) in front of a mask name is a NOT. Additionally, in the cases where adjacent tiles have a meaning, the – and = characters specify one or the other side of a tile. For more information please refer to the SPACE User's Manual [vdMvGBE99].

The condition list editor is shown in Figure 4.9. The condition list can be entered manually or by using the buttons and the listbox. The layer names and descriptions are obtained using the data connection mechanism described in Section 3.7. The special data target **CConditionListDataTarget** has been implemented for this to work. This allows to specify arbitrary sources for the layer names and descriptions in the configuration file.

The listbox on the right side of the dialog box contains the OR expressions. Each line in the listbox is OR-ed together to form the complete condition list.

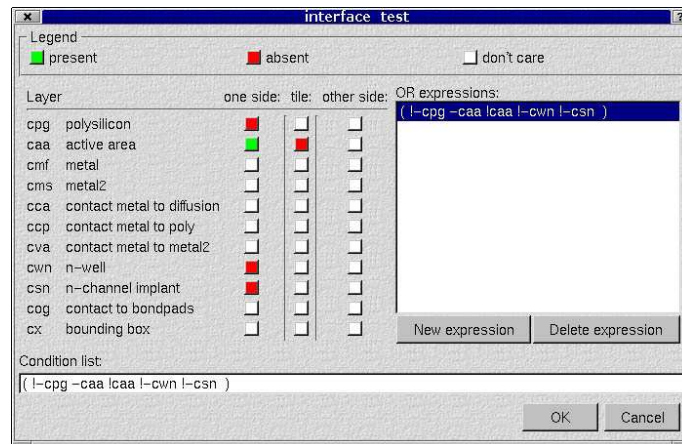


Figure 4.9: The conditionlist editor dialog box.

Selecting an expression in this listbox will cause the buttons on the left side to reflect the selected expression. Changing the button states will immediately update the selected line in the listbox, as well as the line representing the complete condition list.

The condition list editor supports two modes. In the default mode, only the tile of interest is displayed. No neighboring tiles can be selected as present or absent. In the extended mode, neighboring tiles are displayed as “one side” and “other side” and can be selected as normal. The desired mode can be specified in the configuration file.

#### 4.3.5 The dali colorpicker dialog

The dali colorpicker dialog box lets the user select a color and fill style as used in dali, the layout editor. Since dali understands only a limited set of colors and fill patterns, the default Qt color dialog box and fill patterns could not be used.

The colorpicker presents the user with a dialog box as shown in Figure 4.10. The dialog box consists of an array of buttons. All the combinations of available colors with available patterns are present. Clicking a button selects the color and fill represented by the button.



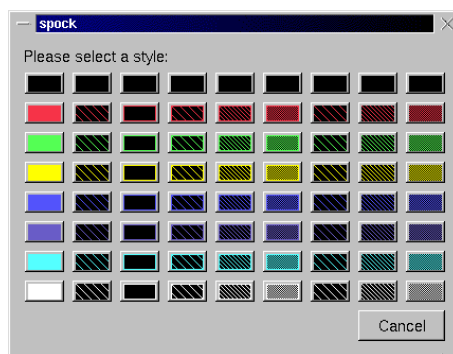


Figure 4.10: The dali colorpicker dialog box.



## Chapter 5

# The configuration file language

The configuration file language was designed to be a flexible and easy to use (and learn) language that can be used to specify a user interface, file generators and the relationship between these two.

The main goal of this chapter is to present and explain the language and its features.

First of all, some of the more general constructions used in the language are presented and explained in Section 5.1. Because the configuration file describes two different phenomena (user interface specification and technology file format specification), a certain division of the file is to be expected. Language constructions related to the user interface are presented in Section 5.2. Generator constructions are described in Section 5.3.

### 5.1 Language design and overview

#### A note about notation

This chapter contains many examples in the form of source code fragments. Source code fragments can be recognized by the **typewriter font**. Identifiers in these examples can be recognized by their *italic* font. Sometimes some lines have been left out for clarity, or additional statements could be present but are not relevant for the example. This is indicated with three dots: ....

The language also supports C++ style comments. If two slash characters (//) are encountered the rest of the line is ignored.

#### 5.1.1 Structuring of components

The parser will have to create the component tree and expression tree described in Section 3.5 and 3.8. It would be helpful for the parser if the tree structure

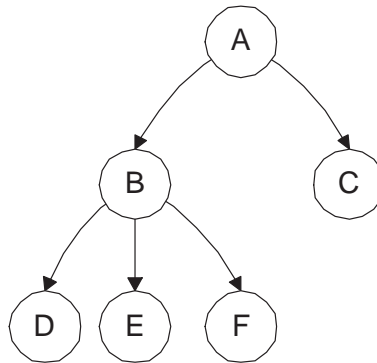


Figure 5.1: Structuring example

of the component and expression trees correspond with the hierarchy in the configuration file language. A common way to structure a language is by using hierarchical blocks. Blocks have a start and end marker and can be nested.

For example: if we assume the block start marker is the { character and the block end marker is the } character we can build the tree structure in Figure 5.1 like this:

```

A {
    ...
    B {
        D {
            ...
        }
        E {
            ...
        }
        F {
            ...
        }
    }
    C {
        ...
    }
    ...
}
  
```

The block start/end markers unambiguously define the tree structure from Figure 5.1. The indentation (leading whitespace) used here is only to clarify the structure, it is not mandatory. We could also have written `A {B { D {} E {} F {} } C {} }`, but this is more obscure.

Note that the tree in Figure 5.1 contains a toplevel node (node A), one embedded node (node B) and several leaf nodes (D, E, F and C). Likewise, the configuration file language distinguishes between toplevel, embedded and leaf components.

The possible components for each of these categories are discussed in Section 5.2.

### 5.1.2 Component types

As mentioned earlier, the components in the tree have type and the type determines what kind of user interface element is created for that component. The types range from strings and numbers to spreadsheets and sections. A common typing mechanism is to put the type name in front of the declaration. For example, if we were to declare an instance of **spreadsheet** type named **capacitances** this would result in:

```
spreadsheet capacitances {  
    ...  
}
```

In this example, **spreadsheet** is the type of the component and *capacitances* is the identifier with which this instance can be referred to. This identifier can be used in the generator specifications to access the values in the spreadsheet entered by the user.

This approach is very usable and is therefore applied in the configuration file language.

### 5.1.3 Properties

Components support properties (see also Section 3.5) and a mechanism to set the properties of a component should be present in the language. This can be done in many ways. The obvious way would be to use the equality sign:

```
section conductors {  
    title = "Conductors list";  
}
```

This approach has a disadvantage however, because this construction looks too much like an assignment. Assignments are normally used to set the value of an identifier. This could lead to confusion because identifiers are used to name the components. The user could think that these identifiers can have values assigned, for example: **conductors** = "Conductor list". This is not the case. In the example above **title** is not an identifier that can have a value assigned, but a property that can have a value set. It is better to reserve the assignment construction for a future version of the language that will support these identifier assignments.

A different construction is therefore needed to set property values. The approach finally taken is to use rounded brackets after the property name. Inside the brackets is the new value for the property:

```
section conductors {  
    title("Conductors list");  
}
```

This is a more function-like approach. Comma's can be used to add additional values, like in the example below:

```
conditionlist condlist_contacts {
    adddatafrom(maskdata_sheet.name, maskdata_sheet.comment);
}
```

The interpretation of the property values (the “arguments”) depend on the context in which they are used. The possibilities are discussed in Section 5.2.

#### 5.1.4 Generators

The technology file generator specifications do not completely “fit” into the component mechanism previously described. The mechanism suffices for naming the generator and setting some basic properties of the generator, but not for the expression tree that will evaluate to the generated text.

Because of this, a special component statement is introduced that switches the parser to a new mode. In this mode, each string is treated as literal text. Identifiers that point to some part of the user interface evaluate to the value entered by the user in that part of the user interface. For example:

```
generator maskdata_gen {
    filename("maskdata");
    title("Maskdata file");

    generate {
        "# Automatically generated file. Do not edit.\n"
        "number_of_rows:"^ maskdata_sheet.numrows "\n"
        "#####\n"
    }
}
```

In this example, the `generate` statement switches the parser to the generator mode. The plain-text and identifier methods are also demonstrated. Some special constructions are available as well, but these are discussed in detail in Section 5.3.

#### 5.1.5 Macro definitions

In some cases it is useful if it is possible to define a macro. For example, condition lists are used in multiple places. An example condition list is given below:

```
conditionlist condlist_simp {
    option("noextended");
    adddatafrom(maskdata_sheet.name, maskdata_sheet.comment);
    title("Condition list"); // Default title
}
```

It would be convenient if there was a way to repeat this condition list without retyping all properties. This can be achieved by placing the word `define` in front of this declaration:

```
define conditionlist condlist_simp {
    option("noextended");
```

```

    adddatafrom(maskdata_sheet.name, maskdata_sheet.comment);
    title("Condition list"); // Default title
}

```

We can now use this `condlist_simp` as a new component type:

```

spreadsheet conductors_sheet {
    ...
    condlist_simp cond_list {
        title("Conductor condition list:"); // Overriding or
        align(right); // adding a property is allowed.
    }
    ...
}

spreadsheet fets_sheet {
    ...
    condlist_simp cond_list {
        title("FET condition list:"); // Overriding a
                                   // property is allowed.
    }
    ...
}

```

The three properties as defined in the macro do not have to be typed again. The example also demonstrates the possibility to set additional properties like the `align` property. It is also possible to override a property. This is demonstrated with the `title` property.

If the `title` property does not need to be redefined an empty block must be specified with curly braces (`condlist_simp cond_list {}`). To people with programming experience this can be confusing. They would expect a semicolon to be sufficient. However, this language construction is already used in the “chopping” mechanism, and using it on a macro reference would result in undefined behaviour. The “chopping” mechanism is explained in Section 5.1.6. Perhaps in a next release of the language this behaviour could be changed.

Another useful application of this mechanism is in dropdowns or comboboxes. For example, if the user can choose the accuracy of an algorithm and there are many algorithms to run:

```

define dropdown algo_acc {
    item highest { title("Highest accuracy - longest runtime"); }
    item high    { title("High accuracy - long runtime"); }
    item normal  { title("Normal accuracy - normal runtime"); }
    item low     { title("Low accuracy - short runtime"); }
    item lowest { title("Lowest accuracy - shortest runtime"); }
}

paramlist algo_params {
    algo_acc alg_A { title("Algorithm A accuracy:"); }
    algo_acc alg_B { title("Algorithm B accuracy:"); }
}

```

```

    ...
}

```

This mechanism could also be used for yes/no or on/off dropdowns. One could argue that these kind of macros should be built-in in the language. The choice was made not to do this. This is the first version of the language and it has not been released to a large public. The interface should therefore be kept as clean as possible. A future version of the language could easily implement these built-ins if the need for them arises.

Nesting of macros is allowed. This means it is possible to use a defined type inside another macro.

### Implementation

Whenever the parser encounters a situation in which a previously declared **define** is used, it copies the branch of the component tree of the **define** to the correct location. If any properties were added or changed like in the first example it updates the copied branch of the component tree to reflect these changes.

Note that the parser performs on-the-fly disambiguation for macro definitions. This means a macro must be defined before it is used.

#### 5.1.6 Reducing indentation

As was mentioned earlier, indentation with whitespace is not necessary. However, it is encouraged because it increases the readability of the configuration file. If the number of nestings becomes too large though, this readability can decrease again. The indentation then becomes too large.

To counter this effect, a “chopping” mechanism is introduced. Chopping makes it possible to place declarations outside the hierarchy if desired. For example, a tabpage usually consists of a scrollframe with several sections:

```

tabpage element_definition {
  scrollframe element_def_frame {
    section fets {
      spreadsheet fets_sheet {
        ...
      }
      ...
    }
    section bjts {
      spreadsheet bjts_sheet {
        ...
      }
      ...
    }
    ...
  }
}

```



Even in this little example the indentation becomes considerable. The “chopping” mechanism is demonstrated below:

```
section fets {
    spreadsheet fets_sheet {
        ...
    }
    ...
}

section bjts {
    spreadsheet bjts_sheet {
        ...
    }
    ...
}

tabpage element_definition {
    scrollframe element_def_frame {
        section fets;
        section bjts;
        ...
    }
}
```

This way, the tabpage declaration remains clear because the finer details are “hidden” in the sections.

This method is called “chopping” because a branch of the component tree is cut off, leaving a reference at the location the chopping occurred.

### Differences with macro definitions

The main differences are listed below:

- References to macros can be used many times. References to a chopped branch can be used only once. If used more than once, the resulting behaviour is undefined.
- References to chopped branches use a different syntax than references to defined macros. A reference to a define macro uses the macro name and must be followed by block markers (`{ ... }`). A reference to a chopped branch must be followed by a semicolon. If a reference to a branch is followed by block markers the resulting behaviour is undefined. If a reference to a macro is followed by a semicolon the resulting behaviour is undefined. As was mentioned in Section 5.1.5, the syntactical difference between an empty macro (no additions or overrides, `{ }`) and a chopped branch can be confusing for experienced programmers. However, it should be noted that chopped branches cannot have additional or overridden properties. Hence, there is no need for curly braces and a semi-colon suffices. In the case of a macro without added or overridden properties, the empty curly braces indicate that something *could* be changed if desired.

As was already mentioned in Section 5.1.5, changing the syntax of these constructions should be considered in a next release.

Like the macro definition, nested choppings are allowed. This means that in the example it is allowed to chop the `scrollframe` *element\_def\_frame* as well.

### Implementation

If we look at it from a tree perspective, defined macro branches are *copied* to new locations in the tree while the chopped of branches are *moved* back to the position where they belong. This is done by the parser. Whenever the parser encounters a reference to a chopped of branch, it tries to find the chopped of branch and moves it to the correct position in the component tree.

Note that the parser performs on-the-fly disambiguation for chopped branches, just like it does for macro definitions. This means the chopped branch must be put before the reference to it.

#### 5.1.7 Data connections

The data connections as discussed in Section 3.7 must have a place in the language as well. A data connection consist of one or more sources linked to a single target. Therefore, the most logical place to define a data connection would be inside the target component. If we make a data connection declaration a property, we can also use the `CFindPropertyVisitor` class to find them when the data connections need to be built. An example of the implemented construction is given below:

```
define dropdown layer {
    adddatafrom(maskdata_sheet.name, layer_synonyms.name);
}
```

In this example we define a dropdown `layer`. The items in the dropdown are the `name` column in the spreadsheet `maskdata_sheet` and the spreadsheet `layer_synonyms`.

The data connections are established right after the `CGUITree` object is created by the `CGUIBuilderVisitor`.

The only datasource currently implemented is a column in a spreadsheet. The currently available data targets can be found in the `adddatafrom` column of Table 5.2.

## 5.2 Language overview for user interface elements

This section will present all the component types that can be used to create a user interface. The context in which a certain component type is used determines what kind of user interface element is created. The context also affects how a component's properties are used.

Table 5.1: Classification of components.

Classification	Component type(s)
toplevel	<code>tabpage</code>
embedded	<code>dropdown</code> , <code>combobox</code> , <code>spreadsheet</code> , <code>section</code> , <code>paramlist</code> , <code>scrollframe</code>
leaf	<code>integer</code> , <code>real</code> , <code>string</code> , <code>identifier</code> , <code>conditionlist</code> , <code>color</code> , <code>dalistyle</code> , <code>item</code>

Since the context largely determines what will happen, the descriptions of the component types and their properties are given relative to the context in which they can be used.

As was mentioned in Section 5.1.1, we can distinguish between toplevel, embedded and leaf components. Table 5.1 lists which component types belong to which category.

The components that can provide a context are the top level and embedded components. Leaf components cannot provide context, but their interpretation strongly depends on the context in which they are used.

In the following subsections, all the context-providing component types are presented. The interpretation of the components inside the context being discussed is explained in these subsections. Small examples with their resulting user interface are given as well. At the end of this section, a complete overview of the currently possible combinations of context, component types and properties is presented in Table 5.2

**Note:** Although it is possible to place components *visually* on the top level (because of the chopping mechanism described in Section 5.1.6), the components *as seen after parsing* still adhere to the classification presented in Table 5.1.

### 5.2.1 Components in a dropdown or combobox context

The dropdown is used exactly the same as the combobox. The only difference is in the behaviour of the widget that will be created. Dropdown widgets present the user with a fixed list of choices. The combobox widget also presents a list of choices. However, the user can also enter text not present in the list of available choices. Because of the similarities we will only discuss the dropdown.

The dropdown can only have `item` children inside. All items will become entries in the dropdown widget. Each item must have the `title` property set to the text that will be displayed as the entry in the dropdown widget. If the `title` property of an item is not specified, an empty entry will be inserted in the dropdown.

**Example:**

```
dropdown def_carr_type {
  item n { title("n doped conductor"); }
```

```

item p { title("p doped conductor"); }
item m { title("metal"); }
default(m);
}

```

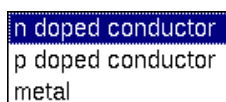


Figure 5.2: Dropdown example result.

### 5.2.2 Components in a spreadsheet context

In a spreadsheet, each child is interpreted as a column. The number of rows changes dynamically, as the user can add or remove rows. The edit mode of the column corresponds to the columns type. This means a **string** column corresponds to a `QLineEdit` as edit widget. These are the types that can be columns in a spreadsheet:

- **string**, **real**, **integer** and **identifier** correspond to a column using `QLineEdit` widgets.
- **color** columns create `CColorCells` that bring up the `QColorDialog` box. They display the selected color.
- **dalistyle** columns are edited using the dali colorpicker dialog. The selected style is displayed in the cells.
- **conditionlist** columns are edited with the condition list editor.
- **dropdown** and **combobox** columns display a dropdown respectively a combobox widget.

**Example:**

```

spreadsheet maskdata_sheet {
  integer      ID { title("Layer ID"); }
  identifier   name { title("Layer name"); }
  def_layer_type layertype { title("Layer type"); } // def_layer_type is a macro
  dalistyle    dali { title("Dali draw style"); }
  color        xspace { title("XSpace draw color"); }
  string       comment { title("Comment"); }
}

```

### 5.2.3 Components in a paramlist context

The parameter list displays a three column widget. The first column contains descriptions of the input fields, the middle column contains the input widgets











Layer ID	Layer name	Layer type	Dali draw style	XSpace draw color	Comment
0	cpg	interconnect		red	polysilicon
1	caa	interconnect		green	active area
2	cmf	interconnect		blue	metal
3	cms	interconnect		blue	metal2
4	cca	normal		black	contact metal to diffusion
5	ccp	normal		black	contact metal to poly
6	cva	normal		gray	contact metal to metal2
7	cwn	normal		pink	n-well
8	csn	normal		gray	n-channel implant
9	cog	normal		cyan	contact to bondpads
10	cx	symbolic			bounding box

Figure 5.3: Spreadsheet example result.

and the last column can contain additional text, for example to display the unit of the value that must be entered.

The children of the `paramlist` component can be any type. The most useful types are the `string`, `real`, `integer`, `dropdown` and `combobox` types.

#### Example:

```
paramlist units_list {
  real resistance {
    title("Sheet resistance unit:");
    unit("in ohm");
    hint("You can specify a base for the values you \n"
        "will enter below. Use as follows:\n"
        "If you do not want to type 3e-6, 8e-6 12.5e-6, etc.\n"
        "but 3, 8, 12.5 instead, type 1e-6 here.");
  }
  dropdown accur {
    title("Accuracy:");
    item low { title("Low accuracy"); }
    item normal { title("Normal accuracy"); }
    item high { title("High accuracy"); }
    default(normal);
  }
}
```

Sheet resistance unit:	<input type="text" value="1"/>	in ohm
Contact resistance unit:	<input type="text" value="1e-12"/>	in ohm per squared meter
Area capacitance unit:	<input type="text" value="1e-6"/>	in farad per squared meter
Edge or lateral capacitance unit:	<input type="text" value="1e-12"/>	in farad per meter
Capacitance unit:	<input type="text" value="1e-15"/>	in micron
Vertical dimension unit:	<input type="text" value="1e-6"/>	in micron
Edge shape list dimension unit:	<input type="text" value="1e-6"/>	in micron

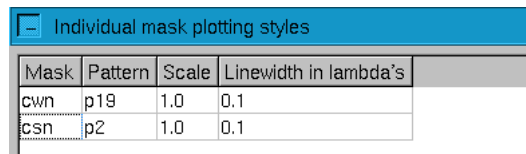
Figure 5.4: Parameter list example result.

### 5.2.4 Components in a section context

Sections are best used inside scrollframes because the total area they can cover can become quite large. The children of a section are placed vertically inside the sections expand/collapse area.

**Example:**

```
section getepslay_body {
  title("Individual mask plotting styles");
  spreadsheet getepslay_sheet {
    layer_combo mask_name { title("Mask"); }
    string      pattern { title("Pattern"); }
    real        scale { title("Scale"); default("1.0"); }
    real        linewidth { title("Linewidth in lambda's"); }
  }
}
```



Mask	Pattern	Scale	Linewidth in lambda's
cwn	p19	1.0	0.1
csn	p2	1.0	0.1

Figure 5.5: Section example result.

### 5.2.5 Components in a scrollframe context

A scrollframe component is useful primarily inside tabpages. The scrollframe will then occupy the entire tabpage area, maximizing the area available to the components inside the frame. Using multiple scrollframes inside a tabpage or nesting scrollframes is not allowed. This is a limitation in the current scrollframe implementation. The resulting behaviour is undefined.

There are no properties for the scrollframe itself, because the scrollframe is not really a visible widget and the behaviour is fixed. Future versions could implement some properties to influence the behaviour. Examples could be properties to influence the used layout mechanism (horizontal/vertical) or child alignment (for example to center or right-justify the widgets instead of the current left justification).

**Example:**

```
scrollframe element_def {
  section units;
  section conductors;
  section fets;
  section bjts;
  section connects;
  section contacts;
```

```

    section capacitances;
}

```

This example shows how some section widgets are placed inside a scrollframe.

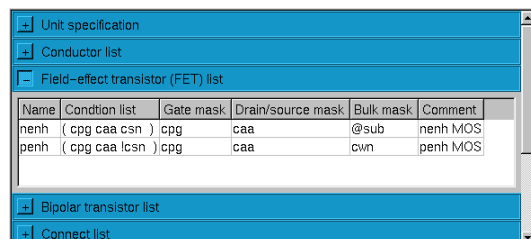


Figure 5.6: Scrollframe example result.

### 5.2.6 Components in a tabpage context

Tabpage components will become the tabpages in the user interface. A tabpage must therefore have a title to distinguish it from the other tabpages.

**Example:**

```

tabpage getepsly_tab {
    title("Postscript plot settings");
    scrollframe getepsly_frame {
        section getepsly_preamble;
        section getepsly_body;
    }
}

```

### 5.2.7 Overview of combinations

Table 5.2 lists the existing relationships between component types in a certain context and the properties that can be used. Note that the table lists the *implemented* relationships, not the *possible* relationships. Additional sensible relationships could be implemented in a future version of the application.

The numbers in the table refer to the explanations given below. Some numbers are followed by a letter. The letter **r** indicates that the property is required. If these properties are not specified the behaviour of the application is undefined. The letter **F** indicates that it is forbidden to specify that property. If the property is specified anyway, the behaviour of the application is undefined.

The label **r.i.s.i.** stands for real, integer, string or identifier. The row is valid for each of these component types. The combobox type is missing, but the properties and contexts listed for the dropdown are identical to the ones listed for the combobox.

If a combination of a component and a context is not listed in the table, the behaviour resulting from this combination is undefined.

Table 5.2: Combinations of components and properties in a context.

Context	Acceptable component types	title	hint	default	align	adddatafrom	unit	pixmap	option
dropdown	item	1 r				F			
spreadsheet	r.i.s.i	2 r		8	10	F			
	color	2 r		8		F			
	dalistyle	2 r		8		F			
	conditionlist	2 r			10	11 r			15
section	r.i.s.i.		5	8		F			
	dropdown		5	8		12			
	spreadsheet		5			F			
paramlist	r.i.s.i	3	6	8		F	13		
	dropdown	3	6	9		12	13		
	spreadsheet	3	6			F	13		
scrollframe	r.i.s.i.		5	8		F			
	dropdown		5	9		12			
	spreadsheet		5			F			
	section	4	7					14	
tabpage	r.i.s.i.		5	8		F			
	dropdown		5	9		12			
	spreadsheet		5			F			
	section	4	7					14	
	scrollframe								



The numbers in Table 5.2 are explained below:

- 1 The value of this property will be used as the text of the corresponding entry in the dropdown.
- 2 The value of this property will be used as the title of the corresponding column in the spreadsheet.
- 3 The value of this property will be used as the label in the first column of the parameter list.
- 4 The value of this property will be used as the title displayed in the section bar.
- 5 The value of this property will be shown as a hint text as the mouse “hovers” over the widget. The hint text can be split over multiple lines by inserting `\n` at the right places. Quotation marks can be inserted with `\`.
- 6 The value of this property will be shown as a hint text as the mouse “hovers” over the widget in the second column. The hint text can be split over multiple lines by inserting `\n` at the right places. Quotation marks can be inserted with `\`.
- 7 The value of this property will be shown as a hint text as the mouse “hovers” over the section bar. The hint text can be split over multiple lines by inserting `\n` at the right places. Quotation marks can be inserted with `\`.
- 8 The value of this property will be used as default value to display in the spreadsheet cells of this column.
- 9 The value of this property must be the name of one of the child items. This item will be shown as the default value in the dropdown/combobox.
- 10 The value of this property can be **left**, **right** or **center**. The spreadsheet column will be aligned accordingly.
- 11 This property accepts two values. Both values must be identifiers that can be unambiguously resolved. The data referenced by the first (required) identifier will be used for the layer names in the conditionlist. The second (optional) identifier references the layer descriptions.
- 12 This property accepts one or more values. Each value is an identifier that can be unambiguously resolved. The data referenced by each identifier will be concatenated and the result will become the entries in the dropdown/combobox.
- 13 The value of this property will be used as the label in the third column of the parameter list, after the input field.
- 14 The value of this property is the filename of a file containing a pixmap. The filename must be given relative to the **ICONPATH** specified at the top of the configuration file. The **ICONPATH** can only contain a single path (for example: **ICONPATH** = `"$(ICDPATH)/lib/spock"`. Environment variables are expanded and can be specified **Makefile** style: between parenthesis preceded by a `$`.)
- 15 This property value must be either **extended** or **noextended**. If **extended** is specified, the conditionlist editor is shown with three columns: ‘one side’, ‘tile’ and ‘other side’. If **noextended** is specified, only the ‘tile’ column will be displayed.

## 5.3 Language overview for generators

This section will present and explain the generator specification in the configuration file language. The generator specification is put into an expression tree that evaluates to the contents of the technology files.

The generator component is described first in Section 5.3.1. After that, the use of value mappings is explained in Section 5.3.2. The sections following Section 5.3.2 describe the expressions that can be used inside the **generate** context.

### 5.3.1 Generator components

The generator components are specified the same way as the other components in the configuration file. An example was already given in Section 5.1.4 and is repeated below:

```
generator maskdata_gen {
    filename("maskdata");
    title("Maskdata file");

    generate {
        "# Automatically generated file. Do not edit.\n"
        "number_of_rows:" maksdata_sheet.numrows "\n"
        "#####\n"
    }
}
```

The generator type component uses a language construction similar to the other component types. Like the other component types, properties can be used and the declaration syntax is the same. However, this component is not translated into a user interface element. More importantly, additional language constructions can be used inside the generator component.

The additional language constructions that can be used are the value mappings and the **generate** statement. The expressions describing the technology file format must be put inside the generate statement

The properties supported by the generator component:

Property	Description
<b>title</b>	Use the <b>title</b> property to set a short descriptive name for the generator. This descriptive name is used in several places in the application, for example in the technology file generation dialog box.
<b>filename</b>	The filename of the technology file to generate. This is only the filename, not the path. The path is specified by the user in the user interface.

If the **title** property is not specified, the description is an empty string. If the **filename** property is not specified, the behaviour of the application is undefined.

### 5.3.2 Value mappings

Besides the properties, the generator component supports value mappings. Value mappings allow the substitution of text used in the **dropdown** and **combobox** items to a string in the technology file. This allows the use of a descriptive text in the user interface and a specialized string in the generated

technology file. An example of the construction is given below. The **dropdown** used in the substitution is also shown to illustrate the mechanism.

```
define combobox def_carr_type {
  item n { title("n doped conductor"); }
  item p { title("p doped conductor"); }
  item m { title("metal"); }
  default(m);
}

map def_carr_type {
  n = "n";
  p = "p";
  m = "m";
}
```

The value mapping could of course be extended into a more general mechanism that allows arbitrary substitutions. This would increase the flexibility of the generators.

Looking at the example above, one could argue that the identifier name could be used as a substitution. This has however, as a disadvantage that it would not be possible to substitute a text that is not an identifier, meaning spaces or other special symbols could not be used.

### 5.3.3 Literal text

Use quotation marks to generate literal text. Literal text may contain tabs and newlines like in C++ using `\t` and `\n`. If quotation marks are needed they too can be escaped (`\`). Concatenation of strings can be used to span long literals over several lines. This is illustrated in the example below.

```
generate {
  "Literal text must be placed between quotation marks (\").\n"
  "\n and \t can be used like in C++. Bordering "
  "literals "   "are "   "concatenated in the output of "
  "the generator."
}
```

### 5.3.4 Identifier substitution

To place values in the generator output we can use the name of any identifier in the user interface specification. The identifier will be replaced with the value that is currently entered in the applications' user interface.

```
generate {
  "The value of getepsplay_pl.mask_order is: "
  getepsplay_pl.mask_order "\n"
}
```

Some component types do not generate any output. These are usually types that contain other components like the scrollframe and the section components.

In some cases, not the actual value of identifier is required but some other property. This is achieved by the use of fields. For example, to generate the number of layers `maskdata_sheet[numrows]` could be used. The field is specified between square brackets after the identifier name. The currently supported fields are listed in Table 5.3

Besides the identifiers present in the user interface specification, there is also a special identifier called **application**. This identifier can be used to retrieve values from the application that are not present in the user interface specification. **application** is a reserved word. If **application** is inadvertently used as an identifier, the parser will exit with an error.

Table 5.3: Supported fields:

Component type	Fields	Description
spreadsheet	numrows	Is replaced with the number of rows present in the spreadsheet.
dalistyle	fill	Is replaced with the selected dali fill number
	clr	Is replaced with the selected dali color number
application	time	Is replaced with the current time
	date	Is replaced with the current date
	processname	Is replaced with name of the process being generated
	processdesc	Is replaces with a description of the process being generated

### 5.3.5 Number addition and subtraction

Simple addition and subtraction is also supported. For example:

```
generate {
    "The number of layers + 1 = " maskdata_sheet[numrows] + 1
}
```

The type of the result depends on the type of the operands. Two integer operands will result in an integer but two reals or a real and a integer will result in a real.

### 5.3.6 Loops

For spreadsheets it is necessary to be able to loop over the rows or all values present in a column of a spreadsheet. To support this, a **foreach** loop is introduced. The foreach loop has two forms. The first form loops over the rows and has no loop variable:

```
foreach(fets.fets_sheet, row) {
    "    " name ":" cond_list ":" mask_g " " mask_ds
    ":" mask_b " # " comment "\n"
```

```
}
  "\n"
```

`fets.fets_sheet` is a hint context. The identifiers inside the loop body are first resolved using this context. If that yields no results a normal disambiguation is tried.

Besides iterating over spreadsheet rows it is also possible to iterate over the values present in a column. Each value in that column is enumerated exactly once:

```
foreach $t (conductors_sheet, cond_type)
  "conductor " $t "\n"
```

Again, the first argument between the brackets is the hint context. The second argument is an identifier that is a spreadsheet column. All values in that column will be enumerated.

The loop variable can be used (here `$t`) to access the current value of an iteration. This makes it possible to nest loops. Each loop variable corresponds to the current iteration values of each of the loops.

If the `cond_type` column contained the values a, b, a, c, d, and a, the result would be:

```
conductor a
conductor b
conductor c
conductor d
```

For an example of some more difficult nested loops please take a look at the sample `spock.uis` configuration file in Appendix D.

### 5.3.7 Conditionals

In some cases it is necessary to generate a piece of text conditionally. To be able to do this, a conditional statement is needed. This is provided by the `if` statement.

```
foreach $t (cap_sheet, cap_type) {
  foreach $s (cap_sheet, subtype) {
    $t " capacitances " $s ":\n"
    "#\tname:condition_list:type:mask1 [mask2]:capacitivity\n"
    foreach(capacitances.cap_sheet, row) {
      if (cap_type == $t) {
        if (subtype == $s) {
          "\t" name ":" cond_list ":" mask_1 " " mask_2 ":" capacitivity "\t\t# " comm
        }
      }
    }
    "\n"
  }
}
```

The possible conditional statements are limited to equal or not equal. The operators for this are `==` and `!=`. Support for additional operators could easily be implemented in a future version.

## 5.4 Recommended language extensions

Language extensions can be grouped into two categories:

1. extensions to the language itself
2. extensions in the form of new types and properties

### 5.4.1 Language extensions

The configuration file language is quite complete in functionality. However, some common language constructions seen in other languages are not yet possible.

- **Operators**

Currently only `+` and `-` are supported. It is of course possible to add multiplication (`*`), division (`/`) and more math-like functions like powers (`^`) and perhaps even `sin()` and `cos()`. This depends on the need for them of course.

- **Conditionals**

The conditional `if` statement could be extended with `else` support and additional comparisons like less then and greater then (`<` and `>`).

- **Loops**

The `foreach` loop construction provided is sufficient for simple loops. It could be useful to implement additional loop structures like `do-while` or `while-do`.

- **Value substitution**

The value substitution mechanism provided by the value maps could be extended to a more general substitution mechanism that can substitute any string for another.

- **Perl**

It is also possible to embed perl into the application. The expressions describing the technology file formats could become a block of perl source.

This mechanism could even be extended to the complete configuration file. Special perl functions internally defined by the application could then be called in the configuration file to build the trees.

The disadvantage of using perl is the addition of a dependency to the application and the possible compilation difficulties on different platforms.

### 5.4.2 New types and properties

Besides extensions to the language itself, it is also possible to add new types and properties to the language.

- **Layout control**

It is possible to add properties or components that use Qt's widget layout mechanism. There would be more control over the arrangement of the components in this case (For example a `vbox{}` and `hbox{}` component that layout their children vertically or horizontally).

- **More editors**

The addition of types would allow more special purpose editors like the condition list editor and the dali color picker. A capacitance value generating editor for example (that could be used to generate the capacitances in the element definition files). Another editor could be a "pattern picker" for the fill patterns used by `geteps`lay.

- **Additional properties**

Additional properties can also be very useful. For example:

- The Qt `whatis` mechanism could be implemented so a `whatis` property can be added. This would enhance the context sensitive help available to the end user.
- More properties that control the looks of a component could be added, for example a `font` property that influences the font used in the component or a `background` property to set the background color of a component.

- **Property warnings**

Warnings or errors should be given if required properties are missing or if forbidden or ignored properties are used

- **Better default values**

If properties are unspecified but can be used in a certain context, sensible defaults could be chosen. For example, if the `title` property of generator component is not given it could use the filename specified in the `filename` property.

- **Better conditional output**

In some cases (a good example is the parameter list) the file generation could be simplified if some built-in construction is available for empty parameter values. This would avoid long lists of `if` constructions. This could be implemented by some general mechanism or a parameter specific mechanism.





## Chapter 6

# Conclusion

In this final chapter we will compare the resulting application with the requirements stated in Chapter 2.

### Platform independence

By using Qt, the STL (standard template library) and the `gcc` compiler we have made sure that the tools necessary to compile the application are available on all target platforms. Both Qt and `gcc` are freely available for all required platforms. All recent versions of `gcc` include a good implementation of the STL.

These are not sufficient conditions to *guarantee* platform independence. However, the areas covered by these tools are usually the areas where platform independencies arise. The application has been successfully tested on both Linux and Solaris. Some tests were also performed on HP-UX and no problems were encountered.

The `makefile` generator tool `tmake` also takes away the need to manually configure the `makefiles` for each target platform.

### Integration with SPACE

Integration with SPACE is available. The application can read the file describing the processes (`processlist`) in the SPACE process tree and the user can integrate the generated technology files into the SPACE process tree if desired. The `processlist` file describing the available processes is automatically updated to reflect the changes that were made.

Unfortunately, the application cannot read the technology files in the SPACE process tree, it can only read its own file format.

### Technology file generation

The technology files specified in the requirements can all be generated. These are:

- *maskdata*, which defines the layers present in the process and the colors used to represent them in the programs and their output.
- *space.xxx.s*, the element definition files used by SPACE.

- *space.xxx.p*, the parameter files used by space.
- *bmlist.gds*, which provides a mapping between the GDS layer format and the format used by SPACE.
- *xspicerc*, a control file that specifies which models are used for the devices.

It should be noted that the space parameter file `space.xxx.p` cannot be completely generated yet. Not all of the parameters have been included in the first version of the configuration file.

## The configuration file language

The configuration file language provides the basic functionality needed. The language can be made more effective if additional language constructions are implemented. The language constructions for “chopping” and macro definitions can appear mangled to new users. A different syntax could be considered.

## Flexibility

Another requirement is the possibility to add new files to the list above or to change the format of these files in a flexible way. This flexibility has been achieved by making use of a configuration file. The configuration file contains the specification of the user interface and the generators that use this user interface to generate the technology files. These files can be changed without the need to recompile the application.

Flexibility with regard to the source code has been achieved by using a good source code documentation application and by using some well-known design patterns that make understanding the code easier.

The final conclusion is that the developed application meets the requirements specified in Chapter 2.

# Bibliography

- [Bur01] Xander Burgerhout. Spock source code documentation. December 2001.
- [FS98] Susan Fowler and Victor Stanwick. *GUI Design Handbook*. McGraw-Hill, New York, 1998.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, first edition, 1995.
- [Tro] TrollTech. Qt, the cross-platform c++ gui framework. <http://www.trolltech.com>.
- [vdMvGBE99] N.P. van der Meijs, A.J. van Genderen, F. Beeftink, and P.J.H. Elias. Space user's manual. CAS Report ET-NT 92.21, Delft University of Technology, Department of Electrical Engineering, Delft, June 1999.
- [vH] Dimitri van Heesch. Doxygen. <http://www.doxygen.org>.



## Appendix A

# Development environment

Developing an application is usually done with many tools. Some tools are required for the development (like the compiler). Other tools increase the programmers productivity.

The main goal of this chapter is to present the environment in which the application was developed. This means the used tools and their interaction will briefly be discussed.

First of all, the compiler and the platform will be described. This is followed by a section on **Makefile** generation. The **Makefile** can be a major headache to developers, especially if multiple platforms are involved. The source code has been documented using a tool called Doxygen. A short description of doxygen is given in Section A.3.

### A.1 Platform and compiler

The platforms used most actively during the development of the application are Solaris and Linux. A few compilations have been done on HP-UX as well and these presented no problems.

The compiler used was **gcc 2.95.2**. The **gcc** compiler is a free compiler that is available on practically every platform.

### A.2 Makefile generation

Makefiles are a necessary evil when developing applications on Unix or Linux platforms. This is because there is no good integrated development environment (like Visual Studio for Windows) available that works on all the platforms that need to be supported. This is mainly due to the peculiarities of each platform.

A solution to this problem is **tmake**. **tmake** is a small utility written by the creators of Qt, Trolltech [Tro]. This utility uses templates and a small input file containing some options and the sources to generate a **Makefile**. The templates hide the peculiarities of each platform, thus establishing a platform independent interface.

Another possibility is to use `autoconf`. The overhead involved with `autoconf` is larger, though.

As was mentioned above, the `tmake` tool uses templates for the generation of makefiles. There are templates for recursive makes and projects. It is also possible to use custom templates. For the development of the application a custom template was created that supports building library and test applications in a single run as well as support for `lex/flex` `yacc/bison` parser compilation.

## A.3 Doxygen

Source code documentation can be generated with `doxygen` [vH]. Another candidate for this position was `kdoc`. However, `kdoc` forces the documentation completely into the header files, making the header files unreadable. `Doxygen` allows documentation virtually everywhere, resulting in clean well-readable source files.

### A.3.1 Possible output formats

As an added bonus, `doxygen` can generate the documentation in many formats:

- HTML format
- $\text{\LaTeX}$
- Man page format
- Rich Text Format (RTF) which can be read by Microsoft Word
- XML

It can even generate a cgi script that can be used as a search engine to search through the HTML documentation.

### A.3.2 Doxygen options

The output of `doxygen` can be influenced with the many features present. It is possible to extract everything, even if the code is undocumented (this shows you the structure of the source code). It is also possible to extract only the documented code. Private and static members can be ignored if desired.

To keep the printed source code reference a reasonable size and to save some trees in general, the documentation was created for documented items only, leaving out the private methods and members.

### A.3.3 Collaboration diagrams

If the `graphviz` package is installed, `doxygen` can make use of `dot`, which is a part of that package. With `dot` it is possible to create collaboration diagrams. Some of the diagrams created by `dot` are used in this report, for example Figure 3.5.

## A.4 Qt Designer

Qt Designer became available near the end of the project. It is a user interface editor like X-Designer. It is possible to click together a dialog box and generate the source code that builds the dialog box. The technology file generation dialog box was created with Qt Designer as an experiment.

To use the generated source a new class has to be derived from the generated class. This derived class implements the desired functionality. In the case of the technology file generation dialog box this is the `CGeneratedDlg` class.

The ease with which a user interface can be created with Qt Designer is remarkable. Qt supports automatic widget layout and with Qt Designer this becomes even easier then before.

Functionality can easily be added to the dialog boxes designed with Qt Designer. By using inheritance, the only code that needs to be written is the code that must provide the needed functionality. This makes Qt Designer a true rapid application development tool.

The only downside of Qt Designer is that is (currently) only possible to create widgets and dialog boxes. It is not possible to create an application's framework.





## Appendix B

# Testing and debugging the application

Testing and debugging applications under development is usually not very difficult. As soon as the application starts to work with events like signals and slots debugging can become difficult.

The main goal of this chapter is to present the testing and debugging methods used, in particular the way Qt's signal/slot mechanism was debugged.

Section B.1 discusses the general test – debug – develop cycle commonly used by the programmer. Section B.2 will focus more on the debugging techniques applicable to signal/slot debugging.

### B.1 The test – debug – develop cycle

The general test – debug – develop cycle is depicted in Figure B.1. The diagram starts with the applications' source code at some point in time. This source must be compiled to obtain the executable. If a typing error was made or a method forgotten to implement, the compiler or linker will not be successful and the source must be edited. This is usually due to a simple mistake on the programmers part.

After the source code has been successfully compiled and linked, the functionality just added to application needs to be tested. This is represented by the input of test vectors. What these test vectors are, depends on the application and the functionality being tested. The tester (in this case the programmer) then either marks the added functionality as successful or unsuccessful.

If testing was successful, the programmer can continue to implement the next feature. If testing failed, a bug is present in the application. The bug must then be located. The bug is usually related to feature that was added but this is not necessarily the case (either because of poor testing or because some other part of the code could not be tested before this part was ready).

Locating a true bug can be quite difficult. It is usually the most time-consuming part of development and is therefore shown in grey in the figure. Once the bug

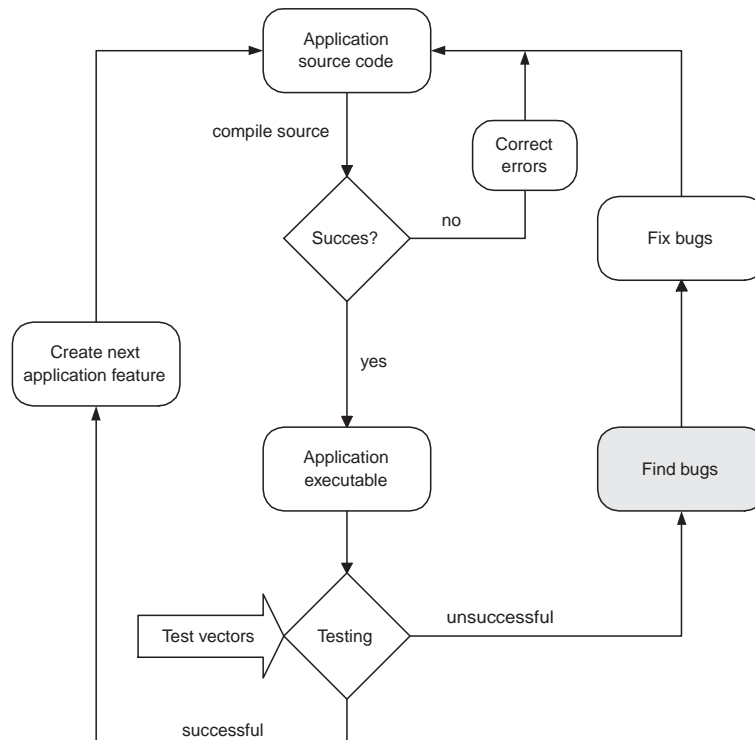


Figure B.1: The test – debug – develop cycle

has been identified a solution can be found easily in most cases. The cycle then continues normally.

## B.2 Debug techniques used

The best technique to find a bug depends on the type of bug encountered. The bugs encountered most are:

- Unexpected behaviour. The application does not perform its task correctly or not all.
- Segmentation faults/bus errors. The application “crashes”. Segmentation faults are usually due to an abused pointer.
- Crashes or unexpected behaviour associated with signals/slots.

Useful approaches for finding the location (or reason) for these bugs will be discussed next.

### B.2.1 Unexpected behaviour

An example of unexpected behaviour could be the erroneous generation of the text resulting from the `foreach` loops in the configuration file. The algorithm

was either not correct or not correctly implemented. To ascertain the exact location or reason for this bug it necessary to gain more insight into how the algorithm is “run”.

The best way to gain insight into the inner workings of an algorithm is by using a debugger. An excellent debugger is `gdb`. Unfortunately, it has no graphical front end. This means some experience is required to use `gdb` effectively. It is therefore recommended to use a graphical front-end to `gdb` like `DDD` (the Data Display Debugger), `xgdb` or even `xemacs`. Another option is `kdbg`. These debuggers allow step-by-step execution of the source code and inspection of the values of the variables used.

Another useful method could be to use debug statements. These print out information about the state of the application at strategic places. If a smart `#define` statement is used it is possible to disable the output for release builds.

### B.2.2 Locating segmentation faults

Contrary to unexpected behaviour, segmentation faults can be located quite easily. If the application is run from the debugger it will report the location the segmentation fault occurred. In `gdb` this can be done with the backtrace command `bt`. This lists a backtrace of the stack, showing exactly how the application reached the point where the segmentation fault occurred.

Once the location of the segmentation fault is known the reason for it is usually (but not always) clear and fixing the bug can begin.

### B.2.3 Signal/slot debugging

The signal/slot mechanism provided by Qt presents a challenge in case of bugs. Segmentation faults that occur right after a signal can be extremely difficult to locate or solve.

The signal/slot mechanism is available to any class that inherits the `QObject` class. This includes all widgets. If a backtrace of the stack is requested right after a segmentation fault it will only display methods used by these Qt objects. It is very difficult to pinpoint the exact object that sent the signal (and often caused the error).

#### A common signal/slot pitfall

An error often made that is difficult to find is the deletion of the sending object from the receiving slot. For example, object `A` sends a signal to object `B`. Because of this signal, `B` tells the manager of `A` objects `C` to delete certain `A` objects. If one of these `A` objects was the object that originally sent the message a segmentation fault is very likely.

The reason for this is the fact that emitting a signal is nothing more than calling a method. Once the signal method completes it will eventually return to the place the signal was emitted. If that object has been deleted this location is no longer valid and a segmentation fault will occur.

For more insights into how the signal/slot mechanism is implemented the interested user is recommend to take a look at the output of the `meta object compiler` and the source of the `QObject` class.

**Tips**

Some additional tips:

- Compile Qt with debugging on and use this library during development. Warnings about unconnected signals/slots will then be issued by Qt. It is also possible to dump information about a `QObject` tree.
- The meta object compiler `moc` also has a debugging feature. Enabling this feature can be useful to track signals/slots.

## Appendix C

# About the CD-ROM

The CD-ROM accompanying this report contains the results of the project. The table below lists the locations of everything on the CD-ROM.

Location	Description
<code>binaries\examples</code>	Contains sample processes for use with the application.
<code>binaries\Linux</code>	Contains binaries for Linux
<code>binaries\Solaris</code>	Contains binaries for Solaris
<code>cvs_repository</code>	Contains the CVS repository used during the development of the application.
<code>documentation\nonpostscript pictures</code>	Contains non-postscript versions of the pictures used in this thesis.
<code>documentation\paper</code>	Contains the $\text{\LaTeX}$ source of the paper written for this project.
<code>documentation\source code</code>	Contains the source code documentation
<code>documentation\source code</code>	Contains the $\text{\LaTeX}$ source of this thesis.
<code>presentation</code>	Contains files used for the presentation.
<code>spock</code>	Contains the sources of the application.



## Appendix D

# A sample configuration file

Below a sample configuration file is given. To save some trees, the example is a *part* of an early configuration file created during development of the application. Many of the language constructions are put into action so it makes a nice example.

```
// This is a SPOCK version 1.0 user interface specification...
//
// =====
// Please do not "just" change this file. You could seriously break
// the spock program. Only edit this file if you know what actions
// will brake compatibility with your saved processes.
// For more information, please refer to the SPOCK source code documentation
// or Xander Burgerhout's graduation report.
//
// Consider this file as part of the source.
//
// !!YOU HAVE BEEN WARNED!!
// =====
// =====
// Icon path.
// The icon (pixmap) path specifies where pixmap(...) properties look for the
// icons.
ICONPATH = "${ICDPATH}/lib/spock"
// =====
// Defines.
define dropdown layer {
    adddatafrom(maskdata_sheet.name);
}

define combobox layer_combo {
    adddatafrom(maskdata_sheet.name);
}

define dropdown layer_gnd {
    item atgnd { title("@gnd"); }
    adddatafrom(maskdata_sheet.name);
}

define dropdown layer_sub {
    item atsub { title("@sub"); }
    adddatafrom(maskdata_sheet.name);
}

define dropdown layer_gndsub {
    item atsub { title("@sub"); }
    item atgnd { title("@gnd"); }
    adddatafrom(maskdata_sheet.name);
}

// Defines a carrier type
define combobox def_carr_type {
    item n { title("n doped conductor"); }
    item p { title("p doped conductor"); }
    item m { title("metal"); }
    default(m);
}

// Defines a transistor type
define dropdown def_trans_type {
    item ver { title("vertical"); }
    item lat { title("lateral"); }
    default(ver);
}
```

```

define dropdown def_layer_type {
    item normal { title("normal"); }
    item interconnect { title("interconnect"); }
    item symbolic { title("symbolic"); }
    default(normal);
}

define conditionlist condlist_simp {
    option("noextended");
    adddatafrom(maskdata_sheet.name, maskdata_sheet.comment);
}

define conditionlist condlist_ext {
    option("extended");
    adddatafrom(maskdata_sheet.name, maskdata_sheet.comment);
}

define dropdown capa_types {
    item normal { title("default"); }
    item junction { title("junction"); }
    default(normal);
}

////////////////////////////////////
// Element definition user interface specification.
//
// This part consists mostly of sections. In the tabpages part of this file,
// these sections are used to create the element definition tabpage.

section units {
    title("Unit specification");
    hint("Use this to specify a base for the values you enter");
    pixmap("section_icons/units.xpm");

    paramlist units_list {
        real resistance {
            title("Sheet resistance unit:");
            unit("in ohm");
            hint("You can specify a base for the values you enter"
                "will enter below. Use as follows:\n"
                "If you do not want to type 3e-6, 8e-6 12.5e-6, etc.\n"
                "but 3, 8, 12.5 instead, type 1e-6 here.");
        }
        real c_resistance {
            title("Contact resistance unit:");
            unit("in ohm per squared meter");
        }
        real a_capacitance {
            title("Area capacitance unit:");
            unit("in farad per squared meter");
        }
        real e_capacitance {
            title("Edge or lateral capacitance unit:");
            unit("in farad per meter");
        }
        real distance {
            title("Capacitance unit:");
            unit("in micron");
        }
    }

    // This is for Space 3D
    real vdimension {
        title("Vertical dimension unit:");
        unit("in micron");
    }

    real shape {
        title("Edge shape list dimension unit:");
        unit("in micron");
    }
}

section conductors {
    title("Conductor list");
    hint ("Definitions for the conducting layers in the circuit");
    pixmap("section_icons/conductor.xpm");

    spreadsheet conductors_sheet {
        identifier cond_type { title("Conductor type"); }
        identifier name { title("Name"); }
        condlist_simp cond_list { title("Condition list"); }
        layer mask { title("Mask"); }
        real sheet_res {
            title("Sheet resistivity");
            align(right);
        }
        def_carr_type carr_type { title("Carrier type"); }
        string comment { title("Comment"); }
    }
}

section fets {
    title("Field-effect transistor (FET) list");
    hint("FET definitions");
    pixmap("section_icons/fet.xpm");

```



```

spreadsheet fets_sheet {
    identifier    name { title("Name"); }
    condlist_simp cond_list { title("Condition list"); }
    layer         mask_g { title("Gate mask"); }
    layer         mask_ds { title("Drain/source mask"); }
    layer_sub     mask_b { title("Bulk mask"); }
    string        comment { title("Comment"); }
}

}

section bjts {
    title("Bipolar transistor list");
    pixmap("section_icons/bipolar.xpm");

    spreadsheet bjts_sheet {
        identifier    name { title("Name"); }
        condlist_ext  cond_list { title("Condition list"); }
        def_trans_type ttype { title("Transistor type"); }
        layer         mask_em { title("Emitter mask"); }
        layer         mask_ba { title("Base mask"); }
        layer         mask_co { title("Collector mask"); }
        string        comment { title("Comment"); }
    }
}

}

section connects {
    title("Connect list");
    pixmap("section_icons/connect.xpm");

    spreadsheet connects_sheet {
        identifier    name { title("Name"); }
        conditionlist cond_list { title("Condition list"); }
        layer         mask_1 { title("Mask 1"); }
        layer         mask_2 { title("Mask 2"); }
        string        comment { title("Comment"); }
    }
}

}

section contacts {
    title("Contact list");
    pixmap("section_icons/contact.xpm");

    spreadsheet contacts_sheet {
        identifier    contact_type { title("Contact type"); }
        identifier    name { title("Name"); }
        condlist_simp cond_list { title("Condition list"); }
        layer_sub     mask_1 { title("Mask 1"); }
        layer_sub     mask_2 { title("Mask 2"); }
        real          resistivity { title("Resistivity"); }
        string        comment { title("Comment"); }
    }
}

}

section capacitances {
    title("Capacitance list");
    pixmap("section_icons/capacitance.xpm");

    spreadsheet cap_sheet {
        capa_types    cap_type { title("Type"); }
        identifier    subtype { title("Sub type"); }
        identifier    name { title("Name"); }
        condlist_ext  cond_list { title("Condition list"); }
        layer         mask_1 { title("Mask 1"); }
        layer         mask_2 { title("Mask 2"); }
        string        capacitivity { title("Capacitivity"); }
        string        comment { title("Comment"); }
    }
}

}

////////////////////////////////////////
// Maskdata specification

section maskdata_info {
    title("Layers");
    pixmap("section_icons/layers.xpm");

    spreadsheet maskdata_sheet {
        integer[0..10] ID { title("Layer ID"); }
        identifier    name { title("Layer name"); }
        def_layer_type layertype { title("Layer type"); }
        dalistyle     dali { title("Dali draw style"); }
        color         xspace { title("XSpace draw color"); }
        string        comment { title("Comment"); }
    }
}

}

////////////////////////////////////////
// Space parameters
section space_params_section {
    title("Space parameters");

    paramlist space_params {
        integer min_art_degree {
            title("Minimal articulation degree");
            hint("The articulation degree is the number of pieces in which\n"
                "the resistance graph would break if the node and its \n"
                "connected resistances were removed");
            default("3");
        }
    }
}

```

---

```

integer min_degree {
  title("Minimal degree");
  hint("Nodes with a degree >= this value and an articulation\n"
        "degree > 1 will also be retained in the final network.");
  default("4");
}
real min_res {
  title("Minimal resistance");
  unit("ohm");
  hint("This heuristic deletes small resistances from the network\n"
        "via Gaussian elimination of one of the nodes that is\n"
        "connected to the resistance.");
  default("100");
}
real min_sep_res {
  title("Minimal separation resistance");
  unit("ohm");
  hint("This heuristic deletes small resistances from the network\n"
        "by joining the two nodes that are connected by the resistance.");
  default("10");
}
real max_par_res {
  title("Maximal parallel resistance");
  unit("ohm");
  hint("This heuristic prevents the occurrence of high-ohmic shunt\n"
        "paths between two nodes.");
  default("25");
}

// no_neg_res comes here.

real min_coup_cap {
  title("Minimal coupling capacitance ratio");
  hint("If, for both nodes a coupling capacitance is connected to,\n"
        "it holds that the ratio between the absolute value of the\n"
        "coupling capacitance and the value of the ground/substrate\n"
        "capacitance of the same type of that node, is less than this\n"
        "parameter, then the value of the coupling capacitance is\n"
        "added to the ground capacitances of the two nodes and the\n"
        "coupling capacitance is removed.");
  default("0.04");
}
}
}

//////////
// Getepsplay

define dropdown term_text_align {
  item al_00 { title("Center of text at center of terminal"); }
  item al_n10 { title("Left-hand of text at center of terminal"); }
  item al_10 { title("Right-hand of text at center of terminal"); }
  item al_01 { title("Text centered below center of terminal"); }
  item al_0n1 { title("Text centered above center of terminal"); }
  default(al_00);
}

section getepsplay_preamble {
  title("General settings");
  paramlist getepsplay_pl {
    string includefile_nelsis {
      title("Include file from nelsis");
      hint("Includes a postscript prolog file.\n"
            "The filename given must be relative to $NELSISHOME/lib/\n"
            "NOTE: only one level of inclusion is supported.");
      default("epsplay.pro");
    }
    string includefile_home {
      title("Include file from home directory");
      hint("Includes a postscript prolog file.\n"
            "The filename given must be relative to the home directory.\n"
            "NOTE: only one level of inclusion is supported.");
      default("");
    }
    string mask_order {
      title("Mask draw order");
      hint("Specify in which order the masks must be drawn.\n");
    }
    string plotfont {
      title("Font for terminal names");
      hint("Any available PostScript font is acceptable.");
      default("Helvetica-Bold");
    }
    integer font_min {
      title("Minimum text size");
      unit("1/72 inch");
      hint("Minimum size of text in printer's points");
      default("6");
    }
    integer font_max {
      title("Maximum text size");
      unit("1/72 inch");
      hint("Maximum size of text in printer's points");
      default("14");
    }
    integer font_lambda {
      title("Preferred size of text in lambda's");
      unit("lambda's");
      hint("Based on the scaling of the particular layout in order to\n");
    }
  }
}

```

[illegible]

```

"#" (terminals/labels may be\n"
"#" defined for this layer).\n"
"#" type = 2: symbolic layer\n"
"#\n"
"#" Pattern-Generate (2): field 4: job number\n"
"#" Only used by PG-tape field 5: mask type\n"
"#" programs type = 0: negative\n"
"#" type = 1: positive\n"
"#\n"
"#" ColorMask Terminals (2) field 6: color number\n"
"#" (Obsolete) field 7: fill style\n"
"#\n"
"#" (Sea)Dali (2) field 8: color number\n"
"#" 0=black, 1=red, 2=green,\n"
"#" 3=yellow, 4=blue, 5=violet,\n"
"#" 6=aqua, 7=white\n"
"#" field 9: fill style\n"
"#" 0=hashed, 1=solid, 2=hollow\n"
"#" 3,4,5 = 12,25,50% hash+outline\n"
"#" 6,7,8 = idem, no outline\n"
"#\n"
"#" Plotter (2) field 10: pen number\n"
"#" (Obsolete) 1=black, 2=red, 3=yellow,\n"
"#" 4=green, 5=brown, 6=violet,\n"
"#" 7=blue, 8=aqua\n"
"#" field 11: fill style\n"
"#\n#\n"

maskdata_sheet[numrows]
" " application[processname] "\n\n" application[processdesc] "\n\n"
" ID\tname\ttype\tPG_tape CM\tDali\tplot comment\n"

foreach(maskdata_sheet, row) {
  ID
  "\t" name "\t"
  layertype
  "\t1 0 1 0\t"
  dali[clr] " "
  dali[style]
  "\t1 0 \n"
  comment
  "\n\n"
}
}

generator space_def_s_gen
{
  filename("space.def.s");
  title("SPACE Element definition file");

  map def_carr_type {
    n = "n";
    p = "p";
    m = "m";
  }

  generate {
    "## This file was generated by SPOCK on " application[date]
    " at " application[time] ".\n"
    "####\n"
    " " application[processname] " - " application[processdesc] "\n"

    "#\n# Masks:\n"
    foreach(maskdata_sheet, row) {
      " " name "\t" comment "\n"
    }
    "## See also the maskdata file.\n"
    "####\n"

    "unit resistance " units.units_list.resistance "\t# ohm\n"
    "unit c_resistance " units.units_list.c_resistance "\t# ohm/m^2\n"
    "unit a_capacitance " units_list.a_capacitance "\t# F/m^2\n"
    "unit e_capacitance " units_list.e_capacitance "\t# F/m\n"
    "unit distance " units_list.distance "\t# micron\n"
    "unit vdimension " units_list.vdimension "\t# micron\n"
    "unit shape " units_list.shape "\t# micron\n\n"

    "colors:\n"
    foreach(maskdata_sheet, row) {
      "\t" name "\t" xspace "\n"
    }
    "\n"

    foreach $t (conductors_sheet, cond_type) {
      "conductors " $t ":\n"
      " # name:condition_list:mask:sheet_resistivity:carrier_type\n"
      foreach(conductors_sheet, row) {
        if (cond_type == $t) {
          "\t" name ":" cond_list ":" mask ":" sheet_res ":" carr_type "\t\t" comment "\n"
        }
      }
    }
    "\n"

    "fets:\n"
    "#\tname:condition list:mask_g mask_ds:mask_b\n"
    foreach(fets.fets_sheet, row) {

```

```

        "\t" name ":" cond_list ":" mask_g " " mask_ds ":" mask_b "\t\t#" comment "\n"
    }
    "\n"

    "bjts:\n"
    "#\tname:condition_list:type:mask_em mask_ba mask_co\n"
    foreach(bjts.bjts_sheet, row) {
        "\t" name ":" cond_list ":" ttype ":" mask_em " " mask_ba " " mask_co "\t\t#" comment "\n"
    }
    "\n"

    "connects:\n"
    "#\tname:condition_list:type:mask1 mask2\n"
    foreach(connects.connects_sheet, row) {
        "\t" name ":" cond_list ":" mask_1 " " mask_2 "\t\t#" comment "\n"
    }
    "\n"

    foreach($t (contacts_sheet, contact_type) {
        "contacts " $t ":\n"
        "#\tname:condition_list:type:mask1 mask2:resistivity\n"
        foreach(contacts.contacts_sheet, row) {
            if (contact_type == $t) {
                "\t" name ":" cond_list ":" mask_1 " " mask_2 ":" resistivity "\t\t#" comment "\n"
            }
        }
    }
    "\n"
}

foreach($t (cap_sheet, cap_type) {
    foreach($s (cap_sheet, subtype) {
        "$t " capacitances " $s ":\n"
        "#\tname:condition_list:type:mask1 [mask2]:capacitivy\n"
        foreach(capacitances.cap_sheet, row) {
            if (cap_type == $t) {
                if (subtype == $s) {
                    "\t" name ":" cond_list ":" mask_1 " " mask_2 ":" capacitivy "\t\t#" comment "\n"
                }
            }
        }
    }
}
}

// //////////////////////////////////////
// This is for space3d only

"vdimensions:\n"
"#\tname:condition_list:mask:bottom thickness\n"
foreach(vdimensions_sheet, row) {
    "\t" name ":" cond_list ":" mask ":" bottom " " thickness "\t#" comment "\n"
}
"\n"

"eshapes:\n"
"#\tname:condition_list:mask:dx b dxt\n"
foreach(eshapes_sheet, row) {
    "\t" name ":" cond_list ":" mask ":" dx b " " dxt "\t#" comment "\n"
}
"\n"

"dielectrics:\n"
"#\tname permittivity bottom\n"
foreach(dielectrics_sheet, row) {
    "\t" name " " permittivity " " bottom "\t#" comment "\n"
}
}

}

generator space_param_gen
{
    filename("space.def.p");
    title("SPACE parameter file");

    generate {
        "## This file was generated by SPOCK on " application[date]
        " at " application[time] ".\n"
        "####\n"
        "## SPACE parameter file for " application[processname] " - " application[processdesc] "\n"
        "#####\n"

        if (space_params.min_art_degree != "") {
            "min_art_degree " space_params.min_art_degree "\n"
        }

        if (space_params.min_degree != "") {
            "min_degree " space_params.min_degree "\n"
        }

        if (space_params.min_res != "") {
            "min_res " space_params.min_res "\t# ohm\n"
        }

        if (space_params.min_sep_res != "") {
            "min_sep_res " space_params.min_sep_res "\t# ohm\n"
        }

        if (space_params.max_par_res != "") {

```

---

```

        "min_par_res          " space_params.max_par_res "\n"
    }

    if (space_params.min_coup_cap != "") {
        "min_coup_cap          " space_params.min_coup_cap "\n"
    }
}

generator getepsly_gen {
    title("Getepsly PostScript settings");
    filename("epsly.def");

    map term_text_align {
        al_00 = "0 0";
        al_n10 = "-1 0";
        al_10 = "1 0";
        al_01 = "0 1";
        al_0n1 = "0 -1";
    }

    generate {
        "% This file was generated by SPOCK on " application[date] " at " application[time] ".\n"
        "% getepsly customization for the " application[processname] " process.\n\n"

        if (getepsly_pl.includefile_nelsis != "") {
            "%Include <" getepsly_pl.includefile_nelsis ">\n"
        }
        if (getepsly_pl.includefile_home != "") {
            "%Include \" " getepsly_pl.includefile_nelsis "\"\n"
        }
        "\n"
        "%Order: " getepsly_pl.mask_order "\n"
        "[ " getepsly_pl.font_min " " getepsly_pl.font_max " " getepsly_pl.font_lambda
        "/" getepsly_pl.plotfont " ] plotFont\n"
        "[ " getepsly_pl.termtext " ] termTextAlignment\n\n"

        "% mask\tpattern\tscale\tlinewidth\n"
        foreach (getepsly_sheet, row) {
            "[ (" mask_name ") \t" pattern "\t" scale "\t" linewidth " ] defineStyle\n"
        }
    }
}

```