The implementation of a system description language and its semantic functions
The implementation of a system description language and its semantic functions

C. van Reeuwijk
Dit proefschrift is goedgekeurd door de promotoren: Prof. dr. ir. J. Davidse en Prof. dr. ir. R.T. Boute.
# Contents

1 Introduction .......................... 7
   1.1 Current system description languages ....................... 8
   1.2 Formal function theory .................................. 8
   1.3 Interpretation of formal functions ......................... 9
   1.4 Semantic functions ...................................... 9
   1.5 System semantics .................................... 11
   1.6 Adirectional systems ................................ 12
   1.7 Applications of system semantics ......................... 12
   1.8 The ESPRIT project FORFUN ............................ 13
   1.9 The contents of this thesis ............................ 14

2 Mathematical background .............. 17
   2.1 Overview of mathematical notation used .................. 18
   2.2 Formal function theory .................................. 18
   2.3 Formal language theory .................................. 21
   2.4 $\lambda$-calculus and $\sigma$-calculus .................... 25

3 Electronics background ............... 31
   3.1 Terminology ........................................... 31
   3.2 Electronic components .................................. 32
   3.3 Deviations from ideal component behaviour ............... 37
   3.4 Circuit analysis ....................................... 39
   3.5 Design methods for analog circuits ....................... 48
4 Software engineering 56
  4.1 Reliability ........................................ 56
  4.2 Openness .......................................... 57
  4.3 Robustness ......................................... 58
  4.4 User friendliness ................................. 58
  4.5 Portability ......................................... 59
  4.6 Impact on the design and implementation of Glass. .... 59

5 The system description language Glass 61
  5.1 Additions to the $\lambda$- and $\sigma$-calculi ........... 62
  5.2 Types ................................................ 66
  5.3 The merger of $\lambda$-calculus and $\sigma$-calculus ... 68
  5.4 Representation in computer character set ............. 69
  5.5 The macro language ................................ 69
  5.6 Library support .................................... 70

6 Support for semantic functions 71
  6.1 Tm .................................................. 72
  6.2 Tm data-structure definition ....................... 73
  6.3 Tm textual data representation ..................... 74
  6.4 The text substitution language of Tm ............... 74
  6.5 Applications of Tm ................................ 78
  6.6 The Glass internal representation .................. 78
  6.7 Netlist representation ................................ 81

7 Implementation of semantic functions 83
  7.1 Flattening ......................................... 84
  7.2 Analog discrete-time signal-flow analysis ............ 91
  7.3 Generation of Maple equations ....................... 93
  7.4 Input generation for existing programs ............... 97

8 Non-standard interpretation of Glass 101
  8.1 Figure specifications .............................. 102
  8.2 Position specification atoms ....................... 102
  8.3 An example of a drawing description ................. 103
Preface

Although a thesis such as this one is supposed to be the work of one person, in most cases the author has had essential support from others. This is especially true for this thesis. In the first place, the entire work described in this thesis was envisioned by Raymond Boute, and in some sense this thesis only adds detail to that vision. Also, the other members of the ESPRIT FORFUN project team provided invaluable support through their own work and through our discussions. Detailed acknowledgements to them are given in chapter 1. I am grateful to all of them. I am also grateful to the ESPRIT organization for their support; it was in its own way also essential.

Although less concrete, the stimulating environment at the Electronics Research Laboratory helped considerably.

A number of people deserve special mention: Onno Kuijken, Martin Middelhoek, Arturo Sarmiento-Reyes and Hans Stoffels for their companionship; Eduard Kleihorst, Hans Stoffels, Chris Verhoeven, Kees Wissenburgh and J.B. Zaat-Jones for proofreading this thesis; and Jan Nusteling and Rob Janse for maintaining the various computers, and for their patience.

In some places in this thesis I was forced to choose between the masculine and feminine form of a word. With reluctance, I have followed the custom in these matters, and used the masculine form. However, the usage should be considered to include the feminine form.

Chapter 1

Introduction

This thesis describes the design of a description language for arbitrary systems, the construction of software to support this language, and the application of this language in the design of analog electronic circuits. In this thesis, a system stands for any composition of objects for which both its behaviour and its structure or realization are important. Therefore, the algebraic calculus expression $x \cdot x + 3 \cdot x$ is not a system since it does not suggest any structure. Conversely, when the structure of such an expression is taken into account, the calculus is no longer the standard algebraic calculus. For example, if the expression $x \cdot x + 3 \cdot x$ is interpreted as a description of the structure of a system (consisting of adders and multipliers), equivalences such as

$$x \cdot x + 3 \cdot x = x \cdot (x + 3)$$

are not valid, since in the first description two multipliers are used, and in the second description only one.

![Circuit Diagram](image)

Figure 1.1: an example of a system description language: a circuit diagram.
1.1 Current system description languages

In many disciplines where systems are designed, specialized languages have emerged to describe them. For example, in analog electronics circuit diagrams (figure 1.1) are widely used. Many other disciplines also use some graphical representation of their systems. In this thesis, however, I will restrict myself to textual languages.

Often such textual languages are input languages for simulation programs. For analog electronics, a popular input language is that of Spice [31, 58]. Many of these languages are derived from programming languages designed around 1960 such as FORTRAN. These older languages usually have a very informal definition, and often have an irregular language structure.

After 1960, new programming languages were developed, for example Pascal, Modula-2 and ADA, that have a more formal definition and a more regular language structure. All these programming languages are still imperative languages\(^1\). For a long time, the techniques used in these new programming languages were not applied to system description languages. Recently, however, a number of new description languages has been developed for digital electronics, for example VHDL [29] and ZEUS [19]. Because they have been derived from programming languages such as ADA and Modula-2, the description languages also have become imperative languages. However, an imperative programming language is an inconvenient basis for a circuit description language; to use such a language as a circuit description language, it must be interpreted as a program to construct the desired circuit or as a program to simulate the circuit. A more appropriate model for a system description language is offered by functional programming languages and formal function theory.

1.2 Formal function theory

I define formal function theory as the branch of mathematics that describes the manipulation rules on functions that are independent of the interpretation of the function. For example, the statement that the function definition

\[ f \in \mathcal{R} \to \mathcal{R}, f(x) = x^2 + x \]

is equivalent to

\[ f \in \mathcal{R} \to \mathcal{R}, f(y) = y^2 + y \]

is part of formal function theory. In this example, \( f \) is called a formal function. Formal function theory does not cover algebraic calculus. For example, the rule that

\[ f \in \mathcal{R} \to \mathcal{R}, f(x) = x^2 + x \]

\(^1\)An imperative language consists of a list of commands to an abstract machine. To execute the program the abstract machine executes the commands in the order they appear in the list. The order of execution may be changed by special commands for repetition, conditional execution, etc.
is equivalent to

\[ f \in \mathcal{R} \rightarrow \mathcal{R}, f(x) = x + x^2 \]

is not part of formal function theory but of algebraic calculus, since it is based on an interpretation of +. Algebraic calculus is one of the possible interpretations of formal functions. In this thesis, alternative interpretations of formal function definitions will be discussed that will have different equivalence rules.

### 1.3 Interpretation of formal functions

As has been suggested in section 1.2, formal function definitions have an obvious interpretation as mathematical functions. I will not elaborate this. Formal function definitions can also be interpreted as algorithms: descriptions of computations. For example, the expression

\[ 3 \cdot x + 5 \cdot y \]

can be interpreted as a set of computation commands:

- Multiply x by 3.
- Multiply y by 5.
- Add these two products.

This is called an operational interpretation or an imperative interpretation. Programming languages contain specialized forms of formal function definitions that have an operational interpretation. In imperative programming languages, the link to formal function definitions is very indirect, but functional programming languages such as Miranda [56, 55] are more directly derived from formal function definitions.

Formal function definitions can also be interpreted as system descriptions. For example, the expression

\[ 3 \cdot x + 5 \cdot y \]

can be interpreted as the description of a system as shown in figure 1.2 or as a system as shown in figure 1.3.

### 1.4 Semantic functions

Descriptions in any form must have an interpretation or meaning that is known to both the creator and the user of the descriptions. Often there are levels of interpretation. For example, if I write:
Figure 1.2: a system interpretation of $3 \cdot x + 5 \cdot y$.

Figure 1.3: another system interpretation of $3 \cdot x + 5 \cdot y$.

$$u \in \mathcal{R} \rightarrow \mathcal{R}, \ u(t) = \sin(\omega \cdot t)$$

to describe the behaviour of a certain voltage in a circuit, I could distinguish the following levels of interpretation:

- Curved lines.
- Symbols.
- Functions.
- A description of voltages in a circuit.

An interpretation can be described as a function that associates a meaning with all the possible descriptions. Such a function is called a semantic function or a meaning function. For example, the interpretation of (Dutch) traffic lights can be described with the following semantic function $m$ (for meaning):

$$m \in \{\text{red, yellow, green}\} \rightarrow \{\text{go, stop, stop if you can}\}$$

$m \ red = \ stop$

$m \ yellow = \ stop \ if \ you \ can$

$m \ green = \ go$
1.5 System semantics

In [6] and in [7], Boute has shown that formal function definitions can be interpreted as system descriptions. For example, the definition

\[ S \in (B \times B \times B) \rightarrow B, \quad S(a, b, s) = (s \land a) \lor (\neg s \land b) \]

can also be interpreted as the description of the structure of a digital circuit with two and gates, one or gate and one inverter, as shown in figure 1.4. The interpretation of \( B \) also varies. For example, in the standard mathematical interpretation it is the set of Booleans: \{true, false\}.

In the same article, he proposes to formalize such interpretations by means of semantic functions. He also remarks that many such semantic functions can be defined to give some behavioural or structural interpretation of a system description.

A similar concept has been used by Sheeran: In [3], Backus describes a programming language, FP, based on combinator calculus. In [52], Sheeran derives a circuit description language from FP called muFP which is intended for the description of digital circuits. However, in this case, the language is based on combinator calculus instead of formal function theory, and their language lacks the concept of multiple semantic functions. Middelhoek derives a different circuit description language from FP, FUN [30]. This language is more suitable than muFP for the description of analog circuits. However, languages based on combinator calculus are difficult to read for casual users (such as designers of electronic circuits).

An important feature of a system description language is that it is not restricted to one or two interpretations, but that it allows many semantic functions that give some behavioural or structural interpretation. For example, for the system description language used in this thesis, Glass, semantic functions have been written that:

- Generate input for circuit analysis programs such as Spice, see chapter 7.
- Do analog signal-flow analysis, see chapter 7.
• Calculate gate delays of digital combinatorial circuits, see [51].

All these semantic functions give a behavioural interpretation, but it is equally possible to incorporate circuit layout software in a semantic function. Moreover, the use of Glass is not restricted to the domain of electronic circuits; it could also be used in other domains. Therefore, Glass is called a system description language instead of a circuit description language.

1.6 Adirectional systems

By using formal function definitions as circuit descriptions, it is implied that there is a direction in the systems that are described. For the cases where this is an obstacle, Boute proposes in [8] to introduce a new notation. Like formal function definitions, these adirectional descriptions can be interpreted in many ways.

1.7 Applications of system semantics

In the design of a system, it is often useful to have multiple interpretations. For example, in the first steps of a design, simplified models of the components of a system are used. These models are refined in the subsequent steps of the design.

Usually only behavioural aspects are considered, since a system is designed for its behaviour. The structure of the system is only used inherently: the purpose of a design is to find a structure that approximates the desired behaviour as well as possible, and that can be implemented. The implementation of a system, once it has been designed, can also be described as a semantic function.

To support this design practice with software, the designer must be able to define his own semantic functions. This requires:

1. A programming environment that allows easy access to system descriptions.
2. Simulation support.

In the ESPRIT project FORFUN, described in the next section, the programming environment has been developed, but not the simulation support. For semantic functions for digital circuits, it is often possible to use existing imperative or functional languages to implement the simulation. In other cases, it may be possible to use existing simulation programs or symbolic computation programs, but in many cases no adequate support is available.
1.8 The ESPRIT project FORFUN

Based on his proposal for system interpretation of formal functions, Boute submitted in 1985 a proposal [5] for a research project to elaborate this idea to the European research funding organization ESPRIT. His proposal was accepted, and in 1986 the project was started. The project was assigned number 881; it had as title “Formal Description of Arbitrary Systems by means of Functional Languages” and as acronym FORFUN. The objectives of the project were (quoted from [5]):

- The design and implementation of an application-independent prototype system description environment (i.e. a coherent and representative set of software packages supporting formal system description on a widely available computer system).

- The further elaboration of the principles of system semantics to cover as yet unresolved issues.

- The design of a prototype system description language and its implementation in the aforesaid environment.

In the original proposal, separate plans were made for the design and implementation of description languages for analog and digital circuits. However, early in the project it was decided to merge these efforts.

During this project, a system description language, called Glass, was developed. This language is, in essence, a standardization of mathematical notation. Also, a representation of all Glass symbols in a common computer alphabet was defined. To make the implementation of semantic functions possible, a number of support programs were written and a set of semantic functions were used as examples.

For the Delft University of Technology, I was the main participant in the project, and the research described in this thesis was mainly done for this project. Inherent in such a project is that its research is not the property of one single person, but more or less of the entire group. Therefore, it is inevitable that some of the research described in this thesis is not my own. To be specific:

- Raymond Boute is the author of the original proposal and is responsible for the idea of system semantics, see [8], [4], [9], and [7].

- Riet Oolman developed the type system of Glass, see [34], and [35].

- Marc Seutter designed the Glass syntax, and wrote some versions of its parser, see [48], and [49].

- Eric Voss wrote the macro expander, and some versions of the Glass parser.

- Johan Vanslembrouck, Simon Klaver and Ron Doesborg wrote semantic functions for digital circuit descriptions, see [17].
The final results of the ESPRIT project are described in [51], [45] and [44].
Emphatically my own are:

- Tm and its libraries, see [42], [43] and chapter 6.
- The internal representation of Glass and work on other implementation issues, see [18] and chapter 6.
- The semantic functions described in chapter 7, see [38], [41], [45] and [40].

I also made some contributions to the design of the Glass language, see [38] and [39].
For Tm and its libraries a manual is available, but it is too large to include in this thesis; see [43].

More time than was originally foreseen was necessary to implement the support software. In fact, the complete prototype support software became available only shortly before the completion of the ESPRIT project, and then only on some systems. This left little time to develop semantic functions, which is reflected in this thesis: a considerable amount of space is devoted to language design and implementation.

One application of Glass envisioned at the beginning of the ESPRIT project was its use as a circuit description language in the circuit synthesis software that is being developed at the Delft University of Technology. Some aspects of this project are described in [54]. This planned application has become reality: Glass is being used in recent versions of a circuit synthesis program called Ampdes. Ampdes is described in chapter 9. As with the ESPRIT project, it is necessary to describe the work of others—in this case Hans Stoffels—in detail. My main contributions to Ampdes are:

- A Glass parser with semantic functions.
- The code generator Tm: Tm has been used extensively in Ampdes.

1.9 The contents of this thesis

The research described in this thesis bridges two very different research areas: declarative language design and the formalization of analog electronics. Therefore, I am faced with the difficult task of explaining my research results in such a way that both groups of potential readers find it sufficiently accessible. This implies in turn that a large number of pages must be spent on matters that are relatively evident to one or the other group.

First, chapters 2 and 3 describe the necessary mathematical and electronic background. They are necessary to understand the principles behind the system description language, Glass, the purpose of the described semantic functions, and the description of an example application: Ampdes. After this, the principles of software
engineering are outlined in chapter 4. These principles have also influenced the design of Glass and its semantic functions, although in a less visible way. In chapter 5, Glass is described and some of the motivations in its design. A brief overview of Glass is given in appendix B.

The implementation of semantic functions is described in chapters 6 and 7. To ensure easy implementation of semantic functions, I have written a special support program, Tm. This program and its application to semantic functions are described in chapter 6. In chapter 7, some example semantic functions are described. One semantic function is listed in appendix C. In chapter 8, an interpretation of Glass is described that was not foreseen during its design: a geometric interpretation.

Chapter 9 describes some (potential) applications of Glass and its semantic functions. It also discusses possible improvements and additions to Glass. Finally, in chapter 10 some conclusions are drawn.
Chapter 2

Mathematical background

To be able to understand this thesis, it is necessary to understand a number of topics in mathematics and computer science that may not be familiar to electronic engineers. Therefore, an informal introduction to these topics will be given in this chapter. Readers who are familiar with these topics may find that the coverage is overly simple. They must keep in mind, however, that a more thorough treatment would burden the reader with many details that are irrelevant for this thesis.

The topics discussed in this chapter are:

- Formal function theory (section 2.2). It describes the formal ‘function’ mechanism and its manipulation rules. Although a large part of the theory falls within the realm of elementary mathematics, there are some deviations from the standard theory that make it necessary to discuss it.

- Formal language theory (section 2.3). It describes the parsing of formal languages. A more complete introduction can be found in [1].

- $\lambda$-calculus and $\sigma$-calculus (section 2.4). A calculus consists of a set of strings (usually defined by a formal language) and a set of equivalence rules. A well-known example is operator calculus, often simply called ‘calculus’, which contains, for example, the equivalence rule

$$a + b = b + a$$

$\lambda$-calculus describes the essence of the abstraction mechanism of formal function theory. A more complete introduction can be found in [37]. The abstraction mechanism of $\lambda$-calculus is only partially suitable for a system description language, since it implies direction. For the more general case, also covering adirectional systems, a new calculus has been proposed in [8] and [7]. This calculus is called $\sigma$-calculus$^1$.

$^1$In some early publications the name $\beta$-calculus has been used. This was an unfortunate choice, and only the newer name should be used.
2.1 Overview of mathematical notation used

The following mathematical notation is used in this thesis:

\[ \mathcal{N} \] The set of all natural numbers.
\[ \mathcal{R} \] The set of all real numbers.
\[ \mathcal{C} \] The set of all complex numbers.
\[ \mathcal{R}_+ \] The set of all positive real numbers.
\[ a \in b \] \( a \) is an element of (is in) set \( b \).
\[ \{x \mid x \in A \land P \, x\} \] The set of all \( x \) in \( A \) with property \( P \, x \). For example:

\[ \{x \mid x \in \mathcal{N} \land \sqrt{x} \in \mathcal{N}\} \]

is the set of all \( x \) in \( \mathcal{N} \) with the property that the square root of \( x \) is also in \( \mathcal{N} \).

\[ \forall \] All quantor. For example,

\[ \forall a \in \mathcal{N} \, a^2 = a \cdot a \]

means: for all \( a \) in \( \mathcal{N} \), \( a^2 \) is equal to \( a \cdot a \).

\[ \exists \] Existence quantor. For example,

\[ \exists a \in \mathcal{N} \, a^2 = a + a \]

means: there is an \( a \) in \( \mathcal{N} \) such that \( a^2 \) is equal to \( a + a \).

2.2 Formal function theory

In the introduction, I define \textit{formal function theory} as the branch of mathematics that describes the manipulation rules on functions that are independent of the interpretation of the function. Formal function theory is concerned with the precise definition of the function manipulation rules of mathematics that every user of mathematics takes for granted. Although the familiar numerical functions are used for illustration, it is important to remember that formal function theory only provides rules for the manipulation of \textit{symbols}. Any interpretation of the symbols is not part of formal function theory.

2.2.1 functions

A function \( f \) from the set \( X \) to the set \( Y \) associates with every element \( x \) of \( X \) one element of \( Y \) called the \textit{image} of \( x \), written\footnote{The reader may be more familiar with the notation \( f(x) \), but, as will be explained in section 2.2.5, the notation without parentheses used in this thesis has some advantages. I will use parentheses only for emphasis or to overrule the precedence rules, for example to distinguish \( x \cdot (y + z) \) from \( x \cdot y + z \).
} \( f \, x \).
For example, the trigonometric function 'cos' associates with every element of \( \mathcal{R} \), one element of \( \mathcal{R} \). Note that the definition of 'function' does not exclude functions on more exotic sets such as the set \( \{a, b, c, d\} \) or the set of all cars in the world.

Some other definitions:

- The set \( X \) is the domain of \( f \), written \( D \ f \).
- The set \( Y \) is the codomain of \( f \), written \( C \ f \).
- The range of \( f \), written \( R \ f \), is the set \( \{f \ x \mid x \in Df\} \). It is a subset of the codomain.
- The set of all functions with domain \( X \) and codomain \( Y \) is written as \( X \rightarrow Y \).
- A partial function is a function that has an image for only a part of its domain. For the remainder of its domain it is said to be undefined.

For example, the function that takes the natural logarithm, \( \ln \), associates with every element of \( \mathcal{R}_+ \) an element of \( \mathcal{R} \). Thus, \( D \ \ln = \mathcal{R}_+ \), \( R \ \ln = \mathcal{R} \), and therefore \( \ln \in \mathcal{R}_+ \rightarrow \mathcal{R} \). Alternatively, \( \ln \) can be described as a partial function in \( \mathcal{R} \rightarrow \mathcal{R} \) which is undefined for \( \mathcal{R}_- \) and 0.

### 2.2.2 types

A type specification is a statement that an object (for example, a function) is an element of a given set of objects. For example:

\[
\ln \in \mathcal{R}_+ \rightarrow \mathcal{R}
\]

or:

\[
\ln \in \{f\mid \forall a \in \mathcal{R}_+ \forall b \in \mathcal{R} \ a > b \Rightarrow f \ a > f \ b\}
\]

or even:

\[
\ln \in \{f\mid f \ x = \ln \ x\}
\]

are correct type specifications. The type specification is intended as a clarification of the function definition.

Computer programs using descriptions based on formal function theory often check whether the definition of the formal functions and their applications agree with the specified type. If this verification is to be done automatically, severe restrictions are must be imposed on the permitted type expressions. Nevertheless, such a type check is a correctness proof of the software, be it a limited one, and is therefore very useful to improve the reliability of the software.
2.2.3 higher-order functions

A higher-order function is a function whose range or domain is itself a set of functions. The Fourier transform is a familiar example:

\[ F \in (\mathcal{R} \to \mathcal{C}) \to (\mathcal{R} \to \mathcal{C}), \quad F f \omega = \int_{-\infty}^{+\infty} (f x) \cdot e^{-j \omega t} \, dt \]

(As is customary in electronics textbooks, I will use \( j = \sqrt{-1} \), since \( i \) is reserved as a symbol for current.) As is indicated by its type, the Fourier transform function, \( F \), maps a function \( f \in \mathcal{R} \to \mathcal{R} \) onto a function \( f' = F f \), with \( f' \in \mathcal{R} \to \mathcal{C} \).

To reduce the number of necessary parentheses, it is customary to assume that, for any function \( f \) of suitable type:

\[ (f g) h = (f (g h)) \]

and that

\[ X \to Y \to Z = X \to (Y \to Z) \]

Thus, function application is left associative, and \( \to \) is right associative, see 2.3.6.

2.2.4 tuples and Cartesian products

A tuple with elements \( x \in X \) and \( y \in Y \) is an ordered list of \( x \) and \( y \), written \( (x, y) \). The type of a tuple is called the Cartesian product of the element types, written \( X \times Y \). The Cartesian product is defined as:

\[ X \times Y = \{ (x, y) | x \in X \wedge y \in Y \} \]

In general, a tuple can have any number of elements, including zero elements and one element. For example, \( (a, b, c) \) and \( (a) \) are valid tuples. The types of these tuples are still written as Cartesian products:

\[ (w, x, y, z) \in W \times X \times Y \times Z \]

If all elements of a tuple are of the same type, it is called an array, and the number of elements in the tuple is called its length. For an array, the Cartesian product can be abbreviated as in this example:

\[ X \times X \times X = X^3 \]

2.2.5 currying

The function notation used in this thesis differs from the usual one. For example, in the usual notation one would write:
\[ f \in \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}, \; f(a, b) = 3 \cdot a^2 + 4 \cdot b^2 \]

As is indicated by the type specification, \( f \) is a first-order function, and its parameters are 'packaged' in a tuple, although in this case the tuple is written as \((a, b)\) instead of \((a, b)\). It is always possible to replace such a definition with a definition of a higher-order function without the tuple packaging. For example, \( f \) could be defined as

\[ f \in \mathcal{R} \rightarrow \mathcal{R} \rightarrow \mathcal{R}, \; f(a, b) = 3 \cdot a^2 + 4 \cdot b^2 \]

This notation was introduced by Schönfinkel [46] and extensively used by Curry [13]; as a result it is known as currying. The advantage of this notation is that each parameter of the function can be applied separately, and results in a new function. For example, \( f 2 \) is a new function. If we call this function \( g \), it could also be defined as:

\[ g \in \mathcal{R} \rightarrow \mathcal{R}, \; g(b) = 3 \cdot 2^2 + 4 \cdot b^2 \]

Since it is always possible to use the curried version of a function without loss of generality, I will use it when convenient.

Currying is used in the macro language of Glass and in many functional programming languages, for example Miranda.

### 2.3 Formal language theory

In mathematics, a list is an array of arbitrary length of elements from a given set, called the base set. The list of length 0 is called the empty list, written \( \epsilon \). In formal language theory, the base set is called an alphabet, and the lists are called strings. A language is an arbitrary set of strings from a given alphabet. Thus, if the alphabet is \( \{\Diamond, \circ, \triangle, \square\} \), a language on this alphabet would be \( \{\epsilon, \square, \square\square, \square\square\square\} \). It is not possible to define larger languages with an exhaustive list of all elements. Instead, they are specified using a grammar.

#### 2.3.1 Grammars

A grammar of language \( L \) is a set of production rules for the strings in \( L \). For example, this is a production rule for \( n \):

\[ n \rightarrow 0 | 1 | 0 \; n \; | \; 1 \; n \]

Informally, this can be read as

To get an \( n \) string, you can take a single \('0'\) or \('1'\) character, or an \( n \) character preceded by a \('0'\) or \('1'\) character.
Obviously this rule will produce all strings consisting of ‘0’ or ‘1’ characters. Thus, the strings ‘0’, ‘00’ and ‘01001’ can be produced from \( a \).

In the grammars in this thesis, I will underline literal strings, as in \( 0 \) and \( 1 \), and separate alternatives with a bar (‘|’). Symbols that are defined and used within the grammar, such as \( a \), are called non-terminal symbols or non-terminals, while all other symbols are called terminal symbols or terminals. Both terminals and non-terminals are called terms.

This is an other example of a grammar:

\[
\begin{align*}
\text{asandbs} & \rightarrow \text{as bs} \\
\text{as} & \rightarrow \epsilon | \text{a as} \\
\text{bs} & \rightarrow \epsilon \\
\text{bs} & \rightarrow \text{b bs}
\end{align*}
\]

(Remember that \( \epsilon \) represents the empty string.) For \( \text{bs} \) two separate production rules are given to show that this style of definition is also possible. Informally, \( \text{asandbs} \) produces strings beginning with zero or more ‘a’ characters followed by zero or more ‘b’ characters. Thus, ‘aabb’, ‘aaaaa’ and ‘b’ can be produced, but ‘abba’ not.

Since constructs such as ‘zero or more “b”’ occur frequently, it is customary to introduce a special notation: ‘zero or more “b”’ is written as ‘\( \text{b}^* \)’. Here ‘*’ is called the Kleene star, and indicates that the preceding term may occur zero or more times. With that convention only one production rule for \( \text{asandbs} \) is necessary:

\[
\text{asandbs} \rightarrow a^* b^*
\]

### 2.3.2 Syntax

In general, the rules that describe the possible strings of a language are called the syntax of the language. These rules are not necessarily grammar rules, but might be described in some other way. For example, the rule that a variable may only be defined once may also be part of the syntax of a language, although it is very difficult to express this in the grammar of the language.

### 2.3.3 Parsing

In the implementation of a language, the set of production rules will not be used to produce strings, but to partition a given string into its relevant parts. This partitioning is called parsing. During parsing, an attempt is made to find a derivation of a given string using the production rules of the grammar.

Thus, to parse a numerical expression string such as ‘1 + 2 × 3’, the following grammar is impractical (‘_’ stands for space):

22
\[ \text{char} \rightarrow 0 | \ldots | \varnothing | \pm | \times | _ \]

\[ n \rightarrow \text{char}^* \]

since it does not result in a useful partitioning. The following grammar is much more useful:

\[ \text{blanks} \rightarrow _* \]
\[ \text{digit} \rightarrow 0 | \ldots | \varnothing \]
\[ \text{number} \rightarrow \text{blanks digit* blanks} \]
\[ \text{prod} \rightarrow \text{number} | \text{number} \times \text{prod} \]
\[ \text{sum} \rightarrow \text{prod} | \text{prod} + \text{sum} \]

This grammar could be used to parse strings such as ‘5 + 7 + 3 × 4’.

### 2.3.4 Parser Generators

Grammars are very suitable for the formal definition and parsing of input languages of computer programs. To be of use for a computer program, the formal language specified in a grammar must be translated to a parser program. This may be done by hand, but this is an error-prone task, and it is difficult to verify the correctness of the translation. Provided that some restrictions are observed, it is possible to do the translation automatically with a program called a parser generator.

A well-known example of a parser generator is the Unix program set yacc [24] and lex [28]. Using these programs, parsing is divided into two stages. The program lex generates a lexical analysis program that partitions the input text into a sequence of tokens such as ‘word’, ‘number’, etc. These tokens form the alphabet of the actual grammar. A parser for this grammar is generated by yacc.

### 2.3.5 Design of Formal Languages

The design of a language that must be parsed by a parser program requires careful consideration, taking into account the following conflicting design objectives:

- **No ambiguity.** Any language expression should have only one derivation.
- **Expressive.** A user should be able to express himself as freely as possible.
- **Compact.** The language should not require lengthy expressions for commonly used constructs.
- **Familiar.** As much as possible, one should use syntax that is familiar to the user of the language.
With some additional information, ambiguities can often be resolved, but this solution complicates the language implementation and often makes the language more difficult to understand. In designing a language, it is therefore necessary to strike the right balance between these design objectives.

The design of Glass has been influenced by these design considerations, although in most cases this will not be discussed explicitly in this thesis.

### 2.3.6 function and operator syntax

An important issue in many languages—including Glass—is the notation of function applications (a function name and some parameters). There are three possible notations:

1. **Prefix notation.** The function name precedes its parameters. For example:

   \[ f \ a \ b \]

   As explained in section 2.2.5, in the notation

   \[ f(a, b) \]

   the expression \((a, b)\) simply represents one—more complicated—parameter.

2. **Postfix notation.** The function name follows its parameters. For example:

   \[ b \ a \ f \]

3. **Infix notation.** In its traditional form, this notation can only be used for functions with two parameters. Usually the function is represented by a special symbol. For example:

   \[ a + b \]

   Symbols of infix functions are sometimes called operators, but this term is also used for higher-order functions, see section 2.2.3.

Despite its restrictions, infix notation is traditionally used for a number of arithmetic functions (+, −, /, etc.). Tradition also dictates a number of rules to resolve the ambiguities in infix notation. They are:

1. **Operator priority.** For example: \(a + b \cdot c = a + (b \cdot c)\).

2. **Associativity.** For example: \(a - b - c = (a - b) - c\). An operator such as \(-\) is said to be **left associative.** Other operators may be **right associative.** For example, \(\rightarrow\) is right associative, since \(X \rightarrow Y \rightarrow Z = X \rightarrow (Y \rightarrow Z)\).
Provided that the type of all functions is known, and that no function has a variable number of parameters, no additional rules are necessary to resolve ambiguities in prefix and postfix notation. For traditional function notation—\(f(a, b)\)—this condition is met, since each function has exactly one (tuple) parameter. For curried functions these conditions could also be met, but this is error-prone. Instead, it is customary to use parentheses to resolve any ambiguities. For example, one is expected to write:

\[ f(a)(b)(c) \]

### 2.4 \(\lambda\)-calculus and \(\sigma\)-calculus

In mathematical function definitions such as

\[ f \in \mathcal{R} \rightarrow \mathcal{R}, \quad f \ x = 2 \cdot x^2 + z \cdot x + 1 \]

it is permitted to replace all occurrences of the variable \(x\) by an occurrence of another variable. For example:

\[ f \in \mathcal{R} \rightarrow \mathcal{R}, \quad f \ a = 2 \cdot a^2 + z \cdot a + 1 \]

is equivalent. It is said that the variable \(x\) is abstracted or bound by the definition of \(f\). The variable \(z\), however, is not bound by the definition of \(f\), and is called free in this expression. There are some restrictions to these substitutions on abstracted values: it is not permitted to replace an abstracted variable by another abstracted one, or one that is free in the expression. Thus,

\[ f \in \mathcal{R} \rightarrow \mathcal{R}, \quad f \ z = 2 \cdot z^2 + z \cdot z + 1 \]

is not equivalent to the previous function definitions.

The same abstraction mechanism occurs in other mathematical constructs; in all of the following expressions variable \(x\) is abstracted:

\[
\forall_{\mathcal{R}} a_x = b_x
\]

\[ \{x | x \in \mathcal{R} \land x^2 = 3\} \]

\[ \int_0^a e^x \cdot dx \]

The abstraction mechanism is also used in most programming languages. For example, in this C function definition variable \(x\) is abstracted:

```c
double sqr( double x )
{
    return x*x;
}
```

25
In addition to the abstraction mechanism of mathematical functions, there is another. This is, for example, found in circuit diagrams, see figure 2.1. In this abstraction mechanism, the convention is to assume that all variables (in this case node names) are only visible within the current definition (circuit diagram) unless they are explicitly indicated as visible outside the definition. In circuit diagrams this is often determined from the context of the variable, or it is indicated by special ‘external connection’ circuit symbols, see figure 2.2.

2.4.1 λ-calculus

The essence of the abstraction mechanism of mathematical functions is described by λ-calculus. λ-calculus works on expressions produced by the following grammar:

\[
\begin{align*}
\text{expr} & \rightarrow \text{variable} \\
& \mid \lambda \text{variable}. \text{expr} \\
& \mid \text{expr expr} \\
& \mid (\text{expr})
\end{align*}
\]

Thus, valid expressions are: ‘a’, ‘x y’, ‘λx.a x’ and ‘(λ x.a x) b’.

An expression of the form λv.e is called an abstraction, λv is called the abstractor of this abstraction, and e is called its abstrahend. Variable v is called the abstracted variable, and occurrences of v in λv.e are called bound occurrences. A variable that is not bound in an expression is called free. An expression that has no free variables is called closed.

If d and e are expressions, and x is a variable, we write \(d[x \leftarrow e]\) for the expression d after properly substituting e for every free occurrence of x in d. Proper substitution means that none of the free variables of e should become bound by an abstractor of
To prevent this, an abstracted variable in \( d \) may be replaced by a new variable before substitution.

On \( \lambda \)-expressions a number of equivalence rules is defined:

- \( \alpha \)-equivalence: \( \lambda x.e = \lambda y.(e [x \leftarrow y]) \), provided \( y \) is not free in \( e \).
- \( \beta \)-equivalence: \( (\lambda x.e) e' = e [x \leftarrow e'] \).
- \( \eta \)-equivalence: \( \lambda x.fx = f \), provided \( x \) is not free in \( f \).

The \( \lambda \)-expressions and equivalence rules form an abstraction mechanism similar to that of mathematical functions. This can be illustrated by the following derivation:

\[
(\lambda x.a \; x) \; b = a \; b \quad (\beta\text{-equivalence})
\]

but also:

\[
(\lambda x.a \; x) \; b \\
= (\lambda z.a \; z) \; b \quad (\alpha\text{-equivalence}) \\
= a \; b \quad (\beta\text{-equivalence})
\]

Clearly \( x \) is abstracted in the sense of section 2.4.

To see why proper substitution is necessary, note that in the expression \( \lambda b.a \; b \) one cannot replace the occurrences of \( a \) by \( b \), since

\[
((\lambda a.(\lambda b.a \; b)) \; b) \; c \neq (\lambda b.b \; b) \; c
\]

instead, a different derivation is necessary:

\[
((\lambda a.(\lambda b.a \; b)) \; b) \; c = ((\lambda a.(\lambda z.a \; z)) \; b) \; c = (\lambda z.x \; z) \; c = b \; c
\]

### 2.4.2 extensions to \( \lambda \)-calculus

Pure \( \lambda \)-calculus as described in the previous section is the core of an abstraction mechanism that is used in many languages. In fact, in many cases it is possible to add syntactic constructs and equivalence rules that map these constructs on existing ones until the entire language has been defined. In this way it is possible to regard the formal function definitions given in section 2.2 as an enhanced version of \( \lambda \)-calculus expressions. Although a complete construction of formal function theory from \( \lambda \)-calculus is beyond the scope of this thesis, the following extensions should serve to illustrate this:

1. Introducing names for expressions, written \( v = e \). Each occurrence of the name \( v \) can be replaced by expression \( e \).

2. Implicit abstraction. \( v \; a \equiv e \) can be defined to be equivalent to \( v = \lambda a.e \).
3. Where clauses. Definitions local to an expression can be introduced. For example,

\[ e \text{ where } v = e' \]

can be defined to be equivalent to \((\lambda v.e)\ e'\). The definition of \(v\) is only valid within \(e\).

In contrast to the previous extensions, the definition of where expressions described here is not valid for all interpretations of \(\lambda\)-calculus expressions. For example, if

\[ (x\ x) \text{ where } x = b\ y \]

is interpreted as a system description, it is not equivalent with

\[ (\lambda x.(a\ x\ ))\ b\ y = (a\ (b\ y)\ (b\ y)) \]

This is because the structural interpretation of both expressions is different: in the first case, one \(a\) and one \(b\) are used, and in the second case, one \(a\) and two \(b\)s are used. Since it is desirable to permit where clauses, in this case the where clauses are an essential addition to traditional \(\lambda\)-calculus.

These extensions allow definitions such as

\begin{align*}
\text{self s0 s1 a b c d} = & \text{ sel s1 (sel s0 a b) (sel s0 c d)} \\
\text{ where } & \\
\text{ sel s a b} = & \text{ or (and s a) (and (not s) b)} \\
\text{ endwhere;} \end{align*}

2.4.3 \(\sigma\)-calculus

In the same way that \(\lambda\)-calculus describes the essence of the abstraction mechanism for function definitions, a calculus can be found to describe the essence of the circuit diagram abstraction mentioned in the introduction to this section. This calculus is called \(\sigma\)-calculus. Like \(\lambda\)-calculus, \(\sigma\)-calculus consists of an expression language and a set of equivalence rules to transform the expressions. I will not give the formal definitions of the equivalence rules, since they are very similar to that of \(\lambda\)-calculus, only more intricate. See [8] for the details.

An example of a valid \(\sigma\) abstraction is:

\[ \sigma(a, b).\{A(a, c), B(c, b)\} \]

It consists of an abstractor, in this case \(\sigma(a, b)\), and an abstractee called an appset, in this case \(\{A(a, c), B(c, b)\}\). Contrary to \(\lambda\)-calculus abstractions, in \(\sigma\)-abstractions all variables (variables within \((\)) are abstracted by an abstractor. It is therefore
necessary to abstract all variables at once. Also, it will prove convenient to introduce expressions of the form \( \star(a, b, c) \) to indicate that the given variables are synonyms.

An appset is an unordered list of applications of \( \sigma \)-abstractions. Although its name may suggest otherwise, an appset is not a set, because it may contain duplicate applications. For example,

\[
\sigma(a, b).\{A(a, b), A(a, b)\} \neq \sigma(a, b).\{A(a, b)\}
\]

\( \sigma \)-calculus has the following equivalence rules:

- \( \alpha \)-equivalence: proper change of name of a bound variable. This is similar to the \( \alpha \)-equivalence in \( \lambda \)-calculus. For example:

\[
\sigma(x, y).\{A(x, z), B(z, y)\} = \sigma(x, y).\{A(x, u), B(u, y)\}
\]

- \( \rho \)-equivalence (restructuring): introducing and removing inner braces and permutations of applications. For example:

\[
\{U, V, W, X\} = \{X, \{U, W, \{V\}\}\}
\]

- \( \epsilon \)-equivalence. Describes the obvious equivalence rules for synonym expressions. For example:

\[
\star(a, b, c) = \star(c, a, b)
\]

\[
\langle a, b, c \rangle = \{\star(a, b), \star(b, c)\}
\]

\[
X = \{X, \star(v)\}
\]

- \( \beta \)-equivalence: if the abstraction is proper (contains no duplicate variables), then

\[
\sigma(v_0, \ldots, v_n).A(w_0, \ldots, w_n) = A[v_0 \leftarrow w_0] \cdots [v_n \leftarrow w_n]
\]

Where \( [v_i \leftarrow w_i] \) designates proper substitution. (This is the same convention as in \( \lambda \)-calculus.)

An example of a valid derivation in \( \sigma \)-calculus:

\[
\sigma(x, y).\{\star(x, a), \star(g, h)\}.\{\star(u, v), A(g, u), B(v, h)\}\{y, a\} = \sigma(a, y).\{\star(g, h)\}.\{A(g, u), B(u, h)\}\{y, a\} = \sigma(a, y).\{A(y, u), B(u, a)\}
\]
Chapter 3

Electronics background

Although this thesis does not deal directly with the design of electronic circuits, the methods used in their design have an impact on the support that must be provided. Therefore, the purpose of this chapter is to give some insight into the design of analog circuits, not to provide a complete introduction to analog electronics. For such an introduction I refer the reader to [20], [21] or [47].

Readers familiar with electronic engineering may want to read this chapter too, since the design method that is described may differ from the methods they are familiar with. Moreover, this chapter uses the mathematical notation introduced in chapter 2, and therefore provides an opportunity to become familiar with it.

3.1 Terminology

For the sake of clarity, I will first define some necessary terminology. One must bear in mind, however, that there is no clear consensus in these matters: different authors may use different definitions or different terms.

An electronic circuit consists of a number of components. Each component has a number of terminals, and these terminals are connected to each other to form the circuit. As an extension to the terminology, a sub-circuit may also have terminals that represent its connections to the outside. Although this may not be true in reality, the component terminals are sometimes considered to be connected to each other at one point, called a node of the circuit.

In the system description language Glass, the more abstract term connection is used. In this context, a node can be seen as a structural interpretation of a connection.
3.2 Electronic components

An electronic component enforces relations between the voltages and currents on its terminals, establishing the behaviour of the component. An ideal component would be one with an exactly known and simple behaviour, but in practice this can only be approximated because of manufacturing tolerances, physical restrictions etc. The challenge in the design of electronic circuits is to create a circuit that approximates as well as possible the desired behaviour despite all these adverse effects.

As is usual in books on electronics, the voltage difference between two circuit nodes $a$ and $b$ is written as $V_{ab}$, and $V_{ab} = -V_{ba}$. A current that is related to $V_{ab}$ is considered to flow from $a$ to $b$, see figure 3.1. If the actual current flows to node $a$, it is considered to have a negative value.

In this thesis, I will distinguish the following classes of components:

- Passive components.
- Sources.
- Theoretical components.
- Transistors.

Apart from the behaviour established by the components, the following laws apply:

- Kirchoff's current law (abbreviated KCL), which states that the sum of the currents flowing into a node must be zero, see figure 3.2.
Figure 3.3: Kirchoff’s voltage law: \( V_{ad} + V_{ba} + V_{cb} + V_{dc} = 0. \)

- **resistor** \( a \rightarrow R \rightarrow b \) \( V_{ab} = R \cdot I_a \)
- **capacitor** \( a \rightarrow C \rightarrow b \) \( V_{ab} = 1/C \cdot \int_{-\infty}^{t} I_a \, dt \)
- **inductor** \( a \rightarrow L \rightarrow b \) \( V_{ab} = L \cdot \frac{di}{dt} \)
- **diode** \( a \rightarrow b \) \( I_a = I_s \cdot (e^{V_{ab}/V_t} - 1) \)

Table 3.1: passive components.

- *Kirchoff’s voltage law* (abbreviated *KVL*), which states that the sum of the voltages over a loop of nodes must be zero see figure 3.3.

### 3.2.1 passive components

Table 3.1 gives an overview of the most important passive components and their symbols for circuit diagrams. Resistors, capacitors and inductors are called linear components, since their behaviour can be described by linear differential equations.

The symbol \( I_s \) in the behaviour equation of a diode is a parameter that depends on the diode type. The symbol \( V_t \) is defined as:

\[
V_t = \frac{k \cdot T}{q} \tag{3.1}
\]

Where \( k \) is Boltzmann’s constant, \( q \) is the charge of an electron, and \( T \) represents the temperature of the diode in Kelvin.

### 3.2.2 sources

Table 3.2 lists the symbols and behaviour of the independent current and voltage sources. According to their behaviour equations, these sources will always supply the same voltage or current, independent from the external circuit on their terminals.
**Voltage source**

\[ V_{ab} = V \]

**Current source**

\[ I_a = I \]

Table 3.2: independent sources.

![Non-ideal voltage source and current source](image)

Figure 3.4: non-ideal voltage source (a) and current source (b).

(Their load). This leads to a conflict for an unconnected current source and a short-circuited voltage source.

The current or voltage of practical sources will be influenced by their load. This is usually modelled by a resistor in series with the voltage source or in parallel with the current source, see figure 3.4.

By a suitable choice of the source value, a non-ideal current source can also be described as a non-ideal voltage source and vice-versa, see figure 3.5. This is called **Norton-Thevenin equivalence**. Because of this equivalence, the distinction between non-ideal current and voltage sources is blurred. Often the representation is chosen to indicate what the information carrying quantity is.

### 3.2.3 Twoport components

A **port** is a pair of terminals that always carry currents that are equal in magnitude.

![Twoport components](image)

Figure 3.5: the Norton-Thevenin equivalence.
Figure 3.6: a port: $I_b = -I_a$ is always true.

Figure 3.7: some twoport components: (a) a voltage-controlled voltage source (VCVS), (b) a nullor, (c) a gyrator and (d) a transformer.

but opposite in direction, see figure 3.6. A twoport component is a component that has two ports as its terminals. The most important twoport components are:

- **Controlled sources.** Figure 3.7a shows a voltage-controlled voltage source. In this case
  \[ V_{cd} = a \cdot V_{ab} \]

- The nullor. The behaviour of a nullor (Figure 3.7b) is described as:
  \[ I_a = 0 \]
  \[ V_{ab} = 0 \]

$I_a$ and $V_{cd}$ are left undetermined. Remember that a nullor is also a twoport component, and therefore $I_b = -I_a = 0$.

- The gyrator. The behaviour of a gyrator (Figure 3.7c) is described as:
  \[ I_a = g \cdot V_{cd} \]
  \[ I_c = g \cdot V_{ab} \]
The transformer. The behaviour of a transformer (Figure 3.7d) is described as:

\[
V_{cd} = \frac{n}{m} \cdot V_{ab} \\
I_c = \frac{m}{n} \cdot I_a
\]

Most of these components are only of theoretical importance: only the transformer has a (non-ideal) realization.

### 3.2.4 active devices

A bipolar transistor is a component with three terminals, called collector, base and emitter, abbreviated c, b and e respectively. There are two variants of bipolar transistors, called an npn transistor and a pnp transistor. Figure 3.8 shows their symbols in circuit diagrams.

An accurate description of the behaviour of a bipolar transistor is complicated, and beyond the scope of this thesis. For design purposes numerous approximating models have been developed, but I will only show the simplest one: the model that designers use as a mental model in for the normal operation area of a transistor. For that application, the behaviour of an npn transistor can be approximated with the following equations:

\[
I_c = I_s \cdot e^{\frac{V_b}{V_t}}
\]

\[
I_c = B_f \cdot I_b
\]

For a pnp transistor the equations are:

\[
I_c = -I_s \cdot e^{-\frac{V_b}{V_t}}
\]

\[
I_c = B_f \cdot I_b
\]

In both cases, \( V_t \) is defined as in equation 3.1, and \( B_f \) and \( I_s \) are parameters that depend on the device.
Although the behaviour model shown here is too simple for most applications, it shows the essential characteristics of the behaviour of a transistor: the exponential relation between \( V_{be} \) and \( I_c \), and the linear relation between \( I_b \) and \( I_c \). For obvious reasons \( B_f \) is often called the current amplification factor.

Next to the bipolar transistor there is the field effect transistor (FET) that behaves differently; I will not discuss it further.

### 3.3 Deviations from ideal component behaviour

The behaviour of practical components will deviate from the idealized behaviour of the components discussed in section 3.2 because of the following effects:

- Noise.
- Distortion.
- Frequency limitations.
- Manufacturing imperfections.
- Influences from the environment.

To achieve high quality electronic circuits, it is important to minimize the influence of these effects.

#### 3.3.1 noise

Random fluctuations in component behaviour are called noise. Noise is caused by a number of physical effects:

- **Thermal noise.** It is caused by the Brownian motion of electrons. Using thermodynamic theory, it can be shown that any resistor at absolute temperature \( T \) will at least contribute a noise voltage of

\[
\overline{u_r^2} = 4 \cdot k \cdot T \cdot R \cdot \Delta f
\]

Where \( k \) is Boltzmann’s constant, and \( \Delta f \) is the observed frequency range.

- **Shot noise.** It is caused by the fact that current is carried by electrons and hence is quantified.

- **Flicker noise.** The cause of this noise is not entirely understood. Known contributions are:

37
Figure 3.9: the model of a noisy resistor; $V_n$ is a noise source.

- Random mechanical fluctuations in components. For example, carbon resistors consist of small carbon particles that move relative to each other because of thermal effects. This movement causes fluctuations in the contact area between particles, and hence in the resistor value.
- Electron distribution in the conductance bands of semiconductors.

Characteristic for flicker noise is that its intensity is higher for lower frequencies. Therefore it is often called $1/f$ noise.

To be able to analyze the influence of noise on the behaviour of a circuit, it is customary to describe the noise by sources with a random current or voltage that are separate from the idealized components. For example, a resistor can be described as shown in figure 3.9.

### 3.3.2 **distortion**

In the previous sections, I have given linear behaviour equations for many of the components I described. However, these equations are only approximations. In practice, significant deviations from these equations are possible, especially for large voltages and currents. These deviations cause distortion of the signals in a circuit. It is difficult to take distortion into account during the design of a circuit, since it requires the use of more complicated behaviour models for the components, and these models are often inaccurate.

### 3.3.3 **frequency limitations**

The component models I have presented are only valid for relatively low frequencies. Above these frequencies physical effects such as electron transition time and eventually the speed of light cause important deviations.

### 3.3.4 **manufacturing imperfections**

As in all manufacturing processes, manufacturing imperfections of the components have to be taken into account. These manufacturing imperfections manifest themselves mainly in the following ways:
• **Component tolerances.** Components can never be made with absolute accuracy. For example, discrete resistors usually have a tolerance of 5\%, while integrated resistors often have a tolerance of about 50\%.

• **Component value restrictions.** Not all component values are available.

Often these imperfections can be removed at additional manufacturing cost.

### 3.3.5 Influences from the environment

A large number of physical effects may have an undesired\(^1\) influence on the behaviour of a component or a circuit. Important influences are: electro-magnetic coupling, thermal influences, mechanical influences and light.

Electromagnetic coupling and thermal effects especially can have a significant influence on the behaviour of a component. To counter these effects, shielding and compensation may be necessary.

### 3.4 Circuit analysis

To design a circuit, it is necessary to be able to predict the behaviour of the circuit before it is realized. This is called **circuit analysis.** Often the circuit analysis is done using a computer with a **circuit simulation** program. These circuit simulation programs almost always use numerical methods for the circuit analysis.

The behaviour of a circuit is determined by the behaviour of the components of the circuit and by Kirchhoff’s laws. Therefore, if the set of behaviour and interconnection equations can be solved, the behaviour of the circuit is known.

In practice, there are some important complications:

1. **Model accuracy.** Since it is not possible to take all physical effects into account, any analysis will be an approximation of reality. Moreover, since computational accuracy and computational power are limited (especially when the analysis is done by hand) the models of the components must be sufficiently simple.

2. **Limitations of numerical methods.** If numerical methods are used to solve the equations, the accuracy of the solutions and the convergence of the iteration method is important. The model behaviour must obey certain laws, such as continuity of the equations, reciprocity and energy conservation.

For circuit simulation, a number of standard simulation forms have evolved:

---

\(^1\)Sometimes the influence is desired, but then the component is called a *sensor.*
• **DC analysis.** This analysis method is used to determine the steady-state currents and voltages of a circuit. That is, the currents and voltages when no signals are applied to the circuit, and all power-on effects have damped out. The resulting currents and voltages are called the operating point or bias currents and voltages.

• **Small signal AC analysis.** In this analysis, it is assumed that for small deviations from the operating point, the signals behave as if the system is **linear.** This assumption makes calculations considerably easier, since in this case only a set of linear differential equations needs to be solved, which is relatively easy. This makes it possible to do the AC analysis for a large number of frequencies. Some analysis programs apply a Fourier or Laplace transform, see section 3.4.1.

• **Transient analysis.** This analysis method solves the non-linear differential algebraic equations by means of numerical integration in the time domain. It is more computationally intensive than DC analysis or small signal AC analysis, but the accuracy for larger signals is much better than that of small signal AC analysis.

Circuit analysis programs require a standard representation of the behaviour and interconnection equations, and a systematic method must be found for their solution. In section 3.4.2, one representation will be described: the nodal admittance (NA) matrix equation. It will be used in an example semantic function in chapter 7. This semantic function uses the Laplace transform, therefore this technique is discussed superficially in section 3.4.1. In [10], a more detailed discussion of the Laplace transform can be found.

I will not discuss solution methods or representation methods of the equations other than the NA matrix equation in this thesis, see [47] for an overview of the methods and representations.

### 3.4.1 the Laplace transform

The behaviour of a linear or linearized continuous-time system can be modelled by a set of linear differential equations with a number of known quantities (excitations) and a number of unknown quantities (responses). Since the differential equations are linear, it is possible to analyze the response to each excitation separately. (This is called the superposition theorem.)

In general, the relation between one input and one output can be written as:

\[
b_0 \cdot (y(t)) + b_1 \cdot \frac{d(y(t))}{dt} + \ldots + b_n \cdot \frac{d^n(y(t))}{dt^n} = a_0 \cdot (x(t)) + a_1 \cdot \frac{d(x(t))}{dt} + \ldots + a_n \cdot \frac{d^n(x(t))}{dt^n}
\]

(3.2)
where

\[ a_0, \ldots, a_n, b_0, \ldots, b_n \in \mathcal{R} \]

Such a differential equation will have \( n \) solutions of the form \( e^{\lambda \cdot t} \), where \( \lambda \) is either real or is an element of a complex conjugate pair. In principle, it is possible to use this knowledge to solve the equation directly. It will prove convenient, however, to transform the differential equation into a polynomial equation from which the \( \lambda \)s are more easily found. This is done by the Laplace transform.

The Laplace transform \( \mathcal{L} \) is a higher-order function:

\[
\mathcal{L} : (\mathcal{R} \to \mathcal{R}) \to (\mathcal{C} \to \mathcal{C}), \quad \mathcal{L} f s = \int_0^\infty e^{-s \cdot t} \cdot (f t) \cdot dt
\]

In this expression, \( s \) is called the Laplace variable. Usually \( s \) or \( p \) is used for it. It is easily shown that the Laplace transform has the following properties:

\[
\mathcal{L} (a \cdot f + b \cdot g) = a \cdot (\mathcal{L} f) + b \cdot (\mathcal{L} g)
\]

\[
\mathcal{L} \left( \frac{d^k(f t)}{dt^k} \right) = s^k \cdot (\mathcal{L} f) - s^{k-1} \cdot (f 0^+) - s^{k-2} \cdot \frac{d(f 0^+)}{dt} \ldots
\]

\[
\mathcal{L} \int_0^t (f t) \cdot dt = \frac{1}{s} (\mathcal{L} f s)
\]

\[
\mathcal{L} f s = \frac{1}{s + a}, \text{ for } f t = e^{-a \cdot t}
\]

\[
\mathcal{L} f s = \frac{\omega}{s^2 + \omega^2}, \text{ for } f t = \sin \omega \cdot t
\]

\[
\mathcal{L} f s = \frac{s}{s^2 + \omega^2}, \text{ for } f t = \cos \omega \cdot t
\]

using these properties and assuming

\[
x 0^+ = \frac{dx 0^+}{dt} = \ldots = 0
\]

it is possible to rewrite (3.2) as

\[
(b_0 + s \cdot b_1 + \ldots + s^n \cdot b_n) \cdot \mathcal{L} y = (a_0 + s \cdot a_1 + \ldots + s^n \cdot a_n) \cdot \mathcal{L} x
\]

or

\[
\mathcal{L} y = \frac{a_0 + s \cdot a_1 + \ldots + s^n \cdot a_n}{b_0 + s \cdot b_1 + \ldots + s^n \cdot b_n} \cdot \mathcal{L} x
\]

41
This expresses the relation between the stimulus $x$ and the response $y$ in the Laplace domain. This relation can be written as:

$$\mathcal{L} y = (H(s)) \cdot \mathcal{L} x$$

Where $H$ is called the transfer function $H$ of $x$ to $y$. Since $H$ describes the behaviour of the circuit for any input function $x$ that has a Laplace transform, it is useful in the design of a circuit.

The transfer function $H$ can be written as

$$H(s) = \frac{a_0 + a_1 \cdot s + \ldots + a_n \cdot s^n}{b_0 + b_1 \cdot s + \ldots + b_m \cdot s^m} = \frac{p}{q} \frac{s}{s}$$

where $a_n \neq 0$ and $b_m \neq 0$. Normally $n \leq m$.

The equation $p \cdot s = 0$ will have $n$ solutions called zeros. Similarly, $q \cdot s = 0$ will have $m$ solutions called poles. Since all $a_i$ and $b_i$ are real, all poles and zeros are either real or part of a complex conjugate pair.

For the analysis of the behaviour of a system, it is useful to represent the poles and zeros graphically in the complex $s$ plane. For example, figure 3.10 shows a pole and zero plot of the function

$$H(s) = \frac{s - 1}{(s^2 - 2 \cdot s + 2) \cdot (s + 3)}$$

Poles are represented by 'X', zeros are represented by 'O'. With some training, inspection of such a pole and zero plot gives insight into the behaviour of a system.

An important property of the Laplace transform is that it is possible to treat a capacitor of value $C$ as a resistor with value $1/(s \cdot C)$, and an inductor as a resistor
\[ C \quad = \quad \frac{1}{sL} \]

\[ L \quad = \quad sL \]

Figure 3.11: a capacitor and an inductor as Laplace domain resistors.

with value \( s \cdot L \), see figure 3.11. Using these equivalences, it is possible to express the behaviour of circuits with linear components as a set of linear equations containing polynomials in \( s \). Linear components are:

- Resistors.
- Coils.
- Capacitors.
- Independent sources.
- Controlled sources, provided that their transfer is constant.
- Gyrators and transformers.
- Nullors.

### 3.4.2 the nodal admittance matrix equation

In practice, the nodal admittance (NA) matrix equation is one of the most convenient representations of behaviour equations. It is derived from the indefinite admittance (IDA) matrix equation. A detailed description of the NA representation is beyond the scope of this thesis, details and references can be found in [47].

To explain the construction of the IDA equation, let us assume for the moment that the circuit to be analyzed only contains the following components:

- Independent current sources.
- Resistors.
- Voltage-controlled current sources (vccs).

Such a circuit can be seen as a network of resistors and voltage-controlled current sources that is supplied by a number of current sources, see figure 3.12. Now, the IDA matrix equation is defined as follows:
Figure 3.12: the general structure of circuits for IDA equations.

\[
\begin{bmatrix}
I_1 \\
\vdots \\
I_n
\end{bmatrix} =
\begin{bmatrix}
m_{1,1} & \cdots & m_{n,1} \\
\vdots & \ddots & \vdots \\
m_{n,1} & \cdots & m_{n,n}
\end{bmatrix}
\begin{bmatrix}
V_1 \\
\vdots \\
V_n
\end{bmatrix}
\]

or in short:

\[
\vec{i} = M \cdot \vec{v}
\]

where \( M \) is the IDA matrix, \( \vec{i} \) is the vector of currents into all terminals, and \( \vec{v} \) is the vector of resulting voltages relative to an arbitrary reference node. Each row and column of \( M \) corresponds to one node in the circuit. An important advantage of the IDA is that it is easily constructed from the contributions of the individual components. These contributions are called stamps.

For a resistor the stamp is:

\[
\begin{array}{c}
a \\
\hline
R \\
\hline
b
\end{array}
\]

\[
\begin{array}{c|cc}
& \text{column a} & \text{column b} \\
\hline
\text{row a} & 1/R & -1/R \\
\text{row b} & -1/R & 1/R
\end{array}
\]

For a voltage-controlled current source the stamp is:
A current source does not contribute to the IDA matrix $M$, but to the current vector $\mathbf{i}$:

\[ \begin{array}{c}
\begin{array}{c}
\text{row a} \mid I \\
\text{row b} \mid -I \\
\end{array}
\end{array} \]

The IDA matrix and current vector of a circuit are found by summing the matrix and vector entries for each of the nodes. For example, the circuit shown in figure 3.13 has the IDA matrix equation:

\[
\begin{bmatrix}
I \\
-I \\
0 \\
0
\end{bmatrix} = \begin{bmatrix}
\frac{1}{r_1} & \frac{-1}{r_1} & 0 & 0 \\
\frac{1}{r_3} & \frac{1}{r_1 + r_3} & 0 & -\frac{1}{r_3} \\
g & g & \frac{1}{r_2} & -\frac{1}{r_2} \\
g & -g - \frac{1}{r_3} & -\frac{1}{r_2} & \frac{1}{r_2} + \frac{1}{r_3}
\end{bmatrix} \cdot \begin{bmatrix}
v_a \\
v_b \\
v_c \\
v_d
\end{bmatrix}
\] (3.3)

In the previous description it was assumed that the circuit only contains resistors, current sources and voltage-controlled current sources. This may seem overly restrictive, but, fortunately, it is possible to construct many other components from them. Figures 3.14, 3.15 and 3.16 show the construction of some important ones. Also, if polynomials in the Laplace variable, $s$, are permitted in the matrix equation, the stamps of inductors and capacitors are easily added, see section 3.4.1.

The set of equations as represented by the IDA is dependent, since according to Kirchoff’s laws the sum of the current vector elements and the sum of the voltage
Figure 3.14: the construction of a gyrator.

Figure 3.15: the construction of a current-controlled current source.

Figure 3.16: the construction of a voltage source.
Figure 3.17: example circuit for Spice and Anp:

Sample SPICE run
R1 1 2 1K
R2 2 0 4K7
C1 2 0 1n
VSRC 1 0 DC 5
.OP
.OPTION CTIME=100
. END

Figure 3.18: example of Spice input.

vector elements are both zero. It is therefore necessary to delete one row and one column from the IDA matrix to make the set of equations solvable. The resulting matrix equation is called a nodal admittance or NA matrix equation. For example, if the last row and column of equation 3.3 are deleted, the following NA matrix equation results:

\[
\begin{bmatrix}
\frac{1}{r_1} & -\frac{1}{r_1} & 0 \\
-\frac{1}{r_1} & \frac{1}{r_1} + \frac{1}{r_3} & 0 \\
-g & g & \frac{1}{r_2}
\end{bmatrix}
\begin{bmatrix}
v_a \\
v_b \\
v_c
\end{bmatrix}
= \begin{bmatrix}
I \\
-I \\
0
\end{bmatrix}
\]

In section 7.3, a semantic function is described that generates statements for the computer algebra program Maple [12]. Using the generated statements, Maple constructs a symbolic NA matrix equation and then solves it. The semantic function closely follows the construction method for the NA matrix equation as described in this section.

The symbolic solution of matrix equations, as done in that semantic function, is only possible for small, linear circuits. For large or non-linear circuits numerical solution methods must be used. These methods require considerable refinements to the construction of the matrix equations. For that reason modern circuit analysis programs often use an extension of the Nodal analysis method, called Modified Nodal Analysis.

3.4.3 circuit analysis programs
The nodal analysis as described in section 3.4.2 is used, with some modifications, in almost all circuit analysis programs. One of the most popular circuit analysis programs is *Spice* [31, 58]. Figure 3.18 shows an example of *Spice* input, a DC analysis of the circuit shown in figure 3.17.

Another program, *Anp*, does a Laplace domain analysis of a circuit, and calculates the poles and zeros of a circuit. Figure 3.19 shows an example of *Anp* input, an analysis of the circuit shown in figure 3.17. Semantic functions that interface to these programs are described in chapter 7.

### 3.5 Design methods for analog circuits

#### 3.5.1 building blocks

Analog circuits usually consist of a number of building blocks that perform well-defined functions. Familiar building blocks are:

1. *Amplifier*. Converts an output signal of a given source to an input signal of a certain load.

2. *Filter*. Selects a certain spectrum from a signal.


All of these building blocks generate or process signals. Broadly speaking, there are the following sources of imperfections in these building blocks that will deteriorate a signal:

1. Noise.

2. Distortion.

3. Frequency limitations.
Figure 3.20: an example of compensation.

4. Inaccuracy.

A designer will specify maximum values (called constraints) for each of these errors. Most of the effort in the design of a building block is invested in reducing these errors enough to meet the constraints, and it is often necessary to find a trade-off between them.

3.5.2 error-reduction techniques

To reduce errors, the obvious first step is to select appropriate components. In addition, two techniques are available to reduce errors: compensation and feedback.

For compensation an attempt is made to counter the error by an influence in the opposite direction. For example: if a system has a transfer function

\[ f(x) = x^\alpha \]

instead of

\[ f(x) = x \]

this may be compensated by cascading it with a system with the transfer function

\[ g(x) = x^{1/\alpha} \]

This is shown in figure 3.20. The cascade of the functions will have the transfer function

\[ g(f(x)) = (x^\alpha)^{1/\alpha} = x \]

For a successful compensation, it is necessary that the error to be compensated is accurately known, or that the error and compensation functions are related. In practice, this is often not possible, and this limits the usefulness of this technique.

For feedback, a system as shown in figure 3.21 is used. For the sake of simplicity, I will assume that the behaviour of the system can be described with linear differential equations. It is then possible to derive the behaviour of the system:

\[ b = i - g \cdot o \]
Figure 3.21: the general feedback circuit.

\[ o = H \cdot b \]

thus:

\[ o = H \cdot (i - g \cdot o) \]

\[ o = \frac{H}{1 + H \cdot g} \cdot i \]

and

\[ \lim_{H \to \infty} \frac{H}{1 + H \cdot g} = \frac{1}{g} \]

Thus, provided that \( H \) has sufficient amplification, and that \( g \) is accurately known, the transfer function of the entire system is also accurately known. In practice, this is very often possible.

A serious complication in the design of feedback systems is that they may become unstable. The prevention of instability often requires careful attention.

3.5.3 the importance of verification

Especially in the manual design of building blocks, it is not possible to use detailed behaviour models of the components. This means that the designed behaviour of a building block is only an approximation of its real behaviour. Better approximations of the real circuit behaviour can be found with numerical circuit simulation, see section 3.4.3. Although the models that are used by these programs can be more accurate than those used for manual design, they are still approximations of the behaviour of real components. It remains necessary to actually construct and measure a circuit to verify the predicted behaviour.

3.5.4 design methods

Designing a circuit means searching among the set of all possible circuits, called the search set, for a circuit that fulfills the required function. To do this, the quality of
the circuits must be evaluated. To determine the quality of a circuit many aspects may be considered, such as the accuracy of signal transfer, distortion, noise, power dissipation, cost, etc. A designer will impose constraints on these quality aspects, and a circuit is only acceptable if it meets all these constraints. Let us call such a circuit a feasible solution.

Thus, designing a circuit means searching the search set for a feasible solution. This search is complicated by the following aspects:

1. It is often difficult to evaluate the quality of circuits, and therefore to indicate what is a 'good' solution.

2. It is often not clear what the search set is. That is, it is often not clear which circuit configurations and component values must be considered.

3. The search set is often very large.

In some cases, a designer may want to find the best solution instead of a feasible solution. This is more difficult, since one must not only ensure that a solution is feasible, but also that there is no solution that is better. This requires much more detailed knowledge of the set of circuits that is searched. Also, it requires the formulation of one quality evaluation function; it is not permitted to have separate evaluation functions for different quality aspects, since conflicts may arise between these functions about the best solution.

3.5.5 search methods

Given the search set and the circuit constraints, the problem remains of finding a feasible solution. Without further knowledge about the search set, the only reliable method is to do an exhaustive search: consider all elements in the search set in arbitrary order until a feasible solution is found. For most search sets this method is far too time consuming. Therefore, more knowledge about the search space is necessary, so that more refined search methods can be used.

Some of the available search methods are:

- **Ordered search**: arrange the order of an exhaustive search so that the circuits that are more likely to be a solution are considered first. Since ordered search is only a refinement of exhaustive search, it is still ensured that, if necessary, all solutions are considered.

- **Algebraic and algorithmic solution methods**. In some cases it is possible to find an algebraic expression for the optimum that can be implemented efficiently in an algorithm.

- **Approximation**. A complicated search problem may be approximated by a simpler one. Inherently, this will introduce the following errors:
- **False solutions.** The solution of the approximating problem is not a solution of the original problem.

- **Missed solutions.** No solution can be found for the approximating problem, although the original one does have a solution.

False solutions can be detected by verification, although not without problems, see section 3.5.3. The only way to recover missed solutions is by using backup methods, if they are available.

A rigorous proof of the soundness of an approximation is often difficult; usually, a verification of the found solution is done instead. An approximation without a rigorous soundness proof is called **heuristic.**

- **Differential search methods.** For some evaluation functions, it is possible to find the derivatives to the parameters. The 'slopes' that can be calculated with the derivatives may be used to estimate the location of an optimum, although this is not necessarily the global optimum. By finding the optimum, one may expect to meet the search constraints.

- **Random search methods.** If differential methods cannot be used, it is necessary to use methods that require less knowledge about the function to be optimized. In this case, random search methods such as the **Monte Carlo** method or **simulated annealing** may be useful. They are based on the evaluation of random points in the solution space. Since in many problems the percentage of feasible solutions in the solution space is low, these methods may require the evaluation of many points in the solution space. Therefore, they are computationally expensive. Moreover, their performance is unpredictable.

- **Projection.** It is often possible to describe the elements in the search set by a list of parameters (for example, component values). Now, the search set can be described as a multi-dimensional space, where each parameter is a dimension in this space. For some problems, it is now possible to make a projection of the problem, and solve that smaller problem. That is, a new problem is formulated wherein values are sought for a subset of the parameters, so that the resulting circuit meets a subset of the constraints. The remaining parameters are then found by solving another projected problem, perhaps using the results of the first search.

- **Backtracking search.** Backtracking search is a variant of projection. An exhaustive or ordered search is done on a subset of the parameters. For each set of values of these parameters, a projected problem is constructed for the remaining parameters, and a solution to that problem is sought. If a solution is found for this subproblem, the entire solution is evaluated to see if the exhaustive search must continue.
3.5.6 application of the search methods to circuit design

Let us now consider how these search methods can be applied to circuit design, both manual and automated.

For electronic circuits, it is useful to describe the search set as a search space, and to divide the dimensions of the search space in two groups:

1. **Configuration dimensions.** They correspond to aspects of the configuration or architecture of a circuit.

2. **Value dimensions.** They correspond to component values.

Actually, the number of components, and therefore the number of value dimensions, will depend on the architecture of the circuit. It is therefore necessary to search for a feasible circuit configuration first, and then search for feasible component values within the search space of the value dimensions for that architecture. This is, in essence, a backtracking search. Since the search space is usually very large, it is necessary to avoid time-consuming calculation and lengthy searches. This is done by using simple behaviour models and by the extensive use of projection.

For manual design, the search space consists of the circuits known to the designer and all circuits in the reference books he has available. Because only limited computations are possible, the designer will restrict himself to simple behaviour models for the circuit or circuit parts, so that projection and algebraic solution methods can be used. This requires drastic approximations and therefore thorough verification is necessary. Verification is usually done with circuit analysis programs such as SPICE.

A drawback of manual design is that only a limited space can be searched, and that drastic approximations are necessary. Moreover, considerable expertise is necessary to introduce the correct approximations.

For automated design, more detailed behaviour models can be used, so that better approximations can be used. Also, it is possible to use differential and random search methods. It is, however, necessary to have a sufficiently formalized design process. This has proved to be a difficult task.

In many circuit design programs, this problem is circumvented by introducing drastic simplifications of the design problem, and considering only a limited set of circuits. This is done in design programs such as IDAC [14]; an overview of these programs is given in [53]. A disadvantage of this approach is that for more demanding applications, no suitable circuit can be designed. Therefore, Stoffels has tried to use more detailed design knowledge in his amplifier design program, and hence to search a large design space. The development of this program, called **Ampdes**, was inspired by the publication of formalized design theory for the manual design of amplifiers by Nordholt [33]. During the project, it became clear that a considerable amount of theory was still lacking, and therefore it had to be developed during the project. At the moment of writing, little has been published on the results of this project, although some earlier results can be found in [54].
Another problem in the automation of circuit design is the necessity for a suitable representation of the circuits being designed. This situation has led to the work described in this thesis. The results of this work, Glass and its semantic functions, have been used in Ampdes. As an illustration of the applications of Glass, Ampdes will be described in some detail in section 9.1.
Chapter 4

Software engineering

As in other branches of engineering, in software engineering an attempt is made to achieve maximal results with minimal effort and cost. Unfortunately, there is at the moment no rigorous method to achieve or even to approximate this objective. Therefore, the discussion in this chapter will have a more qualitative character than that of the previous chapter.

The term 'software' can be interpreted in a very broad sense. In addition to computer programs, all other descriptions that involve a formal language are the subject of software engineering. For example, Glass system descriptions are software.

Software engineering covers the following steps in the construction of software:

1. Specification. Definition of the requirements of a system.
2. Design. Definition of a system that meets the specification.
3. Implementation.

There is at the moment no rigorous method that covers all these steps. There are, however, some methods that cover these steps partially:

1. Structured Analysis and Structured Design. These are more or less formal methods for the specification and design of data transaction systems such as inventory control systems, ordering systems etc. See for example [15] and [36].

2. Documented algorithms. There is a large amount of information on the implementation of specific computer algorithms. See for example [26], [59] and many others. Usually these texts describe algorithms for particular problems, and not a method to arrive at these algorithms.
3. Correctness proofs. Given a mathematical description of the specification and the implementation, it may be possible to prove that a correct implementation satisfies the specification. At the moment, rigorous proofs are possible for a limited class of problems. Less rigorous proofs are possible for a larger range of systems, see for example [16].

4. Transformational reasoning. Instead of trying to connect a specification and its implementation through a proof, it is possible to search for a sequence of proven transformations to derive an implementation from a specification. Thus, the correctness proof does not follow the design, but is the design. This is called transformational reasoning, see for example [11].

The quality of software is determined by the following criteria, roughly in order of importance:

- Reliability (no errors).
- Robustness (withstands 'abuse').
- Openness (easily modified).
- User friendliness.
- Portability.

Obviously these criteria are related: an easily modified program is more portable, and a robust program is also more reliable and user friendly. All these criteria will be discussed in more detail.

### 4.1 Reliability

Reliability simply means that software should contain no errors (deviations from the specified behaviour). Unfortunately this is easier said than done, as is indicated by the programmer's observation that "every non-trivial program contains at least one bug". Thus, in large software projects one has to assume that each fragment of the software has a certain probability of containing an error. There are some measures that can reduce this probability, however, and therefore can lead to the production of more reliable software.

- **Openness.** Software that is easy to understand is less likely to contain errors. Thus, software should have 'obviously no errors' instead of 'no obvious errors'. Openness is discussed in section 4.2.

- **Correctness proofs and checks.** A rigorous proof of correctness is the best guarantee of reliable software. Unfortunately, correctness proofs are often very complicated (often more complicated than the original software), and are as susceptible to errors as other software.
• **Type checking.** If suitable type expressions are used, it is possible to verify automatically the correctness of the type specifications and of the applications of the typed expressions. Type checking of software can be seen as a limited form of correctness proof: it is verified that all applications of a definition are correct with respect to the type specifications. Type checks have the advantage over general correctness proofs in that they can be done automatically.

• **Testing.** Obviously, testing the software will improve its reliability. An important problem, however, is to ensure that all software is actually tested; it is often difficult to predict which parts of the software are used for a given test input.

• **Code re-use.** Any use of software is a limited test of it. Therefore, software that has been used frequently will be more reliable than freshly created software. This has some interesting consequences:
  
  - Software in libraries is more reliable than freshly created software that is written for the same purpose.
  - More general and more portable software is more likely to be used frequently and is therefore more reliable.
  - Multiple copies of software are less reliable than one generally used copy. Obviously this does not apply when the copies are made automatically, for example by the expansion of a macro.
  - Code generators such as yacc [24], lex [28] and tm (chapter 6) generate software from templates. Therefore, these software templates will be tested in all programs that use the generated software. Therefore, the templates are well tested, and the generated software will be more reliable than specific software for that is written for the same purpose.
  - Exceptions make software unreliable. Software fragments that are used only in special circumstances are more likely to contain errors. Therefore, excessive use of ‘if’s leads to unreliable software.
  - No unnecessary constants. Every value that can be derived from other values should be derived.

### 4.2 Openness

Easily understood software is more reliable and portable. Moreover, such software is more easily modified. To achieve openness, the following measures will help:

• **Suitable choice of implementation language.** The language should make it easy to describe the essentials of the software, and should not burden the user with irrelevant details. The one ideal language for this does not exist, and is not likely to exist, since the choice depends on the specific software to be written, the availability of the language, and personal taste.
• Good documentation in and in addition to the software. This is obvious, although there are many software authors who provide neither. There are even software authors who state that software can be self-documenting. This is obviously false, since software always describes only what is done, while the documentation states what should be done.

• Limited context. Every software fragment should be understandable on its own, and rely as little as possible on context.

• Consistent. Software is not the place to introduce surprises. The more predictable the use of names, the behaviour of functions, etc. is, the more easily can the software be understood.

• No unnecessary performance ‘optimizations’. This is a common error in the implementation of computer programs. Since in most programs more than 90% of the execution time is spent in less than 10% of the code, changes in the software to improve performance are wasted in more than 90% of the code. Moreover, it is, without proper tools, often difficult to indicate in which part of the code most time is spent.

• Express what is intended. For example, although \(2 \cdot x\) and \(x + x\) are algebraically equal, one of the two may express more clearly what is intended. For example, the first form is more appropriate if \(x\) has to be multiplied by a constant that ‘happens’ to be 2.

4.3 Robustness

Although reliability and robustness are closely related, they are not equivalent. Reliable software behaves correctly for all valid input, while robust software tries to handle invalid input, or even invalid changes to the software. In order of preference robustness can be achieved as follows:

1. Make software more general. This implies, for example, that a function should behave correctly for the largest possible range of input parameters. Therefore, there should be no arbitrary limits or constants in a program (for example: only 1000 variables per program, the data base is located at `/lib/database`).

2. Detect and handle errors and ‘impossibilities’. The error messages should be informative; the error message ‘something is wrong’ is less useful than the error message ‘No such file: ‘bar.c’.’

4.4 User friendliness

The criterion of user friendliness is more vague than the other criteria. It depends on such things as personal taste, experience with the software and with software in
general etc. For the purpose of this thesis I will assume that user friendly software is:

1. Powerful.
2. Robust and reliable.
3. Consistent.
4. Understandable.

4.5 Portability

Portability means that software must be useful in many environments. This implies that insofar as possible no assumptions must be made about the environment. This in turn implies that to write portable software:

- Insofar as possible, relevant standards must be used.
- Software authors must know what is specific to a certain environment and what is not.
- Software must be tested in a wide range of environments.

4.6 Impact on the design and implementation of Glass.

Software engineering influences the design of a system description language with multiple semantics in two ways:

1. The system description language should encourage the application of software engineering principles.
2. The implementation of semantic functions should follow software engineering principles.

By examining the recommendations in this chapter, it is possible to compile a list for both areas. For the design of Glass the recommendations are:

1. Glass should provide a good type system.
2. Glass should be powerful and consistent.
3. Glass should allow the user to describe the problem from his point of view.
4. Glass should support definition libraries.
5. Glass should provide an easy-to-use comment notation.

6. Glass should allow correctness proofs and checks.

7. There should be a standard for the Glass language.

For the implementation of semantic functions on Glass the recommendations are:

1. Insofar as possible, code should be shared between semantic functions. Thus, standard software should be available for commonly used operations on Glass, for example parsing and macro expansion.

2. The support software should be robust and reliable.

3. The implementation of semantic functions should be made as easy as possible.

4. It should be possible to implement semantic functions in arbitrary programming languages.
Chapter 5

The system description language Glass

As explained in chapter 2, it is possible to use \( \lambda \)-notation and \( \sigma \)-notation expressions to describe arbitrary systems. For a practical realization of this idea, however, some obstacles have to be cleared away:

1. Direct use of \( \lambda \)-calculus and \( \sigma \)-calculus is cumbersome. Thus, a more convenient notation is necessary.

2. Since for the special case of directional circuits, \( \lambda \)-calculus is much too convenient to be replaced by the more general \( \sigma \)-calculus, it should be possible to mix the two styles of notation in the description language.

3. The character set of most computer systems is limited, and usually does not contain symbols such as \( \lambda \) and \( \varepsilon \). Thus, a suitable representation of such symbols must be found.

4. A type system must be designed that allows detailed type specification, but still can be checked easily by the software.

5. Tedium tasks such as repetition of system parts should be handled by the language.

6. Since the language was designed to experiment with semantic functions, an easily accessible representation of Glass for use by semantic functions must be found.

7. The language should support libraries of system definitions.

8. The language should encourage the application of software engineering principles.
The discussion of software engineering principles in chapter 4 resulted in the following recommendations for the Glass language:

- Glass should provide a good type system.
- Glass should be powerful and consistent.
- Glass should allow the user to describe the problem from his point of view.
- Glass should support definition libraries.
- Glass should provide an easy-to-use comment notation.
- Glass should allow correctness proofs and checks.
- There should be a standard for the Glass language.

In this chapter, many of these design issues will be discussed together with the choices made in the final version in Glass. The representation of Glass for semantic functions is important enough to be discussed in a separate chapter, chapter 6. The support for correctness proofs has largely been ignored, although a consistent language would be an advantage for this purpose. A good comment notation is easily provided, and will not be discussed further.

In this chapter, I will not describe Glass in detail, the Glass language definition is given in [50], and an overview is given in appendix B and in [51].

5.1 Additions to the $\lambda$- and $\sigma$-calculi

The direct use of $\lambda$- and $\sigma$-calculus expressions is cumbersome. Thus, a more convenient language is necessary. In principle, the same enhancements to $\lambda$-calculus could be used as are used in formal function definitions. Thus, a definition such as

$$\text{sel} = \lambda a \ b \ s.\text{or} \ (\text{and} \ a \ s) \ (\text{and} \ b \ (\text{not} \ s))$$

could be converted to

$$\text{sel} \ a \ b \ s = (\text{or} \ (\text{and} \ a \ s) \ (\text{and} \ b \ (\text{not} \ s)))$$

This solution has not been chosen for two reasons: currying—as suggested by this notation—is not useful for system descriptions, and the currying notation will be used by a macro language for circuit repetition. The currying of system definitions would imply the introduction of higher-order systems, or in other words the introduction of systems that have other systems as input or output. Even if this would be meaningful, exclusion does not lead to major problems, and makes the design of the language much easier. For these reasons Glass represents connections as tuples and does not use currying for system definitions.
Figure 5.1: a circuit diagram of the Glass system amp10.

It will be necessary to distinguish a system definition from a macro definition. This is done by introducing a system definition with the word def. A macro definition (see section 5.5) will be introduced with the word mac. Thus the definition of \textit{sel} becomes

\begin{verbatim}
def sel (a, b, s) = or (and (a, s), and (b, not s));
\end{verbatim}

Definitions are separated from each other by semicolons (";"). This is necessary to avoid ambiguities in the Glass text.

In Glass, tuples may contain expressions, but may also contain other tuples. This nesting allows the grouping of related connections, for example connections of a data bus, or the two connections of a balanced signal.

It is permitted to let abstraction variables of the definitions abstract entire tuples instead of single connections. Thus, instead of an abstractor

\begin{verbatim}
\langle\langle a,b\rangle,\langle c,d\rangle\rangle
\end{verbatim}

an abstractor such as

\begin{verbatim}
\langle e,\langle c,d\rangle\rangle
\end{verbatim}

or even

\begin{verbatim}
g
\end{verbatim}

is permitted.

An other necessary addition is the introduction of a 'where' clause, so that feedback can be described. For example:

\begin{verbatim}
def amp10 i = o where o = opamp (i, div10 o) endwhere;
\end{verbatim}
Figure 5.2: a circuit diagram of the system \textit{div2}.

Figure 5.3: a circuit diagram of the system \textit{div4}.

Figure 5.1 shows a circuit diagram of \textit{amp10}. Note that in this definition the introduction of a \textit{where} clause is absolutely necessary: a name is necessary for the output of \textit{opamp}, so that the feedback can be described.

An expression such as
\[ o = \text{opamp} \ (i, \ \text{div10} \ o) \]
is called a \textit{connection-pattern definition}. In general, connection patterns may have arbitrary connection patterns as their left-hand side, similar to the abstractors of definitions. For example, with the atom declaration
\[
\textbf{atom} \ Dff \in E \times E \Rightarrow E \times E;
\]
the expression
\[
(q, \ nq) = Dff \ (d, \ clk)
\]
is a valid connection-pattern definition, where the variables \(q\) and \(nq\) match the two output connections of the expression
\[
Dff \ (d, \ clk)
\]
A \textit{where} clause may contain any number of connection-pattern definitions. They are separated from each other by semicolons (';'). A 'where' clause may also contain system and macro definitions. These definitions are valid within the current definition only. For example, the Glass description:
/* The name of the basic connection type. */
definition basetype E;

/* A D-flipflop. \( (q,nq) = \text{Dff} \ (\text{data, clock}) \) */
definition atom Dff \( \in \ E \times E \Rightarrow E \times E \);

definition div4 \( \in \ E \Rightarrow E \);
    div4 i = div2 (div2 i)
where
    definition div2 \( \in \ E \Rightarrow E \);
        div2 clk = q
    where
        \( (q,nq) = \text{Dff} \ (nq,clk) \);
    endwhile;
endwhere;

describes a system div4 that is shown in figure 5.3. The Glass description uses a local system definition, div2, that is shown in figure 5.2. The meaning of the atom and basetype lines and the type expressions div4 \( \in \ E \Rightarrow E \) and div2 \( \in \ E \Rightarrow E \) will be explained in section 5.2.

For \( \sigma \)-calculus definitions (see section 2.4.3) a similar definition style can be introduced:

\[
RC = \sigma(a,b).\{R \ (a, x), \ C \ (x, b)\}
\]

becomes

\[
definition RC \ (a, b) = \{ \ R \ (a, x), \ C \ (x, b) \ \};
\]

Figure 5.4 shows a circuit diagram of RC. Definitions such as that of RC are called adirectional definitions, while definitions such as that of \( \text{sel} \) are called directional definitions.

Adirectional definitions contain an appset, an unordered list of applications. Thus, an adirectional definition can be recognized by the curly brackets (\{\}) that surround its appset.

Like in \( \sigma \)-calculus, appsets may contain synonym expressions. For example,

\[
definition RC \ (a, b) = \{ \ R \ (a, x), \ C \ (y, b), \ *(x,y) \ \};
\]

65
is equivalent to the previous definition of \( RC \).

## 5.2 Types

As has been explained in section 4.1, suitably chosen type expressions can be verified automatically, and serve as a limited correctness check. Therefore, Glass requires a type specification for each system definition. These type expressions describe the interface of a system, and the direction of its connections. To allow Glass to be used to describe arbitrary systems, the user must introduce (declare) a name for every basic connection type. These connection types may represent electrical, magnetic, etc. connections between components. This declaration is done with the keyword `basetype`. For example,

```
basetype \( E \);
```

declares that \( E \) is a basic connection type. I will use \( E \) in this thesis as the name of the basic electrical connection type. From these basic connection types, more complex connection expressions can be composed using the Cartesian product operator \( \times \) and bracketing. For example:

\[
(E \times E \times E) \times (E \times E)
\]

describes the type of a connection such as \( (\langle a, b, c \rangle, \langle d, e \rangle) \). Note that

\[
(E \times E) \times (E \times E) \neq (E \times E \times E \times E)
\]

because the first expression is the type of a connection such as \( (\langle a, b \rangle, \langle c, d \rangle) \), while the second expression is the type of a connection such as \( (a, b, c, d) \).

From these connection type expressions, system type expressions can be composed. *Directional* systems have a type of the form

```
input type \Rightarrow output type
```

For example,

\[
E \times E \Rightarrow E
\]

is the type of a system that has an input connection of type \( E \times E \) and an output connection of type \( E \).

*Adirectional* systems have a type of the form

```
[connection type]
```

For example,
\([E \times E \times E]\)

is the type of a system that has three adirectional connections.

In adirectional types it is possible to indicate the direction of some connections by using the prefix ‘?’ for input and ‘!’ for output. For example:

\([?E \times E \times !E]\)

Although this notation is mainly intended for the reader, for some classes of circuits—for example logical gates without open-collector outputs—it is possible to check these direction annotations for conflicts.

Since Glass requires that all system and macro definitions have a type specification, some of the definitions in the previous sections were incomplete. Their complete definitions are:

```
def sel \in E \times E \times E \Rightarrow E;
  sel (a, b, s) = or (and (a, s), and (b, not s));
```

```
def amp10 \in E \Rightarrow E;
  amp10 i = o
  where
    o = opamp (i, div10 o);
  endwhere;
```

```
def RC \in [E \times E];
  RC (a, b) = \{ R (a, x), C (x, b) \};
```

To enhance the readability of system descriptions, Glass allows the introduction of new names for type expressions. For example,

```
type port \equiv E \times E;
```

defines port to be equivalent to the expression \(E \times E\). Thus,

```
port \times port = (E \times E) \times (E \times E)
```

The type expressions can be used to check the correctness of the system definitions and the applications of the defined systems. To do this, the type of each connection variable is deduced from the connection type of the systems where it is applied, and these deduced types are compared with each other. Similarly, the types of the formal parameters of a system are deduced and compared with the specified connection type.

Obviously, it is not possible to define every system that is applied: the recursion of definitions has to stop somewhere. Therefore, there will be systems that are used in the definition of other systems but are not defined themselves. Such systems are called atoms. Since the type of the atoms must be known to allow type checking, the type of all atoms must be declared with an atom declaration. For example
atom and $\in E \times E \Rightarrow E$;

Unlike a defined system, an atom may have value parameters, and for these parameters a type specification is also necessary. Value parameters can have the following types:

\begin{itemize}
\item int \quad \text{An integer.}
\item float \quad \text{A floating point number.}
\item string \quad \text{A string.}
\item bool \quad \text{A Boolean.}
\end{itemize}

For example, an allowed atom specification is:

atom $R \in \text{float} \rightarrow [E \times E]$;

5.3 The merger of $\lambda$-calculus and $\sigma$-calculus

For directional circuits, $\lambda$-calculus is much too convenient to be replaced by $\sigma$-calculus. The directionality of $\lambda$-calculus allows the use of function notation, and therefore avoids many auxiliary variables. Therefore, the language should allow both notations, and should provide the ability to mix both styles. Thus, a system that has been defined as a $\sigma$-abstraction should be available in $\lambda$-abstractions and vice versa.

This is achieved by introducing the convention that

\begin{verbatim}
def a $\in Ta \Rightarrow Tb$;
a i = x;
\end{verbatim}

is equivalent to

\begin{verbatim}
def a $\in [T a \times T b]$;
a (i, new) = \{ x(new, x) \};
\end{verbatim}

where new is a name that does not clash with existing names. If $x$ contains directional applications they are converted to equivalent adirectional applications that are also added to the appset. Thus, a definition such as

\begin{verbatim}
def a $\in Ta \Rightarrow Tb$;
a i = f i;
\end{verbatim}

would be converted to

\begin{verbatim}
def a $\in [T a \times T b]$;
a (i, new) = \{ f (i, new) \};
\end{verbatim}

68
5.4 Representation in computer character set

Most computer systems have only a limited character set that does not contain symbols such as $\rightarrow$, $\Rightarrow$ and $\in$. Thus, for computer applications a different representation of the language is necessary. For Glass such a representation has been defined. For example,

\[
\begin{align*}
\text{def} & \quad \text{movavr} \in s \Rightarrow s; \\
\text{movavr} & \quad i = o \\
\text{where} & \quad o = \text{scale} \ 0.5 \ \text{(sum} \ \{a, \text{ delay} \ o\}); \\
\text{endwhere};
\end{align*}
\]

is represented as

\[
\begin{align*}
\text{def} & \quad \text{movavr} :: = s \Rightarrow s; \\
\text{movavr} & \quad i = o \\
\text{where} & \quad o = \text{scale} \ 0.5 \ \text{(sum} \ \{a, \text{ delay} \ o\}); \\
\text{endwhere};
\end{align*}
\]

In this thesis I will not use this representation, details are given in [50].

5.5 The macro language

It is often necessary to repeat circuit parts, or to define classes of similar circuits, for example:

\[
\begin{align*}
\text{def} & \quad \text{div4} \in E \Rightarrow E; \\
\text{div4} & \quad i = \text{div2} (\text{div2} \ i); \\
\end{align*}
\]

\[
\begin{align*}
\text{def} & \quad \text{div8} \in E \Rightarrow E; \\
\text{div8} & \quad i = \text{div2} (\text{div2} (\text{div2} \ i)); \\
\end{align*}
\]

\[\vdots\]

Classes of definitions cannot be described in the system description language itself, because it is not possible to give an interpretation to a class of systems, only to individual systems. Therefore, the class definitions must be expanded before any semantic functions are applied to the circuit descriptions. For this purpose, it is useful to introduce a macro language: a language to describe circuit classes, similar to the macro languages of the programming language C and some assemblers. In Glass, this macro language is in fact a small functional language that has fragments of system definitions as one of its data types.

For example, a valid macro definition is:
type \( port = E \times E; \)

\( \text{type} \quad 	ext{twoport} = \text{port} \times \text{port}; \)

\/* The cascade of two twoports. */

\textbf{mac} \quad \text{casc} \in [\text{twoport}] \rightarrow [\text{twoport}] \rightarrow [\text{twoport}];

\text{casc } sa \; sb \; (a,b) = \{ \; sa \; (a,c), \; sb \; (c,b) \; \};

A macro such as \text{casc} cannot by itself be regarded as a system, since it has two systems as parameters.

Similar to other functional languages, the macro language of Glass allows \textit{pattern matching}. This means that a macro may have several partial definitions that may contain values instead of abstractor variables. The associated function definition is used only if the applied value matches the value expression. The function definitions are tried from top to bottom. For example:

\textbf{mac} \quad \text{div} \in \text{int} \rightarrow E \Rightarrow E;

\text{div } 2 \; i = \text{div2 } i;

\text{div } n \; i = \text{div2 } (\text{div } (n/2) \; i);

In this case the first definition is applied only if the first parameter of \text{div} equals 2.

The Glass language including macros is called \textit{full Glass}, the Glass language without macros (or the result of the expansion of the macros) is called \textit{kernel Glass}.

\section*{5.6 Library support}

To facilitate re-use of software, Glass should support libraries of system descriptions. Ideally, it should be possible to hide definitions that are local to a library from other libraries.

Unfortunately, there was no time available for the design and implementation of such a module system. Instead, the file inclusion facility of the C preprocessor [25] is used, which is much more limited.

70
Chapter 6

Support for semantic functions

As explained in chapter 4, the support for semantic functions should meet the following requirements:

1. Insofar as possible, code should be shared between semantic functions. Thus, standard software should be available for commonly used operations on Glass, for example parsing and macro expansion.

2. The support software should be robust and reliable.

3. The implementation of semantic functions should be made as easy as possible.

4. It should be possible to implement semantic functions in arbitrary programming languages.

After some experimentation it was decided to implement the semantic functions as shown in figure 6.1: the Glass text is processed by a common parser, macro expander and type checker. The semantic functions work on a special internal representation that only represents the kernel language, i.e. the language without macros.

The representation for semantic functions should be easy to access from the language in which the semantic function is implemented, and should only contain essential information for the semantic functions. As internal representation, a format similar to that of the data structures of the programming language Miranda [55] has been chosen. Since it is necessary to allow the implementation of semantic functions in many programming languages, a special code generator, called Tm, has been developed that allows easy access to the internal representation from common programming languages such as C and Pascal. Tm is not only useful to generate interface code, many of the available functions it generates are of general use.

Since Tm is a code generator that is frequently used, it may be assumed that its code is reliable. For this reason it has been used in a number of other projects, and often the generated Tm code is used for purposes other than interfacing.
6.1 Tm

The transfer of structured data (such as the Glass internal representation) between programs is often carried out using ad hoc binary or textual formats. However, this can result in ambiguous and non-portable file formats. For example, the Pascal type declaration

```pascal
record foo x, y: integer; c: char; end;
```

is 'equivalent' to the C type definition

```c
typedef struct { int x, y; char c; } foo;
```

This does not imply that it is easy to transfer data in these records and structures from one language to the other. Facilities that are provided for this purpose, such as `get` and `put` in Pascal and `fread` and `fwrite` in C are useless, and may even cause problems if data is transferred between different implementations of the same language.

An effective way to solve this problem is to introduce a textual representation of the data, and replace the binary read and write routines by text printing and parsing routines. It is now necessary, however, to define a suitable language for this representation. If this is not done properly, it may lead to inconsistency or system dependency.

The program Tm (for template manager) allows such textual representations to be defined in a special data-structure definition language. Tm is able to generate the
data-structure definitions for a number of programming languages from Tm data-structure definitions. It can also generate code to read the textual representation into internal data structures, and code to write these internal data structures to the textual representation.

At the moment, Tm can generate code for C, Lisp, Miranda and Pascal. Code generation for any other language is easily added, because the generation is based on 'templates': source texts for the target programming language interspersed using text substitution and repetition commands for Tm.

6.2 Tm data-structure definition

The central information source for Tm is a file containing a list of type definitions. Both the syntax of the data-structure definition and the textual representation are derived from the tuple and the algebraic types of Miranda, see [55], [37] or [56]. In fact, it is often possible to use the textual representation as Miranda source text.

The types that can be defined in this type-definition file are tuple types and a constructor types.

A tuple is a group of elements of fixed length and order, see section 2.2.4. Such a tuple type is very similar to the records of Pascal, the structures of C and the tuples used in data-base systems. In Tm, a tuple type definition consists of a list of element types; each element is also given a name. For example,

\[
\text{foo} == ( \text{x: int}, \text{y: int}, \text{c: char} );
\]

defines a tuple type with element types int, int and char. The element names are x, y and c.

A constructor consists of a number of alternative element groups. Constructors are convenient for the representation of recursive structures. For example:

\[
\text{tree} ::= \text{Tree v: int } 1:\text{tree } r:\text{tree } | \text{TreeNIL};
\]

defines a constructor type.

The words Tree and TreeNIL are used to differentiate between possible alternatives and are called constructors. The character 'l' is used to separate alternatives. By convention, constructors start with an upper case character, while type and element names start with a lower case character.

Constructor and tuple elements may be constructors and tuples, but they may also be of any other type. If the type name is not known to Tm, it is assumed that the type is defined elsewhere; such a type is called a primitive type. For example, in type tree type int is primitive.

Constructors and tuple elements can be grouped into lists of arbitrary length (including length 0). All elements of such a list must be of the same type. To indicate
that a list is intended, the type name is surrounded by square brackets. For example, 
[[foo]] is a valid list type. Like constructor and tuple elements, list elements may 
be tuples, constructors or elements of primitive types. However, for implementation 
reasons Tm does not allow lists of lists. Thus, [[[foo]]] is not allowed.

6.3 Tm textual data representation

The type definitions described in section 6.2 are used as the syntax of the actual 
textual representation. For example, an instance of type [foo] is:

[[ 3, 5, 'c', ( -1, 42, '?' )]]

and an instance of type tree is:

Tree 2 (Tree 6 TreeNIL TreeNIL) TreeNIL

An other instance of type tree is:

TreeNIL

The exact rules for this representation are described in [43].

This representation can be parsed by a simple recursive descent parser. Such a parser 
can be implemented in almost any programming language.

6.4 The text substitution language of Tm

Tm can generate code to read and write the tuples, constructors and lists described 
in section 6.3 for a number of programming languages. This is done by providing 
templates for the various languages that are filled in using the given data-structure 
definitions (see figure 6.2). Templates are source texts for the target programming 
language interspersed with text substitution and repetition commands for Tm. An 
example of a template is given at the end of this section.
The templates for the various languages are implemented using the text substitution language described in this section. The casual user of Tm is expected to use only the standard templates that have been prepared for various programming languages, and needs only limited knowledge of the text substitution language. While using the standard templates, he only needs the following constructs of the text substitution language of Tm:

1. **Comment**.

2. **Variable assignment**. This is used to specify the list of functions that Tm should generate, and to provide other configuration information.

3. **File inclusion**. File inclusion is used to invoke the standard templates.

I will only describe the text substitution language superficially, the defining document is [43].

Tm will copy all text from the template to the output file, with two exceptions:

1. All lines starting with a period (`.`) are interpreted as commands to Tm. They are called line commands. Some line commands 'bracket' a number of lines; for example:

   ```
   .if 0
   True
   .else
   False
   .endif
   ```

   results in:

   ```
   False
   ```

   since Tm interprets 0 as the Boolean value 'false'. There are also line commands for repetition and file inclusion.

   An other important line command is `.set`. It is used to assign lists of words to variables. For example:

   ```
   .set blah foo bar
   ```

   Assigns the words `foo` `bar` to the Tm variable `blah`.

   Comment can be included in the template using two periods at the beginning of the line. For example:

   ```
   .. This is a comment line.
   ```
2. Within all lines (both line commands and normal lines), the character ‘$’ indicates the start of an expression that is evaluated by Tm. In the output of Tm, the expression is replaced by the result of evaluation.

There are three forms of $ expressions:

1. Expressions of the form ${\text{fn parm ... parm}}$ are function applications. The first word within the brackets is the function name, the remaining words are the parameters of the function. Tm provides functions for access to type definitions, arithmetic, comparison, list manipulation, version control, and various other functions. For example:

```
.set 1 h a c b d c
.sort $1
.uniq $1
```

results in:

```
a b c c d h
a b c d h
```

An important class of Tm functions is that for data-structure access. These functions return information about the defined data structures. For example, `${\text{typelist}}$ returns the list of defined types.

2. Expressions of the form $[{\text{expr}}]$ are numerical expressions. All the standard arithmetic operators are available. For example:

```
[1+2+3]
```

results in:

```
6
```

3. Expressions of the form ${\text{varname}}$ are variable references. These variables are set with the line command .set, see above. If the variable name consists of a single character, the parentheses may be omitted. For example:

```
.set bar text
.set z substitution
Tm does $(\text{bar})$ $z$
```

results in:

```
Tm does text substitution!
```
To see how this language can be used for code generation, consider the following Tm code to generate a Lisp list reflecting the data-structure definition:

```
.. Reflect datastructure definitions in a Lisp list.

( .. Set 't' to each defined type and do everything
   .. up to last '.endforach'..
   .foreach t ${typelist}
   .  if $\{len ${telmlist $t}\}$
   .. 't' is a tuple type, show its elements.
      (tuple $t$ ${telmlist $t}\})
   .else
   .. 't' is a constructor type, show its constructors
   .. and the elements of each constructor.
      (constructor $t$
      .foreach c ${conslist $t}$
      ($c$ ${celmlist $t$ $c}$)
      .endforeach
      )
    .endif
   .endforach
)
```

With this Tm code and a data-structure definition file that contains the type definitions of foo and tree, Tm will produce the following output:

```
( (tuple foo x y c)
  (constructor tree
   (Tree v l r)
   (TreeNIL )
  )
)
```

In the example the following functions are used:

- `typelist`  
  Return the list of types defined in the data-structure file.
- `telmlist t`  
  Given a tuple type $t$, return the list of element names of that tuple. For a constructor type return an empty list.
- `conslist t`  
  Given a constructor type $t$, return the list of constructors of that type. For a tuple type return an empty list.
- `celmlist t c`  
  Given a constructor type $t$ and a constructor name $c$, return the list of element names of that constructor. If $t$ is a tuple type, an error message is given.

The text substitution language is powerful enough to allow the implementation of complicated templates. For example, the templates of all supported languages require that the user supplies a list of wanted functions. From this list, the list of functions that are necessary to support these functions is deduced using the text substitution language.
6.5 Applications of Tm

The original purpose of Tm was to provide a well-defined interface between the Glass parser and its semantic functions. Tm has proved to be very useful for this purpose, especially since it allows interpretation functions to be written in several languages. However, it has also been used for other, similar applications.

Tm has also been used extensively for data structures that are only used within one program. This was done for the following reasons:

- Code that is generated by Tm is well tested. Essentially the same code (only differing in type names) is used for all data types in all generated code. Therefore, it has been exposed to a large number of runs in other generated code, and therefore has been better tested than similar code written by hand for the same purpose.

- The functions are documented.

- Tm code assists debugging. For example, the C templates of Tm generate code to count the number of allocations and deallocations of each particular type. This has proved to be invaluable for the detection of memory leakage (memory allocation not followed by deallocation) and repeated deallocation of the same memory. Also, since there is a textual representation, inspection of the data is easily possible.

- Tm encourages a standard coding style. The standardization of Tm functions also forces more standardization on the functions that use the Tm functions. This enhances the clarity of the code.

These advantages have led some users of Tm not to use it to generate interface code, but simply to generate correct and debug-assisting code.

One program that uses Tm-generated code extensively is Ampdes, see section 9.1. From a total of about 125000 lines of code, 70000 are generated by Tm.

6.6 The Glass internal representation

In this section the actual Tm data-structure definitions for the representation of Glass are described. The complete definition file is listed in appendix A.

Constructor type def represents the possible definitions and declarations that make up a Glass description:

\[
\text{def} ::= \begin{array}{l}
\text{DefAtom atorig:orig atnm:symbol atptyp: [partyp] atctyp:typ |} \\
\text{DefBasetype baseorig:orig basename:symbol |}
\end{array}
\]
DefVal valorig:orig valnm:symbol valtyp:typ valas:val |  
DefCon conorig:orig defcon:val conas:val |  
DefTyp typorig:orig typnm:symbol typas:typ ;

Connection equivalences (DefCon) should only occur within where clauses, while atom and basetype definitions (DefAtom and DefBasetype) should not occur within where clauses.

The origin file and line number of Glass description parts are represented by type orig.

orig == (file:string, line:inum);

This information is used in error messages and for debugging purposes.

Type typ represents connection and system type expressions.

typ ::=  
TypBase basenm:symbol | // basetype  
TypIn ityp:typ | // ?ityp  
TypOut otyp:typ | // !otyp  
TypUni utyp:typ utotyp:typ | // utyp => utotyp  
TypMon nontyp:typ | // adirectional system  
TypProd ptypes:[typ] | // Cartesian product  
TypSym sym:symbol; // symbolic type

The software must ensure that a TypSym never refers to a basetype, a TypBase must be used for this instead.

Only atoms may have value parameters (e.g. resistor values). These value parameters are represented by parval:

parval ::=  
ParInt i:inum | // int  
ParFlo f:fnum | // float  
ParStr s:string | // string  
ParBool b:bool ; // Boolean

while the type of these value parameters is represented by partyp:

partyp ::= PTInt | PTFlo | PTStr | PTBool;

Type val represents connection expressions, directional and adirectional applications and aspects.

val ::=  
VSym symorig:orig sym:symbol | // Symbol  
VLambda lpar:formcon lval:val | // Lambda abstraction
\texttt{V\textit{Sigma} spar:\textit{formcon} sval:val \mid \text{Sigma abstraction}} \\
\texttt{V\textit{Apply} aval:val apar:val \mid \text{Apply (asval aspar)}} \\
\texttt{V\textit{Where} wdefs:[\texttt{def}] wval:val \mid \text{local context}} \\
\texttt{V\textit{List} l:[\texttt{val}] \mid \text{List expression}} \\
\texttt{V\textit{Appset} asp:[\texttt{val}] \mid \text{appset}} \\
\texttt{V\textit{Atom} atorig:orig atm:atom symbol atvpar:[\texttt{parval}] atcpars:val \mid} \\
\texttt{V\textit{Syn} synlist:[\texttt{val}]; \mid \text{Synonym expression}}

The software must ensure that a \texttt{VSym} never refers to an \texttt{atom}, a \texttt{VAtom} must be used for this instead. The formal parameters of \(\lambda\) and \(\sigma\)-abstractions (\texttt{VLambda} and \texttt{V\textit{Sigma}}) are represented by the type \texttt{formcon}:

\texttt{formcon ::= F\textit{CList} l:[formcon] \mid F\textit{CSym} sym:symbol;}

As an example, consider the following \texttt{Glass} description:

\begin{quote}
\begin{verbatim}
basetype  A;

atom B \in int \rightarrow A \Rightarrow A;

def C \in A \Rightarrow A;
    C i = B 1 (B 2 i);
\end{verbatim}
\end{quote}

This description is represented as (assuming this description is in the file \texttt{t}):

\begin{verbatim}
[ 
    (DefBasetype ("t", 1) "A"),
    (DefAtom
        ("t", 2)
        "B"
        [PTInt]
        (TypUni (TypBase "A") (TypBase "A")))
    ),
    (DefVal
        ("t", 4)
        "C"
        (TypUni (TypBase "A") (TypBase "A"))
        (VLambda
            (FCSym "i")
            (VAtom
                ("t", 4)
                "B"))
    )
\end{verbatim}
6.7 Netlist representation

The representation described in the previous section is a complete representation of kernel Glass. The representation is, however, quite complicated to handle for a semantic function. Some semantic functions require (or work just as well on) a flattened circuit representation. A special flattener has been written to accomodate these functions, and a special, much simpler, representation format is available for them. Its data-structure definitions are listed in appendix A.
Chapter 7

Implementation of semantic functions

In this chapter some example implementations of semantic functions on Glass are described. These semantic functions are:

- Flattening: removal of hierarchy from Glass descriptions (section 7.1).
- Analog signal-flow analysis (section 7.2).
- Generation of behaviour equations for Maple (section 7.3).
- Interfacing to existing circuit analysis programs such as Spice (section 7.4).

By convention, flattening is in fact not considered a semantic function, since it only converts the Glass description into another form, and it is not concerned with the actual contents of the Glass description, such as the atoms that occur in it. For implementation, the difference is unimportant.

The semantic functions have been chosen to demonstrate an aspect of the implementation of semantic functions: the flattening function demonstrates the transformations that can be done on Glass descriptions. Moreover, some semantic functions require a flattened Glass description. Analog discrete-time signal-flow analysis is an easily implemented semantic function, and is useful to demonstrate the principle of a semantic function. The other semantic functions show the use of existing programs in semantic functions.

Most of these semantic functions give a behaviour interpretation rather than a structural interpretation to Glass. This is done for two reasons:

- The behaviour of a system is more important than structure ("form follows function").
• Behaviour semantic functions are easier to use as examples.

Important structural semantic functions are layout generation and the derivation of a circuit diagram. Both require complex algorithms that would distract from the interpretation aspect that is important in this thesis. For circuit diagrams there is another complication: there is no satisfactory algorithm for the generation of clear and organized circuit diagrams from netlists or Glass descriptions, although [2] describes an interesting proposal in this direction.

7.1 Flattening

The flattening function converts an arbitrary Glass description into a description with only one adirectionial definition, that only contains applications of atoms, and no application of other definitions or synonym expressions. Moreover, it is ensured that each connection variable in the definition represents a basetype connection, not a composite connection. Thus, flattening a system description means that all hierarchy in the description is removed. A definition of this form is easily interpreted as a netlist.

For example, given the Glass description

```plaintext
type port = (E×E);

def RC ∈ [port×port]
    RC ((a,b),(c,d)) = { R "1K" (a,c), C "1n" (c,d), *(b,d) };

def RC2 ∈ [port×port]
    RC2 (x,y) = { RC (x,z), RC (z,y) };
```

flattening RC2 would result in:

```plaintext
def RC2 ∈ [(E×E)×(E×E)]
    RC2 ⟨⟨a,b⟩,(c,d)⟩ =
    {
        R "1K" (a,c),
        C "1n" (c,d),
        *(b,d),
        R "1K" (e,c),
        C "1n" (c,d),
        *(f,d)
    }
```

For flattening, the following transformations are necessary:

1. Conversion of directional definitions to adirectionial ones. In general one must assume that a description contains both. Directional definitions can always be converted to adirectionial ones, but the converse is not always possible.
2. Conversion of connection equivalences to synonym expressions. The connection equivalences must be removed to allow the removal of where clauses.

3. Expansion of abstraction applications. This is the essential transformation of flattening, since it removes the hierarchy.

4. Expansion of connection variables. This ensures that each variable represents a basic connection, not a tuple.

5. Removal of synonym expressions.

These conversions must be done in the given order.

As an example I will show each of the steps in the flattening of the $\text{div4}$:

\[
\text{atom } Dff \in E \times E \Rightarrow E \times E;
\]

\[
\text{def } \text{div4} \in E \Rightarrow E;
\]

\[
\text{div4 } i = \text{div2} \ (\text{div2 } i)
\]

where

\[
\text{def } \text{div2} \in E \Rightarrow E;
\]

\[
\text{div2 } i = q
\]

where

\[
(q, nq) = Dff \ (nq, i);
\]

endwhere;

\[
\text{endwhere;}
\]

7.1.1 conversion of directional abstractions to adirectional ones

As explained in section 5.3, the convention has been adopted that a system definition such as

\[
\text{def } a \in Ta \Rightarrow Tb;
\]

\[
a \ i = f \ i;
\]

may be converted to its adirectional equivalent

\[
\text{def } a \in [Ta \times !Tb];
\]

\[
a \ (i, \ new) = \{ \ *\ (\text{new}, \ nI), \ f \ (i, \ nI) \} ;
\]

where new and nI are new variable names. In this example the definition could be converted to:
\[
def \ a \in \{T\times T, T\}; \\
a \langle i, \text{new} \rangle = \{ \ f \langle i, \text{new} \rangle \};
\]

thereby removing one of the new variables, but in general this is not possible. As explained in section 4.1, it is bad practice to handle exceptions such as this one separately, since this would make the software less reliable, and, therefore, the more general conversion is done.

The example definition, \( \text{div}_4 \), is now converted to:

\[
\text{atom} \ \text{Diff} \in \{E \times E \times E \};
\]

\[
def \ \text{div}_4 \in \{E \times E \}; \\
\text{div}_4 \langle i,n1 \rangle = \{ \ \langle n1,n2 \rangle, \ \text{div}_2 \langle n3,n2 \rangle, \ \text{div}_2 \langle i,n3 \rangle \}
\]

where

\[
def \ \text{div}_2 \in \{E \times E \}; \\
\text{div}_2 \langle i,n4 \rangle = \{ \ \langle q,n4 \rangle \}
\]

where

\[
\langle q,nq \rangle = \text{Diff} \langle nq,i \rangle;
\]

endwhere;

7.1.2 removal of connection equivalences

Where clauses might contain connection equivalences that must be removed. For example, the above where clause of \( \text{div}_2 \) contains such a connection equivalence. Assuming that the abstraction is adirectional, the where clauses can be converted to synonym expressions and applications in the appset. This is done as follows:

- If the equivalence does not contain system applications, it can be converted to a synonym expression. For example:

\[
a = b
\]

is converted to the appset

\[
\{ \ \langle a,b \rangle \}
\]

- If the equivalence expression contains system applications, they are added to the appset as separate applications. For example:

\[
f \ (g \ i)
\]

is converted to

\[
\{ \ f \ (n2,n1), \ g \ (i,n2) \}
\]
where \( n1 \) and \( n2 \) are names that do not clash with existing names. The variable \( n1 \) now represents the output of the expression.

The example definition, \( \text{div}_4 \), is now converted to:

\[
\text{atom } \text{Diff} \in [?/(E \times E)]
\]

\[
\text{def } \text{div}_4 \in [?E \times !E];
\]

\[
\text{div}_4 \ (i,n1) = \{ \ {n1,n2}, \ \text{div}_2 \ (n3,n2), \ \text{div}_2 \ (i,n5) \ \}
\]

where

\[
\text{def } \text{div}_2 \in [?E \times !E];
\]

\[
\text{div}_2 \ (i,n4) =
\]

\[
\{ \ *\langle n4,q \rangle , \ *\langle \langle q,nq \rangle ,n5 \rangle , \ \text{Diff} \ \langle \langle nq,i \rangle ,n5 \rangle , \}
\]

7.1.3 expansion of abstraction applications

For this transformation it is necessary that directional abstractions have been rewritten into their adirectional equivalents. Applications of adirectional abstractions can be replaced by:

1. The appset of the abstraction.

2. A synonym expression equating the abstractor of the applied abstraction with the connection expression of the application.

This conversion is equivalent to \( \beta \)-conversion, see section 2.4.3.

It must be ensured that the connection names within the abstraction differ from the names that are 'visible' from the application. In the implemented flattening function this is done by generating new names for all connection variables.

The example definition, \( \text{div}_4 \), is now converted to:

\[
\text{atom } \text{Diff} \in [?/(E \times E)]
\]

\[
\text{def } \text{div}_4 \in [?E \times !E];
\]

\[
\text{div}_4 \ (i,n1) =
\]

\[
\{ \ *\langle n1,n2 \rangle , \{ \ *\langle n7,n8 \rangle , \ *\langle \langle n8,n9 \rangle ,n10 \rangle , \ \text{Diff} \ \langle \langle n9,n6 \rangle ,n10 \rangle \}, \ *\langle n6,n7 \rangle ,\langle n3,n2 \rangle \}, \{ \ *\langle n14,n15 \rangle , \ *\langle \langle n15,n16 \rangle ,n17 \rangle , \ \text{Diff} \ \langle \langle n16,n19 \rangle ,n17 \rangle \}, \ *\langle n13,n14 \rangle ,\langle i,n5 \rangle \}
\]
The nested appsets in this definition can be flattened without problems, they have been added here for clarity.

In the first expansion of `div2` its variables are renamed as follows:

\[
\begin{align*}
&i \quad n6 \\
&n4 \quad n7 \\
&q \quad n8 \\
nq &n9 \\
&n5 \quad n10
\end{align*}
\]

in the second expansion they are renamed as:

\[
\begin{align*}
&i \quad n13 \\
&n4 \quad n14 \\
&q \quad n15 \\
nq &n16 \\
&n5 \quad n17
\end{align*}
\]

### 7.1.4 expansion of connection variables

For semantic functions that use the flattening transformation, it is convenient if it is ensured that each connection variable in the flattened definition represents a basetype connection, and not a composite connection. This is easily accomplished by adding a synonym expression of the form

\[*(c, \text{full})\]

to the appset for each atom in the appset, where \(c\) is the existing connection expression of the atom, and \(\text{full}\) is a fully expanded connection expression with new connection names. Using the conversion rules for synonym expressions described in section 7.1.5, this will ensure the connection expressions are expanded.

For example,

\[
\begin{align*}
\text{atom } R \in \text{num} &\rightarrow [E \times E]; \\
\text{def } Rl &\in [E \times E]; \\
&\quad Rl c = \{ \ R \ 1e3 \ c \ \};
\end{align*}
\]

is converted to:

\[
\begin{align*}
\text{atom } R \in \text{num} &\rightarrow [E \times E]; \\
\text{def } Rl &\in [E \times E]; \\
&\quad Rl c = \{ \ R \ 1e3 \ c, \ *(c, (n1,n2)) \ \};
\end{align*}
\]

The `div4` example remains unchanged.
7.1.5 removal of synonym expressions

To remove synonym expressions, they are converted to a set of synonym expressions of the following forms:

\[ *(a, b) \]
\[ *(a, (b0, b1 \ldots bn)) \]

Synonym expressions of these forms are easily removed from an adirectional definition by replacing all occurrences of \( a \) in the appset and the abstractor by \( b \) or \( (b0, b1 \ldots bn) \) respectively.

It is possible to convert arbitrary synonym expressions into the canonical forms using the following rewrite rules:

1. \[ *(e0, e1 \ldots en) \Rightarrow *(e0, e1), *(e0, e2), \ldots *(e0, en) \]
2. \[ *(\langle a0 \ldots an \rangle, \langle b0 \ldots bn \rangle) \Rightarrow *(\langle a0, b0 \rangle, \langle a1, b1 \rangle, \ldots \langle an, bn \rangle) \]
3. \[ *(\langle a0 \ldots an \rangle, b) \Rightarrow *(b, \langle a0 \ldots an \rangle) \]

Moreover, synonym expressions of the following forms can be deleted:

\[ *(e, e) \]
\[ *(e) \]
\[ *() \]

Rule 1 ensures that each synonym expression is converted to one of the following forms:

1. \[ *(a, b) \]
2. \[ *(a, \langle b0 \ldots bn \rangle) \]
3. \[ *(\langle a0 \ldots an \rangle, b) \]
4. \[ *(\langle a0 \ldots an \rangle, \langle b0 \ldots bn \rangle) \]

Form 1 and 2 are canonical forms. Form 3 is directly converted to one of the canonical forms using rule 3. Form 4 is converted to smaller synonym expressions using rule 2. Since synonym expressions have a finite size, eventually these smaller expressions will be converted to one of the canonical forms.

Note that in rule 2 it is assumed that both tuples are of the same size. In correct Glass descriptions this must be true, since otherwise the expressions would have different types, which is not allowed.

For example, the synonym expression

\[ *(\langle a, b \rangle, c, \langle d, b \rangle) \]

is converted as follows:

89
\*(\langle a, b \rangle, c, \langle d, b \rangle) \\
= \*\langle (a, b), c \rangle, \*\langle (a, b), (d, b) \rangle \\
= \*\langle c, (a, b) \rangle, \*\langle a, d \rangle, \*\langle b, b \rangle \\
= \*\langle c, (a, b) \rangle, \*\langle a, d \rangle

Using these rewrite rules for synonym expressions, the definition of \textit{div4} is now:

\begin{verbatim}
atom Diff ∈ [?(E×E)×!(E×E)];
def div4 ∈ [?E×!E];
div4 (i, n1) =
{
    \*\langle n1, n2 \rangle,
    \*\langle n7, n5 \rangle,
    \*\langle n10, \langle n8, n9 \rangle \rangle,
    Diff ((\langle n9, n6 \rangle, n10),
    \*\langle n6, n3 \rangle,
    \*\langle n7, n2 \rangle,
    \*\langle n14, n15 \rangle,
    \*\langle n17, \langle n15, n16 \rangle \rangle,
    Diff ((\langle n16, n19 \rangle, n17),
    \*\langle n14, n3 \rangle,
    \*\langle n19, i \rangle
}
\end{verbatim}

Applying the substitutions implied in these synonym expressions yields:

\begin{verbatim}
atom Diff ∈ [?(E×E)×!(E×E)];
def div4 ∈ [?E×!E];
div4 (i, n2) =
{
    Diff ((\langle n9, n3 \rangle, \langle n2, n9 \rangle),
    Diff ((\langle n10, i \rangle, \langle n3, n16 \rangle),
}
\end{verbatim}

And \textit{Rl} is now converted to:

\begin{verbatim}
atom R ∈ num → [E×E];
def Rl ∈ [E×E];
Rl \langle n1, n2 \rangle = \{ R \ l\ e\ s \ \langle n1, n2 \rangle \};
\end{verbatim}

These appsets are easily interpreted as netlists.
7.2 Analog discrete-time signal-flow analysis

This semantic function is described mainly because it approximates the ideal semantic function: it is simple and powerful. Also, it shows the use of the directional part of Glass. Since this semantic function is so small, it is reasonable to list it entirely. This is done in appendix C.

In analog discrete-time signal-flow analysis, signals are represented by streams: sequences of analog values of arbitrary length. For example:

\[ [0, 0.25, 0.5, 0.75, 1, 0, 0.25, \ldots] \]

These streams represent signal values at fixed times. The individual values are referred to by using an index. For example:

\[ a = [a_0, a_1, \ldots, a_n] \]

Each component in the system has a number of these streams as its input and output. The behaviour of such systems is easily described as a set of functions in a functional programming language. Table 7.1 lists the atoms of analog discrete-time signal-flow analysis and their behaviour. Figure 7.1 shows an example system. The Glass description of this system is (assuming appropriate atom definitions):

```plaintext
basetype S;

def movavr ∈ S ⇒ S;
    movavr i = o
where
    o = scale 0.5 (add (a, delay o));
endwhere;
```

The atoms and definitions of movavr can be translated into functions in a programming language. These functions simulate the behaviour of the system. For this semantic function, the functional programming language Miranda [56] was chosen,
Table 7.1: atoms of analog discrete-time signal-flow analysis.
since it supports lists of arbitrary length. These lists are very convenient to represent the data streams that are used in this semantic function. For example, the behaviour of the system `movavr` could be simulated by the following Miranda script:

```plaintext
-- A signal is a list of numbers of arbitrary length.
s :: [num];

-- The function 'scale' scales all samples in a signal. For a signal with no samples this is easy: it results in another signal with no samples. For other signals the first sample (called 'a') is scaled and put in front of the list of scaled remaining samples. The remainder of the samples is scaled in a recursive application.
scale :: num->s->s
scale n []     = []
scale n (a:b)  = n*a:(scale n b)

-- The function 'add' adds the samples of two signals to make one new signal. If one of the signals has no samples, the resulting list will have no samples, else the first samples of both the lists ('xh' and 'yh') are added and put in front of the list of remaining added samples. The remainder of the samples is added in a recursive application.
add :: (s,s)->s
add ([],b)    = []
add (a,[])    = []
add ((xh:xt),(yh:yt)) = (xh+yh):(add (xt,yt))

-- The function 'delay' shifts all samples to the next time by putting a new sample with value 0 in front of the original signal.
delay :: s->s
delay a = 0:a

-- The function 'movavr' simulates the behaviour of the system 'movavr'.
movavr :: s->s
movavr i = o
    where
        o = (scale 0.5 (add (i, (delay o))))

These functions will simulate the behaviour of movavr.

### 7.3 Generation of Maple equations

Using the symbolic formula manipulation program Maple [12], it is quite simple to derive a symbolic transfer function of a circuit.

The behaviour equations are represented using the Nodal Admittance (NA) matrix, see section 3.4.2. As explained in that section, the equations of only three components can be directly inserted in the NA matrix equation: a resistor, a current source and
a voltage-controlled current source. Obviously these are the atoms of this semantic function. One other atom is added: a voltage detector.

Maple can handle symbolic expressions. Therefore, it is useful to represent component values as strings, since this way they may contain arbitrary Maple expressions. Because this semantic function produces symbolic transfer functions anyway, it is possible to produce Laplace domain transfers by simply generating the appropriate polynomials in the Laplace variable $s$. In principle this could be done by using an application such as

$$ R \ "10e-9s" \ (a,b) $$

for an inductance, but this is not very convenient. Therefore, capacitors and inductors are introduced as separate atoms.

The atom declarations of this semantic function are:

```plaintext
basetype $E$

atom $R \in string \rightarrow [E \times E]$;
atom $C \in string \rightarrow [E \times E]$;
atom $L \in string \rightarrow [E \times E]$;
atom $I \in string \rightarrow [E \times E]$;
atom $meter \in string \rightarrow [E \times E]$;
atom $vccs \in string \rightarrow [(E \times E) \times (E \times E)]$;
```

Many other useful components can be defined using these atoms with the equivalences described in section 3.4.2. For example:

```plaintext
/* A gyrator is two vccs back to back. */
mac gyrator $\in string \rightarrow [(E \times E) \times (E \times E)]$;
 gyrator $g (a,b) \ = \ \{ \ vccs \ g \ (a,b), \ vccs \ g \ (b,a) \ \}$;

/* A cccs is a vccs and a gyrator. */
mac cccs $\in string \rightarrow [(E \times E) \times (E \times E)]$;
 cccs $g (a,b) \ = \ \{ \ gyrator \ "1" \ (a,c), \ vccs \ g \ (c,b) \ \}$;
```

This corresponds to the equivalences shown in figures 3.14 and 3.15. A semantic function has been written, similar to the semantic function listed in appendix C, that constructs an NA matrix from a flattened Glass description in a form that is suitable for Maple. The semantic function also adds Maple statements to compute the transfer of all current sources to all detectors in the circuit. As an example, consider the following Glass text:
def main ∈ [ ];
main () =
{
  src "is" (a, gnd),
  R "r1" (a, b),
  C "c1" (b, gnd),
  L "l1" (b, c),
  R "r2" (c, gnd),
  meter "vc" (c, gnd)
};

Figure 7.2 shows a circuit diagram of this circuit. The nodal analysis matrix for this circuit is:

\[
\begin{bmatrix}
\frac{1}{r_1} & \frac{-1}{r_1} & 0 \\
\frac{-1}{r_1} & \frac{1}{s \cdot l_1} + s \cdot c_1 & \frac{-1}{s \cdot l_1} \\
0 & \frac{1}{s \cdot l_1} + \frac{1}{r_2} & 0
\end{bmatrix} \cdot \begin{bmatrix}
v_a \\
v_b \\
v_c
\end{bmatrix} = \begin{bmatrix}
i_s \\
0 \\
0
\end{bmatrix}
\]

The implemented semantic function represents the matrix in a form that is suitable for the symbolic analysis program Maple. For example, for the circuit above the following Maple input is generated:

readlib(linalg):
readlib(latex):
with(linalg,submatrix):
with(linalg,inverse):
with(linalg,transpose):
with(linalg,multiply):

$\$ Convert list 'l' to a form that 'latex' will print properly.
latexvec:=proc(l) transpose(convert([l],array)) end:

$\$ Assign indices to the nodes.
nd_a:=1:
nd_b:=2:
nd_c:=3:
nd_gnd:=4:

95
$\text{Construct and fill the IDA matrix}$

\begin{verbatim}
ida := array(1..4,1..4,sparse):
ida[nd_a,nd_a] := simplify(ida[nd_a,nd_a]+(1/(r1))):
ida[nd_b,nd_b] := simplify(ida[nd_b,nd_b]+(1/(r1))):
ida[nd_a,nd_b] := simplify(ida[nd_a,nd_b]+(-1/(r1))):
ida[nd_b,nd_a] := simplify(ida[nd_b,nd_a]+(-1/(r1))):
ida[nd_gnd,nd_gnd] := simplify(ida[nd_gnd,nd_gnd]+(s*(c1))):
ida[nd_b,nd_gnd] := simplify(ida[nd_b,nd_gnd]+(-s*(c1))):
ida[nd_gnd,nd_b] := simplify(ida[nd_gnd,nd_b]+(-s*(c1))):
ida[nd_b,nd_b] := simplify(ida[nd_b,nd_b]+(1/(s*l1))):
ida[nd_c,nd_c] := simplify(ida[nd_c,nd_c]+(1/(s*l1))):
ida[nd_b,nd_c] := simplify(ida[nd_b,nd_c]+(-1/(s*l1))):
ida[nd_c,nd_b] := simplify(ida[nd_c,nd_b]+(-1/(s*l1))):
ida[nd_c,nd_c] := simplify(ida[nd_c,nd_c]+(1/(r2))):
ida[nd_gnd,nd_gnd] := simplify(ida[nd_gnd,nd_gnd]+(1/(r2))):
ida[nd_c,nd_gnd] := simplify(ida[nd_c,nd_gnd]+(-1/(r2))):
ida[nd_gnd,nd_c] := simplify(ida[nd_gnd,nd_c]+(-1/(r2))):
\end{verbatim}

$\text{Construct and fill the current vector.}$

\begin{verbatim}
invec := convert([0,0,0,0],array):
invec[nd_a] := simplify(invec[nd_a]+(is)):
invec[nd_gnd] := simplify(invec[nd_gnd]+(-is)):
\end{verbatim}

$\text{Delete a redundant row and column from the matrix.}$

\begin{verbatim}
na := submatrix(ida,1..3,1..3):
\end{verbatim}

$\text{Delete a redundant row from the current vector.}$

\begin{verbatim}
insubvec := [convert(invec,list)][1..3]:
\end{verbatim}

$\text{Solve the equations.}$

\begin{verbatim}
outsubvec := multiply( inverse(na), insubvec):
\end{verbatim}

$\text{Construct the full output vector.}$

\begin{verbatim}
outvec := convert(outsubvec,array,1..4):
outvec[4] := 0:
\end{verbatim}

$\text{Format the results for LaTeX.}$

\begin{verbatim}
outlv := latexvec([v_a,v_b,v_c]):
inlv := latexvec(convert(insubvec,list)): latex(map(eval,[na,outlv,inlv]),'mplna.txh'): writeto('mpldemo.txh'):
\end{verbatim}

$\text{Extract detector data from output vector and make LaTeX for it.}$

\begin{verbatim}
latex(vc := simplify(outvec[nd_c]-outvec[nd_gnd])):
quint
\end{verbatim}
Using this input, Maple will produce the following symbolic expression for \( v_c \):

\[
v_c = \frac{r_2}{s^2 \cdot i_1 \cdot c_1 + s \cdot c_1 \cdot r_2 + 1} \cdot i_s
\]

### 7.4 Input generation for existing programs

Semantic functions that analyze the behaviour of systems often use existing circuit analysis programs. These programs usually use a circuit description in the form of a netlist, and therefore it is often not possible to translate all constructs in kernel Glass into constructs in their input language.

Fortunately, a flattened Glass definition is easily interpreted as a netlist, so, after flattening, conversion to the appropriate input format for these programs is straightforward. Such translations have been implemented for the programs Anp and Spice. Ideally, all constructs of a target program should be available in Glass, but this is often difficult or impossible. The problems that arise in the implemented semantic functions are representative:

- Optional parameters.
- Non-electrical component coupling.
- Connection numbering.
- Component identification.
- Handling of analysis requests.

#### 7.4.1 optional atom parameters

Often components have parameters that are hardly ever used and have a reasonable default. Since Glass does not allow defaults, these parameters always have to be supplied. For example, the Spice resistor has two optional parameters to specify its behaviour for temperatures other than the default. Therefore, the Spice resistor should have the atom declaration

\[
\text{atom } Rt \in \text{float} \rightarrow \text{float} \rightarrow \text{float} \rightarrow [E\times E]
\]

Since it is very inconvenient to add these temperature parameters to all resistor applications, some shortcut has to be found. At least two solutions are possible:

1. Introduce a macro with reasonable defaults. For example,

   \[
   \text{mac } R \in \text{float} \rightarrow [E\times E];
   \]

   \[
   R \ v \ c = \{Rt \ 0 \ 0 \ v \ c\};
   \]
2. Delegate this shortcut to Spice, and pass the parameter or parameters as a string. Thus the atom declaration becomes:

\[
\text{atom } R \in \text{string } \rightarrow [E \times E];
\]

and one can use \( R \) as follows:

\[
R \ "1k\" \ (a,b)
\]

or as

\[
R \ "1k\ \ TC=0.001,0.0015\" \ (a,b)
\]

Solution 1 is more elegant, but is impossible to use it when the number of parameters is variable. For example, Spice independent voltage and current sources have an entire sub-language to describe periodic signals. Since solution 1 requires a macro definition, and since the macro expander has only recently become available, solution 2 has been chosen for the implemented semantic functions.

### 7.4.2 component coupling

Electrical connections in Glass are easily interpreted as nodes, but in some cases circuit simulators allow non-electrical connections between components. For example, both Spice and Anp support coupled inductors. Again, two solutions are possible:

1. Introduce connection variables in the non-electrical domain. For example, a base type for magnetic connections could be introduced:

   \[
   \text{basetype } M;
   \]

   However, the interpretation of such connections would complicate the semantic function considerably.

2. ‘Pack’ the non-electrical connection in an atom. For example, one atom could describe two inductors and their coupling.

Since solution 1 would involve considerably more bookkeeping, solution 2 has been chosen in the currently implemented semantic functions. Future semantic functions certainly have to do better in this respect.

### 7.4.3 node numbering

Spice and Anp require node numbers instead of node names. This conversion is easily done by first constructing a list of node names used, and assigning a unique number to each of these names.
7.4.4 global connection

Spice uses the node with number 0 as a common ground node. The best representation of such a node would be a global connection variable, but Glass does not allow these. As a way around this, for such a connection an atom \textit{gnd} is introduced:

\[
\text{atom gnd} \in E;
\]

An application of \textit{gnd} will label a connection as a connection to ground. This solution has as a disadvantage that the equivalence of the labelled nodes is not apparent. For example, for all interpretations the following is valid:

\[
\{ \text{gnd } a, \text{ gnd } b \} = \{ \text{gnd } a, \*_{a,b} \}
\]

But this is still an interpretation of \textit{gnd}, and therefore it is not covered by the standard conversion rules of Glass that are used by programs such as the flattener.

7.4.5 component identification

Spice and Anp require that each application of an atom is named, so that it can be distinguished from other applications of the same atom. Thus, in Spice the lines

\[
\begin{align*}
\text{R 1 2 1K} \\
\text{R 2 3 2K}
\end{align*}
\]

are not allowed. Instead, lines such as

\[
\begin{align*}
\text{RA 1 2 1K} \\
\text{RB 2 3 2K}
\end{align*}
\]

must be used.

In the semantic functions these names are generated by numbering the applications, and using the number as the name. This will distinguish all applications from each other, not only applications of the same atom.
Chapter 8

Non-standard interpretation of Glass

Although Glass was designed to support multiple interpretations, the interpretation described in this chapter lies outside the intended application area: in this chapter Glass descriptions are interpreted as geometric descriptions. This is not only a curiosity; this semantic function has been implemented in the program GLdraw.

In this interpretation, Glass connection variables of type Coord are interpreted as coordinates in a two-dimensional space. All coordinates have the same scale factor, where the ‘normal’ distance between two text lines is defined as ‘1’ in the coordinate system. Angles are given in degrees.

There are two distinct classes of atoms: atoms for figure elements, and atoms for position specification. The latter specify the relation between the coordinates of the various variables. The semantic function determines the relative position of all variables in the given definition using the position specifications. After this, the figure specifications are used to generate the drawing.

Using a textual language to describe pictures is not a new idea: it has been implemented in such languages as Metafont\(^1\) [27], Juno [32], and Ideal [60]. All of these languages use sets of equations, called constraints, that describe the relations between various coordinates in the picture. An equation solver is then used to determine the actual value of each of the coordinates.

The semantic function has been implemented as a program that converts the Glass description to a list of geometric elements. This list of geometric elements is then converted to various formats by separate programs that are called format converters.

\(^1\)Apart from the fonts used in the figures, the fonts used in this thesis have been defined using Metafont.
8.1 Figure specifications

All figure specifications have an attribute string \( \alpha \) as the first value parameter. The meaning of this string is determined by the various format converters, and it is suggested that the user can configure this translation. All format converters should at least understand the specification 'normal' that gives a reasonable default. Ideally, format converters should have a (user specified) translation table to provide a flexible choice of representations on various devices.

\[
\begin{align*}
\text{arc} \alpha \phi (c,r) & \quad \text{An arc with centre } c \text{ starting from point } r \text{ and ending at a point with angle } \phi \text{ from } r. \\
\text{line} \alpha (b,e) & \quad \text{A line between point } b \text{ and } e. \\
\text{triangle} \alpha (a,b,c) & \quad \text{A triangle between points } a, b \text{ and } c. \\
\text{rect} \alpha (a,b) & \quad \text{A rectangle with horizontal and vertical lines between points } a \text{ and } b. \\
\text{text} \alpha t (a,b) & \quad \text{Text } t \text{ at position } a. \text{ The line between point } a \text{ and } b \text{ is part of the base line of the text. Actual drivers may round the angle implied by this baseline to the nearest convenient angle.}
\end{align*}
\]

The atom declarations are as follows:

\[
\begin{align*}
\text{atom} \ arc & \in \text{string} \rightarrow \text{float} \rightarrow [\text{Coord} \times \text{Coord}]; \\
\text{atom} \ line & \in \text{string} \rightarrow [\text{Coord} \times \text{Coord}]; \\
\text{atom} \ rect & \in \text{string} \rightarrow [\text{Coord} \times \text{Coord}]; \\
\text{atom} \ triangle & \in \text{string} \rightarrow [\text{Coord} \times \text{Coord} \times \text{Coord}]; \\
\text{atom} \ text & \in \text{string} \rightarrow \text{string} \rightarrow [\text{Coord} \times \text{Coord}];
\end{align*}
\]

8.2 Position specification atoms

Gldraw determines the relative position of all variables in the given definition using the position specifications. The choice of position specification atoms is rather arbitrary; it is certainly possible to define a better (more powerful) set.
dir d (a,b,c)  Coordinate c lies on the line from a to b, and has a distance d between a and c.
trans tx ty (a,b)  Coordinate b is (tx, ty) from coordinate a. Can also be used to determine position of a from the position of b.
polar φ r (a,b)  Coordinate b has angle φ and distance r from coordinate a. Can also be used to determine position of a from the position of b.
pcljs φ (a,b,c)  Given a point b on a circle with centre a, point c lies on the same circle, but is rotated over an angle φ. Can also be used to determine the position of b from a and c.
between f (a,b,c)  Point c lies on the line from a to b at f times the distance between a and b. Can also be used if one of the others is unknown.
xyproj (a,b,c)  Point c has x coordinate of a and y coordinate of b. This can be seen as a projection of b on a vertical line through a, or a projection of a on a horizontal line through b.

The atom declarations are as follows:

atom trans ∈ float → float → [Coord×Coord];
atom xyproj ∈ [?Coord×?Coord×!Coord];
atom polar ∈ float → float → [Coord×Coord];
atom between ∈ float → [Coord×Coord×Coord];

8.3 An example of a drawing description

As an example, consider the following Glass text:

basetype Coord;

atom trans ∈ float → float → [Coord×Coord];
atom line ∈ string → [Coord×Coord];

def atriangle ∈ [Coord×Coord];  
atriangle (b1,b2) =
{  
  trans 10 0 (b1,b2),
  trans 5 5 (b1,top),
  line "normal" (b1,b2),
  line "normal" (b1,top),
  line "normal" (b2,top)
};

This Glass text describes the following figure:
Chapter 9

Applications of Glass

A general system description language such as Glass has to compete with the special-purpose description languages that have evolved in several disciplines. For example, in analog circuit design the most popular description languages are circuit diagrams and input languages for popular circuit simulators such as Spice. The replacement of these languages with Glass will only be useful if there are some clear advantages over existing languages. The most important advantages of Glass are that it permits multiple interpretations and that it is a rigorously defined hierarchical language. Also, the fact that it permits multiple interpretations implies that it could replace several special-purpose description languages, thereby introducing a standard.

In this chapter, some applications and potential applications are described that use the advantages of Glass. The amplifier design program Ampdes uses Glass for its hierarchy and rigorous definition. This is described in section 9.1. In section 9.2, another application is described: the description of measurements. In section 9.3, I will speculate about possible improvements of Glass to increase its range of applications.

9.1 Ampdes

As explained in section 3.5, Ampdes is a program for the design of high-performance amplifiers. The design of a circuit, in this case an amplifier, involves the following steps:

1. Specification. The required behaviour of the circuit must be specified, and its environment (source and load) must also be specified.

2. Design. Given the specification, it is possible to search for a circuit that meets the specification (a feasible solution).
3. **Verification.** For the search for a feasible solution, mathematical models for the behaviour are used. Usually these models are simple enough to enable rapid search. Therefore, approximations have been introduced that must be verified.

For Ampdes, specification and design will be described in the following sections. The design space is described separately. Verification is done using standard programs such as Spice and Anp. Also, the structure of the program and the way Glass is used is described.

### 9.1.1  specification

Ampdes is a program to design high-performance amplifiers. An amplifier can be seen as a circuit to convert the output signal of a certain source to the input signal for a certain load without loss of information. To be able to do this, it is necessary to characterize the source and load of the amplifier. For the characterization of the source, the following information is necessary:

- The physical quantity that represents the signal. For example, in a photodiode (a light sensor) the amount of light shining on the photodiode is most accurately reflected by the current that flows through it. Thus, a photodiode is best described as a **current** source. Other sources may have other signal quantities, for example charge, or the product of current and voltage (power). For the sake of simplicity, I will assume in this chapter that sources are either current sources or voltage sources.

- The spectrum and amplitude of the source signal.

- The noise properties of the source signal.

A similar characterization is necessary for the load.

An ideal amplifier should make an accurate conversion of the input signal to the output without loss of information. Practical amplifiers will never meet this objective because of the component imperfections mentioned in section 3.3. It is, however, possible to exchange one sort of imperfection for another. Therefore, constraints on each of the imperfections must be given. Also, the designer will want to impose constraints on other behavioural aspects such as power dissipation, chip area and cost.

### 9.1.2  the search space

Because Ampdes must be able to design high-performance amplifiers, it is not possible to use predesigned configurations only, as is done in many other design programs. Instead, an amplifier configuration as shown in figure 9.1 is assumed: an amplifier
Figure 9.1: General amplifier configuration.

Figure 9.2: possible amplifier stage configurations (shown for bipolar npn transistor): (a) a CB stage, (b) a CE stage, (c) a CC stage.

Figure 9.3: variations on stage configuration (shown for CE stage): (a) single, (b) balanced, (c) mirrored.
consists of an arbitrary number of stages, and has overall negative feedback\(^1\). For each stage the following choices must be made:

- Technology of the active devices: for example bipolar NPN transistor or P-channel FET.
- Component type of the active devices: for example ‘BC549’.
- Stage configuration: common emitter, common base or common collector for BJT, and similar configurations for FET, see figure 9.2.
- Stage configuration variation: single, balanced or mirrored, see figure 9.3.

The bias circuit, which determines the operating point of the circuit, is shared by the stages.

To compensate for the behaviour of parasitic components, special compensation components are added.

All in all the search space is very large. Therefore, an elaborate search strategy is necessary.

**9.1.3 search strategy**

By approximation, it is assumed that each of the design objectives can be met separately by the proper design of a part of the amplifier. This is a projection in the sense of section 3.5. The design objectives are partitioned as follows:

- The nature of the source and load quantity determine the nature of the feedback configuration, see figure 9.4.
- The desired transfer function amplitude and spectrum is determined by the feedback network.
- Inaccuracy in transfer amplitude is caused by the finite gain and bandwidth of the active part. Since the gain per stage is finite, the number of stages must be increased to improve the accuracy of the transfer function. However, increasing the number of stages will make it more difficult to keep the amplifier stable.
- Noise is caused by the first amplifier stage. Thus, it is assumed that after the first stage the signal is sufficiently amplified.
- Bandwidth limitations and instabilities are caused by parasitic components. To some extent these limitations can be countered by special compensation components.
- Distortion is caused by the last stage of the active circuit.

\(^1\)This actually is also a presdesigned configuration, but a much more abstract one.
Figure 9.4: amplifier configurations: (a) voltage amplifier, (b) current amplifier, (c) voltage-to-current amplifier, and (d) current-to-voltage amplifier.

As a further complication, not all combinations of stages will produce useful amplifiers. Combinations of stages may be rejected for the following reasons:

- The resulting active part does not invert. Inversion is essential for negative feedback.
- The combination of stages is incompatible. That is, interaction between the stages has the effect that they do not function properly together.
- The resulting circuit cannot be biased.
- The resulting circuit cannot be compensated properly.
- The resulting circuit has an offset that is too large.

9.1.4 the implementation of Ampdes

In Ampdes, the design is divided into many small steps. Each of these steps has a specified objective, and may require other steps to be taken first. Examples of design steps are:

- Construct the fully designed amplifier.
• Establish the configuration of the amplifier source.

• Find values for the compensation components.

The results of these steps are stored in a global ‘pool’ of results. This pool is frequently saved to a file, so that the design can be restarted from that point. The order in which the design steps are executed is not fixed in Ampdes’ code, but is determined by the requirements of each step. Thus, if some of the specifications are changed, only a minimal number of design steps are re-executed.

For each design step the following actions are undertaken:

1. Examine the data that is required for this step. Construct each item that does not exist or is outdated by its own required data.

2. Execute the construction code of the design step.

3. Verify the correctness of the results of the step.

4. Allow the user to verify the results.

5. Allow the user to repeat a design step.

The actual actions are different for each design step. In principle, the user must verify each step, but for trivial steps this may be skipped, depending on the actual user-verification code. Also, the user may specify at the start of the design that he wants to verify only the crucial steps in the design. What a crucial step is, is coded in the program.

Ampdes consists of the following major blocks:

• Design step sequencer. This block selects and executes design steps to prepare sufficient information for further design steps.

• Noise optimizer. Given the transfer function to which the noise sources of the input stage transistor is subjected, this program selects the component with minimal noise contribution, and it finds the optimal bias current for it.

• Frequency optimizer. Given the transfer function of the amplifier containing compensation components, this program will select values for the compensation components so that the best transfer function is attained.

• Amplifier stage generator. This block generates amplifier configurations for the consideration of the evaluation software. Since detailed analysis of amplifier configurations requires considerable computational effort, insofar as this is possible, faster tests are used to reject useless configurations.

• User interface. This block lets the user supply necessary data, inspect and modify data, restart design steps, etc. To give the user insight into, and influence on the design process, Ampdes provides the opportunity to inspect and modify the input data.
<table>
<thead>
<tr>
<th></th>
<th>hand-written</th>
<th>generated by Tm</th>
</tr>
</thead>
<tbody>
<tr>
<td>programs</td>
<td>36000</td>
<td>57000</td>
</tr>
<tr>
<td>libraries</td>
<td>6500</td>
<td>13000</td>
</tr>
<tr>
<td></td>
<td>42500</td>
<td>70000</td>
</tr>
</tbody>
</table>

Table 9.1: origin statistics of Ampdes source code.

In the implementation of Ampdes and the supporting programs, Tm has made a considerable contribution to the speed of implementation and the reliability of the software. This is illustrated by the statistics presented in table 9.1. Thus, of a grand total of 112500 lines of C code, 70000 were generated by Tm. Based on a previous implementation in a different programming language, this code was developed by one person in about four months. The program as it is described here is not complete, since it does not cover the entire design of an amplifier, however, the design steps that have been implemented work as planned.

9.1.5 the use of Glass in Ampdes

In Ampdes, Glass is used for all circuit descriptions. Thus:

- The user must specify the source and load circuit in Glass.
- In some design and verification steps a circuit simulation is done. The circuit to be simulated is described in Glass. This circuit description is converted by a semantic function to input for Spice or Anp.
- The resulting circuit is described in Glass.

The current implementation of Glass has caused some problems in Ampdes:

- Glass text is unsuitable for use in a program since it can only be represented as strings. A representation with more structure would be desirable.
- The internal representation for semantic functions that is described in chapter 6 is difficult to understand and is too general since too many expressions are possible that are not valid Glass expressions. Also, it only covers the kernel language, not the macro language.

For this reason, Ampdes uses internally a different, more limited, representation of the circuits. Ampdes also contains some semantic functions on this representation, one of which generates Glass text.
9.2 Measurement description

The recent addition of computational capacity to measurement equipment, and the advent of a standard for the interconnection of measurement equipment (the IEEE 488 standard), has made a more formal description of measurement circuits and procedures desirable. Some attempts in this direction have been made, for example TMSL [22]. Their main shortcoming is that they describe a measurement circuit with specific equipment. Also, they tend to be oriented to one specific discipline.

A language to describe arbitrary measurements in a more abstract way would be more desirable. For example, one should be able to use a generic voltage metre in the description, not a specific type of voltage metre.

Under the supervision of M.G. Middelhoek and myself, Eugene Jansen made an attempt to use Glass as a measurement description language, see [23]. The requirements were:

- Description of the measurement circuit and the measurement procedure.
- Descriptions should be independent of specific equipment.
- Implementation should be easily possible.
- The system should employ Glass where relevant.

As the project has shown, to fulfil these requirements the following additions to Glass would be necessary:

- An additional language to describe the necessary computations and actions to perform the measurement. For example, the order and timing of measurement steps must be described, the reaction to any interesting event that occurs, etc.

- A language to describe the details of measurement equipment. This is a complicated task for the following reasons:

  - The organization of measurement equipment is often complex and insufficiently documented. There is little standardization.

  - The equipment descriptions must contain many details concerning accuracy, range and operating procedures. Again, there is little standardization in this area.

Since the design of either language was beyond the scope of the project, temporary solutions were necessary. The measurement computations are described as systems working on abstract signals, and an appropriate set of atoms is provided for this purpose. Thus, to describe the sum of two signals, an adder atom could be introduced:
basetype S;  /* The 'signal' connection type. */

/* An adder of two signals */
atom sigadd ∈ S×S ⇒ S

Although it is possible to describe the computations and operations in this way, it is very cumbersome. Additions to Glass for this purpose would be welcome.

For the description of equipment properties, no satisfactory solution was found.

Provided that suitable description formats are developed to describe the equipment and the signal operations (and this should pose no problems) it is possible to write a program that implements the abstract measurement circuit by using the actual equipment described in the equipment data base.

9.3 Shortcomings of Glass and its implementation

The purpose of the Glass language and support software is to enable a user to create specific interpretations of system descriptions. To date, this idea has not been very successful because of the following reasons:

1. For a long time the software was not reliable enough for useful work. For example, Ampdes still uses a Glass parser program for the kernel language. This program was intended as a temporary program to replace the parser, macro expander and type checker until they were completed, but at the moment the program is still necessary.

2. The Tm representation for semantic functions that is described in chapter 6 is difficult to understand and is too general since too many expressions are possible that are not valid Glass expressions.

3. Especially for analog electronics, interesting semantic functions require a considerable amount of numerical computation. As long as no easy-to-use software is available that provides these capabilities, writing semantic functions for analog electronics will remain difficult. Symbolic algebra programs such as Maple [12] are not good enough for this purpose since they cannot solve sets of nonlinear equations. Circuit simulation programs such as Spice are not general enough.

4. As explained in section 9.1, the use of Glass in synthesis programs is hampered by the lack of a good Tm representation of the full language.

Another problem is that many designers prefer a graphical representation of a system above a textual representation such as Glass. Conversion to a currently used graphical representation would mean, however, that the fine-grained hierarchy of a textual representation would be lost. The design of a graphical system that provides the same fine-grained hierarchy as Glass would require considerable research.
Chapter 10

Conclusions

The objective of the Esprit project FORFUN has been to prove that it is possible
to define and implement a system description language that permits multiple inter-
pretations. An advantage of such a language is that it can be used as a standard
description format for a large number of tools. Furthermore, it is inherently neces-
sary that such a system description language be well-defined. This implies that it is
suitable as a target language for circuit synthesis, and that it can be used for formal
manipulation.

As is to be expected for such an experimental language implementation, the current
language can be improved upon on several points, see section 9.3. Nevertheless, the
Esprit project has demonstrated that it is possible to design a system description lan-
guage for multiple semantics. The software that has been developed to demonstrate
this makes it relatively easy to implement semantic functions in a compact way. This
is illustrated in appendix C, where an entire semantic function is listed.

One of the reasons for this success is Tm. This program provides a standard file
interface for structured data to many languages. Thus, it ensures easy implementa-
tion of semantic functions in many programming languages. This has made it easy to
partition the software into separate programs, which is very valuable in a cooperative
project such as the FORFUN project where software had to be developed at a num-
ber of sites. Also, Tm has provided considerable support for the implementation of
Ampdes, a program for the synthesis of high-performance amplifiers. It has allowed
a program of about 125000 lines of code (70000 generated by Tm) to be written by
one person in about four months.

As I explained in section 9.3, it is my opinion that the combination of a system
description language such as Glass, a fast functional or hybrid language and a general
equation solving program would yield a very powerful package. Such a package would
be of interest to many designers in many disciplines.
Appendix A

Glass representations for semantic functions

This appendix lists the two Tm data structure representations that are used by semantic functions. They are explained in chapter 6.

The first uses a hierarchical representation that represents the Glass kernel language:

```
orig == (file:string, line:inum);
```

|| Definitions.
```
def ::=
  DefAtom atorig:orig atnm:symbol atptyp:[partyp] atctyp:typ |
  DefBasetype baseorig:orig basename:symbol |
  DefVal valorig:orig valnm:symbol valtyp:typ valas:val |
  DefCon conorig:orig defcon:val conas:val |
  DefTyp typorig:orig typnm:symbol typas:typ ;
```

|| Possible types of parameters types
```
partyp ::= 
  PToInt | 
  PTFlo | 
  PTStr | 
  PTBool ;
```

|| Type expressions
```
typ ::= 
  TypBase basenm:symbol | || basetype
  TypIn ityp:typ | || ?ityp
  TypOut otyp:typ | || !otyp
```
TypUni uityp:typ uotyp:typ \ uityp => uotyp  \adirectional system
TypNon nontyp:typ \ Cartesian product
TypProd ptypes:[typ] \ symbolic type
TypSym sym:symbol;

Formal connection patterns.
formcon ::=  
FList 1:[formcon] \ tuple
FSym sym:symbol; \ variable

parameters of parameterized atoms
parval ::=  
ParInt i:inum \ int
ParFlo f:fnum \ float
ParStr s:string \ string
ParBool b:bool ; \ bool

Value expressions.
Atoms and apply-s have a source file and line.
val ::=  
VSym symorig:orig sym:symbol \ Symbol
VLambda lpar:formcon lval:val \ Lambda abstraction
VSigma spar:formcon sval:val \ Sigma abstraction
VApply v:val v:val \ Application
VWhere w:val:val[def] v:val:val \ Local context
VList v:val \ List expression
VAppset aps:[val] \ appset
Atom application
VAtom atorig:orig attm:symbol atvpar:[parval] atvpar:val \ Synonym

The other representation is a simple netlist:
File: flat.ds
Simple net-list like representation of a circuit.

flatorig => ( file:string, line:inum );

flat => (  
or:flatorig,
name:symbol,
params:[flatparam],
con:[symbol]
);

flatparam ::=  
FlatParmInt i:inum \ int
FlatParmFlo f:fnum \ float
FlatParmStr s:string \ string
FlatParmBool b:bool ; \ bool
Appendix B

An overview of Glass

This appendix gives a brief overview of Glass. It is not intended as a tutorial on Glass, such a tutorial can be found in [51].

The notation of self-evident parts of the language, such as numbers, strings and variables will not be explained explicitly. Comment can be added to a Glass text by surrounding it with /* */. For example:

/* This is a comment. */

B.1 Definitions

A Glass text consists of a list of definitions. The classes of definitions are:

- **Base type declaration.** Declares a symbol to be a basic connection type of the description. It is not permitted to have basetype declarations in where clauses. For example:

  basetype $E$;

  declares $E$ to be a basic connection type.

- **Atom declaration.** Declares a symbol to be an atomic system with the given connection type. An atomic system is a system that is not defined further in the Glass text. The atom may have parameters, each with a certain parameter type. See the sections B.3 and B.2 for a description of parameter and connection types respectively. It is not permitted to have atom declarations in where clauses. For example:

  atom $R \in float \rightarrow [E \times E]$;
means: "adirectional atomic system $R$ has a float parameter and two connections". And as another example:

atom and $\in E \times E \Rightarrow E$;

This means: "directional atomic system and has no parameters, two inputs, and one output". Finally,

atom $Z \in [(E \times E) \times (E \times E)]$;

means: "adirectional atomic system $Z$ has no parameters, and two groups of two connections".

- **Type equivalence.** Defines a symbol to be equivalent to a given type. For example:

  type port $= E \times E$;

means: "type port is equivalent to $E \times E$". This definition can now be used in any type expression; for example:

  type twoport $= [port \times port]$;

means: type twoport is equivalent to $[(E \times E) \times (E \times E)]$.

- **System definition.** System definitions come in two forms: directional definitions and adirectional definitions. Both directional and adirectional systems have a formal connection pattern (abstractor). A formal connection pattern can be of the form

  name

or of the form

( formcon, ... formcon )

An example of a directional system is:

\[
\text{def} \quad \text{sel} \in E \times E \times E \Rightarrow E; \\
\text{sel} \ (a, b, s) = \text{or} \ (\text{and} \ (a, s), \ \text{and} \ (b, \text{not} \ s));
\]

A circuit diagram of the system is shown in figure B.1. An example of an adirectional system is:

\[
\text{def} \quad \text{source} \in [E \times E]; \\
\text{source} \ (a, b) = \{ \ \text{VS} \ (c, b), \ \text{RS} \ (a, c) \};
\]
Adirectional definitions contain an *appset*, an unordered list of applications. Thus, an adirectional definition can be recognized by the curly brackets (\{\}) that surround its appset.

- *Connection equivalence definition.* A connection equivalence definition may only occur within *where* clauses, where they define that a connection pattern is equivalent to a connection expression. See the description of a system definition for the description of formal connection patterns. For example:

\[
\text{atom } \text{Dff} \in E \times E \Rightarrow E \times E;
\]
\[
\text{def } \text{div2} \in E \Rightarrow E;
\]
\[
\text{div2} \ \text{clk} = q
\]
\[
\text{where}
\]
\[
(q, nq) = \text{Dff} (nq, \text{clk});
\]
\[
\text{endwhere};
\]

A circuit diagram of the system is shown in figure B.2.

- *Macro definition.* Macro definitions resemble system definitions, but are expanded before any semantic function is applied. Also, macros may have parameters. For example:
mac div ∈ int → E ⇒ E;
    div 2 i = div 2 i;
    div n i = div 2 (div (n/2) i);

defines a macro to generate dividers for any power of 2.

As is shown by this definition, a macro definition may consist of a number of
definitions for specific values of its parameters or specific connection expressions.
Only if the actual parameters match these values does the given definition part
apply. These definition parts are searched from top to bottom.

As an example of an adirectional macro definition, the macro

    type port = E×E;
    type twoport = port×port;

    /* The cascade of two twoports. */
    mac casc ∈ [twoport] → [twoport] → [twoport];
    casc sa sb (a,b) = { sa (a,c), sb (c,b) };

defines the cascade interconnection of two twoports, where a twoport is a system
with two input connections and two output connections\footnote{In network theory, a stricter definition of a twoport is used: each port must meet the port
condition. This means that the currents through the two connections are of equal magnitude
and opposite direction.}.

B.2 Connection types

The type of a single connection is specified by base types. Base types are declared
using the basetype declaration. The type of a connection list is the Cartesian
product of the types of the elements in the connection (see section 2.2.4). Thus,
assuming that a and b are of type E, the list \langle a,b \rangle is of the type E×E. It may be
necessary to use parentheses () to indicate what is meant. For example, (E×E)×E is
not equivalent to E×E×E, since the first is the type of connection expressions such
as \langle \langle a, b \rangle, c \rangle, and the second is the type of connection expressions such as \langle a, b, c \rangle.

For systems, it must be indicated whether the system is directional or adirectional.
A directional system has a type of the form
\textit{intype} \Rightarrow \textit{outtype}

where \textit{intype} and \textit{outtype} are the type of the input and output connections respectively. An adirectional system has a type of the form

\[ \text{[contype]} \]

where \textit{contype} is the connection type.

Within adirectional systems some connections may have a direction. This may be indicated by prefixing the type of that connection with $? \ or \ !$ for input and output respectively. For example:

\[ ?(E \times E) \times !((E \times E) \times (E \times E)) \]

is the type of an adirectional system with an input of the form (a, b), and an output of the form ((a, b), (c, d)).

At each place where a \textit{basetype} can be used, one may also use a type declared with a \textit{type} declaration. Thus, with

\texttt{type port = E \times E;}

\texttt{type}

\[ [port \times port]\]

is equivalent to

\[ [(E \times E) \times (E \times E)] \]

\section{B.3 Parameter types}

Each parameter of an atom or macro must have a type. The following two types are possible:

\begin{tabular}{ll}
\texttt{int} & An integer. \\
\texttt{float} & A floating point number. \\
\texttt{string} & A string. \\
\texttt{bool} & A Boolean. \\
\end{tabular}

Atom and macro types are constructed from value and connection types by using the function type operator, $\rightarrow$. For example, the following types could be constructed:

\begin{tabular}{ll}
\texttt{int} & \texttt{int} \\
\texttt{float} & \texttt{float} & \texttt{E} & \Rightarrow & \texttt{E} \\
\end{tabular}
The first example is the type of a function with an integer parameter and an integer result. The second example is the type of a function with two float parameters resulting in a directional system with one input and one output. One should not confuse the directional system type operator, ⇒, with the function type operator.

### B.4 Values

As indicated in a previous section, system definitions have the form

```
def name ∈ contyp;
    name formcon = val;
```

For **directional** systems, *val* is a connection expression. Such a connection expression can be of the following forms:

- `name`
- `<connection, connection ... connection>`
- `systemname connection`
- `atomname parameter ... parameter connection`
- `macroname parameter ... parameter connection`

Examples of connection expressions are:

```
a
and ⟨b, not c⟩
⟨⟨d, e, f⟩, ⟨⟨g, h⟩, i⟩⟩
⟨a, not b⟩
```

Thus, a valid directional definition is:

```
def sel ∈ E×E×E ⇒ E;
    sel ⟨a, b, s⟩ = or ⟨and ⟨a, s⟩, and ⟨b, not s⟩⟩;
```

This definition consists of an application of *or*, and it contains nested applications of *and* and *not*.

When atoms or macros are applied, the application may have parameters. For example, an atom that is declared as follows:

```
atom A ∈ int → string → bool → [E×E];
```

might be applied as follows:

```
A 3 "bar" true ⟨a, b⟩;
```

For **directional** systems, *val* consists of an *appset*, a list of system applications surrounded by {} and separated by commas. For example:
\textbf{def} \quad \textit{source} \in [E \times E];
\textit{source} \langle a, b \rangle = \{ \textit{VS} \langle c, b \rangle, \textit{RS} \langle a, c \rangle \};

The appset contains applications of \textit{VS} and \textit{RS}.

In appsets, it is not permitted to have applications in connection expressions. Therefore, appsets such as

\{ \textit{not} \, \langle \textit{not} \, a, b \rangle \}  

are not permitted. An appset may also contain other appsets and \textit{synonym} expressions. A synonym expression has the form

*( connection, connection \ldots connection *)

It indicates that each of the connection expressions within the synonym expression are synonymous. For example:

*(a, \langle b, c \rangle);*

states that \( a \) and \( \langle b, c \rangle \) are synonymous. Thus, at all places where \( a \) is used, it is also permitted to use \( \langle b, c \rangle \) and vice versa.

Both directional and adirectional definitions may contain 'where' clauses of the form

\begin{verbatim}
val where definitions endwhere
\end{verbatim}

See section B.1 for the possible contents of \textit{definitions} and an example.

\section{B.5 Directional systems as a subclass of adirectional systems}

To permit the use of directionally defined systems in adirectional systems and vice-versa, a sub-class of the adirectional systems is considered equivalent to directional systems.

Systems with types of the form

\(< \textit{?intype} \times \textit{!outtype} >\)

are considered equivalent to systems of the type

\textit{intype} \Rightarrow \textit{outtype}

Thus, suppose

\textit{atom and} \in E \times E \Rightarrow E;
Figure B.4: a possible implementation of the oscillator cell.

Now it is possible to use \textit{and} in appsets as a system with type

\[ [(E \times E) \times !E] \]

For example:

\[
\{ \ldots \text{and} \ (\langle \langle a, b \rangle, c \rangle, \ldots ) \}
\]

As another example:

\begin{verbatim}
def a in Ta implies Tb;
a i = x;
\end{verbatim}

is equivalent to

\begin{verbatim}
def a in [(T\textasciitilde a) \times !Tb];
a (i, new) = \{ *(new, x) \};
\end{verbatim}

Where \textit{new} is a name that does not clash with existing names.

\section*{B.6 A larger example}

As a larger example, the coupled oscillators from Verhoeven's thesis [57] are described. As is explained in his thesis, it is possible to construct an entire family of oscillators—called \textit{coupled} oscillators—from copies of a single oscillator cell. By coupling the oscillators, the noise and component tolerances of the individual cells are averaged, and therefore reduced. A circuit diagram of a possible implementation of the oscillator

126
cell is shown in figure B.4. I will not discuss the behaviour properties of these oscillators, but will only show how they could be described in Glass.

In Glass, the cell shown in figure B.4 is best described as an adirectional definition:

```
basetype E;

atom Vcc \in [E];
atom Gnd \in [E];

/& Transistor: \langle collector, base, emitter \rangle /*
atom T \in \text{string} \rightarrow [E \times E \times E];
atom C \in \text{float} \rightarrow [E \times E];
atom I \in \text{float} \rightarrow [E \times E];
atom R \in \text{float} \rightarrow [E \times E];

/* A possible oscillator cell. */
def icell \in [?(E \times E) \times !((E \times E))];
icell \langle \langle \text{in+}, \text{in-}, \rangle, \langle \text{out+}, \text{out-} \rangle \rangle =
{
T "BC549" \langle \text{in-}, \text{in+}, \text{out+} \rangle,
T "BC549" \langle \text{in+}, \text{in-}, \text{out-} \rangle,
C 1n \langle \text{out+}, \text{out-} \rangle,
I 1m \langle \text{out+}, \text{ground} \rangle,
I 1m \langle \text{out-}, \text{ground} \rangle,
R 1k \langle \text{supply}, \text{in-} \rangle,
R 1k \langle \text{supply}, \text{in+} \rangle,
Vcc \text{ supply},
Gnd \text{ ground}
};
```

Some remarks:

- A more elaborate description of the cell could use the symmetry in the cell.
- Globally used nodes such as Vcc and ground must be declared as systems instead of connections. This is a shortcoming of Glass.
Using this cell, a family of oscillators can be constructed. The simplest one uses one copy of the cell, see figure B.5, and consists simply of the cell itself with nothing connected to its inputs. This oscillator can be described in Glass as an adirectional definition:

```plaintext
def osc1 ∈ ![E×E];
    osc1 out = { icell (in,out) };
```

It can also be described as a directional definition:

```plaintext
def osc1 ∈ ⇒ (E×E);
    osc1 () = icell in;
```

This is a somewhat odd directional system since it does not have any inputs, hence the empty tuple as the formal connection expression and the empty left-hand side of ⇒ E×E. Also, remember that it is possible to use a certain class of adirectional systems as directional systems, see section B.5. This property is used in this application of icell.

It is also possible to construct more complicated oscillators by using cells in cascade or in parallel. For example, an oscillator with two cells in cascade (see figure B.6) can be described in Glass as an adirectional definition:

```plaintext
def osc2 ∈ ![E×E]×(E×E];
    osc2 (out1,out2) =
        *
        *{(out2, (out2+, out2-))},
        icell ((out2-, out2+), out1),
        icell (out1, out2)
    }
```

And again, it can also be described as a directional definition:
Figure B.7: an oscillator from an arbitrary number of cells.

/* Exchange two connections. */
def twist \in E \times E \Rightarrow E \times E;
twist \langle a,b \rangle = \langle b,a \rangle;

def osc2 \in \Rightarrow (E \times E) \times (E \times E);
osc2 (\cdot) = \langle \text{out1, out2} \rangle
where
\text{out1 = twist (icell out2);}
\text{out2 = icell out1;}
endwhere;

This construction scheme can be extended to employ an arbitrary number of integrator cells, but this is not very useful.

Oscillator cells can also be used in parallel, and in this case any number of cells can be used, see figure B.7. In Glass, this is best described as a macro definition. This macro has a numerical parameter that indicates the number of integrator cells to be used. Again, it is possible to use an adirectional macro definition:

mac oscn \in \text{int} \rightarrow [!/(E \times E)];
oscn n out = \{ \text{nicells n (in, out)} \}
where
mac nicells \in \text{int} \rightarrow [?(E \times E) \times !/(E \times E)];
nicells 1 c = \{ \text{icell c} \};
nicells n c = \{ \text{icell c, nicells (n-1) c} \};
endwhere;

or a directional macro definition:

129
/* Connect two systems in parallel. */
mac par ∈ ![E×E] → ![E×E] → ![E×E];
    par sysa sysb c = { sysa c, sysb c };

mac oscn ∈ int → (E×E);
    oscn n () = nicells n in
    where
    mac nicells ∈ int → E×E ⇒ E×E;
        nicells 1 in = icell in;
        nicells n in = par icell (nicells (n-1)) in;
    endwhere;
Appendix C

Signal-flow analysis using Miranda

This appendix lists the files of an example semantic function: signal-flow analysis using Miranda. It uses a Glass description, and generates a Miranda script to simulate the signal-flow system.

To understand the working of this semantic function, it is necessary to have some knowledge of Miranda. I will not give an introduction in this thesis, the interested reader is referred to [37], [55] or [56].

C.1 Tm interface code

The interfacing functions to the Glass internal representation are generated by Tm, only a small header file is necessary to invoke the appropriate Tm template:

```
.. File: glassds.mt
|| File: glassds.m

.include $(libpath)$<pathsep>mira.mt

string == [char]
symbol == [char]
inum == num
fnum == num
```

This will result in the following script:

```
|| File: glassds.m
```
orig == (string, inum);

def ::=  
DefAtom  orig symbol [partyp] typ |  
DefBasetype orig symbol |  
DefVal   orig symbol typ val |  
DefCon   orig val val |  
DefTyp   orig symbol typ;

partyp ::=  
PTInt    |  
PTFlo    |  
PTStr    |  
PTBool   ;

typ ::=  
TypBase  symbol |  
TypIn    typ |  
TypOut   typ |  
TypUni   typ typ |  
TypNon   typ |  
TypProd  [typ] |  
TypSym   symbol;

formcon ::=  
FCList   [formcon] |  
FCSym    symbol;

parval ::=  
ParInt   inum |  
ParFlo   fnum |  
ParStr   string |  
ParBool  bool;

val ::=  
VSym     orig symbol |  
VLambda  formcon val |  
VSigma   formcon val |  
VApply   val val |  
VWhere   [def] val |
C.2 The compiler

The compiler itself consists of one file:
|| File: df.m
|| Compile a data flow system to a Miranda script to
|| simulate it.

string == [char]

symbol == [char]
inum == num
fnum == num

|| tm code to read glass data structures
%include "glassds.m"

|| Separate strings in list from each other by a comma.
sepcomma :: [string]->string
sepcomma [] = ""
sepcomma [a] = a
sepcomma (a:b) = a++", "++(sepcomma b)

|| Convert a Glass def name to a Miranda name.
comp_symbol :: symbol->string
comp_symbol s = "g_"++s

|| Construct a tuple from the given list of strings.
mk_tuple :: [string]->string
mk_tuple l = "(" ++ (sepcomma l) ++ ")"

|| Compile a formal connection expression.
comp_formcon :: formcon->string
comp_formcon (FCSym s) = comp_symbol s
comp_formcon (PCList l) = mk_tuple (map comp_formcon l)

|| Compile a parval.
comp_parval :: parval->string
comp_parval (ParInt i) = (show i)++"":
comp_parval (ParFlo f) = (show f)++":
comp_parval (ParStr s) = (show s)++":
comp_parval (ParBool b) = (show b)++":

|| Compile a type expression.
comp_typ :: typ->string
comp_typ (TypBase nm) = comp_symbol nm
comp_typ (TypUni i o) = (comp_typ i)++"->"++(comp_typ o)
comp_typ (TypProd l) = mk_tuple (map comp_typ l)
comp_typ (TypSym nm) = comp_symbol nm

|| Compile a 'val' expression.
comp_val :: num->val->string
comp_val i (VSym or nm) = comp_symbol nm
comp_val i (VLambda fc v)
  = dummynm ++ "\n" ++ whereline ++ dummyabstr ++ dummyval
    where
      dummynm = "d"++(show i)
      ind = rep i :,
      whereline = ind++"where\n"
      fcstr = comp_formcon fc
      dummyabstr = ind++dummynm++"++fcstr++\n"
      dummyval = ind++"="++(comp_val (i+2) v)

comp_val i (VApply v p)
  = abstr ++ " " ++ abpar
    where
      abstr = "("++(comp_val i v)++")"
      abpar = ")("++(comp_val i p)++")"

comp_val i (VList v1) = mk_tuple (map (comp_val i) v1)
comp_val i (VWhere dl v)
  = vv ++ "\n" ++ whereline ++ wheredefs
    where
      vv = comp_val i v
      ind = rep i :,
      whereline = ind++"where\n"
      wheredefs = concat (map (comp_def i) dl)

comp_val i (VAtom or nm vp cp)
    = "("++(comp_symbol nm) ++ " " ++ vps ++ cps++)")"
    where
    vps = concat (map comp_parval vp)
    cps = comp_val i cp

|| Compile a 'val' definition.
comp_valdef :: num->symbol->typ->val->string
comp_valdef i nm t v
    = typespec ++ "\n" ++ bhead ++ "\n" ++ body ++ "\n"
    where
    ind = rep i ' '
    typespec = ind ++ (comp_symbol nm) ++ "::" ++ (comp_typ t)
    bhead = ind ++ (comp_symbol nm)
    body = ind ++ "=" ++ (comp_val (i+2) v)

|| Compile a 'defcon' definition.
compdefcon :: num->val->val->string
compdefcon i c v
    = indstr ++ cs ++ " = " ++ vs ++ "\n"
    where
    indstr = rep i ' '
    cs = (comp_val 0 c) || should need no indent
    vs = (comp_val 0 v) || should need no indent

|| Compile a 'typ' definition.
comp_typdef :: num->symbol->typ->string
comp_typdef i nm t
    = indstr ++ (comp_symbol nm) ++ " = " ++ comp_typ t ++ "\n"
    where
    indstr = rep i ' '

|| Compile a definition.
|| Each def type is handled by a separate function.
comp_def :: num->def->string
comp_def i (DefAtom or nm pt ct) = ""
comp_def i (DefBasetype or nm) = ""
comp_def i (DefVal or nm t v) = comp_valdef i nm t v
comp_def i (DefCon or c v) = compdefcon i c v
comp_def i (DefTyp or nm t) = comp_typdef i nm t

|| Given a list of Glass definitions, compile the Glass
|| descriptions in the input file to a Miranda out file.
comp_defs :: [def]->string
comp_defs dl
    = header ++ defs
    where
header = "%include "dfatoms.m"
defs = concat (map (comp_def 0) dl)

|| Shortcut definition for demo
demo :: string
demo = comp_defs (%insert "demo.ig")

This script will generate a Miranda script to simulate the signal-flow system. Some parts of this code require clarification.

1. For the sake of brevity, error checking is minimal.

2. To avoid confusion with existing Miranda definitions, all system and type names in the generated Miranda script are preceded by "g_".

3. Because of a peculiarity of Miranda, it is necessary to ensure that all ‘where’ clauses have an appropriate indentation. Hence, most functions that generate Miranda code have a parameter ‘i’ that indicates the current indent level.

4. Since Miranda does not support nameless λ-abstractions, a constructor such as

VLambda (FCSYN "a") (VSym ("f", 1) "b")

(representing \(\lambda a.b\)) must be translated to:

v2
where
v2 a = b

C.3 An example of a Glass system

As an example, the compilation of the following system will be shown:
basetype $S$;

atom \texttt{scale} \in \texttt{float} \rightarrow S \Rightarrow S;
atom \texttt{sum} \in S \times S \Rightarrow S;
atom \texttt{sub} \in S \times S \Rightarrow S;
atom \texttt{prod} \in S \times S \Rightarrow S;
atom \texttt{div} \in S \times S \Rightarrow S;
atom \texttt{delay} \in \texttt{float} \rightarrow S \Rightarrow S;
atom \texttt{const} \in \texttt{float} \rightarrow \Rightarrow S;

/* Summing system. The output is the sum of the input sequence. */
def \texttt{summer} \in S \Rightarrow S;
\texttt{summer} \ i = o
\texttt{where}
\hspace{1cm} o = \texttt{sum} \ (i, \texttt{delay} \ o);
\texttt{endwhere};

/* The average of the input sequence. */
def \texttt{average} \in S \Rightarrow S;
\texttt{average} \ i = \texttt{div} \ ((\texttt{summer} \ i), \texttt{summer} \ (\texttt{const} \ 1 \ () ));

/* The variance of the input sequence. */
def \texttt{variance} \in S \Rightarrow S;
\texttt{variance} \ i = \texttt{average} \ \texttt{dsq}
\texttt{where}
\hspace{1cm} \texttt{dsq} = \texttt{prod} \ (d, d);
\hspace{1cm} d = \texttt{sub} \ (i, \texttt{average} \ i);
\texttt{endwhere};

Figures C.1, C.2 and C.3 show diagrams of the system. Compilation of this system
will result in file \texttt{demo.m}:
\%include "dfatoms.m"
g\_integr::g\_S->g\_S
\texttt{g\_integr}
Figure C.2: a signal-flow system for the average of a signal.

Figure C.3: a signal-flow system for the variance of a signal.

\[
= d2
\quad \text{where}
\quad d2 \ g_i
= g_o
\quad \text{where}
\quad g_o = (g_{\text{sum}} (g_i, (g_{\text{delay}} g_o)))
\]

\[g_{\text{average}}::g_S \rightarrow g_S\]
\[g_{\text{average}} = d2\]
\quad \text{where}
\quad d2 \ g_i
= (g_{\text{div}} ((g_{\text{intgr}}) (g_i), (g_{\text{intgr}}) ((g_{\text{const}} 1 ()))))
\]

\[g_{\text{variance}}::g_S \rightarrow g_S\]
\[g_{\text{variance}} = d2\]
\quad \text{where}
\quad d2 \ g_i
= (g_{\text{average}}) (g_{\text{dsq}})
\quad \text{where}
\quad g_{\text{dsq}} = (g_{\text{prod}} (g_d, g_d))
\quad g_d = (g_{\text{sub}} (g_i, (g_{\text{average}}) (g_i)))
\]

Function definitions for the atoms are also necessary. They are defined in the file dfatoms.m:

```plaintext
|| File: dfatoms.m
```

138
Atom definitions as used in generated Miranda code for
for data-flow semantic function.

A g_S is a (possibly infinite) list of values.

\[ g_S = [\text{num}]; \]

\[
g_{\text{scale}} :: \text{num} \to g_S \to g_S
\]
\[
g_{\text{scale}} n [] = []
\]
\[
g_{\text{scale}} n (a:b) = n*a:(g_{\text{scale}} n b)
\]

\[
g_{\text{sum}} :: (g_S,g_S) \to g_S
\]
\[
g_{\text{sum}} ([],b) = []
\]
\[
g_{\text{sum}} (a,[]) = []
\]
\[
g_{\text{sum}} ((xh:xt),(yh:yt)) = (xh+yh):(g_{\text{sum}} (xt,yt))
\]

\[
g_{\text{sub}} :: (g_S,g_S) \to g_S
\]
\[
g_{\text{sub}} ([],b) = []
\]
\[
g_{\text{sub}} (a,[]) = []
\]
\[
g_{\text{sub}} ((xh:xt),(yh:yt)) = (xh-yh):(g_{\text{sub}} (xt,yt))
\]

\[
g_{\text{div}} :: (g_S,g_S) \to g_S
\]
\[
g_{\text{div}} ([],b) = []
\]
\[
g_{\text{div}} (a,[]) = []
\]
\[
g_{\text{div}} ((xh:xt),(yh:yt)) = (xh/yh):(g_{\text{div}} (xt,yt))
\]

\[
g_{\text{prod}} :: (g_S,g_S) \to g_S
\]
\[
g_{\text{prod}} ([],b) = []
\]
\[
g_{\text{prod}} (a,[]) = []
\]
\[
g_{\text{prod}} ((xh:xt),(yh:yt)) = (xh*yh):(g_{\text{prod}} (xt,yt))
\]

\[
g_{\text{delay}} :: g_S \to g_S
\]
\[
g_{\text{delay}} a = 0:a
\]

\[
g_{\text{const}} :: \text{num} \to () \to g_S
\]
\[
g_{\text{const}} n () = \text{repeat } n
\]

This Miranda program can be used to simulate the behaviour of the system ‘variance’.
For example:

\[
g_{\text{variance}} [1,2,3,1,2,3,1]
\]

will result in:

\[
[0.0,0.5,1.0,0.375,0.38,0.65,0.312244897959]
\]
Appendix D

Summary

This thesis describes the design of a language to describe arbitrary systems, called Glass. An important feature of Glass is that it allows multiple interpretations of the system descriptions. For example, when digital circuits are described in Glass, it is possible to derive a program from this description to simulate the behaviour of a circuit, and for Glass descriptions of suitable analog circuits, it is possible to derive an input file for circuit simulation programs like Spice. Each of these interpretations is called a semantic function. Because of this feature, Glass is said to have multiple semantics.

Having multiple semantics is only useful if it is easy to define new semantic functions on Glass. Therefore, difficult parts of a semantic function, such as parsing and correctness checking, should be provided to the author of semantic functions, so that only the essence of the semantic function must be written. For this reason I have developed a program, called Tm, to generate the necessary code to communicate with these parsing and checking programs. Tm is able to generate code for a number of programming languages, providing more flexibility in the implementation of semantic functions. Tm is a general program; it has also been used to generate code for other programs, including Ampdes, see below.

I have also written a number of example semantic functions that use this support. The example semantic functions were designed to be used in a synthesis program for high-performance amplifiers that is being developed at the Electronics Research Laboratory. This program is called Ampdes. Ampdes itself is also described.
Appendix E

Samenvatting

Dit proefschrift beschrijft een taal om willekeurige systemen te beschrijven, genaamd Glass. Een belangrijke eigenschap van Glass is dat het meerdere interpretaties van de systeem beschrijvingen toestaat. Bij voorbeeld, wanneer digitale circuits beschreven worden in Glass, is het mogelijk van deze beschrijving een programma af te leiden om het gedrag van een circuit te simuleren, en voor Glass beschrijvingen van geschikte analoge circuits is het mogelijk een invoer file af te leiden voor een circuit simulatie programma zoals Spice. Elk van deze interpretaties wordt een semantische functie genoemd. Men kan zeggen dat Glass meerdere semantieken heeft.

Het bezit van meerdere semantieken is alleen nuttig als het eenvoudig is nieuwe semantische functies op Glass te definiëren. Daarom moeten moeilijke delen van de semantische functie, zoals ontleiding en controle van de correctheid van de beschrijving, ter beschikking worden gesteld aan de auteur van semantische functies, zodat alleen het essentiële deel van de semantische functie geschreven moet worden. Daarom heb ik een programma gemaakt, Tm genaamd, om de code te genereren die noodzakelijk is om te communiceren met de ontledings en controle programma's. Tm is in staat code te genereren voor een aantal programmeertalen, en levert daarmee meer flexibiliteit in de implementatie van de semantische functies. Tm is een algemeen bruikbaar programma: het is ook gebruikt om code te genereren voor andere programma's, inclusief Ampdes.

Ik heb als voorbeeld ook een aantal semantische functies geschreven, gebruik makend van deze ondersteuning. Deze voorbeeld programma's zijn ontworpen om gebruikt te worden in een synthese programma voor hoogwaardige versterkers dat ontwikkeld wordt op bij de Vakgroep Elektronica. Dit programma heet Ampdes. Ampdes zelf wordt ook beschreven.
Appendix F

Biography

Cornelis (Kees) van Reeuwijk was born in Rotterdam on 3 February 1962. When he was eight, he decided that the price that Fisher Technik was asking for its logic building blocks was much too high, and that with some work on his own he could do much better. Needless to say, the logic building blocks were never built. Another of his important decisions was to turn to computers (a NASCOM computer kit) instead of electronics, because then you did not have to do all the tedious things associated with electronics, such as making printed circuit boards and buying components. Nevertheless, as long as you didn’t have to build anything, electronics was still interesting and, therefore, in 1980 he decided to study at the Faculty of Electrical Engineering instead of the Faculty of Mathematics at the Delft University of Technology. After his graduation in 1986 at the Department of Computer Architecture, decided to continue his studies at the Department of Electronics, since it offered the same interesting mixture of analog electronics and computers. Besides, it was much more fun than being drafted into the army.
Bibliography


## Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ-calculus</td>
<td>26</td>
</tr>
<tr>
<td>σ-calculus</td>
<td>17, 28</td>
</tr>
<tr>
<td>abstracted</td>
<td>25</td>
</tr>
<tr>
<td>abstracted variable</td>
<td>26</td>
</tr>
<tr>
<td>abstraction</td>
<td>26</td>
</tr>
<tr>
<td>abstractor</td>
<td>26, 28</td>
</tr>
<tr>
<td>abstrahend</td>
<td>26</td>
</tr>
<tr>
<td>add</td>
<td>92</td>
</tr>
<tr>
<td>adirectional definitions</td>
<td>65, 120</td>
</tr>
<tr>
<td>algebraic</td>
<td>73</td>
</tr>
<tr>
<td>algorithms</td>
<td>9</td>
</tr>
<tr>
<td>All quantor</td>
<td>18</td>
</tr>
<tr>
<td>alphabet</td>
<td>21</td>
</tr>
<tr>
<td>Ampdes</td>
<td>14, 53, 105</td>
</tr>
<tr>
<td>Anp</td>
<td>48</td>
</tr>
<tr>
<td>appset</td>
<td>28, 65, 121, 124</td>
</tr>
<tr>
<td>architecture</td>
<td>53</td>
</tr>
<tr>
<td>array</td>
<td>20</td>
</tr>
<tr>
<td>Associativity</td>
<td>24</td>
</tr>
<tr>
<td>Atom declaration</td>
<td>119</td>
</tr>
<tr>
<td>atom declaration</td>
<td>67</td>
</tr>
<tr>
<td>atoms</td>
<td>67</td>
</tr>
<tr>
<td>Backtracking search</td>
<td>52</td>
</tr>
<tr>
<td>base</td>
<td>36</td>
</tr>
<tr>
<td>base set</td>
<td>21</td>
</tr>
<tr>
<td>Base type declaration</td>
<td>119</td>
</tr>
<tr>
<td>behaviour</td>
<td>32</td>
</tr>
<tr>
<td>bias</td>
<td>40</td>
</tr>
<tr>
<td>bipolar transistor</td>
<td>36</td>
</tr>
<tr>
<td>bound</td>
<td>25</td>
</tr>
<tr>
<td>bound occurrences</td>
<td>26</td>
</tr>
<tr>
<td>calculus</td>
<td>17</td>
</tr>
<tr>
<td>capacitor</td>
<td>33</td>
</tr>
<tr>
<td>Cartesian product</td>
<td>20</td>
</tr>
<tr>
<td>circuit analysis</td>
<td>39</td>
</tr>
<tr>
<td>circuit simulation</td>
<td>39</td>
</tr>
<tr>
<td>closed</td>
<td>26</td>
</tr>
<tr>
<td>codomain</td>
<td>19</td>
</tr>
<tr>
<td>collector</td>
<td>36</td>
</tr>
<tr>
<td>compensation</td>
<td>49</td>
</tr>
<tr>
<td>components</td>
<td>31</td>
</tr>
<tr>
<td>Configuration dimensions</td>
<td>53</td>
</tr>
<tr>
<td>connection</td>
<td>31</td>
</tr>
<tr>
<td>Connection equivalence definition</td>
<td>121</td>
</tr>
<tr>
<td>connection-pattern definition</td>
<td>64</td>
</tr>
<tr>
<td>const</td>
<td>92</td>
</tr>
<tr>
<td>constraints</td>
<td>49</td>
</tr>
<tr>
<td>constructors</td>
<td>73</td>
</tr>
<tr>
<td>Controlled sources</td>
<td>35</td>
</tr>
<tr>
<td>coupled</td>
<td>126</td>
</tr>
<tr>
<td>current amplification factor</td>
<td>37</td>
</tr>
<tr>
<td>current source</td>
<td>34</td>
</tr>
<tr>
<td>currying</td>
<td>21</td>
</tr>
<tr>
<td>delay</td>
<td>92</td>
</tr>
<tr>
<td>diode</td>
<td>33</td>
</tr>
<tr>
<td>directional definitions</td>
<td>65, 120</td>
</tr>
<tr>
<td>distortion</td>
<td>38</td>
</tr>
<tr>
<td>div</td>
<td>92</td>
</tr>
<tr>
<td>domain</td>
<td>19</td>
</tr>
<tr>
<td>electronic circuit</td>
<td>31</td>
</tr>
<tr>
<td>emitter</td>
<td>36</td>
</tr>
<tr>
<td>empty list</td>
<td>21</td>
</tr>
<tr>
<td>exhaustive search</td>
<td>51</td>
</tr>
<tr>
<td>Existence quantor</td>
<td>18</td>
</tr>
<tr>
<td>feasible</td>
<td>105</td>
</tr>
<tr>
<td>feasible solution</td>
<td>51</td>
</tr>
</tbody>
</table>
feedback, 49
FET, 37
field effect transistor, 37
formal function, 8
formal function theory, 8, 18
format converters, 101
Fourier transform, 20
FP, 11
free, 25, 26
full Glass, 70
FUN, 11
function, 18
functional, 9

Glass, 13
grammar, 21
gyrator, 35

heuristic, 52
higher-order function, 20

IDA, 43
image, 18
imperative, 8, 9
indefinite admittance, 43
inductor, 33
Infix, 24
interpretation, 9

KCL, 32
kernel, 71
kernel Glass, 70
Kirchoff’s current law, 32
Kirchoff’s voltage law, 33
Kleene star, 22
KVL, 33

language, 21
Laplace transform, 41
Laplace variable, 41
left associative, 24
length, 20
lexical analysis, 23
line commands, 75
linear, 40

linear components, 33
list, 21
load, 34

Macro definition, 121
macro language, 69
meaning, 9
meaning function, 10
Modified Nodal Analysis, 47
Monte Carlo, 52
muFP, 11

NA, 43, 47
nesting, 63
nodal admittance, 43, 47
node, 31
noise, 37
non-terminal symbols, 22
non-terminals, 22
Norton-Thevenin equivalence, 34
nnp, 36
nullor, 35

operating point, 40
operational, 9
Operator priority, 24
operators, 24

parser generator, 23
parsing, 22
partial function, 19
pattern matching, 70
pnp, 36
poles, 42
port, 34
port condition, 122
Postfix, 24
Prefix, 24
primitive type, 73
prod, 92
projection, 52
proper, 29
Proper substitution, 26

range, 19
resistor, 33
right associative, 24

scale, 92
search set, 50
semantic function, 10
semantic functions, 11
sensor, 39
signals, 112
simulated annealing, 52
Spice, 48
stamps, 44
streams, 91
strings, 21
sub, 92
superposition theorem, 40
syntax, 22
system, 7
System definition, 120

template manager, 72
templates, 74
terminal symbols, 22
terminals, 22, 31
terms, 22
Tm, 71, 72
tokens, 23
transfer function, 42
transformational reasoning, 56
transformer, 35, 36
tuple, 20, 73
twoport, 35
type check, 19
Type equivalence, 120
typelist, 77

undefined, 19

Value dimensions, 53
voltage source, 34
voltage-controlled voltage source, 35
zeros, 42