Glass:
A system description language and its environment

Introduction and User manuals

M. Seutter (ed.)

Catholic University of Nijmegen

January 20, 1992

Preface

The ESPRIT project #881 – FORFUN (FORmal description of systems by means of FUNctional languages) – is a feasibility study concerning the application possibilities and the implementation of system description languages with multiple interpretations (structural, behavioural, performance, etc.) in the area of analog and digital electronic systems and circuits.

The main result of this project is a prototype system description language for analog and digital circuits called Glass (*G*eneral *La*nguage for *s*ystem *s*emantics), in conjunction with a prototype user environment Glue (*Gl*ass *u*ser *e*nvironment) supporting the description language and the definition of semantic functions expressing the various interpretations of the language.

This book is meant to make the public domain release of the Glass language implementation and the Glue environment more readily accessible to people wishing to experiment with various styles and principles of system description. It is addressed at designers and implementors of system description languages and CAD environments as well as designers of electronic systems and circuits, who in fact constitute the end user community for such languages.

It will be clear from the above that the Glass/Glue system is intended for experimentation, not for usage in actual production CAD. The latter was not a goal of the project (which mainly originates from university research interests), nor would such a goal have been realistic because

a) history shows that system description languages can gain acceptance in production CAD only through large-scale standardisation (as is the case e.g. for VHDL);

b) the implementation of a production-quality environment (robust, easily maintainable software) would require a manpower investment at least three times the 20 man-years reserved for this project, using an experienced programming team rather than researchers.

It must also be emphasized that Glass describes (concrete) system and circuit *realizations*. With respect to (abstract) specification, it can be used for *verification* (compliance of the behavioural interpretation of a system description with the abstract specification). In the reverse direction (from specification to realization), the usage of Glass depends on the degree of design automation. In fully automated design (the ultimate role of "silicon compilation"), Glass can assume only the secondary role of internal representation. If the design is not (or can not be) fully automated, Glass can greatly facilitate human intervention by providing easily readable realization description.

Obviously, this line of research does not end with this project and with Glass. Future work will concentrate on a language tentativily called Funmath (*Fun*ctional *math*ematics) which is meant to be a wide-spectrum language for expressing abstract specifications, algorithmic realizations and circuit realizations in a uniform notational framework that is very well-suited towards transformational reasoning and design. The way in which circuit realizations are descibed will depend to a large extent on the experience gained with Glass.

Therefore readers of this book and experimenters with the Glass/Glue system are kindly invited to share their experience with us.

Prof. Dr. Ir. R.T. Boute

Chapter 1

Introduction

R.T. Boute

1.1 The FORFUN view on system description languages

1.1.1 The inadequacy of algorithmic and programming language concepts in view of system description

Most existing system description languages are based on syntactic and semantic principles derived from traditional (imperative) programming languages. This is due to a number of historical factors such as

a) the theory and the design of formal languages was originally developed in the area of programming and hence constituted a readily available basis;

b) the idea of formal system description was mainly restricted to simulation, with little awareness of other important issues such as formal (e.g. transformational) reasoning and symbolic manipulation (by human and by machine).

These factors jointly create a vicious circle, because the syntactic and semantic principles of progamming languages are suitable only for expressing algorithms, and hence their usage in system description prevents the other issues from being addressed effectively.

The inadequacy of the algorithmic paradigm as a basis for system and circuit description can be inferred from the various definitions in the literature where the algorithm concept is carefully analyzed, e.g. [Knu73]. By definition, an algorithm is a set of instructions (or recipe) describing the (serial or parallel) steps in a computation. Hence an algorithm constitutes a (concrete) *realization* by means of a *computational process*. As a consequence, algorithms are not suitable for expressing

- (abstract) *specifications*, i.e. black-box descriptions of the input/output behaviour of programs or circuits without prejudicing particlular realizations;

- (concrete) *realizations* by means that cannot be properly characterized as computational processes (in the sense elaborated rigorously in the theory of computation.)

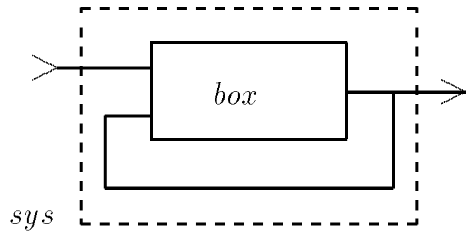
For the fact that electronic circuits and systems indeed belong to the class of systems whose behaviour cannot be properly characterized as computational processes, we present an informal epistomological argument and a formal mathematical example.

Most natural and technical systems are essentially non-algorithmic. Firstly, their *behaviour* is governed by the laws of nature, which are better modelled mathematically by functions and equations than by algorithms. For instance, a falling object does not execute "Newton's gravitional algorithm" but behaves according to the equations of motion in mechanics. A similar observation holds for electronic circuits, both digital and analog. Superficially, synchronous digital circuits appear to be amenable to algorithmic descriptions, but further reflection reveals these to be mere simulations of one particular view of behaviour in which both the time space and the signal space are discretized. Hence this situation constitutes a degenerate particular case. Secondly, the *structure* of systems, being static in nature, is even less well-represented by algorithmic descriptions.

The following example clearly illustrates that circuit behaviour is "equational" rather than "algorithmic". Slightly running ahead of the more carefully designed notation of Glass, consider the following definition:

$$sys\ x = box\ (x,\ sys\ x)$$

In a rather straightforward intuitive *structural* interpretation x is the internal name for the system input wire (the name is externally irrelevant as can be seen by substituting y for x), and the occurrence of sys at the right-hand side suggests feedback. Hence the structural interpretation of the entire definition can be represented schematically as



More interestingly, consider the *behavioural* interpretation in the very simple case where box is a differential amplifier with gain A, that is

$$box\ (x, y) = A \cdot (x - y)$$

Substituting this definition of *box* in the definition of *sys* yields

$$sys\ x = A \cdot (x - sys\ x) \tag{1.1}$$

The *syntax* (textual form) maintains the close correspondence with structure. If interpreted as a (functional) *program* (a *program* is an algorithm expressed in a formal language), definition 1.1 describes a nonterminating computation: "to compute $sys\ x$, subtract the result of computing $sys\ x$ from x and multiply the result of the subtraction by A". The reason for nontermination is that this program is not recursive (as in $fac\ 0 = 1, fac\ n = n \cdot fac\ (n-1)$, where the argument n is decremented with every call, until the value 0 is reached.

However, the behaviour can be correctly obtained by considering definition 1.1 as an *equation*, whose closed-form solution is

$$sys\ x = \frac{A \cdot x}{A + 1}$$

In a very real sense, the circuit actually *solves* the equation in a way that cannot be described as algorithmic (it is even not directly obvious how to approximate this process by simulation).

1.1.2 A different paradigm and its supporting notation

We have seen that the algorithmic paradigm and its supporting notation, namely that of programming languages, is not adequate for the description of general systems and circuits.

On the other hand, the *dynamical system* paradigm that has proven its value over the centuries in physics and engineering, and in recent years especially in systems theory, electronics and communications, provides the necessary generality, even including algorithms as a particular case. The formalisms used in these disciplines have evolved in close correspondence with mathematics, in particular algebra and analysis. The most powerful concept in this respect is that of a mathematical *function*, together with the commonly used notation attributed to Euler. The advantages of functional formalisms over the concepts and notations from programming languages are the following:

- *Expressive power:*

  - semantically not restricted to operational descriptions or to discrete models; hence suitable also for e.g. specification purposes as well;
  - syntactically better matched to the wide variety of system classes and views of systems to be expressed (in particular also a more direct mapping to hardware realizations);

- closer to the notations commonly used in systems theory, electronics and related disciplines for the development of theory and for design.

- *Manipulative power:*

  - the function notation and the equational style of expressions are eminently suitable for transformational reasoning;
  - concepts and theorems from mathematical disciplines (algebra, analysis, etc.) relevant to systems engineering are more directly applicable.

Moreover, the developments in functional programming languages during the past decades, [Lan66], [Bur75], [Bac78], [Tur79], [Bur81], [Tur85], [Mee86], [Mee87], [Bir88] show that the same advantages are gradually being realized in the area of programming (algorithmics) itself. In particular, reasoning about algorithms expressed in functional notation follows the algebraic/transformational style based on equalities and substitutions rather than the cumbersome logic/deductive style based on pre- and postconditions. This is why in this project the notion of *function* has been chosen as the basis for system description. The project acronym *FORFUN* stands for *For*mal system description by means of *fun*ctional languages. As shown in the next chapter, the notation for mathematical functions that is commonly used in systems theory, electronics and related disciplines, can be given various interpretations that are of interest in formal system description. The principle of *system semantics* [Bou86],[Bou88] is to distinguish between these interpretations explicitly by means of semantic functions. This makes it possible to use a single formal language rather than a collection of different languages to express various system aspects such as structure, behaviour, performance, cost.

Chapter 2

Basic Principles

2.1 Existing hardware description languages

J. Vanslembrouck, ABT

2.1.1 Introduction

Hardware descripton languages (HDLs) were conceived for the description, documentation and design of large digital circuits and to simulate these circuits before implementation. Many HDLs therefore have two aspects: the structural part of the language is used to describe the structure of a circuit as an interconnections of subcircuits while the behavioural part describes how the circuit should behave. The latter part is most often used for simulation purposes and as input for logic synthesis tools.

2.1.2 An overview of HDLs

It is surely not the intention of this overview to give a description of all known HDLs. Only a few languages will be described to provide the reader with a flavour of how contempory HDLs look like. The description of IDL, TI-HDL, AHPL and Zeus has largely been taken from [Ayl86].

2.1.2.1 IDL - interactive design language

The original purpose of IDL [Mai82], [Mai83], [Mai84] was to facilitate the design of PLAs embodying highly complex algorithms; it is now used for the design of random logic as well. IDL is nonprocedural; the order in which IDL statements are listed is unimportant. IDL is hierarchical with some limitations. Data entry may be in graphical flow chart or text form. IDL also allows blocks of logic to be represented as truth tables. High-level behavioural descriptions can be modeled with APL programs.

IDL is used for both design and description, since it generates two-level logic from the high-level description. Multilevel logic can also be generated.

The adopted format is a series of two-level boxes connected by signals. Design verification under IDL is achieved via simulation. IDL is suitable for both structural and behavioural descriptions.

2.1.2.2 TI-HDL

TI-HDL [Pre82], [Tex83] stands for "Texas Instruments hardware description language". TI-HDL is a block structured hardware design and description language permitting the description of hierarchical designs. TH-HDL can outline a digital design from abstract functional behaviour down through detailed logic design.

Simulation can be performed at any design hierarchy or design abstraction level. It allows validation of the lower level design against the higher level functional definition. Cost and time of simulation are reduced by modeling at a higher level of abstraction.

TI-HDL supports the design of synchronous, asynchronous, and combined synchronous/asynchronous machines.

2.1.2.3 AHPL - A hardware programming language

AHPL [Hil78] is a hardware description language based upon the notational conventions of APL [Ive62]. AHPL was developed in an academic environment to teach digital system design. Its basic structural similarity to other high-level programming languages makes is accessible to a wider class of users than would be the case with a more specialized hardware language. Since Iverson originally developed APL as a means of describing computer organisation, developing the language into an HDL was very natural. APHL is procedural, as is APL, so that the order of statements is important. Extensions have been made to the original AHPL, and the current version is AHPL III [Hil81].

AHPL makes use of only those APL operators which can be readily interpreted as hardware primitives. A few conventions have been added to AHPL representing unique hardware capabilities such as parallel control sequences, asynchronous transfers, and conditional transfers.

Syntax has been added to the APL base, enabling AHPL to describe synchronous and asynchronous delays. Syntax in AHPL also describes digital system structure, while constructs express repetitive connections between all elements at compile time. Design verification under AHPL is achieved via a function-level simulation.

2.1.2.4 Zeus

Zeus [Lie83], [Lie84] is a general-purpose hardware description language based on Hades [Wir82] and Modula-2 [Wir82]. Principles of Zeus structuring and much of its syntax was copied from Modula-2, but their semantics are radically different. Compared to other hardware description languages, the main extension of Zeus lies in hardware structuring facilities.

Zeus can provide a design's structural and functional description. The functional description is usually linked to the structural description by giving each structural element a functional meaning. However, Modula-2 can describe the behaviour of a component without providing a structural description. Hierarchical abilities are provided through the use of so-called component types within a description. Zeus can also generate hardware conditionally through the use of a simple metalanguage. However, it disallows feedback loops that don't lead through registers. This restriction is claimed to prevent poor design approaches and to simplify simulation. Inherently, Zeus is a nonprocedural language.

Time in Zeus is introduced only because of the storage element. Time is assumed to proceed in discrete steps – so-called clock cycles. Clock cycles are assumed to be long enough to allow signals sufficient time to ripple through the combinational logic. No provision is made for the insertion of propagation, or rise- and fall time delays in either the storage element or the combinational logic.

2.1.2.5 The N.2 environment

N.2 [Abt1] is a collection of languages and programs for designing complex digital systems. N.2 allows the description of designs at a fairly high level – from the register transfer level up to the highest levels for system description. The design can be simulated at the same high level in order to find performance, timing, and functional flaws. Systems can be simulated which have been described at mixed levels of abstraction, from the gate/switch level up to purely algorithmic descriptions written in C.

N.2 allows the description and simulation of all components of the system design – hardware, firmware, and software. N.2 simulation supports a broad range of user-defined breakpoints, counters, timers and other analytic tools.

Broadly speaking, the programs, libraries, and languages which comprise N.2 can be classified into three categories – the hardware tools, the software tools, and the postprocessing and utility tools. The hardware tools consists of a pair of languages (see below) for describing a hardware design, the compilers of these languages, a simulator, and a command language for controlling the simulator. The software tools consist of a meta-assembler and a meta-loader, plus assorted minor programs. These tools are important for any designer who works at a level where firmware or software is part of the complete system design. The postprocessors provide powerful mathematical and display capabilities for analyzing and comparing the results of simulation runs. These utilities allow statistical analysis of design performance, graphical analysis of design output, and comparative functional analysis of related designs.

In the N.2 system, hardware behaviour is described in the language ISP', which is based on an earlier language ISP (Instruction Set Processor) [Bel71]. The original ISP was used to describe the instruction set of a computer. ISP' is more powerful and more general in purpose. In fact, ISP' is comparable in size and power to a general purpose programming language such as C or Pascal. System structure is described in the Topology Language. A Topology file is always required in order to run a simulation.

2.1.2.6 VHDL

Because VHDL (VHSIC HDL where VHSIC stands for Very High Scale Integrated Circuits) has become the IEEE 1076 standard, this language will be described in a little more detail than the other languages. Informal introductions to the language can be found in [Sha85] and [Lip86].

Design entity. The principal hardware abstraction in VHDL is the *design entity* [Lip86]. The design entity provides for effective separation of interface and function, thus allowing hierarchical design decomposition. A design entity models hardware of any complexity; for example, a design entity may model a logic gate, a flip-flop, a control unit, or a computer system. A design entity is composed of an interface description and one ore more alternative bodies.

Interface description. The *interface* description describes the component's external view, and its operating characteristics and conditions (such as timing assertions) that need to be known by component users. The interface contains information common to its alternative bodies. A subset of this information (namely the specification of ports and generics) is externally visible.

- Ports define communication channels between design entities and the outside world. A port definition involves description of its mode and type. The port's mode specifies the direction of information flow through the port. A port type specifies the set of values a port may assume. Port values may be represented by voltage levels, truth values, binary digits, or multiple logic values.

- A design entity interface may also define generics. Such a generic design entity defines a class of components. When used, a generic design entity is particularized to select one component in the class. To particularize a generic design entity, desired values are supplied for corresponding generics.

Body description. A design entity *body* describes the organization and/or operation of a component. An interface may possess multiple bodies, each representing a different implementation or emphasizing a different design aspect. For example, one body may describe a component's behaviour while another describes its structure. Alternative structural implementations of the same hardware entity can be modeled to evaluate cost and speed factors. VHDL provides two body description types: architectural bodies and configuration bodies.

Architectural bodies describe how the input and output ports of a design entity relate, either by expressing the involved input/output data transformation or by connecting those ports to subcomponents. Three general styles of description are possible within an architectural body: structural, dataflow and behavioural.

- Structural descriptions capture the schematic view of hardware and consist primarily of interconnected components. The behaviour of the subcomponents is defined externally (in the form of another design entity). One advantage of structural descriptions is the ability to express the parallelism inherent in hardware operation. Potentially, every component of a structural description may operate in parallel with any number of other components; however, purely structural descriptions cannot express *what* each component actually does since component behaviour is externally defined.

- Dataflow descriptions, a little more abstract, specify data transforms being performed in terms of concurrently executing RTL statements. Dataflow descriptions allow the simultaneous expression of parallelism and behaviour. Dataflow descriptions are created through the use of *concurrent signal assigment statements*. Such statements execute in response to *events* or changes in signal values referenced within the statements.

- Behavioral descriptions, the most abstract, specify data transforms in terms of algorithms for computing output responses to input changes. When it is necessary to describe behaviour without biasing that description to a particular implementation, behavioural descriptions, entailing little or no structural information, can be used. Behavioral descriptions are written using *process* statements. A process statement contains a declarative part and a statement part. However, only *sequential* statements such as if, case, and loop statements can be written in its statement part. Such statements are used to describe algorithms that specify how a component will respond to changes on input signals by causing changes on output signals. Such and algorithm need imply nothing about structural characteristics of the component whose behaviour is being modeled because it contains no parallelism.

A given architectural body may make use of any combination of these styles of description, for they are all defined under a common set of semantics.

A configuration body contains global information (such as which design entities model subcomponents used in an architectural body or how global signals are distributed). For more information on this subject, the reader is referred to [Lip86].

2.1.2.7 Verilog

Verilog HDL [Abt2] can be considered as the "de facto" standard and is widely spread in the U.S. Lots of libraries are available in Verilog HDL.

Verilog is a rather concise and powerful language, specifically developed for modelling and simulation. Apart from the many modelling constructs, the languages provides specific constructs for stimuli generation and debugging.

The levels of abstraction covered by Verilog range from the architectural level (where stochastic and statistics simulation is possible) to the switch level (where four values and seven strengths are available to model signals).

2.1.2.8 muFP

muFP [She83] is a VLSI design language which is a variant of Backus' FP [Bac78]. There are several reasons why muFP was based on FP. Functional programs are easier to write and to reason about than imperative ones. muFP descriptions are just expressions built from a small number of primitive functions and combining forms (functionals that map functions into functions). These functions and combining forms (CFs) have nice algebraic properties. Circuit descriptions are concise and can be manipulated using the algebraic laws of the language. Each CF has a simple geometric interpretation, so that every muFP expression has an associated schematics.

2.1.2.9 ELLA

ELLA [Mor82] is a hardware description language designed to cover a wide range of hardware design, from below gate level to above register transfer level. The language includes facilities for the description of networks and the behaviour of nodes. Because of the wide range that ELLA can cover, nodes can be anything from transistors to microprocessors.

ELLA contains no inbuilt signals. Users can either build their own representation of signals by defining appropriate types or they can use types declared for them in a library.

ELLA is a functional language, and as such there are no assignments of signal values to variables. Values are transformed by a network of functions.

2.1.2.10 Using other formalisms as HDL

Existing general purpose programming and specification languages have been proposed and used for the behavioural and structural description of digital hardware. Examples are Concurrent Prolog [Suz85], Prolog [Clo87], Temporal Logic [Mos85] and HOL [Han85].

2.1.3 For further reading

Three issues of *Computer* magazine handled hardware description languages (HDLs). The first issue (December 1974) discussed the similarities and differences between programming and hardware design and the necessity for a high-level language in which designers could express asynchronous operations, parallel control, and the structure of a hardware system. In the second issue (June 1977), the use of a common language for hardware designers and a design language that reflects a common modeling philosophy for software and hardware were suggested. The articles in the issue revealed a growing interest in applications such as synthesis and test sequence generation. The third special issue on HDLs (February 1985) includes results of efforts to develop a common language as well as several new languages with special applications, such as formal verification of hardware designs. Papers handled Temporal Logic, DDL (Digital Design Languae), Concurrent Prolog, Zeus, Conlan and VHDL. The April 1986 issue of *IEEE Design and Test of Computers* was devoted to VHDL which was about to become the standard HDL at that time.

2.1.4 Conclusion

Existing hardware description languages are mainly used for the behavioural description, simulation and recently also the synthesis of large digital designs. Their syntax and semantics has often been based on existing (procedural) programming languages. The separation is between behavioural (which have no structural interpretation) and structural descriptions (which have no behavioural interpretation), if made at all, is usually not very strict.

Little attention has been paid to the description of analog circuits and to the readibility of the structural part. Structure is often considered as a derived concept and is entered by means of schematic entry or generated by means of synthesis tools. Therefore, structural descriptions still resemble netlist-like descriptions.

Behaviour (with or without timing information) is considered to be the central semantics of many existing HDLs. Simulation is most often the only means for verification.

2.2 Formal system description with functional formalisms

R.T. Boute, KUN

Reading Suggestion As is often the case with an introduction, it is advisable to read this section only superficially before the remainder of this book, and perhaps more thoroughly afterwards.

Moreover, this introduction contains informal discussion interspersed with more technical material that is of interest to specialists in language design only. Much of the latter material may be skipped at first reading. In particular, it is advisable to look initially only at the paragraphs dealing with the *structural* model for each example and leave the *behavioural* model until later on.

2.2.1 Mathematical function definitions

We recall that a *function* (or *mapping*) from (set) A to (set) B is a mathematical object that associates with every element of A a single element of B. We write $A \to B$ for the set of all functions from (the *domain*) A to (the *codomain*) B. An expression of the form $f \in A \to B$ is called a *type definition* for f. If $x \in A$ and $f \in A \to B$, then $f\ x$ denotes the corresponding element in B (called the *image* of x under f). A function is completely defined by its type definition and its image definition (i.e. the image of every element in the domain). Therefore we write a function definition as follows:

$$\textbf{def}\ type\ definition\ \textbf{with}\ image\ definition$$

The two main styles for an image definition are the following:

- *Extensional definition*: the image is specified individually for every value in the domain, for instance by means of a table. Of course, this is practical only for small (and, a fortiori, finite) domains. An example is the following (where $\mathbb{B}$denotes the set $\{0, 1\}$):

$$\begin{array}{lll} \textbf{def}\ sel \in \mathbb{B}^3 \to \mathbb{B}\ \textbf{with} & & \\ & sel\ \langle 0,0,0\rangle = 0 & sel\ \langle 0,0,1\rangle = 1 \\ & sel\ \langle 0,1,0\rangle = 0 & sel\ \langle 0,1,1\rangle = 1 \\ & sel\ \langle 1,0,0\rangle = 0 & sel\ \langle 1,0,1\rangle = 0 \\ & sel\ \langle 1,1,0\rangle = 1 & sel\ \langle 1,1,1\rangle = 1 \end{array}$$

- *Intensional definition*: by means of an expression for the image of an arbitrary value in the domain (designated by a variable), for instance:

$$\textbf{def}\ sel \in \mathbb{B}^3 \to \mathbb{B}\ \textbf{with}\ sel\ \langle s,x,y\rangle = s\ ?\ x;\ b$$

  Here the notation $c\ ?\ a;\ b$ abbreviates **if** c **then** a **else** b.

2.2.2 Formal function definitions and their system semantics

In the above discussion, we have interpreted the notation according to its *standard mathematical interpretation*, namely as the definition of mathematical functions (mappings). The notation itself can be given many other useful interpretations. In order to "make room" for these interpretations, we must start by considering the definitions as mere pieces of text, so-called *formal function definitions (ffds)* without any a priori or privileged interpretation, and express the desired *interpretation* (or *model*) explicitly by means of a *semantic* (or *meaning*) function). A *meaning function* is defined in a general way on the syntax of the language. Its *domain* is the set of all possible sentences (in this case ffds) in the language and its *codomain* is the space of all possible values (in the widest sense) that a certain system property or aspect may assume. For instance, the codomain for the meaning function **struct** describing the structural model is the space of all possible system structures (for the considered class of systems). The most elegant method for describing meaning functions was developed in the context of programming languages by Knuth [Knu67]. How to extend this method to formal system description languages is shown in [Bou86], [Bou88]. In this section, however, we will not give the definition of the semantic functions themselves, but only illustrate their usage by showing the image of specific example sentences under the considered semantic function. Moreover, these images will be expressed informally, viz.

- using schematics for the structural model
- using mathematical functions for the behavioural model.

Finally, when the meaning function is clear from the context, we designate its application by underscoring the argument, e.g., when dealing with the structural model, $\underline{system}$ abbreviates **struct** *system*. This reduces notational clutter and saves space.

2.2.3 Examples for digital and analog systems

Preliminary remark: not all expressions one encounters in mathematics can be seen as ffds having system-oriented interpretations. In fact, only a special set of primitive functions and function composition rules can be given meaningful structural intrepretations: for instance, in the synchronous digital circuit case, the primitive functions are those corresponding to gates and memory cells (D-flipflops, JK-flipflops etc.), and the syntactic composition rules are only *function application* (representing input/ouput connection in the structural model) and the **where**-construct (structurally: fanout and/or feedback). The Glass language is designed such that only these combinations are possible.

2.2.3.1 A combinational circuit example (a data selector)

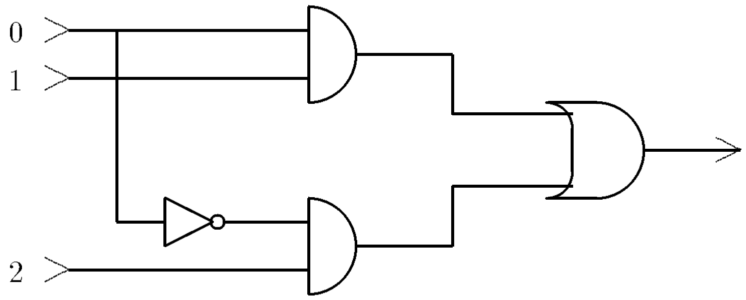
Consider the following two ffds:

$$\textbf{def } hsel \in \mathbf{E}^3 \Rightarrow \mathbf{E} \textbf{ with } hsel\ \langle s, x, y\rangle = (s \wedge x) \vee (\neg s \wedge y)$$
$$\textbf{def } ssel \in \mathbf{E}^3 \Rightarrow \mathbf{E} \textbf{ with } ssel\ \langle s, x, y\rangle = (s \wedge x) \vee (\neg s \wedge y) \vee (x \wedge y)$$

Here **E** designates a so-called base type (still uninterpreted!).

- In the *structural model*

  $\textbf{struct}\ (\mathbf{E}^3 \Rightarrow \mathbf{E})$ = the set of 3-input, 1-output digital systems
  $\textbf{struct}\ hsel =$



  and similarly for ssel. Clearly $\textbf{struct}\ hsel \neq \textbf{struct}\ ssel$.

- In the *simplex behavioural model* (with $\textbf{simp}\ \mathbf{E} = \mathbb{B}$), the operators $\vee, \wedge, \neg$ are interpreted according to their standard mathematical interpretation (logical *or*, *and*, *not*). Hence

  $$\textbf{simp}\ (\mathbf{E}^3 \Rightarrow \mathbf{E}) = \mathbb{B}^3 \rightarrow \mathbb{B} \qquad \text{(recall: } \mathbb{B} = \{0, 1\})$$
  $$\textbf{simp}\ hsel = sel \qquad \textbf{simp}\ ssel = sel.$$

  where *sel* is the *mathematical* function defined earlier. Clearly $\textbf{simp}\ hsel = \textbf{simp}\ ssel$.

- In the *multiplex behavioural model* we deal with sequences, representing discrete-time behaviour. We may choose to model this by either infinite (functions of type $\mathbb{N} \rightarrow \mathbb{B}$, abbreviated $\mathbb{N}^\infty$) or finite sequences (type $\mathbb{N}^*$). The choice is of little importance. For instance, letting $\textbf{mult}\ \mathbf{E} = \mathbb{N}^\infty$, we obtain

  $$\textbf{mult}\ hsel \in (\mathbb{N}^\infty)^3 \rightarrow \mathbb{N}^\infty$$
  $$\textbf{mult}\ hsel\ \langle s, x, y\rangle\ n = sel\ \langle s\ n,\ x\ n,\ y\ n\rangle$$

  Here also $\textbf{mult}\ hsel = \textbf{mult}\ ssel$. Memoryless systems (such as this example) typically satisfy the simple relationship:

  $$\textbf{mult}\ sys\ input\ n = \textbf{simp}\ sys\ (input\ n)$$

Hence we shall also give an example of a digital circuit with memory later on.

- The Eichelberger behavioural model can reflect static hazards by using ternary algebra $\mathbb{E} = \{0, 1, \perp\}$, where $\perp$ stands for "undefined" (in this case modelling a signal in transition). The primitive operators are interpreted as follows:

$$\begin{aligned} \underline{\neg} a &= a = 0?1; a = 1?0; \perp \\ a \underline{\wedge} b &= a = 0?0; a = 1?b; b = 0?0; \perp \\ a \underline{\vee} b &= a = 1?1; a = 0?b; b = 1?1; \perp \end{aligned}$$

The elaboration of **eich** $hsel$ and **eich** $ssel$, both of type $\mathbb{E}^3 \to \mathbb{E}$, is left as an exercise for the reader. Notice, however, that **eich** $hsel \ \langle \perp, 1, 1 \rangle = \perp$ whereas **eich** $ssel \ \langle \perp, 1, 1 \rangle = 1$ and hence **eich** $hsel \neq$ **eich** $ssel$. This reflects the fact that $hsel$ is hazardous and $ssel$ is safe with respect to static hazards.

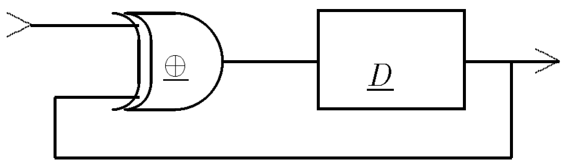
2.2.3.2 A sequential circuit example (a parity checker)

Consider the following ffd

$$\mathbf{def}\ par \in \mathbf{E} \Rightarrow \mathbf{E}\ \mathbf{with}\ par\ x = y\ \mathbf{where}\ y = D\ (x \oplus y)$$

- The *structural model* yields

struct par =



- The *simplex behavioural model* is mathematically trivial and practically meaningless for systems with memory.

- The *multiplex behavioural model* yields

$$\begin{aligned} &\mathbf{mult}\ par\ x \in \mathbf{mult\ E} \to \mathbf{mult\ E\ with} \\ &\mathbf{mult}\ par\ x = y\ \mathbf{where}\ y = \underline{D}\ (x \underline{\oplus} y) \end{aligned}$$

With the following auxiliary functions, taking **mult E** $= \mathbb{N}^\infty$:

$$\begin{aligned} &\underline{D} \in \mathbb{N}^\infty \to \mathbb{N}^\infty\ \mathbf{with}\ \underline{D}\ a\ n = n = 0?0; a\ (n-1) \\ &\underline{\oplus} \in (\mathbb{N}^\infty)^2 \to \mathbb{N}^\infty\ \mathbf{with}\ (a \underline{\oplus} b)\ n = a\ n \oplus b\ n \end{aligned}$$

where $\oplus$ (not underscored) designates exclusive or, this yields

$$\underline{par} \in \mathbb{N}^\infty \to \mathbb{N}^\infty\ \mathbf{with}$$

$\underline{par}\ x = y$ **where** $y\ n = n = 0?0; x\ (n-1) \oplus y\ (n-1)$

The elaboration with **mult** $\mathbf{E} = \mathbb{N}^*$ is left as an exercise.

2.2.3.3 An analog control system example

Consider the ffd

def $sys \in \mathbf{E} \Rightarrow \mathbf{E}$ **with** $sys\ x = y$ **where** $y = PF\ \langle x, y \rangle$

assuming that P (the "plant") and F (the "feedback") are primitive formal functions or are defined separately.

- The *structural interpretation* yields

  struct $sys =$

- The *behavioural interpretation* yields:

$$\underline{sys}\ x = y \textbf{ where } y = \underline{P}.\underline{F}\ \langle x, y \rangle \qquad (2.1)$$

With time functions ($\in \mathcal{T} \to \mathbb{R}$) for the base domain:

$$\begin{array}{rcl} \underline{sys} & \in & (\mathcal{T} \to \mathbb{R}) \to (\mathcal{T} \to \mathbb{R}) \\ \underline{F} & \in & (\mathcal{T} \to \mathbb{R})^2 \to (\mathcal{T} \to \mathbb{R})^n \\ \underline{P} & \in & (\mathcal{T} \to \mathbb{R})^n \to (\mathcal{T} \to \mathbb{R}) \end{array}$$

where $\underline{F}$, $\underline{P}$ depend on the definitions of F and P. In general, such functional equations cannot be solved in closed form. However, with Fourier transforms ($\in \mathbb{R} \to \mathbb{C}$) for the base domain (useful for linear systems only), formal functions are interpreted as transfer functions, application as multiplication, and equation (2.1) can be solved in closed form. For instance, with $\underline{F}\ \langle x, y \rangle = G.x + H.y$, equation (2.1) becomes

$$\underline{sys}\ x = y\ where\ y = \underline{P}.(G.x + H.y)$$

and, following solution of the algebraic equation for y,

$$\underline{sys} = \frac{\underline{P}.G}{1 - \underline{P}.H}$$

2.2.3.4 Language syntax considerations

From the above examples, it becomes apparent that, for the considered class of systems (made more specific later on), as few as *two* language constructs are sufficient to make a fully-fledged system description language, namely *function application* (with structural interpretation as a signal flow description) and the **where**-*expression* (with structural interpretation as a description for fanout, i.e. output sharing, and feedback). This illustrates the fact that *syntax design* for system description languages is essentially a matter of *variable scoping and binding rules.*

For systems with large repetitive structures, the usage of *macro instantiations* may avoid tedious textual repetitions. The *macro definitions* may be *parameterized* to cater for minor variations. Strictly speaking, macro definition is not a language design issue, in the sense that it does not depend on the semantics and is similar for "all" languages. Yet it is good practice to make macro definitions syntactically identical to formal function definitions, except for the fact that they may contain free variables to reflect bindings in the context of instantiation. The following definition of a synchronous counter may serve as an illustration:

Def
$\quad counter \in B^8 \Rightarrow B^5;$
$\quad counter\langle d_0, d_1, d_2, d_3, nload, nclear, enap, enat\rangle =$
$\quad\quad \langle q_0, q_1, q_2, q_3, carry\rangle$
$\quad\quad$ **where**
$\quad\quad\quad carry = and5\ \langle q_3, q_2, q_1, q_0, enat\rangle;$
$\quad\quad\quad q_3 = stage\langle d_3, preset, and4\ \langle q_2, q_1, q_0, en\rangle, nclr\rangle;$
$\quad\quad\quad q_2 = stage\langle d_2, preset, and3\ \langle q_1, q_0, en\rangle, nclr\rangle;$
$\quad\quad\quad q_1 = stage\langle d_1, preset, and2\ \langle q_0, en\rangle, nclr\rangle;$
$\quad\quad\quad q_0 = stage\langle d_0, preset, en, nclr\rangle$
$\quad\quad\quad preset = ornot\ \langle nload, nclr\rangle;$
$\quad\quad\quad nclr = buf\ nclear;$
$\quad\quad\quad en = and2\ \langle enap, enat\rangle;$
$\quad\quad\quad$ **mac**
$\quad\quad\quad\quad stage \in B^4 \Rightarrow B;$
$\quad\quad\quad\quad stage\langle d, preset, toggle, nclr\rangle =$
$\quad\quad\quad\quad\quad jkff\ \langle and2\langle ja, ea\rangle, and2\ \langle ka, ea\rangle\rangle$
$\quad\quad\quad\quad\quad$ **where**
$\quad\quad\quad\quad\quad\quad ja = nand\ \langle ka, preset\rangle;$
$\quad\quad\quad\quad\quad\quad ka = nand3\ \langle d, preset, nclr\rangle;$
$\quad\quad\quad\quad\quad\quad ea = or\ \langle toggle, preset\rangle$
$\quad\quad\quad\quad\quad$ **endwhere**
$\quad\quad$ **endwhere**

The *structural* interpretation is represented schematically in figure 2.1. Notice that, in the macro definition, only d and *toggle* occur as formal arguments, because they correspond to inputs to *stage* which are different for

Figure 2.1: structural interpretation of the counter

each of the four instantiations. By contrast, *nclr* and *preset* are distributed over all four instantiations and therefore are represented by free variables which become bound in the context of the **where**-expression.

2.2.4 Description of adirectional systems

2.2.4.1 Problem statement

The examples given thus far show that formal function definitions constitute an ideal syntax for so-called (uni)directional systems i.e. systems, or conceptual views of systems, for which it is meaningful to attribute a direction to the signal or information flow inside and between the components, and hence to distinguish between input and output. Moreover, the notation is close to the standard mathematical notation for describing behaviour. We have seen that, for the examples given, it suffices to replace the formal primitive functions by the mathematical functions expressing their behaviour. Hence formal function definitions may be subject to transformations as if they were mathematical expressions, and the behavioural interpretation remains unaffected. This results in a high manipulative power for transformational design.

However, it has been shown [Bou88] that for certain classes systems, even undirectional ones, this attractive property may be invalidated by loading effects.

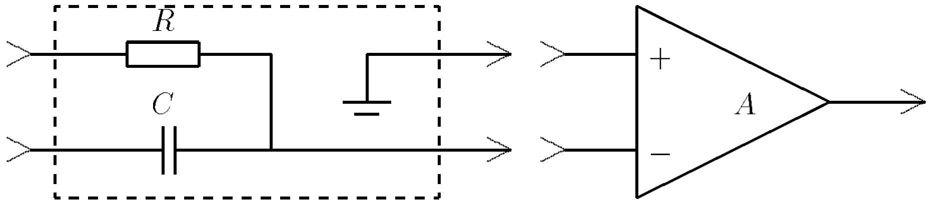
Moreover, formal function definitions do not have any known interpretation that makes them useful as a syntax basis for describing adirectional systems, as is illustrated by the following example.

Consider the definition

$$\textbf{def } sys \in \mathbf{E} \Rightarrow \mathbf{E} \textbf{ with } sys\ x = y \textbf{ where } y = PF\ \langle x, y \rangle$$

where we assume the case of an integrator with

struct $F =$ **struct** $P =$



We observe that an ffd is inadequate for describing F. Indeed, a naive syntactic attempt such as

$$\textbf{def } F\ \langle x, y \rangle = \langle ground, node\ \langle R\ x, C\ y \rangle \rangle$$

is not generalizable to express structure in all cases of interest, and has no systematic behavioural interpretation that yields correct results. For instance, in the linear case and assuming low-Z source and high-Z load, one should find (for the voltage transfer model):

$$\textbf{behav } F\ \langle x,\ y \rangle = \langle 0,\ \frac{x + \jmath\omega RCy}{1 + \jmath\omega RC} \rangle$$

which cannot be obtained from the above syntax in a compositional way.

2.2.4.2 Sigma terms for adirectional systems

The ffds as shown are based on the syntax and on the variable binding and scoping rules of lambda terms, whose general form (in abstract syntax) is

$$\lambda varlist.expression$$

For adirectional systems, it is more advantageous to introduce the syntax, variable binding and scoping rules of sigma terms. These are described in more detail in [Bou90]; here we suffice with a small example. The general form (in abstract syntax) for a sigma term is:

$$\sigma varlist.appset$$

where an *appset* is itself of the following form: {*sigterm varlist*, ..., *sigterm varlist*}.

The following illustrates an unparameterized definition expressed in the concrete syntax:

$$\textbf{def } star \in [\mathbf{E}^3] \textbf{ with } star\ \langle x,y,z\rangle = \{A\langle x,c\rangle, B\langle y,c\rangle, C\langle z,c\rangle\}$$

where square brackets designate the type of a sigma term. Parameterization is introduced, as usual, using the ordinary conventions for function definitions (lambda terms), for instance

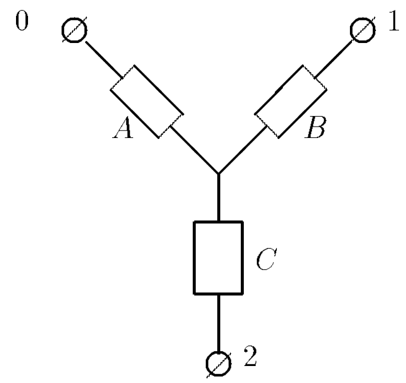
$$\begin{array}{ll}\textbf{def} & Y \in [\mathbf{E}^2]^3 \to [\mathbf{E}^3] \textbf{ with} \\ & Y\ \langle A,B,C\rangle\langle x,y,z\rangle = \{A\langle x,c\rangle, B\langle y,c\rangle, C\langle z,c\rangle\}\end{array}$$

where the image part is concrete syntax for

$$Y = \lambda\ \langle A,B,C\rangle.\sigma\langle x,y,z\rangle.\{A\langle x,c\rangle, B\langle y,c\rangle, C\langle z,c\rangle\}$$

- In the *structural interpretation* of a sigma term for electrical circuits, the σ-abstractor ($\sigma varlist.$) describes the externally accessible connections, and the appset the internal connections, every variable name standing for an interconnection net. Different names represent disjoint nets, unless specified otherwise by a *synterm* (or *synonym term*, with syntax $\star varlist$) appearing in the appset. For instance,

  struct $Y\ \langle A,B,C\rangle =$



- In the *behavioural interpretation* of a sigma term for electrical circuits, the most general model describes the circuit equations. Let

$\mathbb{E} \subset (\mathcal{T} \to \mathbb{R}) \times (\mathcal{T} \to \mathbb{R})$ be the set of pairs $\langle i,\ v\rangle$ describing current and voltage time functions. Then the domain of interpretation is the set of predicates $\mathbb{E}^* \to \mathbb{B}$.

Elaboration of the meaning function for the example yields:

$$
\begin{array}{l}
\underline{Y\ \langle A, B, C\rangle} \in \mathbb{E}^3 \to \mathbb{B} \\
\underline{Y\ \langle A, B, C\rangle}\ \langle\langle i_0, v_0\rangle,\ \langle i_1, v_1\rangle,\ \langle i_2, v_2\rangle\rangle = \\
\qquad (i_0 + i_1 + i_2 = 0) \wedge \\
\qquad \exists v.(\underline{A}\langle\langle i_0, v_0\rangle, \langle -i_0, v\rangle\rangle \wedge \\
\qquad\qquad \underline{B}\langle\langle i_1, v_1\rangle, \langle -i_1, v\rangle\rangle \wedge \\
\qquad\qquad \underline{C}\langle\langle i_2, v_2\rangle, \langle -i_2, v\rangle\rangle)
\end{array}
$$

where, as usual, $\underline{A}$ stands for **behav** A etc. Notice that different occurrences of a same variable (e.g. c) in the sigma term may yield, in the behavioural model, $\langle i,\ v\rangle$ pairs with different i-components, reflecting Kirchhoff's laws for electrical nets.

In general, these equations have no closed form solutions. However, for special classes of circuits (e.g. allowing linear approximations), the usage of Fourier transforms reduces them to algebraic equations, which can be reflected by another behavioural model. In the same vein, various models can reflect various circuit analysis techniques, such as the DC (quiescent point) model, the small signal (linear) model, the large-signal (distortion) model, the noise model.

Chapter 3

Informal Introduction to the language Glass

M. Seutter, KUN

3.1 Introduction

Glass (**G**eneral **la**nguage for **s**ystem **s**emantics) is a language for describing arbitrary systems, although its present usage is limited to electronic systems. The ideas for this language and its "describing environment" (as opposed to the term "programming environment") are based upon systems semantics [Bou88].

Inspired by the Algol-68 tradition [Wij76], [Lin80] the language is documented in two reports namely the "Revised report on the language Glass" plus "Grammar for Glass, context-sensitive part", which can be found in an other volume, and the "Informal introduction to the language Glass". The first describe the exact syntax and typing system of the language. The last is intended to introduce the language Glass in an informal way to those readers that have not seen any Glass description before. Thus we will pay more attention to describing the ideas behind the language and elucidating the language concepts by many examples than to describing its exact syntax.

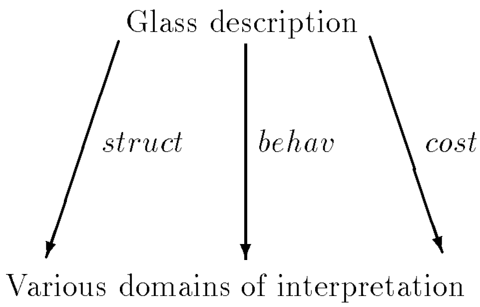
Glass also has two language representations, namely the *reference language* and the *implementation language*. The first is intended as *publication language* and is therefore used throughout this chapter in all of the examples. However it is not suited for automated interpretation. For this purpose the implementation language may be used. In an appendix of this book the implementation language equivalents of the examples presented in this chapter will be given.

In this "Informal Introduction" we will first concentrate on the basic principles of the language. After that we will try and learn the language constructs one by one.

3.1.1 Basic principles

> *"When I use a word,", Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean, neither more nor less"*
> *Through the Looking-glass* *Lewis Carroll*

In ordinary hardware description languages several languages are used to describe different aspects of a circuit, like its behaviour, its structure, its layout etc. Opposed to this is the idea of systems semantics: Use only *one* language to describe formally the decomposition of a system into subsystems and their connectivity: Glass. Such a formal description may then be assigned a meaning by a *semantic function.* A semantic function is a function from the set of all legal Glass descriptions to a certain domain of interpretation. A description in Glass therefore has no a priori meaning at all: it is merely a piece of text. You can only derive a meaning of it by applying a semantic function to it. So by applying several semantic functions to *one* description one may get several meanings of the described circuit, like structure, its cost, its behaviour, etc. In fact, you derive some kind of *multi-view* of the described system:



Examples of semantic functions will be given as soon as we can comprehend small system descriptions.

Every semantic function should, if possible, be *compositional*, that is, the semantic function expresses the meaning of a composite system in terms of the meanings of the constituents. This of course means that the semantic function itself is based upon the meaning of certain primitive systems. These latter are called *atoms* in Glass.

Descriptions in Glass define systems in terms of subsystems and their connectivity. Not all interconnections are of the same nature: systems may be coupled electrically (by wires), magnetically, pneumatically, optically (by glass fibers), etc. We will therefore use the term *connection* in a very general sense, not only for electric connection, but also for connection via magnetics, optics etc. We will also use the word *terminal* in a general sense, being an interface between (sub)systems. Also two systems may be multiply connected: the interface between the two may be decomposed into sets of subconnections.

A system description is therefore characterized not only by its decomposition, but also by its external interface. The specification of such an external interface is called the *type* of a system.

Basic interconnections are introduced by means of so called *basetype* declarations, which serve to introduce a name for an elementary connection. In all of our examples there will be only one basetype E, which indicates the type of the wire.

Systems may be differentiated into three large classes namely the *directional*, the *adirectional* and the *hybrid* ones. In directional systems there is a clear flow of information from a certain set of terminals (oft called inputs) to another set of terminals (outputs). Digital circuits typically belong to this class. In the adirectional systems such a clear flow of information is missing. Analogue circuits typically belong to this class. Of course hybrid systems (systems that are neither purely directional nor purely adirectional but more a kind of mixture of both) can also exist, as we will see later.

This differentiation can be made for the interface of a system as well as for its decomposition. A system may well be directional for the outside world whereas it can be composed out of adirectional subsystems. For our first descriptions we will focus on directional digital circuits. As examples will be freely mixed with remarks the reader is warned that all descriptions in one section of this document must be combined to make a correct Glass text.

3.2 Directional systems

3.2.1 A very simple system

"So it does!" said Pooh. "It goes in!"
"So it does!" said Piglet. "And it comes out!"
"Doesn't it?" said Eeyore.
"It goes in and out like everything."
Winnie-the-Pooh *A.A. Milne*

Consider the following description:

Basetype E;

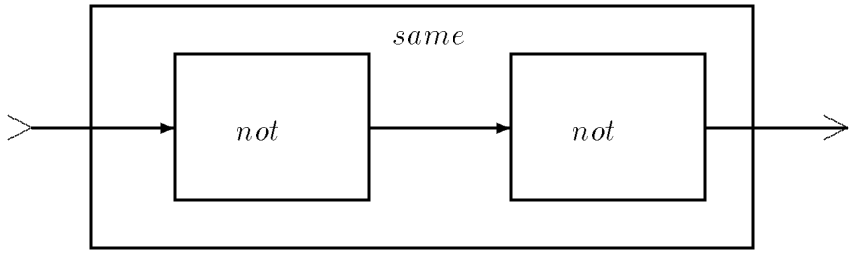
Atom
$not \in E \Rightarrow E$;

The first sentence of this Glass text introduces E as a basetype, which will stand for the type of a wire, as explained on the previous page. The second introduces an atomic component called *not*. Atomic system definitions can not be decomposed into subsystems as they stand for primitive components, but their external interface *must* be specified. We will call such a specification of the interface a *declaration*. In the declaration we see a first example of a typeconstructor namely the $\Rightarrow$. In general $U \Rightarrow V$ stands for the set of directional systems having terminals of type U as input and terminals of type V as output. Thus *not* is declared to be an element of the set of directional systems having one input of type E and one output of type E.

Now here is some very trivial component:

Def
$same \in E \Rightarrow E;$
$same\ x \ = \ not\ (not\ x);$

This is the first composite system definition we meet. A composite system definition starts with a declaration (the type declared must of course be valid for the description), followed by its decomposition into subsystems. In directional systems this decomposition looks a bit like a function definition in a functional language. In a functional language $f\ u$ stands for the application of the function f to the parameter u. This notation is also used in Glass. At the left hand side of the $=$ you see the system name and the name of its input parameter. At the right hand side stands an expression. In this expression application of a system to a subexpression means the connection of output to input or if that subexpression is a name, connection of that terminal to a input. In this way *same* is described as a system with one input terminal named x, and one output terminal which is connected to a row of two *not*s fed by x:



Mark that the parentheses in the right hand side of the definition are really necessary: without them one would read this expression as the application of a system to a system applied to x (i.e. $(not\ not)\ x$), which clearly is faulty Glass. Check the typing for instance: From the expression $not\ e$, we derive that e must have type E. In the expression $not\ not$ the second not has type $E \Rightarrow E$.

As a general rule, parentheses ('(' and ')') are used for indicating priority.

3.2.2 Multiple connections

In the coming example the systems may get more than one input. More precisely: their input parameter may be a compound one. The type of a cable of connections will be specified by means of the cartesian product of the constituents. Thus $E \times E$ is the type of a bundle of two wires. You may also use a shorthand for specifying repeated cartesian products: U^k stands for the k-fold cartesian product $U \times U \times \ldots \times U$, provided $k \geq 2$. What this construct means for the cases $k = 0$ and $k = 1$ will be explained later on. Now let us introduce some well known systems with two inputs:

Basetype E;

Atom
$\quad$ $and \in E \times E \Rightarrow E,$
$\quad$ $or \in E \times E \Rightarrow E,$
$\quad$ $xor \in E \times E \Rightarrow E;$

Now suppose that we have two terminals of type E, say x and y, how can we combine them into a terminal of type $E \times E$? The way to do this in Glass is by making a tuple $\langle x, y \rangle$ for it. In general if you have a set of terminals $a_1, a_2, \ldots, a_n$ so that a_i is a terminal of type U_i then $\langle a_1, a_2, \ldots, a_n \rangle$ is a terminal of type $U_1 \times U_2 \times \ldots \times U_n$. Let us continue the description:

Atom
$\quad$ $not \in E \Rightarrow E;$ $\quad$ */* we have met this one before! */*

Def
$\quad$ $select \in E \times E \times E \Rightarrow E;$
$\quad$ $select\ \langle s, a, b \rangle = or\ \langle and\ \langle not\ s, a \rangle, and\ \langle s, b \rangle \rangle;$

Tupling (making a tuple) is allowed in the formal arguments of a system definition as well as in expressions. You can use this construct also for making circuits with more than one output:

Def
$\quad$ $halfadder \in E \times E \Rightarrow E \times E;$
$\quad$ $halfadder\ \langle x, y \rangle = \langle xor \langle x, y \rangle,\ and\ \langle x, y \rangle \rangle;$

3.2.3 Nameless systems

In all our previous examples the described systems had names. Sometimes however it is useful to describe *nameless systems*. A nameless directional system is introduced by a λ-abstractor. With

$$\lambda\ \langle a, b \rangle . not\ (or\ \langle a, b \rangle)$$

we mean a nameless system with two inputs (i.e. having type $E \times E \Rightarrow E$), consisting of an *or* and a *not* gate. These λ-abstractors may appear anywhere in expressions. In particular, the right hand side of a definition may consist of a single λ-abstractor. In fact the following two definitions are completely equivalent:

Def

$ornot \in E \times E \Rightarrow E;$
$ornot\ \langle a, b\rangle = not\ (or\ \langle a, b\rangle);$

Def

$ornot \in E \times E \Rightarrow E;$
$ornot = \lambda\ \langle a, b\rangle.not\ (or\ \langle a, b\rangle);$

The first definition can be seen as a *syntactically sugared* version of the second definition.

3.2.4 Feedback and Fanout

But Eeyore wasn't listening.
He was taking the balloon out,
and putting it back again
as happy as could be.
Winnie-the-Pooh *A.A. Milne*

In many systems *feedback* may be encountered. In the digital field feedback is used to build sequential circuits, synchronous and asynchronous ones. In the analog domain, feedback is used for instance to reduce the distortion in amplifiers.

With the language we know so far, we cannot describe feedback yet. In Glass *feedback* and also *fanout* (that is the connection of several inputs to one output) may be described by a **where**-*clause*, with which you may introduce *local* definitions. Consider for instance the following definition:

Basetype E;

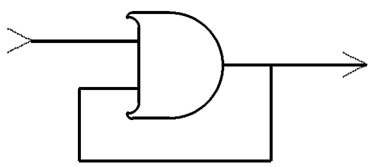
Atom

$or \in E \times E \Rightarrow E;$

Def

$SetOnce \in E \Rightarrow E;$
$SetOnce\ i = o$ **where** $o = or\ \langle i, o\rangle$ **endwhere**;

which has as structural interpretation:



Another good example is the description of the *reset-set-flipflop*:

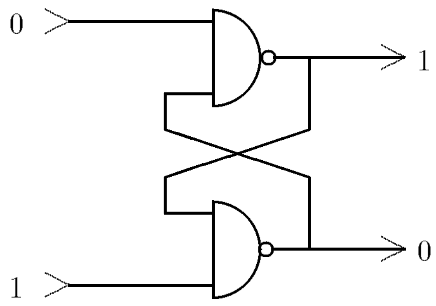
Atom

$nand \in E \times E \Rightarrow E$;

Def

$RSFF \in E^2 \Rightarrow E^2$;
$RSFF\ \langle R', S' \rangle = \langle q, q' \rangle$
where
$q = nand\ \langle S', q' \rangle$;
$q' = nand\ \langle R', q \rangle$
endwhere;

whose structural interpretation can be represented by this schematic:



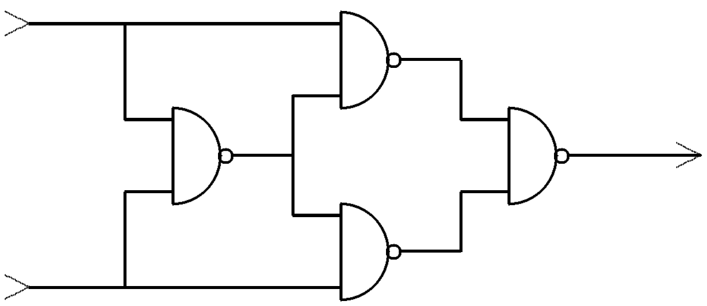
In a where clause local definitions must be separated by semicolons.

An example of expressing fanout with the where construct is the following somewhat strange implementation of a exclusive or:

Def

$my\_xor \in E \times E \Rightarrow E$;
$my\_xor\ \langle x, y \rangle = nand\ \langle nand\ \langle b, x \rangle, nand\ \langle b, y \rangle \rangle$
where $b = nand\ \langle x, y \rangle$ **endwhere**;

whose structural interpretation is represented by this schematic:



Later on we will learn other possibilities of using the **where**-construct.

3.2.5 Type naming

To save writing in type declarations one may introduce a name for a type:

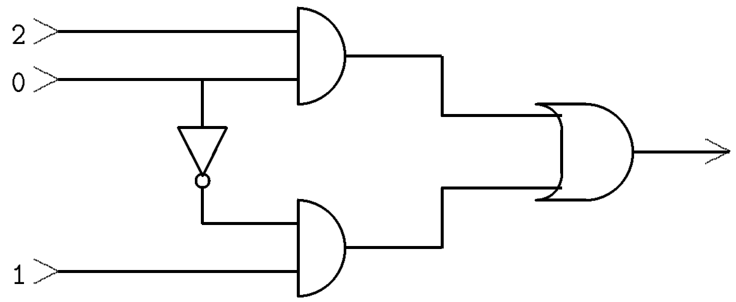
Type
$tp = E \times E \Rightarrow E$

One may then use this name as a synonym of the righthand side in other declarations, like $my\_xor \in tp$.

3.3 Some simple semantic functions

In this section we will introduce some very simple semantic functions. Remember however that neither semantic functions nor their implementation are part of the language. They only take Glass texts as input. Those interested in the actual implementation of the semantic functions are referred to later chapters of this book.

The first semantic function we introduce is *struct*, which yields the structural interpretation of a description. Now if we apply *struct* to our *select* description of the previous section we would derive its structure, represented here by its schematic:



Another semantic function *simplex* yields the truth table semantics of a description. So if we apply this semantic function to *select* it will generate a function of type $\mathbb{B}^3 \rightarrow \mathbb{B}$ with $\mathbb{B} = \{0, 1\}$, describing its truth table behaviour:

$$(simplex\ select)(0,1,0) = 1$$

A third one is the *cost* function, which yields the cost of a description based on a certain cost model. For instance if the cost model used is the number of gates in the circuit, then $cost\ select = 4$.

3.4 Adirectional and hybrid systems

3.4.1 Pure adirectional systems

In adirectional systems there is no clear flow of information. Such a system may have terminals connecting it to the outside world but you can not indicate any input or output. Think for instance of a resistor: it has two terminals to connect it with other components but current may flow into a terminal (and at the same time out of the other (Kirchhoff!)) and at next moment of time the current through the resistor may be reversed.

As all system definitions must contain a declaration we must be able to specify that a circuit is adirectional. With $[U]$ we mean the set of all adirectional systems having terminals of type U. So a resistor has type $[E \times E]$. Let us introduce some typical adirectional systems:

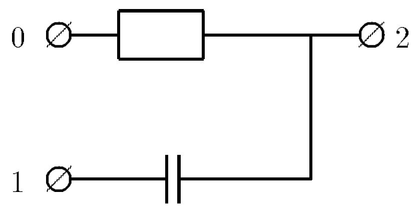
Basetype E;

Atom
$R \in [E \times E]$,
$C \in [E \times E]$,
$NPN \in [E \times E \times E]$; /* *emittor, base, collector* */

As there is no output in an adirectional system the right hand side of a composite system definition also gets another form. You decompose an adirectional system by making a set of all its constituents. We will call this set an *appset* (which has type **Appset**). In an appset connectivity is indicated by using common variables. Because you do not want to let all local connections be visible to the outside, names are bound within the appset unless they are already bound outside the appset (as formal parameter of the system for instance). Now let us make a simple example:

Def
$RCnet \in [E \times E \times E]$;
$RCnet\ \langle a, b, c\rangle = \{\ R\ \langle a, c\rangle,\ C\ \langle b, c\rangle\}$;

with the following structural interpretation:



In appsets systems applications, appsets and the so called synonym constructions may be used.

A synonym construction is the application of a special constructor indicated by a star ('$\star$'), which stands for the connection of terminals mentioned as the actual argument of that star (the connected terminals should of course be composed in the same way, i.e. have the same type). An example of this construct is the following:

Def
$RCnet2 \in [E \times E \times E]$;
$RCnet2\ \langle a, d, x\rangle = \{\ R\ \langle a, x\rangle,\ C\ \langle y, d\rangle,\ \star\ \langle x, y\rangle\}$;

which has the same *structural interpretation* as the previous description.

3.4.2 Supplies in descriptions

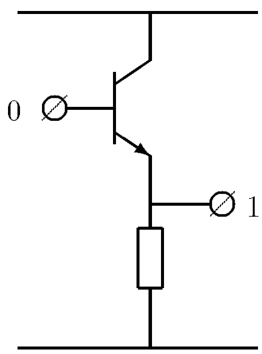
In analog systems components are often connected to one of the supplies. Think for instance of a NPN-transistor whose emittor is grounded. One way of describing connections to supplies is to declare supplies as atomic components having only one adirectional terminal

Atom
$Plus12V \in [E]$,
$Gnd \in [E]$;

An example then of their use is the following description of a one-transistor voltage-follower:

Def
$VoltageFollower \in [E \times E]$;
$VoltageFollower\ \langle in, out \rangle =$
$\{\ NPN\ \langle out, in, plus \rangle,$
$R\ \langle out, gnd \rangle,$
$Plus12V\ plus,$
$Gnd\ gnd \}$;

which has the following structural interpretation:



3.4.3 Nameless adirectional systems

As for directional systems it is possible to describe *nameless adirectional* systems. A nameless adirectional system is introduced by a σ-abstractor. With

$$\sigma\langle a, b\rangle.\{\ R\ \langle a, c\rangle,\ C\ \langle c, b\rangle\}$$

we mean an adirectional system with two terminals (i.e. having type $[E \times E]$) consisting of a resistor and a capacitor in series.

3.4.4 Partially adirectional systems

In this paragraph we will encounter circuits that are partially adirectional, that is, some of the terminals of a system are truly adirectional, whereas others serve as inputs or as outputs. For this purpose it is possible to specify that several terminals of such a system are directional by specifying that directionality in the type of the system. With $?U$ is meant the type of a input terminal of type U, and $!U$ stands for the type of an output terminal of type U. Consider for instance the declaration of an atomic monostable flipflop, which has one input, two terminals to connect it to an external timing capacitor, and one output:

Basetype E;

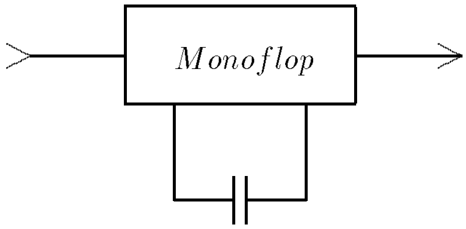
Atom
$Monoflop \in [?E \times E \times E \times !E]$;

Now how can you combine directional systems with such partial adirectional systems? The solution to this problem is that in Glass the type $U \Rightarrow V$ is equivalent to the type $[?U \times !V]$. Thus $E^2 \Rightarrow E$ is equivalent to $[?(E^2) \times !E]$. This equivalence may be propagated through products (and their shorthands): $?(U_1 \times U_2 \times \ldots \times U_n)$ is equivalent to $?U_1 \times ?U_2 \times \ldots \times ?U_n$. So $E^3 \Rightarrow E^2$ is equivalent to $[(?E)^3 \times (!E)^2]$.

In the same way you can decompose directional systems in a set of (partially) adirectional components. Consider for instance the following description of a delay element:

Def
$delay \in E \Rightarrow E$;
$delay \ \langle in, out \rangle = \{ \ Monoflop \ \langle in, x, y, out \rangle, C \ \langle x, y \rangle \}$;

which has the following structural interpretation, represented as schematic:



3.4.5 A nice hybrid one

Remark that the concepts introduced in the previous paragraph can also be used to describe systems that are directional to the outside world but internally are built with adirectional components. An example of such a system is for instance an amplifier: it has one input and one output, but internally it is built with adirectional elements like transistors, resistors, capacitors, etc.

Another example of such systems can be seen in the following description of an invertor in CMos technology:

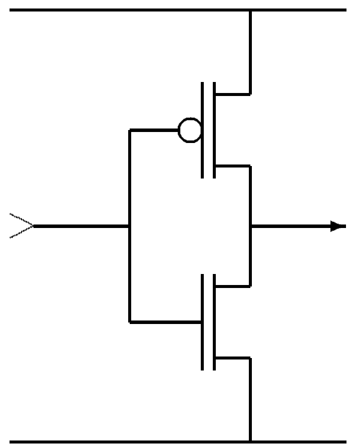
Basetype E;

Atom
$Nenh \in [E \times E \times E]$, /* *gate, source, drain* */
$Penh \in [E \times E \times E]$; /* *idem* */

Atom
$Vdd \in [E]$,
$Gnd \in [E]$;

Def
$CMosInvertor \in E \Rightarrow E$;
$CMosInvertor\ \langle in,\ out\rangle =$
$\{\ Penh\ \langle in,\ plus,\ out\rangle,$
$Nenh\ \langle in,\ gnd,\ out\rangle,$
$Vdd\ plus,$
$Gnd\ gnd\}$;

which has the following *structural interpretation*:



3.5 Macros

One of the design issues of the language was that the language should support the description of regular structures, as they are often encountered in VLSI design. With the language we know so far this is only possible by writing out every element of such a regular structure.

The language therefore contains a macro expansion mechanism, with which you can describe regular systems. It is also useful for obtaining compact descriptions. Now a macro describes only how to generate all the descriptions that make up a regular circuit. It does not have a meaning nor can a semantic function ascribe a meaning to it.

When using the Glass "describing environment", it is first checked if a Glass text is syntax/type correct. If so, the macro expander replaces all applications of macros by their expansions. That is, macro names applied to actual arguments are replaced by the body of the macro with the actual arguments filled in for the formal ones (after renaming to prevent name-clashes). This gives a text in a subset of the language. To texts belonging to this subset semantic functions may be applied.

3.5.1 A first expansion

We introduce a new type constructor namely the function arrow $\rightarrow$. With $U \rightarrow V$ we denote the set of all functions with U as domain and V as codomain. In the case of macros, it is the type of macros with a formal argument from U, giving a result in V after rewriting.

The reader is warned that he should not confuse the two arrows $\rightarrow$ and $\Rightarrow$. The first expresses a function type, whereas the second expresses a system type (and has higher precedence).

When expanding a macro of type $U \rightarrow V$, only the formal arguments of type U are replaced by their corresponding actual arguments. As macro expansion results in system descriptions, the result type of a macro is always of the form $U \Rightarrow V$ or $[U]$. Therefore the last parameter of the macro is of type U and stands for terminals. For this last parameter a λ/σ abstraction is generated by macro expansion.

Let us try a first example:

Basetype E;

Atom
$\quad$ $divide\_by\_two \in E \Rightarrow E$;

Mac
$\quad$ $triple \in E \Rightarrow E \rightarrow E \Rightarrow E$;
$\quad$ $triple\ A\ in = A\ (A\ (A\ in))$;

Def
$\quad$ $divide\_by\_8 \in E \Rightarrow E$;
$\quad$ $divide\_by\_8\ in = triple\ divide\_by\_two\ in$;

Now macro expansion (of *divide_by_8*) will rewrite *triple divide_by_two*. This results in:

$$\lambda in.(divide\_by\_two\ (divide\_by\_two\ (divide\_by\_two\ in)))$$

Thus *divide_by_8* turns out to be a cascade of three *divide_by_two* circuits.

3.5.2 Expanding one by one

The previous example is not quite satisfactory: it would be nicer if we had ways to describe how many times a circuit should be replicated, which necessitates us to introduce ways to calculate in Glass.

We therefore introduce some new types and constants of these types in Glass :

- **Int** (the set of integer numbers)
- **Float** (the set of floating point numbers)
- **String** (the set of character strings)
- **Bool** (the set of booleans)

The language also contains several well known predefined operators for these sets like $+$, $-$, $\times$, **Div** and **Mod**, etc, which are evaluated by the macro expander. As it must be possible to direct the expansion process dynamically the language has a conditional, whose syntax was adopted from SASL [Tur79]: with

$$b \rightarrow e_1; e_2$$

is meant that the macro expander has to evaluate the condition b; if the result hereof is true then the expander has to expand e_1 and else e_2. Using these constructs we can now give a more general description of a replicated system, which also shows the possibilities of using recursive macros:

Basetype E;

Atom
$\quad divide\_by\_two \in E \Rightarrow E;$

Mac
$\quad replicate \in \mathbf{Int} \rightarrow E \Rightarrow E \rightarrow E \Rightarrow E;$
$\quad replicate\ n\ A\ in = n = 0 \rightarrow in;\ \ A\ (replicate\ (n-1)\ A\ in);$

Def
$\quad divide\_by\_1024 \in E \Rightarrow E;$
$\quad divide\_by\_1024\ \ in = replicate\ 10\ divide\_by\_two\ \ in;$

3.5.3 Parametrized atoms

In section 3.4.1 it was suggested to declare a resistor as $R \in [E \times E]$. This is not very realistic, since one would like to have resistors of some specific value. The $\rightarrow$ and constant types provide us a way:

Atom $R \in \mathbf{Float} \rightarrow [E \times E]$

declares that R, applied to some floating point number (e.g. $R\,4.7$), is an adirectional atom of type $[E \times E]$.

These parametrized atoms are not macros, but are introduced here because they use the same notation.

3.5.4 An interlude on typing

3.5.4.1 On compound types

We may now be able to specify an arbitrary replication of systems, but what we can not yet do is describing macros that describe structures with an arbitrary set of terminals nor can we describe arbitrary interconnection patterns, which occur for instance in butterfly networks.

In order to understand how you describe such structures we must take a more precise look on the cartesian product. With $\Box n$ we will mean the set of integers $\{0, \ldots, n-1\}$. We will understand the product $U_0 \times \ldots \times U_{n-1}$ to be the set of functions $f \in \Box n \to \bigcup_{i \in \Box n} U_i$, such that $f\ i \in U_i$. This gives a way to select elements from a tuple. For example:

$$\begin{aligned}&\langle a, b, c\rangle\ 0 = a\\&\langle a, b, c\rangle\ 1 = b\\&\langle a, b, c\rangle\ 2 = c\end{aligned}$$

This definition also gives a meaning to E^1 and E^0. Note that this definition implies that $E^1 \neq E$.

Expressions of the form $\langle a_0, \cdots a_{n-1}\rangle i$ are also rewritten by the macro expander (to a_i), if i is in range.

Now suppose you have an expression a of type U and an expression b of type U^n. You may then combine them into an expression of type U^{n+1} by *cons*ing them: $a : b$. In general, $a_0 : \ \ldots \ : a_{n-1} : \langle\rangle$ is equivalent to $\langle a_0,\ \ldots,\ a_{n-1}\rangle$.

The language also contains possibilities to introduce parametrized types, although that parameter is restricted to the domain of the integers. By writing **Int** $\ni n$ in a type you introduce n as a type parameter, which may then be used to specify the sizes of the constituents in the rest of the type. Thus one may specify

$$\mathbf{Int} \ni n \to E^n \times E^n \Rightarrow E^n$$

A macro of this type yields a system taking two n-tuples as input and yielding one n-tuple as output.

Type checking is a necessary part of the Glass "describing environment". Since it is not decidable (statically) if a type U^n is of exactly the right size (because of the **Int** $\ni n$ types), all types of the form U^n are treated in the same way during a first typecheck, namely as types for lists of undetermined size. After macro expansion it is checked if they are of exactly the right size.

Since U^n and the n-fold product $U \times \ldots \times U$ are equivalent, the latter type is treated in the same way (So it may happen that if you use *or* $\langle a, b, c\rangle$, the fact that there is one input too much will be discovered later than you expected).

3.5.4.2 Type variables

As you may have noticed, some macro definitions are insensitive to the exact form of (part of) their argument. Take for example *triple* (section 3.5.1): this macro definition is perfectly valid whatever the exact form of A is, as long as its inputs and outputs are of the same type (because these are connected in series). One is therefore allowed to introduce so called em type variables. By

$$\{U\}\ V$$

we mean any type V in which U is replaced by an arbitrary type (for a connection). Therefore one is allowed to declare *triple* in the following way:

$$triple \in \{U\}\ U \Rightarrow U \rightarrow U \Rightarrow U$$

This saves defining several variants of *triple*, that differ only in the type. In this way one may use *triple halfadder* and *triple divide_by_two*.

3.5.5 Using patterns

In order to enable case distinction in a more elegant way the language contains pattern matching facilities for macros. Macros may consist of several alternatives which are tried from top to bottom at expansion time. The first alternative whose formal arguments match the actual ones is then expanded. Patterns may be:

- a name, matching everything.
- a constant, matching only if the actual argument equals that constant after its expansion.
- the empty tuple $\langle\rangle$ matching only the empty tuple.
- a tuple $\langle f_1, \ldots, f_n\rangle$, matching of the actual argument has the form $\langle a_1, \ldots, a_n\rangle$ and if f_i matches a_i.
- a consing $a : b$, matching only if the actual argument has the form $c : d$, and a matches c and b matches d.

We must keep in mind, that the actual argument, after expansion, must have precisely the required syntactic form. Using pattern matching (on the integer argument) we are now able to specify an adder chain:

Basetype E;

Atom

$adc \in E \times E \times E \Rightarrow E \times E$;

Mac

$nbitsadder \in \mathbf{Int} \ni n \rightarrow E^n \times E^n \times E \Rightarrow E \times E^n$;
$nbitsadder\ 0\ \langle\langle\rangle, \langle\rangle, c\rangle = \langle c, \langle\rangle\rangle$;
$nbitsadder\ n\ \langle a : as, b : bs, c_{in}\rangle = \langle c_{out},\ s : ss\rangle$
where
$\langle v, s\rangle = adc\ \langle a, b, c_{in}\rangle$;
$\langle c_{out}, ss\rangle = nbitsadder\ (n-1)\ \langle as, bs, v\rangle$;
endwhere;

Def

$Fourbitsadder \in E^4 \times E^4 \times E \Rightarrow E \times E^4$;
$Fourbitsadder\ \langle as,\ bs,\ c_{in}\rangle = nbitsadder\ 4\ \langle as,\ bs,\ c_{in}\rangle$;

In the expansion (of *Fourbitsadder*) the second alternative of *nbitsadder* is used 4 times (for arguments 4, 3, 2 and 1: these do not match the formal argument 0). Finally the first alternative is used (because the actual and formal argument are both 0).

3.5.6 A larger example

A nice example of the application of (local) macros is the following description of a four-bit counter:

Atom

$jkff \in B \times B \Rightarrow B$,
$and2 \in B \times B \Rightarrow B$,
$and3 \in B^3 \Rightarrow B$,
$and4 \in B^4 \Rightarrow B$,
$and5 \in B^5 \Rightarrow B$,
$or \in B \times B \Rightarrow B$,
$buf \in B \Rightarrow B$,
$nand \in B \times B \Rightarrow B$;

Def

$counter \in B^8 \Rightarrow B^5$;
$counter\langle d_3, d_2, d_1, d_0, nload, nclear, enap, enat\rangle =$
$\langle carry, q_3, q_2, q_1, q_0\rangle$
where
$carry = and5\ \langle q_3, q_2, q_1, q_0, enat\rangle$;
$q_3 = counterstage\ \langle d_3, preset, and4\ \langle q_2, q_1, q_0, en\rangle, nclr\rangle$;
$q_2 = counterstage\ \langle d_2, preset, and3\ \langle q_1, q_0, en\rangle, nclr\rangle$;
$q_1 = counterstage\ \langle d_1, preset, and2\ \langle q_0, en\rangle, nclr\rangle$;
$q_0 = counterstage\ \langle d_0, preset, en, nclr\rangle$;

$preset = nand\ \langle nload,\ nclr \rangle;$
$nclr = buf\ nclear;$
$en = and2\ \langle enap, enat \rangle;$
Mac
  $counterstage \in B^4 \Rightarrow B;$
  $counterstage\ \langle d, preset, toggle, nclr \rangle =$
    $jkff\ \langle and2 \langle ja, ea \rangle, and2\ \langle ka, ea \rangle \rangle$
    where
      $ja = nand\ \langle ka, preset \rangle;$
      $ka = nand\ \langle d, nclr \rangle;$
      $ea = or\ \langle toggle, preset \rangle;$
    endwhere;
endwhere;

whose structural interpretation can be found in figure 2.1.

Note that the **where**-clause is now used for two purposes: to give names to outputs and to introduce local macros.

3.6 Remarks

"I can explain all the poems, that ever were invented – and a good many that haven't been invented just yet"
Through the Looking-glass *Lewis Carroll*

Having read this chapter we hope that the reader now has an idea of the possibilities of the language Glass. We warn however that this chapter is not intended to be a language definition nor is it intended to fully cover every bit of the language. We hope that the reader can now better understand the hows and whys of some language constructs and make a good start in writing his own Glass descriptions.

Chapter 4

Simple semantic functions

M. Seutter

4.1 Introduction

In the following chapters we will investigate the theoretic side of the implementation of semantic functions. As we already explained in a previous chapter semantic functions map descriptions in Glass (i.e. Glass syntax) into descriptions in particular domains of interest. As syntax is very important in describing these semantic functions we will introduce a specific notation to indicate what parts of the description is Glass syntax and what is not. The two special brackets '⟦' and '⟧' will be placed around Glass syntax or Glass syntactical constructs in the tradition of denotational semantics.

Semantic functions may be classified in two classes, namely the compositional ones and the non compositional ones. Compositional semantic functions compose the properties of compound system definitions from the properties of the subsystems. This process is carried on in a recursive way. For those subsystems that are not compound, but are primitive (atomic systems) the semantic function yields predefined semantics. Non compositional semantic functions do not comply to this way of evaluation.

We will describe the semantic functions initially as mappings from Glass syntax to Funmath. The domain of interest will therefore also be described in Funmath. Later on we will also indicate how one transforms these mappings into mappings from Glass syntax to programs in existing programming languages. Of these latter we will consider especially mappings towards the functional language Miranda and the imperative language C.

The first of these is interesting because one can do a one-to-one translation from the mapping towards Funmath into the mapping towards Miranda. The second is interesting because much efficiency can be gained if one can map into an imperative language, due to the fact that really efficient implementations of lazy functional languages still do not exist.

We will now focus on compositional semantic functions for directional system descriptions. In general they all conform to the following generic model, presented in [Bou88].

4.2 The generic model

One can express a semantic function by giving rules for every syntactic construct in the directional subpart of Glass how properties must be propagated from those of the constituting parts. In fact what we define in this section is the specification of a transducer from Glass syntax to Funmath syntax. The semantic functions defined in the following sections and chapters all adhere to these rules. The only difference between them lies in the definition of the properties of the atoms.

In this generic model we will consider only a small subset of Glass namely that subpart of the language which describes directional system descriptions without local definitions or system parameters. In the next chapter we will try and amend this assumption.

Let $\mathcal{D}$ be the domain of interpretation and $\mathcal{A}$ the set of the atoms. The interpretation of the atoms then takes as type:

$$\tau_A \in \mathcal{A} \rightarrow \mathcal{D}_A \qquad \text{where } \mathcal{D}_A = \bigcup_{n \in \mathbb{N}} (\mathcal{D}^n \rightarrow \mathcal{D})$$

To be more specific, for an atom $c \in \mathcal{A}$ of type $\mathbf{E}^n \Rightarrow \mathbf{E}$ (having $\mathbf{E}$ as base-type):

$$\tau_A \, [\![c]\!] \in \mathcal{D}^n \rightarrow \mathcal{D}$$

In the following rules $\mathcal{E}$ and $\mathcal{F}$ stand for the set of (syntactic) correct Glass expressions and system definitions, respectively. The set $\mathcal{I}$ is a set of mappings from Glass variables to Funmath variables. In the following description $i \in \mathcal{I}$ is used to carry the environment in the transduction.

Def

$$\begin{array}{ll}
\tau_E \in \mathcal{E} \rightarrow \mathcal{I} \rightarrow \mathcal{D} & \\
\tau_E \, [\![v]\!] \; i = i \; v & \text{if v is a variable} \\
\tau_E \, [\![f \; e]\!] \; i = \tau_F \, [\![f]\!] \; i \; (\tau_E \, [\![e]\!] \; i) & \\
\tau_E \, [\![\langle e_0, e_1, \ldots, e_{n-1} \rangle]\!] \; i = (\tau_E \, [\![e_0]\!] \; i, \ldots, (\tau_E \, [\![e_{n-1}]\!] \; i) &
\end{array}$$

Def

$$\begin{array}{ll}
\tau_F \in \mathcal{F} \rightarrow \mathcal{I} \rightarrow \mathcal{D}_A & \\
\tau_F \, [\![c]\!] \; i = \tau_A \, [\![c]\!] & \text{if } c \in \mathcal{A} \\
\tau_F \, [\![\lambda \, t.e]\!] \; i \in \mathcal{D} \rightarrow \mathcal{D} & \text{with} \\
\tau_F \, [\![\lambda \, t.e]\!] \; i \; d = \tau_E \, [\![e]\!] \;\; i[\,^d/_t] & \\
\tau_F \, [\![\lambda \, \langle t_0, t_1, \ldots, t_{n-1} \rangle.e]\!] \; i \in \mathcal{D}^n \rightarrow \mathcal{D} \qquad \text{with} & \\
\tau_F \, [\![\lambda \, \langle t_0, t_1, \ldots, t_{n-1} \rangle.e]\!] \; i \; (d_0, d_1, \ldots, d_{n-1}) = & \\
\qquad \tau_E \, [\![e]\!] \;\; i[\,^{d_0}/_{t_0}] \ldots [\,^{d_{n-1}}/_{t_{n-1}}] &
\end{array}$$

where we use the following function to update the environment:

Def

$[\,{}^{d}/_{t}] \in \mathcal{I} \to \mathcal{I}$

$i\,[\,{}^{d}/_{t}]\ t' = \ (t' = t)?d; i\ t'$

Remark that we differentiate in the transduction of a system definition having only one formal parameter and the transduction of one having a tuple of formal parameters. We must do so because in Glass these are two different syntactical forms.

The meaning of a specific system definition may then be found as follows:

Def

$\tau_S \in \mathcal{F} \to \mathcal{D}_A$

$\tau_S\ [\![s]\!] = \tau_F\ [\![s]\!]\ i$ where i may be any interpretation

4.3 The simplex model

One of the easiest semantic functions to implement is the semantic function *simplex*, which yields the static (truth table) semantics of a description. As domain of interest we take the set $\mathbb{B} = \{0, 1\}$. For this particular semantic function we will extensively show how it corresponds to the generic model. However we will not repeat this exercise for the other semantic functions presented in other sections of this chapter. This can be done in a quite analog and straightforward way.

First we introduce the interpretations of our atoms in our domain of interest:

Def

$s\_not \in \mathbb{B} \to \mathbb{B}$

$s\_not\ 0 = 1$

$s\_not\ 1 = 0$

Def

$s\_and \in \mathbb{B}^2 \to \mathbb{B}$

$s\_and\ (0, k) = 0$

$s\_and\ (k, 0) = 0$

$s\_and\ (1, 1) = 1$

Def

$s\_nand \in \mathbb{B}^2 \to \mathbb{B}$

$s\_nand\ (0, k) = 1$

$s\_nand\ (k, 0) = 1$

$s\_nand\ (1, 1) = 0$

Def

$s\_or \in \mathbb{B}^2 \to \mathbb{B}$

$s\_or\ (1, k) = 1$

$s\_or\ (k, 1) = 1$

$s\_or\ (0, 0) = 0$

Def

$s\_nor \in \mathbb{B}^2 \to \mathbb{B}$

$s\_nor\ (1, k) = 1$

$s\_nor\ (k, 1) = 1$

$s\_nor\ (0, 0) = 0$

Def

$$s\_xor \in \mathbb{B}^2 \to \mathbb{B}$$
$$s\_xor\ (0, k) = k$$
$$s\_xor\ (1, k) = s\_not\ k$$

Def

$$s\_xnor \in \mathbb{B}^2 \to \mathbb{B}$$
$$s\_xnor\ (1, k) = k$$
$$s\_xnor\ (0, k) = s\_not\ k$$

We have given explicite definitions for the above functions to indicate the difference with the semantic function presented in the next section. However one can easily verify that:

$$s\_not\ x = \neg\ x$$
$$s\_and\ (x, y) = x \wedge y$$
$$s\_or\ (x, y) = x \vee y$$
$$s\_xor\ (x, y) = x \oplus y$$
$$s\_xnor\ (x, y) = (x = y)$$

using the normal logical operators.
We then define:

$$\tau_A\ [\![not]\!] = s\_not$$
$$\tau_A\ [\![and]\!] = s\_and$$
$$\tau_A\ [\![nand]\!] = s\_nand$$
$$\tau_A\ [\![or]\!] = s\_or$$
$$\tau_A\ [\![nor]\!] = s\_nor$$
$$\tau_A\ [\![xor]\!] = s\_xor$$
$$\tau_A\ [\![xnor]\!] = s\_xnor$$

Let us consider the following description of a simple selector:

Def

$$hsel \in \mathbf{E} \times \mathbf{E} \times \mathbf{E} \Rightarrow \mathbf{E};$$
$$hsel\ \langle s, x_0, x_1 \rangle = or\ \langle and\ \langle not\ s, x_0 \rangle, and\ \langle s, x_1 \rangle \rangle$$

We can now express the meaning of our selector description by application of the generic model:

$$
\begin{aligned}
&(simplex\ [\![hsel]\!]) \in \mathbb{B}^3 \to \mathbb{B}\\
&(simplex\ [\![hsel]\!])\ \ (d, d_0, d_1) =\\
&\quad \tau_F\ [\![\lambda(s, x_0, x_1).or\ (and\ (not\ s, x_0), and\ (s, x_1))]\!]\ \ i\ \ (d, d_0, d_1) =\\
&\quad \tau_E\ [\![or\ (and\ (not\ s, x_0), and\ (s, x_1))]\!]\ \ i[\ {}^{d}/_{s}][\ {}^{d_0}/_{x_0}][\ {}^{d_1}/_{x_1}] =\\
&\quad \tau_E\ [\![or\ (and\ (not\ s, x_0), and\ (s, x_1))]\!]\ \ i' =\\
&\quad \tau_F\ [\![or]\!]\ i'\ \ (\tau_E\ [\![and\ (not\ s, x_0)]\!]\ i', \ldots, \tau_E\ [\![and\ (s, x_1)]\!]\ i') =\\
&\quad \tau_A\ [\![or]\!]\ \ (\tau_E\ [\![and\ (not\ s, x_0)]\!]\ i', \ldots, \tau_E\ [\![and\ (s, x_1)]\!]\ i') =\\
&\quad s\_or\ (\tau_E\ [\![and\ (not\ s, x_0)]\!]\ i', \ldots, \tau_E\ [\![and\ (s, x_1)]\!]\ i') =\\
&\quad s\_or\ (\tau_F\ [\![and]\!] i'\ \ (\tau_E\ [\![not\ s]\!]\ i', \tau_E\ [\![x_0]\!]\ i'), \tau_F\ [\![and]\!] i'\ \ (\tau_E\ [\![s]\!]\ i', \tau_E\ [\![x_1]\!]\ i')) =\\
&\quad s\_or\ (\tau_A\ [\![and]\!]\ \ (\tau_E\ [\![not\ s]\!]\ i', \tau_E\ [\![x_0]\!]\ i'), \tau_A\ [\![and]\!]\ \ (\tau_E\ [\![s]\!]\ i', \tau_E\ [\![x_1]\!]\ i')) =\\
&\quad s\_or\ (s\_and\ (\tau_E\ [\![not\ s]\!]\ i', \tau_E\ [\![x_0]\!]\ i'), s\_and\ (\tau_E\ [\![s]\!]\ i', \tau_E\ [\![x_1]\!]\ i')) =\\
&\quad s\_or\ (s\_and\ (\tau_F\ [\![not]\!]\ i'\ (\tau_E\ [\![s]\!]\ i'), d_0), s\_and\ (d, d_1)) =\\
&\quad s\_or\ (s\_and\ (\tau_A\ [\![not]\!]\ (\tau_E\ [\![s]\!]\ i'), d_0), s\_and\ (d, d_1)) =\\
&\quad s\_or\ (s\_and\ (s\_not\ (\tau_E\ [\![s]\!]\ i'), d_0), s\_and\ (d, d_1)) =\\
&\quad s\_or\ (s\_and\ (s\_not\ d, d_0), s\_and\ (d, d_1))
\end{aligned}
$$

where we have substituted

$$
\begin{array}{l}
i' \in \mathcal{I} \\
i' = i\ [\ ^{d}/_{s}][\ ^{d_0}/_{x_0}][\ ^{d_1}/_{x_1}]
\end{array}
$$

We can use this result to study the static behaviour of this system. For instance:

$$(\mathit{simplex}\ [\![\mathit{hsel}]\!])\ \ (0,1,0) = 1$$

One can easily implement a semantic function which will yield corresponding executable definitions in a programming language like C or Miranda.

4.4 Eichelberger algebra

Another easy to implent semantic function is a variant on the semantic function *simplex*, based on the Eichelberger algebra [Eic65]. We will denote this semantic function by *eich*.

Our domain of interpretation in this model is the set $\mathbb{E} = \{0, \perp, 1\}$, where $\perp$ stands for "undefined value". As previously, we must first specify the properties of our atoms:

Def

$$
\begin{array}{ll}
\mathit{e\_not} \in \mathbb{E} \rightarrow \mathbb{E} & \qquad \tau_A\ [\![\mathit{not}]\!] = \mathit{e\_not} \\
\mathit{e\_not}\ 0 = 1 & \\
\mathit{e\_not}\ 1 = 0 & \\
\mathit{e\_not}\ \perp = \perp &
\end{array}
$$

Def

$$
\begin{array}{ll}
\mathit{e\_and} \in \mathbb{E}^2 \rightarrow \mathbb{E} & \qquad \tau_A\ [\![\mathit{and}]\!] = \mathit{e\_and} \\
\mathit{e\_and}\ (1,1) = 1 & \\
\mathit{e\_and}\ (0,k) = 0 & \\
\mathit{e\_and}\ (k,0) = 1 & \\
\mathit{e\_and}\ (k,l) = \perp &
\end{array}
$$

Def

$$
\begin{array}{ll}
\mathit{e\_or} \in \mathbb{E}^2 \rightarrow \mathbb{E} & \qquad \tau_A\ [\![\mathit{or}]\!] = \mathit{e\_or} \\
\mathit{e\_or}\ (0,0) = 0 & \\
\mathit{e\_or}\ (1,k) = 1 & \\
\mathit{e\_or}\ (k,1) = 1 & \\
\mathit{e\_or}\ (k,l) = \perp &
\end{array}
$$

Def

$$e\_or3 \in \mathbb{E}^3 \to \mathbb{E} \qquad \tau_A \; [\![or3]\!] = e\_or3$$
$$e\_or3\;(0,0,0) = 0$$
$$e\_or3\;(1,l,m) = 1$$
$$e\_or3\;(k,1,m) = 1$$
$$e\_or3\;(k,l,1) = 1$$
$$e\_or3\;(k,l,m) = \perp$$

Let us consider a slightly changed version of the selector of the previous section:

Def

$$ssel \in \mathbf{E} \times \mathbf{E} \times \mathbf{E} \Rightarrow \mathbf{E};$$
$$ssel\;\langle s, x_0, x_1\rangle = or3\;\langle and\;\langle not\; s, x_0\rangle, and\;\langle s, x_1\rangle, and\;\langle x_0, x_1\rangle\rangle$$

This semantic function also conforms to our generic model. If we apply *eich* to our selectors *hsel* and *ssel* one deduces that:

$$(eich\;[\![hsel]\!]) \in \mathbb{E}^3 \to \mathbb{E}$$
$$(eich\;[\![hsel]\!])\;(d, d_0, d_1) = e\_or\;(e\_and\;(e\_not\; d, d_0), e\_and\;(d, d_1))$$

$$(eich\;[\![ssel]\!]) \in \mathbb{E}^3 \to \mathbb{E}$$
$$(eich\;[\![ssel]\!])\;(d, d_0, d_1) =$$
$$e\_or3\;(e\_and\;(e\_not\; d, d_0), e\_and\;(d, d_1), e\_and\;(d_0, d_1))$$

From this we can derive that:

$$(eich\;[\![hsel]\!])\;(\perp, 1, 1) =$$
$$e\_or\;(e\_and\;(e\_not\;\perp, 1), e\_and\;(\perp, 1)) =$$
$$e\_or\;(\perp, \perp) =$$
$$\perp$$

and that:

$$(eich\;[\![ssel]\!])\;(\perp, 1, 1) =$$
$$e\_or3\;(e\_and\;(e\_not\;\perp, 1), e\_and\;(\perp, 1), e\_and\;(1,1)) =$$
$$e\_or3\;(\perp, \perp, 1) =$$
$$1$$

Thus we have shown that $eich\;[\![hsel]\!] \neq eich\;[\![ssel]\!]$, although it can be shown using ordinary boolean algebra that $simplex\;[\![hsel]\!] = simplex\;[\![ssel]\!]$. This observation is related to the fact that *ssel* is safe with respect to static hazards and *hsel* is not.

4.5 Some very simple cost models

When designing hardware it is advisable to have some measure of the cost of the design. If our design will be realized in VLSI the cost of the design must be measured by the chip area needed for the design. To calculate an estimate of this area one must place the subsystems of a design on the silicium in an optimal way. This latter is unfortunately a highly uncompositional and very time consuming process.

There are also other models with which we may obtain some measure of the cost of a design. We may consider for instance the number of gates in a design. These kind of models do comply to a compositional model although they do not comply to the generic model.

As stated when we defined the generic model we must provide rules for every syntactic construct. In the following rules we use $\mathbb{R}_0^+ = \{x \in \mathbb{R} | x \geq 0\}$. Define:

$$\begin{array}{l} \tau_A \in \mathcal{A} \rightarrow \mathbb{R}_0^+ \\ \tau_A \llbracket c \rrbracket = \text{the cost of the atom } c \end{array}$$

Likewise we define the other rules:

$$\begin{array}{ll} \tau_E \in \mathcal{E} \rightarrow \mathbb{R}_0^+ & \\ \tau_E \llbracket v \rrbracket = 0 & \text{if } v \text{ is a variable} \\ \tau_E \llbracket f\ e \rrbracket = \tau_F \llbracket f \rrbracket + \tau_E \llbracket e \rrbracket & \\ \tau_E \llbracket \langle e_0, \ldots, e_{n-1} \rangle \rrbracket = \sum\limits_{i=0}^{n-1} \tau_E \llbracket e_i \rrbracket & \end{array}$$

$$\begin{array}{ll} \tau_F \in \mathcal{F} \rightarrow \mathbb{R}_0^+ & \\ \tau_F \llbracket c \rrbracket = \tau_A \llbracket c \rrbracket & \text{if } c \in \mathcal{A} \\ \tau_F \llbracket \lambda t.e \rrbracket = \tau_E \llbracket e \rrbracket & \\ \tau_F \llbracket \lambda \langle t_0, \ldots, t_{n-1} \rangle .e \rrbracket = \tau_E \llbracket e \rrbracket & \end{array}$$

If we are interested in the number of gate in a design we must define that $\tau_A \llbracket c \rrbracket = 1$ for every gate c. The above rules will then yield the number of gates in a system. We might refine this a bit by giving each atom its appropriate cost in which case the above rules will yield the sum of the costs of all gates in the design.

A special case of this can be defined is we take the cost of a gate to be the number of inputs it has. It is this cost model that forms the basis of the optimizing processes using Karnaugh diagrams or Quine- McCluskey techniques which we may use to reduce the number of literals (i.e. the number of inputs in the circuit) in a sum of products.

Chapter 5

Discrete timing function models

M. Seutter, KUN

5.1 A taxonomy

Our previous models concerning behaviour only handled static behaviour. The static behaviour of a system is oft quite trivial. For other systems it may even not exist, for instance systems that contain feedback do not exhibit any static behaviour. In this chapter (and later ones) we will take a closer look on semantic functions handling dynamic behaviour and the corresponding models.

One abstracted form of dynamic behaviour is of particular interest if one is interested in the values of the inputs and outputs at fixed unidistant instants in time. This model is especially usefull for designing synchronous sequential circuits, whose behaviour is governed by the ticks of the external clock and thus only change state at certain fixed moments in time.

We will denote our value domain by $\mathcal{V}$. One may then describe the domain of interest in two ways

a) as *streams*, that is we use as domain of interest $\mathcal{D} = \mathcal{V}^*$, with the convention that succession in the sequence corresponds to succession in time. We will call this model the *sequence* model.

b) as *discrete time functions*. Our domain of interest then becomes $\mathcal{D} = \mathbb{N} \to \mathcal{V}$ with the convention that the value of signal x at time t is $x\ t$. This model will be called the *discrete time function* model.

5.2 Some new atoms

As we are going to design some synchronous sequential circuits we must introduce some new atomic components, namely the flipflops. In the coming sections we will assume that their clock input is an implicit input which 'ticks' at every new signal instance.

Because we will use some new atomic components we must include their declaration in the Glass text:

Atom

$$dff \in \mathbb{B} \to \mathbf{E} \Rightarrow \mathbf{E},$$
$$jkff \in \mathbb{B} \to \mathbf{E}^2 \Rightarrow \mathbf{E},$$
$$tff \in \mathbb{B} \to \mathbf{E} \Rightarrow \mathbf{E};$$

We will use the extra value parameter to denote the initial state of the flipflops. As we are also going to allow feedback loops in our descriptions our semantic functions must be able to cope with local definitions. Our generic model must therefore be extended to handle this extra syntax, which can be done in quite a straightforward way.

5.3 The sequence model

We will use as value domain $\mathcal{V} = \mathbb{B}$, which incurs that our domain of interpretation is $\mathcal{D} = \mathbb{B}^*$. We will denote this semantic functions by *seq*.

We start by defining the properties of our atoms:

Def

$$seq\_not \in \mathbb{B}^* \to \mathbb{B}^* \qquad \tau_A \llbracket not \rrbracket = seq\_not$$
$$seq\_not\ [] = []$$
$$seq\_not\ (a : d) = s\_not\ a : seq\_not\ d$$

Def

$$seq\_and \in (\mathbb{B}^*)^2 \to \mathbb{B}^* \qquad \tau_A \llbracket and \rrbracket = seq\_and$$
$$seq\_and\ (k, []) = []$$
$$seq\_and\ ([], k) = []$$
$$seq\_and\ (a : d, b : f) = s\_and\ (a, b) : seq\_and\ (d, f)$$

We can define the properties of the other atomic combinatorial components (ports) in the same way. We must also define the properties of our flipflops:

Def

$$seq\_dff \in \mathbb{B} \to \mathbb{B}^* \to \mathbb{B}^* \qquad \tau_A \llbracket dff \rrbracket = seq\_dff$$
$$seq\_dff\ q_0\ l = q_0 : l$$

or alternatively:

Def

$$seq\_dff \in \mathbb{B} \to \mathbb{B}^* \to \mathbb{B}^*$$
$$seq\_dff\ q_0\ [] = [q_0]$$
$$seq\_dff\ q_0\ (a : l) = q_0 : seq\_dff\ a\ l$$

Although these definitions seem different, they effectivily define the same function. The second definition expresses more precisely what the effect is of the tick of the clock by indicating what the next state of the flipflop is. The first definition expresses that a D-flipflop actually performs a delay of one tick of the clock. Later in this chapter we will give a proof of the equivalence for the corresponding discrete time function models.

The other two flipflops can be defined in the same way:

Def

$$\begin{array}{ll} seq\_jkff \in \mathbb{B} \rightarrow (\mathbb{B}^*)^2 \rightarrow \mathbb{B}^* & \tau_A \, [\![jkff]\!] = seq\_jkff \\ seq\_jkff \; q_0 \; ([], k) = [q_0] & \\ seq\_jkff \; q_0 \; (k, []) = [q_0] & \\ seq\_jkff \; q_0 \; (j : p, k : q) = q_0 : seq\_jkff \; newq \; (p, q) & \\ \qquad\qquad \mathbf{where} \;\; newq = \;\; (q_0 = 0)?j; \neg\, k & \end{array}$$

Def

$$\begin{array}{ll} seq\_tff \in \mathbb{B} \rightarrow \mathbb{B}^* \rightarrow \mathbb{B}^* & \tau_A \, [\![tff]\!] = seq\_tff \\ seq\_tff \; q_0 \; [] = [q_0] & \\ seq\_tff \; q_0 \; (t : p) = q_0 : seq\_tff \; newq \; p & \\ \qquad\qquad \mathbf{where} \;\; newq = \;\; (t = 0)?q_0; \neg\, q_0 & \end{array}$$

As the sequence model also conforms to the (revised) generic model we can immediately derive the semantic function *seq* from these properties. Suppose we have the following Glass description of a parity checker:

Def

$$\begin{array}{l} parity \in \mathbf{E} \Rightarrow \mathbf{E}; \\ parity \; x = y \; \mathbf{where} \; y = \, dff \; 0 \; (xor \; \langle x, y \rangle); \; \mathbf{endwhere} \end{array}$$

By applying *seq* to this description we obtain:

Def

$$\begin{array}{l} (seq \; parity) \in \mathbb{B}^* \rightarrow \mathbb{B}^* \\ (seq \; parity) \; x = y \; \mathbf{where} \; y = seq\_dff \; 0 \; (seq\_xor \; (x, y)) \end{array}$$

Thus

$$(seq \; parity) \; [1, 0, 1, 1] = [0, 1, 1, 0, 1]$$

Although these definitions appear to be very lucid there are some disadvantages if one tries to implement this model in a functional programming language like Miranda. If one uses the second definition of *seq_dff*, the Miranda implementation fails, because it needs to know that y is nonempty in order to correctly do the pattern matching to select the second alternative of the definition of *seq_dff* which will then evaluate the first element of y. In Miranda terms: these definitions constitute a so called “Black Hole”. Likewise problems occur when using the other types of flipflops. Because of these implementation problems we did not pursue the implementation of the sequence model any further and turned towards the discrete time function model.

5.4 The discrete time function model

In this model our value domain will remain $\mathcal{V} = \mathbb{B}$ (although one can quite easily adept the model to other value domains like $\mathbb{E}$). Thus the domain of interest becomes $\mathcal{D} = \mathbb{N} \to \mathbb{B}$. First we define the properties of the atoms:

Def

$$dtm\_not \in (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B} \qquad \tau_A \, [\![not]\!] = dtm\_not$$
$$dtm\_not \; x \; n = s\_not \; (x \; n)$$

Def

$$dtm\_and \in (\mathbb{N} \to \mathbb{B})^2 \to \mathbb{N} \to \mathbb{B} \qquad \tau_A \, [\![and]\!] = dtm\_and$$
$$dtm\_and \; (x, y) \; n = s\_and \; (x \; n, y \; n)$$

Def

$$dtm\_xor \in (\mathbb{N} \to \mathbb{B})^2 \to \mathbb{N} \to \mathbb{B} \qquad \tau_A \, [\![xor]\!] = dtm\_xor$$
$$dtm\_xor \; (x, y) \; n = s\_xor \; (x \; n, y \; n)$$

Our gates therefore have no delay in this model. We define the D-flipflop:

Def

$$dtm\_dff \in \mathbb{B} \to (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$$
$$dtm\_dff \; q_0 \; d \; 0 = q_0$$
$$dtm\_dff \; q_0 \; d \; n = d \; (n - 1)$$

or alternatively using an auxilary function:

Def

$$tail \in (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$$
$$tail \; d \; n = d \; (n + 1)$$

Def

$$dtm\_dff \in \mathbb{B} \to (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$$
$$dtm\_dff \; q_0 \; d \; 0 = q_0$$
$$dtm\_dff \; q_0 \; d \; n = dtm\_dff \; (d \; 0) \; (tail \; d) \; (n - 1)$$

We define the other two flipflops in the same way:

Def

$$dtm\_jkff \in \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^2 \to \mathbb{N} \to \mathbb{B}$$
$$dtm\_jkff \; q_0 \; (j, k) \; 0 = q_0$$
$$dtm\_jkff \; q_0 \; (j, k) \; n = dtm\_jkff \; new \; (tail \; j, tail \; k) \; (n - 1)$$
$$\textbf{where } new = \; (q_0 \; = 0) ? j \; 0 ; \neg \; (k \; 0)$$

Def

$$dtm\_tff \in \mathbb{B} \to (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$$
$$dtm\_tff \; q_0 \; t \; 0 = q_0$$
$$dtm\_tff \; q_0 \; t \; n = dtm\_tff \; new \; (tail \; t) \; (n - 1)$$
$$\textbf{where } new = \; (t \; 0 = 0) ? q_0 ; \neg \; q_0$$

with the usual rules that:

$$\tau_A \; [\![dff]\!] = dtm\_dff$$
$$\tau_A \; [\![jkff]\!] = dtm\_jkff$$
$$\tau_A \; [\![tff]\!] = dtm\_tff$$

Again we can use the conformity to our generic model to derive the corresponding semantic function, which we will denote by *dtm*. Thus:

Def

$$(dtm \; [\![parity]\!]) \in (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$$
$$(dtm \; [\![parity]\!]) \; i = o \;\textbf{where}\; o = dtm\_dff \; 0 \; (dtm\_xor \; (i, o))$$

If one tries to implement this model in a functional programming language like Miranda, one does not encounter problems in the definitions because of the lazy evaluation mechanism in the language implementation. The only remaining disadvantage is that this mechanism is slow. Experiments showed that for $n = 30$ one had to wait several minutes before $(dtm \; [\![parity]\!]) \; i \; n$ was evaluated for any input i.

We will come back to this problem later on, when we will try to develop an implementation of the discrete time function model in the imperative language C.

5.5 Other definitions for the flipflops

Consider the two definitions to define the properties of the D-flipflop in the previous section. It would benefit us to know that these two are equivalent so that we may substitute one for the other to obtain a (more) efficient implementation. We will prove this conjecture by induction. In this proof we will denote the function defined first by *dtm_dff1* and the second by *dtm_dff2*.

As can be seen from the definitions the case $n = 0$ is trivial. Now suppose that $n > 0$ and that we we have already proven that for all $k \leq n$ holds that $dtm\_dff1 \; q_0 \; d \; k = dtm\_dff2 \; q_0 \; d \; k$ where $q_0 \in \mathbb{B}$ and $d \in \mathbb{N} \to \mathbb{B}$ may be chosen arbitrarily. Then:

$$\begin{aligned} dtm\_dff2 \; q_0 \; d \; (n+1) &= dtm\_dff2 \; (d \; 0) \; (tail \; d) \; n \\ &= dtm\_dff1 \; (d \; 0) \; (tail \; d) \; n \\ &= (tail \; d) \; (n-1) \\ &= d \; n \\ &= dtm\_dff1 \; q_0 \; d \; (n+1) \end{aligned}$$

which concludes the proof. In the same way the following definition

Def

$$\begin{aligned} dtm\_jkff \; q_0 \; (j,k) \; 0 &= q_0 \\ dtm\_jkff \; q_0 \; (j,k) \; n &= (q' = 0)?j \; (n-1); \neg \; (k \; (n-1)) \\ &\quad \textbf{where}\; q' = dtm\_jkff \; q_0 \; (j,k) \; (n-1) \end{aligned}$$

can be shown to be equal to the definition given for the properties of the JK-flipflop in the previous section. This latter definition however will be more convenient when we will try and find a more efficient evaluation scheme for this model. This definition is also more beneficial if one wants to prove some properties of the hardware described.

5.6 Explicit clock signals

In our previous model the flipflops were clocked implicitly, that is at every step in time they received a tick of the clock. However in practical circuits this is not a valid assumption. Flipflops are oft clocked by (other) subparts of the system. One may think for instance of a counter that is built as a chain of $JK-$flipflops in which every flipflop is clocked by the output of its predecessor in the chain. Moreover many synchronous Mosfet networks employ dynamic latches which may be modelled at logic level by transparent latches.

For these reasons we will try and extend our previous model by allowing to use explicit clock signals. This means that we have to change the declaration of our flipflops in the Glass text to incorporate the extra clock signal:

Atom

$$\begin{aligned}
&\mathit{dff} \in \mathbb{B} \to \mathbf{E}^2 \Rightarrow \mathbf{E},\\
&\mathit{jkff} \in \mathbb{B} \to \mathbf{E}^3 \Rightarrow \mathbf{E},\\
&\mathit{tff} \in \mathbb{B} \to \mathbf{E}^2 \Rightarrow \mathbf{E},\\
&\mathit{tlatch} \in \mathbb{B} \to \mathbf{E}^2 \Rightarrow \mathbf{E},\\
&\mathit{tglatch} \in \mathbb{B} \to \mathbf{E}^3 \Rightarrow \mathbf{E};
\end{aligned}$$

We must also fix the properties of our atoms. Some flipflops are level sensitive whereas others are positive edge or negative edge triggered. In the following definitions we will encounter examples of all three types. We start by we define the properties of a positive edge triggered D-flipflop:

Def

$$\begin{aligned}
&\mathit{dtm_dff} \in \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^2 \to \mathbb{N} \to \mathbb{B}\\
&\mathit{dtm_dff}\ q_0\ (ck, d)\ 0 = q_0\\
&\mathit{dtm_dff}\ q_0\ (ck, d)\ n = (ck\ n \wedge \neg(ck\ (n-1)))?d\ (n-1); q'\\
&\qquad\qquad\qquad\qquad\qquad \textbf{where } q' = \mathit{dtm_dff}\ q_0\ (ck, d)\ (n-1)
\end{aligned}$$

The JK-flipflop and T-flipflop are clocked by the trailing edge of the clock:

Def

$$\begin{aligned}
&\mathit{dtm_jkff} \in \mathbb{B} \to (\mathbb{N} \to \mathbb{B})^3 \to \mathbb{N} \to \mathbb{B}\\
&\mathit{dtm_jkff}\ q_0\ (ck, j, k)\ 0 = q_0\\
&\mathit{dtm_jkff}\ q_0\ (ck, j, k)\ n =\\
&\qquad (ck\ (n-1) \wedge \neg(ck\ n))?((q' = 0)?j\ (n-1); \neg\ (k\ (n-1))); q'\\
&\qquad\qquad \textbf{where } q' = \mathit{dtm_jkff}\ q_0\ (ck, j, k)\ (n-1)
\end{aligned}$$

Def

$dtm\_tff \in \mathbb{B} \rightarrow (\mathbb{N} \rightarrow \mathbb{B})^2 \rightarrow \mathbb{N} \rightarrow \mathbb{B}$
$dtm\_tff\ q_0\ (ck, t)\ 0 = q_0$
$dtm\_tff\ q_0\ (ck, t)\ n =$
$\quad (ck\ (n-1) \wedge \neg(ck\ n))?((t\ (n-1) = 0)?q'; \neg\ q')$
$\quad\quad$ **where** $q' = dtm\_tff\ q_0\ (ck, t)\ (n-1)$

The following definition fixes the properties of a transparant latch, that is a storage element whose clock input is level sensitive. This in effect models a dynamic storage element in nMos at logic level. (Such a dynamic storage element consists of a parasitic capacitor which may be charged or discharged by a pass transistor. In most cases it is followed by a (signal restoring) invertor. It may thus be modelled in this model by a transparent latch followed by an invertor).

Def

$dtm\_tlatch \in \mathbb{B} \rightarrow (\mathbb{N} \rightarrow \mathbb{B})^2 \rightarrow \mathbb{N} \rightarrow \mathbb{B}$
$dtm\_tlatch\ q_0\ (s, i)\ n = (s\ n = 1)?i\ n; q'$
$\quad$ **where** $q' = (n = 0)?q_0; dtm\_tlatch\ q_0\ (s, i)\ (n-1)$

5.7 Towards an efficient implementation

As we have seen, the discrete time function model serves excellently to aid in designing synchronous circuits. The only problem that must be solved before such a semantic function can be used in practice is the slow evaluation of our discrete time functions in a functional language (Miranda), due to the necessity of a lazy evaluation.

We will try and find a solution to this problem by taking a second look to the function we obtain if we apply *dtm* to the parity checker:

Def

$(dtm\ [\![parity]\!]) \in (\mathbb{N} \rightarrow \mathbb{B}) \rightarrow \mathbb{N} \rightarrow \mathbb{B}$
$(dtm\ [\![parity]\!])\ i = o$ **where** $o = dtm\_dff\ 0\ (dtm\_xor\ (i, o))$

A first transformation we apply to this definition is that we lambda lift the where clause and then introduce the time parameter t by η-converting the definitions:

Def

$(dtm\ [\![parity]\!]) \in (\mathbb{N} \rightarrow \mathbb{B}) \rightarrow \mathbb{N} \rightarrow \mathbb{B}$
$(dtm\ [\![parity]\!])\ i\ t = \_o\ i\ t$

Def

$\_o \in (\mathbb{N} \rightarrow \mathbb{B}) \rightarrow \mathbb{N} \rightarrow \mathbb{B}$
$\_o\ i\ t = dtm\_dff\ 0\ (dtm\_xor\ (i, \_o\ i))\ t$

As a third step we try and evaluate the body of $\_o$ partially by using the properties of our atoms:

$$
\begin{aligned}
\_o\ i\ t &= dtm\_dff\ 0\ (dtm\_xor\ (i, \_o\ i))\ \ t \\
&= (t = 0)?0; (dtm\_xor\ (i, \_o\ i)\ (t-1)) \\
&= (t = 0)?0; (s\_xor\ (i\ (t-1), \_o\ i\ (t-1))
\end{aligned}
$$

After substituting this form for the body of $\_o$ we obtain the following two definitions:

Def

$(dtm\ [\![parity]\!]) \in (\mathbb{N} \to \mathbb{B}) \to\in \mathbb{N} \to \mathbb{B}$

$(dtm\ [\![parity]\!])\ i\ t = \_o\ i\ t$

Def

$\_o \in (\mathbb{N} \to \mathbb{B}) \to \mathbb{N} \to \mathbb{B}$

$\_o\ i\ t = (t = 0)?0; (s\_xor\ (i\ (t-1), \_o\ i\ (t-1))$

If we want to use these definitions we must provide the input signal i, which is passed on recursively until it is used. If we now lift the i to the global scope, our definitions are no longer higher order functions:

Def

$i \in \mathbb{N} \to \mathbb{B}$

Def

$(dtm\ [\![parity]\!]) \in \mathbb{N} \to \mathbb{B}$

$(dtm\ [\![parity]\!])\ t = \_o\ t$

Def

$\_o \in \mathbb{N} \to \mathbb{B}$

$\_o\ t = (t = 0)?0; (s\_xor\ (i\ (t-1), \_o\ (t-1))$

We may now evaluate these definitions using a strict evaluation mechanism: we have removed the necessity of lazy evaluation by partially evaluating the original right hand side using the properties of our atoms. Instead of formulating these definitions in Miranda we can now chose to formulate them in C.

It is clear that this removal can only be done in full if the right hand sides of our expressions only contain applications of atoms. So in order to guarantee this we must remove all applications of λ-abstractions and user defined system definitions by means of β-reduction.

Because this removes in effect all hierarchy in the description we can view this removal as a kind of flattening. What remains is only one definition containing applications of atoms and where clauses. Hereafter all where clauses are lambda lifted followed by the partial evaluation of the bodies of the resulting set of definitions.

The current implementation of the discrete time function model therefore first removes all λ-abstractions and user defined system definitions by means of β-reduction. We may view this removal as a kind of flattening. What remains is only one system definition containing only applications of atoms and local definitions (where clauses). To this definition the above process is applied and corresponding C definitions are generated. Memoizing is used to lessen the time spent in recursion. In order to visualize the inputs and outputs as well as to have the possibility of editing the input signals a user interface based on the window system X is generated with these definitions. The program thus generated is then compiled and linked using the standard C compiler.

5.8 Using the current implementation

The current implementation of the discrete time function model falls apart in three pieces:

a) A program called *uflat* (*u*nidirectional *flat*tener), which implements the flattening process mentioned above. It takes its input from *stdin* and yields its output on *stdout*.

b) A program called *dtm*, which takes as input a flattened description and yields a C program according to the process described. Input and output must be provided in the same way as for the flattener. Flags may be given to steer the generation of the C program:

   - The *-e* flag indicates that the Eichelberger set is used for the value domain instead of the booleans.
   - The *-i* flag indicates that the program must use an implicit clock signal for the flipflops.

c) Two scripts that are generated to combine these two programs with the C compiler and linker, which are called *Xdtm*, *Xextm*. To use these scripts one must compile the Glass description using *glass* (see also next chapter) and then apply these scripts to the resulting datastructures. The linkage with the X support library is done automatically.

Chapter 6

Glass user environment

R. Doesborg, S. Klaver, SAG

6.1 Users view of the environment

6.1.1 Introduction

This chapter describes the user interface of the Glass system description environment. The user interface to the environment is called Glue (Glass User Environment). Before you can use Glue, installation must have completed succesfully. Make sure that the directory FORFUNDIR/bin is in your command search path. See the installation guide for more details.

The user interface of the environment is menu-driven. The command 'glue' starts the environment. The menus for the environment can be changed. New programs and semantic functions can easily be added to the menus. See the chapter on Glue in the installation and maintenance document.

6.1.2 Walking through menus

Figure 6.1 shows the screen when you start the environment. On the screen is the title of this menu, and a list of menu items. How to walk through the menus is easier to do then to explain. One item is the current item. This item is displayed in reverse video. You can change the current item with the arrow keys or with the space bar. You can select the current item by pressing <return>. You can directly select an item by pressing the character that uniquely identifies the item. Identifying characters of the items are also indicated in reverse video. Selecting an item either brings you to another menu or starts one or more actions.

Press 'x' or arrow up, to go to a previous menu. You can leave the environment by pressing 'x' when in the root menu. While in Glue, pressing '!' puts you in the shell. If you logout, you return to Glue. When you select an item, you are sometimes confronted with a parameter input screen, as depicted in figure 6.2. On a parameter screen is a list of parameters and

Figure 6.1: Glue start up screen

Figure 6.2: Glue parameter input screen

their current value. One of the parameter names is in reverse video. This is the current parameter. If you press <return> you can edit its value in the edit box below the list of parameters. If you press <return> on a parameter that has an enumerated type, the parameter value toggles. While editing a parameter value, <cntrl-U> erases the current parameter value.

You can change the current parameter with the up and down arrows and with the <space>-bar. If you select the item "accept", you leave the parameter input screen. If you select the item "restore", the parameter values are restored to the values they had when you entered the parameter input screen.

6.1.3 Items in the main menu

The goal of working in Glue is creating a Glass file, compiling it and viewing the result of a semantic function.

By selecting an item in the main menu you can perform the following actions:

"Defaults"
: The items: editing a Glass file, compiling a Glass file and viewing the result of a semantic functions all work on a default Glass file and a default compiled Glass file. This item changes the name of the default Glass file.

"Edit Glass"
: This item starts vi with the default Glass file, its ".sf" file and its ".listing" file. You can inspect error messages and change the Glass file. See the following subsection for an explanation of the ".sf" and ".listing" file.

"Compile"
: Compiles a Glass file. See the following subsection for more details.

"Meaning functions"
: Enters a submenu. If the default Glass file has been compiled you can select meaning functions from this submenu. No check is done on the currency of compiled Glass files. If you change a Glass file without recompiling it, the changes will have no impact on the result of semantic functions. What semantic functions are present and how they are to be used is explained later in this document.

"Edit file"
: You are prompted for a filename. After that you can edit this file.

"Info"
: Enters a submenu. The submenu contains items to get help on Glue, help on the functionality of keys in Glue and you can list the Glass files present in your current directory.

"Exit"

Leaves Glue.

6.1.4 Glass compilation

If you select "Compile", the following messages will appear on your screen:

```
glass: cpp ...
glass: parsing ...
glass: checking types ...
glass: expand ...
glass: size check ...
glass: listing ...
```

Each line stands for a part in the total compilation process. Compilation is comprised of the following actions:

1) Preprocessing
   The C-preprocessor is used to remove (among other statements) '#include'-statements that are used to include other Glass files.

2) Parsing
   Checking of the syntax. Generation of a compiled Glass file (in a not readable format). All further actions work on this file.

3) Checking types
   Checking of types, excluding checking of the correct size of busses that are used in macros.

4) Expanding macros
   Macros are expanded

5) Size check
   An extension of typechecking: the size of busses after macro- expansion is checked.

6) Listing
   A (readable) kernel Glass file is generated. This file does not contain macros.

Glass compilation can end in four situations:

1) Syntax errors
   Errors are displayed on screen.

2) Typechecker errors
   Errors are displayed on screen and in a ".listing" file.

3) Macroexpand errors
   Errors are displayed on screen and in a ".sf" file.

Figure 6.3: Glass environment, users view

4) No errors, succesfull compilation
Compiled Glass in ".sf" file.
Expanded Glass (kernel Glass) in ".listing" file.

An overview of these situations is presented in figure 6.3. When you have compiled a Glass file you can apply semantic functions. How these semantic functions may be used and how they are implemented may be read in the subsequent sections and chapters of this book.

6.2 User manual for digital semantic functions

J. Vanslembrouck, ABT

6.2.1 Outline

This section contains the user's documentation for digital semantic functions implemented at Bell.

The report is organized as follows. In subsection 6.2.2, an overview of the functions will be given. Subsection 6.2.3 describes general information to invoke them. The last subsection 6.2.4 describes the different aspects of each function and gives an example of their use.

6.2.2 Overview of semantic functions for digital systems

Each semantic function has its own subdirectory under the main directory. The following table presents an overview of all implemented functions. The first column gives the directory where the function can be found, the second column indicates if the function is a compiled (with target) or interpreted one, the third column refers to the manual of the function and the last column gives a brief description. A similar table can be found in the chapter on semantic functions for digital systems. There, also a reference to a more detailed description, the theoretical backgrounds, and the motivation can be found.

Directory	Target	Manual	Description
cpm	Miranda	6.2.4.1	worst case delay calculation
mossimII	(interp.)	6.2.4.2	switch level simulator (Bryant's model)
mult	Miranda	6.2.4.3	multiplex behaviour
nndf	NNDF	6.2.4.4	translation to neutral network description format
prolog	Prolog	6.2.4.5	simplex behaviour
senspath	(interp.)	6.2.4.6	calculation of sensitized paths
sfdig	Miranda		simplex behaviour
	Miranda		multiplex behaviour
simlog	SIMLOG	6.2.4.7	input format for ITT SIMLOG simulator
simp	Miranda	6.2.4.8	simplex behaviour
simulation	(interp.)	6.2.4.9	event-driven/selective-trace simulator
symbolic	Miranda	6.2.4.10	symbolic simulator

The simplex and multiplex semantic functions have been implemented at Bell and Sagantec (sfdig version). The Sagantec version is documented in a separate manual.

6.2.3 Semantic function invocation

6.2.3.1 Introduction

We can distinguish between compiled and interpreted semantic functions. Both use the Glass abstract syntax tree as input.

In the case of *compiled* functions, a translation is made to a description in another language which has a predefined semantics (except for NNDF). If additional inputs such as signal values are required, these have to be given in the syntax of the target language. In case the target is a Miranda script, there is a possibility to compile the script immediately and to enter the Miranda environment for evaluation of the Glass description. For all other targets, such as Prolog , NNDF or SIMLOG, no further actions will be taken after compilation, because these targets will not (always) be available on each system. In the case of *interpreted* functions, additional inputs will have to be provided in Miranda syntax.

We will use a Glass file named "example.gl" as an example to illustrate the process. This file is present in a subdirectory "works" of the main directory. We will assume that this file has already been parsed, type-checked and macro-expanded, i.e. the file "example.sf" is available. Several scenario's for semantic function invocation will be presented going from completely manually (for UNIX minded users or for maintenance people and semantic function writers who want to understand the information flow) over half automatic (where scripts are used in order to relieve the user from some work) to integrated in the Glass user environment.

6.2.3.2 Invocation of compiled functions

Miranda as target This scenario is valid for functions in the *cpm*, *mult*, *simp* and *symbolic* directory. A symbolic directory name *xxx* will be used to illustrate the translation.

Scenario 1 - completely manually.

- Step 1: "cd ../xxx" : Enter the semantic function directory.
- Step 2: "cp ../works/example.sf kernel.m" : Copy the kernel Glass abstract syntax tree to the kernel file in this directory.
- Step 3: "mira main" : Enter the Miranda translator.
- Step 4: Enter "compile". This is a Miranda function which compiles the Glass abstract syntax tree present in the file "kernel.m" to a Miranda script. The script is listed on the screen. It is of course practical to redirect this output to a file. This can be accomplished by entering (e.g.) "compile &> example.xxx.m" instead. A batch process is started. Wait some time until the file has been generated...

- Step 5: Enter "\f example.xxx" to compile and type check this Miranda script.
- Step 6: System evaluation: see corresponding manual.

Scenario 2 - half automatic.

- Step 1: "cd ../xxx" : Enter the semantic function directory.
- Step 2: "cp ../works/example.sf ." : Copy the kernel Glass file to this directory.
- Step 3a: Enter "compile example" to produce the Miranda target. The reader is referred to the corresponding manual for the correct target name. This file will automatically be compiled and the Miranda environment is entered.
- Step 3b: This is an alternative to step 3a. Enter "compil example" to produce the Miranda target. This target file will not be compiled automatically, which gives the user the possibility to inspect the generated code. Compile the Miranda target and enter the Miranda environment.
- Step 4: System evaluation: see corresponding manual.

Scenario 3 - automatic.

Select the correct menu of the Glue environment. Continue with the "system evaluation" subsection in the semantic function manual.

Non-Miranda targets The scenario's presented in this subsection are valid for the Prolog , *NNDF* and *SIMLOG* semantic functions. The symbolic directory name *xxx* will be used to illustrate the translation.

Scenario's

Scenario 1 - completely manually.

- Step 1: "cd ../xxx" : Enter the semantic function directory.
- Step 2: "cp ../works/example.sf kernel.m" : Copy the kernel Glass abstract syntax tree to the kernel file in the xxx directory.
- Step 3: "mira main" : Enter the Miranda translator.
- Step 4: Enter "compile". This is a Miranda function which compiles the Glass abstract syntax tree present in the file "kernel.m" to a Miranda script. The script is listed on the screen. It is of course practical to redirect this output to a file. This can be done by entering (e.g.) "compile > example.xxx" instead. A batch process is started. Wait some time until the file has been generated...

- Step 6: System evaluation: see corresponding manual.

Scenario 2 - half automatic.

- Step 1: "cd ../xxx" :Enter the semantic function directory.
- Step 2: "cp ../works/example.sf . " : copy the kernel Glass abstract syntax tree to the xxx directory.
- Step 3: Enter "compile example" to generate the correct target. The target language compiler or interpreter is not invoked because this program will in general not be present on each system. Extension of the "compile" script to have the same functionality as in the case of Miranda targets is left as an exercise to the maintenance people.
- Step 4: System evaluation: see corresponding manual.

Scenario 3 - automatic.

Select the correct menu of the Glue environment. Continue with the "system evaluation" subsection in the semantic function manual.

Invocation of interpreted functions The scenario's presented in this subsection are valid for the *mossimII*, *senspath* and *simulation* semantic functions. The symbolic directory name *xxx* will be used to illustrate the translation.

Scenario 1 - completely manually.

- Step 1: "cd ../xxx" : Enter the semantic function directory.
- Step 2: "cp ../works/example.sf kernel.m " : Copy the kernel Glass file to the kernel file in the xxx directory.
- Step 3: "mira main" : Enter the Miranda environment.
- Step 4: System evaluation: see corresponding manual.

Scenario 2 - half automatic.

- Step 1: "cd ../xxx" : Enter the semantic function directory.
- Step 2: "cp ../works/example.sf ." : Copy the kernel Glass file to this directory.
- Step 3: Type "compile example" to enter the Miranda environment.
- Step 4: System evaluation: see corresponding manual.

Scenario 3 - automatic.

Select the correct menu of the Glue environment. Continue with the "system evaluation" subsection in the semantic function manual.

6.2.4 Semantic function manuals

The manuals will be structured as follows.

- Identity: presents the most important information to identify the function.
  - Directory: directory where the function can be found.
  - Description: short description of the function.
  - Type: is the semantic function a compiled or interpreted one? In the first case, the target language will be given.
  - Semantics of the atoms: where can the semantic definitions of the Glass atoms be found? In many cases, these definitions are placed in a separate file called "atomics.m".
- Restrictions: some restrictions on the Glass descriptions are given.
- Invocation: for the scenario's to invoke the function, the reader will be referred to the subsection 6.2.3.2.
- System evaluation: Some additional syntax is often required to evaluate a Glass system description. This syntax will be presented in a BNF-notation. Terminal symbols (which have to appear literally) are placed between single quotes ('). If such a quote itself appears in the syntax, it will be proceeded by a backslash (\).
- Example: an example is given to illustrate the system evaluation described before. The Glass descriptions can be found in the "test.gl" file of the directory. For compiled functions, the target should be present in the file "test.xxx.good", where *xxx* is the extension corresponding to the semantic function. Some system evaluation examples (if applicable) can be found in the "test.examp" file. These files should be present on the distribution tape.

6.2.4.1 cpm - Critical path method

Identity

- **Directory:** cpm
- **Description:** calculates the worst case delay from every input in the system to every output. Registers are removed in sequential circuits and the delay is calculated for all combinational subcircuits.
- **Type:** compiled to Miranda script.
- **Semantics of atoms:** present in file "atomics.m".

Restriction on the descriptions Be sure that the semantics of your atoms have been defined in the file "atomics.m".

Only the directional subset of Glass is supported. There are no further restrictions on the syntax of the atoms or definitions.

Invocation See subsection 6.2.3.2. The target for the Glass file "example.gl" is "example.cpm.m".

System evaluation You can ask for the delay of a system by just entering the name of the system in the Glass description after the Miranda prompt. Because the function is input pattern independent, no values are needed for the formal connection parameters.

Example For the following Glass description

```
BASETYPE B ;

ATOM
  and :- B&B => B,
  or  :- B&B => B,
  not :- B => B ;

DEF
    sel :- B&B&B => B;
    sel [s,a,b]  = or [and [not s, a], and [s,b]]
```

the generated Miranda script looks like

```
%include "aux.m"

%include "atomics.m"

s = newarrl "s"
a = newarrl "a"
b = newarrl "b"

sel = pp1 (sel' (s,a,b))
sel' (s,a,b)
 = (rf "or0" or0)
   where
     or0 = cpm_or (and00,and01)
     and00 = cpm_and (not0000,a)
     not0000 = cpm_not s
     and01 = cpm_and (s,b)

pp1(a1) = heading ++ pprint a1
```

The evaluation of the system *sel* gives the following output:

```
Miranda sel

The delay seen at each output from each input:
output                   delay     input

or0                      5         a
or0                      5         b
or0                      6         ena
```

6.2.4.2 mossimII - Switch-level simulator

Identity

- **Directory:** mossimII
- **Description:** simulates a switch-level description using the Bryant model.
- **Type:** Interpreted.
- **Semantics of atoms:** see function *mk_matrix_elem* in the file "mossimII.m".

Restriction on the descriptions Be sure that the semantics of your atoms have been defined in the file "mossimII.m" (function mk_matrix_elem).

Only adirectional descriptions using n-type and p-type transistors are supported. The transistor ports should be ordered as *gate*, *source* and *drain*. A size should be assigned to all nodes inside the network and to the output node. A strength should be assigned to all transistors. Sizes and strengths are subsets of the same enumeration type defined in "mossimII.m" as

size ::= L | K1 | K2 | Y1 | Y2 | W

This set can be expanded at wish. The example below should make things more clear.

Invocation See subsection 6.2.3.2. No target file is produced.

System evaluation In the Miranda environment, on-line help about the semantic function is available. Just type "help" to obtain explanation about the syntax of the simulation command. This help panel looks like:

```
                     MOS simulator

 The syntax for invoking the simulator is:
```

```
    'mossim' <system-name> <states> <inputs>
 where
   <states>
       ::= '[]' | '['<state> {',' <state>}* ']'
   <state>
       ::= '(' <internal-pin-name> ',' <value> ')'
   <inputs>
       ::= '[]' | '['<input> {',' <input>}* ']'
   <input>
       ::= '(' <input-pin-name> ',' <value> ')'
   <value>
       ::= 'S0' | 'S1' | 'SX
   <internal-pin-name> == <input-pin-name> == <pin-name>
   <system-name>
      ::= any system in the Glass description (between double quotes)
   <pin-name>
      ::= any pin in your system description (between double quotes)
          for originally nameless pins, use the 'display' function
 Some examples of use are included in the file 'test.examp'


 Enter 'display' <system-name> to generate the transformed
     description which is used by the simulator.
```

Example The following Glass description will be used to illustrate system evaluation.

```
BASETYPE B ;

ATOM ntran :- STRING -> [B&B&B],
     dtran :- STRING -> [B&B&B],
     node  :- STRING -> [B] ;

DEF inverter :- [B&B&B] ;
    inverter [in,pass,b]
      = {ntran "Y1" [a,vdd,a],
         ntran "Y2" [in,a,vss],
         ntran "Y2" [pass,a,b],
         node "K1" a,
         node "K1" b,
         node "W" vdd,
         node "W" vss
        }
```

The function "display" can be used to display the transformed Glass description used by the simulator. (In the case of the description above, the same description will result). In case of hierarchical descriptions, the hierarchy will be flattened and unique names will be generated for the connections. These names can be used for assigning initial states to connections inside the circuit.

The following state vectors

```
y1 = [("vdd",S1),("vss",S0),("a",S1),("b",S0)]
y2 = [("vdd",S1),("vss",S0),("a",S1),("b",SX)]
y3 = [("vdd",S1),("vss",S0),("a",S1),("b",S1)]
```

and the following input vector

```
yin1 = [("in",S1),("pass",SX)]
```

will be used for the evaluation of the *inverter* circuit:

```
Miranda mossim "inverter" y1 yin1
[("a",S0),("b",S0),("vdd",S1),("vss",S0)]
Miranda mossim "inverter" y2 yin1
[("a",S0),("b",SX),("vdd",S1),("vss",S0)]
Miranda mossim "inverter" y3 yin1
[("a",S0),("b",SX),("vdd",S1),("vss",S0)]
```

6.2.4.3 mult - Multiplex semantic function

Identity

- **Directory:** mult
- **Description:** calculates the multiplex behaviour
- **Type:** compiled to Miranda script.
- **Semantics of atoms:** present in file “atomics.m”.

Restriction on the descriptions Be sure that the semantics of your atoms have been defined in the file “atomics.m”.

Directional descriptions at the gate-level or higher are supported. There are no restrictions on the syntax of the atoms or definitions.

Invocation See subsection 6.2.3.2. The target for the Glass file “example.gl” is called “example.mp.m”.

System evaluation To evaluate a Glass description, enter the name of system with the arguments between parantheses instead of the square brackets of the original descriptions. The most usual interpretation of the basic type is a stream of binary values. In the example below, however, a somewhat different approach has been taken.

Example For the following Glass description,

```
BASETYPE B;

ATOM out :- B&B => B,
     next :- B&B => B,
     dffint :- B&B => B;

DEF fsa :- B&B => B;
    fsa [q0,is] = out [is,qs]
       where
         qs = dffint [q0, next [is,qs]]
       endwhere ;
```

the generated Miranda scripts looks like

```
%include "atomics.m"

fsa (q0,is)
 = out (is,qs)
   where
     qs = dffint (q0,next (is,qs))
```

With the following definitions of the atoms (to be found in the file "atomics.m")

```
dffint (q0,qs) = q0:qs

out (x:xs,y:ys) = x*y : out (xs,ys)
out (xs,ys) = []

next (x:xs,y:ys) = x+y : next (xs,ys)
next (xs,ys) = []
```

the evaluation of the circuit for a sample input leads to

```
Miranda fsa (1,[1,2,3,4,5])
[1,4,12,28,55]
```

6.2.4.4 nndf - Translation to NNDF

Identity

- **Directory:** nndf
- **Description:** translates the Glass description to the Neutral Network Description Format used at Alcatel Bell.
- **Type:** compiled to NNDF
- **Semantics of atoms:** present in the file "atomics.m"

Restriction on the descriptions As Glass , NNDF has no predefined semantics. The definitions in "atomics.m" are only used for unparsing to NNDF. The real semantics are present in libraries used by the seperate tools which take NNDF as input format. Therefore, the names of the atoms should be restricted to names which can be found in standard library data books. No check is provided, however, to check these names.

Both directional and adirectional descriptions at any level of abstractions are supported.

Invocation See subsection 6.2.3.2. The target for the Glass file "example.gl" is called "example.nnd".

System evaluation Sorry, you need the NNDF environment...

Example A sample Glass description can be found in "test.gl". Because the NNDF output in "test.nnd.good" is rather long, even for small Glass descriptions, the translation is not included here.

6.2.4.5 prolog - Translation to Prolog

Identity

- **Directory:** prolog
- **Description:** generates Prolog rules from Glass circuit descriptions. Corresponds to the simplex semantic function but can also be used to simulate certain kinds of switch-level circuits.
- **Type:** compiled to Prolog program
- **Semantics of atoms:** present in file "atomics.m".

Restriction on the descriptions The function works for directional (at the logic gate level or higher) as well as for adirectional switch-level descriptions.

Be sure that the semantics of your atoms have been defined in the file "atomics.m". There are no further restrictions on the syntax of the atoms or definitions.

Invocation See subsection 6.2.3.2. The target for the Glass file "example.gl" is called "example.pro".

System evaluation You need a Prolog interpreter. To evaluate a Glass description, enter the name of system with the arguments between parantheses instead of the square brackets of the original descriptions. Do not forget that additional arguments should be used for the outputs of the circuit. Therefore, it is advisable to check the generated Prolog program first.

The most usual interpretation of the basic type is a binary value, but numeric types may be taken as well. This depends on the definition of the atoms.

subExample For the following Glass description,

```
BASETYPE B ;

ATOM
  and :- B&B => B,
  or  :- B&B => B,
  not :- B => B ;


DEF
    imec :- B&B&B => B;
    imec [i,j,k] = sel [and[k,j], and[and[i,j],k], i] ;

DEF
    sel :- B&B&B => B;
    sel [s,a,b]  = or [and [not s, a], and [s,b]]
```

the generated Prolog program looks like

```
/* file atomics.m   */

and(b0,X,b0).
and(b1,X,X ).

or(b0,X,X).
or(b1,X,b1).

p_not(b0,b1).
p_not(b1,b0).

sel(b0,X,Y,X).
sel(b1,X,Y,Y).

/* end file atomics.m */

imec(I,J,K,Sel0):-
    and(K,J,And00),
    and(I,J,And0010),
    and(And0010,K,And01),
```

```
    sel(And00,And01,I,Sel0).

sel(S,A,B,Or0):-
    p_not(S,Not0000),
    and(S,B,And01),
    and(Not0000,A,And00),
    or(And00,And01,Or0).
```

The *sel* circuit will be evaluated in the *backward* direction (i.e. given the output of the circuit, which inputs should be applied?). The output of the Prolog interpreter may look as follows

```
?- sel(S,A,B,b0).
** sel(b0, b0, _1, b0) ? ;
** sel(b1, _1, b0, b0) ? ;
no
```

6.2.4.6 senspath - Sensitized path calculation

Identity

- **Directory:** senspath
- **Description:** calculates the path expressions in switch level networks.
- **Type:** interpreted.
- **Semantics of atoms:** see function *pe_mkadjlist* in the file "route.m".

Restriction on the descriptions Only adirectional descriptions at the switch-level using n-type and p-type transistors are supported. The transistors ports should be ordered as *gate*, *source* and *drain*.

Invocation See subsection 6.2.3.2. No target file is produced.

System evaluation . In the Miranda environment, on-line help about the semantic function is available. Just type "help" to obtain explanation about the syntax of the simulation command. This help panel looks like:

```
              Sensitized path semantic function

 Enter 'pathex' <system> <point1> <point2> to display
   the boolean expression which states the condition under which
   there is a connection between <point1> and <point2>
   in system <system>.
  Enter 'display' <system> to generate the flattened Glass description
   which is used by the path function.
 <system>, <point1> and <point2> are identifiers between double quotes.
```

Example The following Glass description will be used to illustrate system evaluation

```
BASETYPE B ;

TYPE tran = [B&B&B] ;

ATOM
     ntran :- [B&B&B],
     ptran :- [B&B&B],
     Vdd :- [B],
     Gnd :- [B] ;

DEF cmods :- [B&B&B&B];
    cmods [Cin,A,B,Ncout] =
       { ntran [A,P1,b0],
         ntran [B,P1,b0],
         ntran [Cin,Ncout,P1],
         ntran [A,Ncout,P2],
         ntran [B,P2,b0],
         ptran [A,b1,P3],
         ptran [B,b1,P3],
         ptran [Cin,P3,Ncout],
         ptran [A,P4,Ncout],
         ptran [B,b1,P4]
       } ;
```

The path expressions between the high voltage source and the output, between the ground and the output, and between the high voltage source and the ground will be calculated.

```
Miranda pathex "cmods" "b1" "Ncout"
~Cin.~A + ~Cin.~B + ~A.~B
Miranda pathex "cmods" "b0" "Ncout"
Cin.A + Cin.B + A.B
Miranda pathex "cmods" "b1" "b0"
A.Cin.~Cin.~A + A.Cin.~Cin.~B + A.Cin.~A.~B + B.Cin.~Cin.~A +
B.Cin.~Cin.~B + B.Cin.~A.~B + B.A.~Cin.~A + B.A.~Cin.~B + B.A.~A.~B
```

6.2.4.7 simlog - Translation to SIMLOG

Identity

- **Directory:** simlog
- **Description:** generates input for the ITT SIMLOG simulator.
- **Type:** compiled to SIMLOG format

- **Semantics of atoms:** present in proprietary library.

Restriction on the descriptions The function works for directional (at the logic gate level) as well as adirectional (switch-level) descriptions. There are no further restrictions on the syntax of the atoms or definitions.

Invocation See subsection 6.2.3.2. The target for the Glass file "example.gl" is called "example.sml".

System evaluation Sorry, you need the SIMLOG simulator and more information about this tool.

Example Because the SIMLOG output in "test.sml.good" is rather long, even for small Glass descriptions, the translation is not included here.

6.2.4.8 simp - Simplex semantic function

Identity

- **Directory:** simp
- **Description:** calculates simplex behaviour for combinational circuits.
- **Type:** compiled to Miranda script.
- **Semantics of atoms:** present in file "atomics.m".

Restriction on the descriptions Be sure that the semantics of your atoms have been defined in the file "atomics.m". There are no restrictions on the syntax of the atoms or definitions.

Invocation See subsection 6.2.3.2. The target for the Glass file "example.gl" is called "example.sp.m".

System evaluation To evaluate a Glass description, enter the name of system with the arguments between parantheses instead of the square brackets of the original descriptions. The most usual interpretation of the basic type is a binary value, but numeric types may be taken as well. This depend on the definition of the atoms.

Example For the following Glass descriptions,

```
BASETYPE B ;

ATOM
  and :- B&B => B,
```

```
  or  :- B&B => B,
  not :- B => B ;

DEF
    imec :- B&B&B => B;
    imec [i,j,k] = sel [and[k,j], and[and[i,j],k], i] ;

DEF
    sel :- B&B&B => B;
    sel [s,a,b]  = or [and [not s, a], and [s,b]]
```

the generated Miranda script looks like

```
%include "atoms_sp.m"

imec (i,j,k)
 = sel (sp_and (k,j),sp_and (sp_and (i,j),k),i)

sel (s,a,b)
 = sp_or (sp_and (sp_not s,a),sp_and (s,b))
```

The evaluation of the system "imec" for some sample input combinations gives the following output:

```
Miranda imec(B0,B1,B1)
B0
Miranda imec(B1,B1,B0)
B0
Miranda imec(B1,B1,B1)
B1
```

6.2.4.9 simulation - Event-driven/selective-trace simulator

Identity

- **Directory:** simulation
- **Description:** simulates a circuit description in the event-driven/selective-trace mode.
- **Type:** interpreted.
- **Semantics of atoms:** present in file "atomics.m".

Restriction on the descriptions Be sure that the semantics of your atoms have been defined in the file "atomics.m".

The function supports directional system descriptions at the gate-level. There are no further restrictions on the syntax of the atoms or definitions.

Invocation See subsection 6.2.3.2. No target file is produced.

System evaluation In the Miranda environment, on-line help about the semantic function is available. Just type "help" to obtain explanation about the syntax of the simulation command. This help panel looks like:

```
                    Event-driven/selective-trace simulator

The syntax for invoking the simulator is:
   'sim' <simulation-mode> <system-name> <lut> <eventlist>
where
  <simulation mode>
     ::= 'sw1' (transport delay mode, all pulses passed)
       | 'sw2' (inertial delay of element = propagation delay)
       | 'sw3' (inertial delay of element < propagation delay)
  <lut>   /* look-up table */
     ::= '[]' | '[' <lut-entry> {',' <lut_entry>}* ']'
  <lut-entry>
     ::= '(' <pin-name> ',' <pin-value> ')'
  <eventlist>
     ::= '[]' | '[' <event>, {',' <event>}* ']'
  <event>
     ::= '(' <time> ',' <pin-name> ',' <pin-value> ')'
  <system-name>
     ::= any Glass system you have defined (between double quotes)
  <pin-name>
     ::= any pin in your system description (between double quotes)
         for originally nameless pins, use the "display" function.
  <time>
     ::= a numeric value, giving the time at which an event occurs
  <value>
     ::= '0' | '1' | '2' (2 stands for the intermediate value 'X'
Some examples are included in the file "test.examp".


Enter 'display' <system-name> to generate the transformed
    description which is used by the simulator
```

Example The following Glass description will be used to illustrate system evaluation.

```
BASETYPE B ;

ATOM and :- B&B => B ,
     or2 :- B&B => B ;
```

```
DEF aoa2 :- B&B&B => B ;
    aoa2 [in1,in2,s] = and[or2[and[in1,in2],s],s] ;
```

First, the function "display" can be used to display the transformed Glass description used by the simulator. Names are generated for originally nameless connections in the description. These names are useful 1) to initialize nodes in the circuit and 2) to be able to follow the activity in this circuit during simulation.

```
Miranda display "aoa2"
  DEF aoa2 :- (B&B&B) => B ;
      aoa2 [in1, in2, s] = and0
            where
              and0 = and [or200,s] ;
              or200 = or2 [and0000,s] ;
              and0000 = and [in1,in2]
            endwhere
```

The following contains the output of the simulator for a sample input.

```
Miranda sim sw2 "aoa2" [("and0",2)]
                       [(-100,"in2",1),(0,"in1",1),(4,"in1",0)]
      time     value          node
      -100         0          in1
      -100         1          in2
      -100         0          s
         0         1          in1
         3         1          and0000
         4         0          in1
         7         0          and0000
```

6.2.4.10 symbolic - Symbolic simulator

Identity

- **Directory:** symbolic
- **Description:** simulates a circuit description with the possibility for complete symbolic and complete numeric simulations or a mixture of the two. Corresponds to the simplex semantic function, but allows to simulate the combinational parts of sequential circuits.
- **Type:** compiled to Miranda
- **Semantics of atoms:** present in file "atomics.m" and "symbolics.m".

Restriction on the descriptions Be sure that the semantics of your atoms have been defined in the files "atomics.m" and "symbolics.m".

Directional descriptions at the gate-level or higher are supported. There are no further restrictions on the syntax of the atoms or definitions.

Invocation See subsection 6.2.3.2. The target for the Glass file "example.gl" is called "example.sym.m".

System evaluation In the Miranda environment (generated Miranda script), on-line help about the semantic function is available. Just type "help" to obtain explanation about the syntax of the simulation command. This help panel looks like:

```
                         Symbolic simulation

  A symbolic simulation of a system description is
  made by entering
    'sym\''<system name>
  where system name is the name of a defined circuit, or
  by entering
     <system name> '(' <input> {',' <input>}* ')'
  where
     <input>
        ::= 'Var' <identifier>
          | 'B0'
          | 'B1'
    <identifier>
        ::= any identifier (between double quotes) for denoting
            symbolically some input.
  More complicated inputs can be made by using symbolic expressions
  instead of variables or constants (see file "symbolic.m")
```

Example For the following Glass descriptions,

```
BASETYPE B ;

ATOM
  and :- B&B => B,
  or  :- B&B => B,
  not :- B => B ;


DEF
    imec :- B&B&B => B;
    imec [i,j,k] = sel [and[k,j], and[and[i,j],k], i] ;
```

```
DEF
    sel :- B&B&B => B;
    sel [s,a,b]  = or [and [not s, a], and [s,b]]
```

the generated Miranda script looks like

```
%include "atomics.m"

sym'imec = pp1 (imec' (Var "i",Var "j",Var "k"))
imec(i,j,k) = pp1 (imec' (i,j,k))
imec' (i,j,k)
 = (Out "sel0" sel0)
   where
     (sel0) = sel' (and00,and01,i)
     (and00) = s_and (k,j)
     (and01) = s_and (and0010,k)
     (and0010) = s_and (i,j)

sym'sel = pp1 (sel' (Var "s",Var "a",Var "b"))
sel(s,a,b) = pp1 (sel' (s,a,b))
sel' (s,a,b)
 = (Out "or0" or0)
   where
     (or0) = s_or (and00,and01)
     (and00) = s_and (not0000,a)
     (not0000) = s_not (s)
     (and01) = s_and (s,b)
```

Three evaluations are included. The first one is completely symbolic. Therefore, *sym'imec* has been entered. The second example illustrates a mixture of symbolic and numeric simulation, while the last evaluation is completely numeric.

```
Miranda sym'imec
sel0 = (((~(k.j)).((i.j).k))+((k.j).i))
Miranda imec(Var "x",Var "y",B1)
sel0 = (((~y).(x.y))+(y.x))
Miranda imec(Var "x",Var "y",B1)
sel0 = (((~y).(x.y))+(y.x))
Miranda imec(B1,B1,B1)
sel0 = 1
```

6.3 Using simplex and multiplex

R. Doesborg, S. Klaver, SAG

These semantic functions have a similar user interface. After you have compiled a Glass file you can select simplex or multiplex. After some time you enter the Miranda system. The file that is loaded is a simulator of the system you have compiled. All systems in the Glass text are translated into a Miranda function.

The file c3.gl (supplied with the semantic functions) contains the following systems:

```
DEF exor :- B&B => B;
DEF log3 :- B&B&B => B;
DEF counter3 :- B => B&B&B;    /* reset => (q2, q1, q0) */
```

In the generated simulator you will find the functions:

```
exor:: ((stream,stream) -> stream)
log3:: ((stream,stream,stream) -> stream)
counter3:: (stream -> (stream,stream,stream))
```

By evaluation of these functions with appropriate stimuli you can simulate them. The basic types in the generated functions are:

```
base ::= B0 | B1 | BX
stream == [base]
```

Simulation is three-valued: with values 0, 1 and X. These values are represented by B0, B1 and BX. Examples of simulation are:

```
Simplex:    log3 (B0, B1, B0)
Multiplex:  par3 (counter3 (B1:v0))

The result of the second simulation is:
    x x x
    0 0 0
    0 0 1
    0 1 0
    0 1 1
    ... etc.
```

Multiplex also generates some auxiliary functions that assist in feeding test values to a function and displaying the result of a function. These functions are:

- v0, v1, v01, ... : Streams of bits that can serve as test inputs
- par2, par3, ... : Functions that display a number of streams of bits in parallel.

The functions generated by simplex and multiplex simulate systems based on the definitions of the atoms. These atoms should be defined in the Miranda files:

- for simplex: FORFUNDIR/sfdig/atoms_sf.m
- for multiplex: FORFUNDIR/sfdig/atoms_mp.m

These files contain the definitions of the atoms that are present in the Glass file FORFUNDIR/sfdig/sag_atoms.gl. If you use other atoms, add their definition to the Miranda files mentioned above.

Simplex and multiplex can also work with Glass files that have macros in them. An example is the supplied file FORFUNDIR/sfdig/add.gl. This file contains the definition of an n-bits adder, as a macro, and the definition of an 8-bits adder that uses this macro.

Note that generation of a simulator, for a Glass file that contains deeply nested macros, takes a long time. This is because, for each level of nesting in the original system, an auxiliary function has to be generated.

Chapter 7

A set of semantic functions for analog circuits

C. van Reeuwijk, TUD

7.1 Data structures for semantic functions

Since one of the important features of Glass is that the *user* is able to supply semantic functions, it is essential to have a well-chosen representation of Glass for these semantic functions. This representation should meet the following requirements:

1. The representation is easy to access from the language in which the semantic function is implemented.

2. It abstracts from parsing details. Also, the implementor should not have to worry about macro expansion and type checking.

After some experiments it has been decided to use a separate parser, type checker and macro expander that will convert the full Glass text to a special internal file representation. As internal representation Miranda algebraic types, see [Tur79], have been chosen. Since it is necessary to access the internal representation from other languages, a program has been developed, tm, that generates code for a number of languages (C , Pascal to access the algebraic data types. If necessary support for other languages is easily added. For a detailed description of Miranda algebraic types and tm see [Ree90].

7.1.1 Glass internal representation

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

```
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 ;
```

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 of Glass description parts is recorded with type `orig`.

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

This information can be used in error messages.

Type `typ` represents connection and system type expressions.

```
typ ::=
    TypBase basenm:symbol |              || basetype
    TypIn ityp:typ |                     || ?ityp
    TypOut otyp:typ |                    || !otyp
    TypUni uityp:typ uotyp:typ |         || uityp => uotyp
    TypNon nontyp:typ |                  || adirectional system
    TypProd ptypes:[typ] |               || Cartesian product.
    TypSym sym:symbol;                   || symbolic type
```

A writer of semantic functions may assume that a `TypSym` never refers to a **basetype**, a `TypBase` will be used for that instead.

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 ;                     || bool
```

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

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

Type `val` represents connection expressions and appsets.

```
val ::=
    VSym symorig:orig sym:symbol |       || Symbol
    VLambda lpar:formcon lval:val |      || Lambda abstraction
    VSigma spar:formcon sval:val |       || Sigma abstraction
    VApply aval:val apar:val |           || Apply (asval aspar)
    VWhere wdefs:[def] wval:val |        || local context
    VList l:[val] |                      || List expression
    VAppset aps:[val] |                  || appset
    VAtom atorig:orig atnm:symbol atvpar:[parval] atcpar:val |
    VSyn synlist:[val];                  || Synonym expression
```

A writer of semantic functions may assume that a `VSym` never refers to an **atom**, a `VAtom` will be used for that instead. The formal parameters of λ- and σ-abstractions (`VLambda` and `VSigma`) are represented by the type `formcon`:

```
formcon ::= FCList l:[formcon] | FCSym sym:symbol;
```

7.1.2 An example of an internal representation

For example, the Glass description:

> **basetype** A;
>
> **atom** $B \in \text{int} \rightarrow A => A$;
>
> **def** $C \in A \Rightarrow A$;
> $C\ i = B\ 1\ (B\ 2\ i)$;

Would be represented as (assuming this description would be in the file `t`):

```
[
 (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"
    [(ParInt 1)]
    (VAtom ("t", 4) "B" [(ParInt 2)] (VSym ("t", 4) "i"))
   )
  )
 )
]
```

7.2 Flattening

Even the kernel language of Glass provides more syntactic constructs than are available in many domains of interpretation, for example:

- Both directional and adirectional abstractions.
- Where clauses.
- Synonym terms.
- Connection tuples.

For many semantic functions it is necessary to remove these constructs, in particular when an existing program is used in the semantic function. This transformation is called *flattening*.

The flattening function has been implemented in the program `kgflat`. It transforms an arbitrary Glass description into a description with only one adirectional definition. This definition will only contain 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 compound connection. A definition of this form is easily interpreted as a net-list.

For flattening the following transformations are necessary:

- Conversion of directional abstractions.
- Conversion of connection equivalences in **where** clauses.
- Expansion of abstraction applications.
- Removal of synonym expression.
- Expansion of connection variables.

7.2.1 Conversion of directional abstractions

A flattener must assume that a description contains adirectional abstractions. Therefore, the directional abstractions must be converted to adirectional abstractions.

To convert directional abstractions to adirectional abstractions the following rewrite rule is used: a definition of the form

$$\textbf{def} \quad s\ i = f\ i$$

is converted to

$$\textbf{def}\ s\ [i, new] = \{f\ [i, new]\}$$

Where *new* is a name that does not clash with existing names. If i contains directional applications, they must be re-written in a similar way.

For example,

$$\textbf{def} \quad n\ i = \text{not (not } i);$$

is converted to:

$$\textbf{def} \quad n\ [i, n_o] = \{\text{not } [n_i, o], \text{not } [i, n_i]\};$$

7.2.2 Removal of connection equivalences

Where clauses may contain connection equivalences that must be removed. Assuming that the abstraction is adirectional, they 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 an equivalent synonym expression. For example

  $$a = b$$

  is converted to the appset

  $$\{*[a, b]\}$$

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

  $$a = f\ i$$

  is converted to

  $$\{*[a, n], f\ [i, n]\}$$

  where n is a name that does not clash with existing names.

7.2.3 Removal of synonym expression

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

$$*[a, b]$$
$$*[a, [b_0, b_1 \ldots b_n]]$$

Synonym expressions of these forms are easily removed from a adirectional definition by replacing all occurrences of a in the appset and abstractor by b or $[b_0, b_1 \ldots b_n]$ respectively.

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

$$
\begin{array}{lll}
*[e_0, e_1 \ldots e_n] & \Rightarrow & *[e_0, e_1], *[e_0, e_2], \ldots * [e_0, e_n] \\
*[[a_0 \ldots a_n], [b_0 \ldots b_n]] & \Rightarrow & *[a_0, b_0], \ldots * [a_n, b_n]
\end{array}
$$

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

$$
\begin{array}{l}
*[e, e] \\
*[e] \\
*[]
\end{array}
$$

7.2.4 Expansion of abstractions

It is assumed that directional abstractions have been rewritten into their adirectional equivalent. 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.

It must be ensured that the connection names within the abstraction differ from the names that are 'visible' from the application. If necessary some of the connections in the applied abstraction must be renamed.

7.2.5 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 compound connection. This is easily accomplished by adding a synonym expression of the form

$$*[c, \mathit{full}]$$

to the appset for each atom in the appset, where c is the existing connection expression of the atom, and *full* is a fully expanded connection expression with 'fresh' connection names. Using the re-write rules for synonym expressions described in section 7.2.3, this will ensure the connection expressions are expanded.

7.3 Semantic functions for behavior

Semantic functions that analyze the behavior of systems will 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 to constructs in their input language.

Fortunately a flattened Glass definition is easily interpreted as a netlist, and therefore after flattening the interface to the appropriate input format for these programs is straightforward. Such translations have been implemented in the programs `kganp` that converts to input for the program `anp` , `kgspice` that converts to input for the program `spice`, and `kggnl` that converts to a simple general netlist format.

Ideally all constructs of a circuit simulation program should be available in Glass , but this is often difficult or impossible. The complications that arise in the implemented semantic functions are representative:

- Optional parameters.
- Component coupling.
- Connection numbering.
- Application numbering.

7.3.1 Optional atom parameters

Often components have parameters that are hardly ever used and have a reasonable default. Since Glass does not handle defaults these parameters always have to be passed. For example, the `spice` resistor has two optional parameters to specify its temperature behavior. Therefore, the `spice` resistor should have the atom declaration

$$\textbf{atom} \quad R_t \in \text{float} \to \text{float} \to \text{float}[E \times E]$$

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

1. Introduce a macro with reasonable defaults. E.g.

   $$\begin{array}{ll} \textbf{mac} & R \in \text{float} \to [E \times E]; \\ & R\ v\ c = \{R_t\ 0\ 0\ v\ c\} \end{array}$$

2. Rely on the syntax of `spice`, and pass the parameter or parameters as a string. Thus the atom declaration becomes:

   $$\textbf{atom} \quad R \in \text{string} \to [E \times E];$$

   and one can use R as follows:

   $$R\ \text{"}\mathtt{1K}\text{"}\ [a, b]$$

   or as

   $$R\ \text{"}\mathtt{1K\ TC = 0.001, 0.0015}\text{"}\ [a, b]$$

Solution 1 is more elegant, but is impossible to use when there is a variable number of parameters. For example, `spice` independent voltage and current sources have an entire sub-language to describe periodic signals.

For the implemented semantic functions solution 2 has been chosen.

7.3.2 component coupling

In some cases circuit simulators use non-electrical coupling between components. For example, both spice and anp support coupled coils. 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:

   basetype M;

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

Since solution 1 would involve considerably more bookkeeping, solution 2 has been chosen in the implemented semantic functions.

7.3.3 Connection numbering

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

As a further complication, spice requires that a connection number 0 is used for a common ground node that is (implicitly) used by transistors. This can be handled by introducing an atom **gnd**:

atom $\mathrm{gnd} \in E$

to label this node in the Glass description.

7.3.4 Application numbering

Spice and anp require that each application of an atom contains a 'label' that distinguishes it from other applications of the same atom. This is done by numbering the applications, and using the number as the label. This will distinguish *all* applications from each other, not only applications of the same atom.

7.4 Geometric interpretation of Glass

Although Glass was not designed for it, it is possible to use Glass as geometric description language. This is *not* only a curiosity, the programs `gldraw` and `drawps` implement this semantic function.

For this interpretation Glass connection variables of type *Coord* are interpreted as coordinates in a 2-dimensional space. All coordinates have the same scale factor. The 'normal' distance between two text lines is defined as '1' in the coordinate system. 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. Gldraw determines the relative position of all variables in the given definition using the position specifications. After that the figure specifications will be used to generate the drawing.

7.4.1 figure specifications

All figure specifications have an attribute string α as 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 converters should at least understand the specification 'normal' that gives a reasonable default. Ideally device drivers should have a (user specified) translation table to allow a flexible choice of representations on various devices.

arc α ϕ $[c, r]$	An arc with center c starting from point r and ending at a point with angle ϕ from r.
line α $[b, e]$	A line between point b and e.
triangle α $[a, b, c]$	A triangle between points a, b and c.
rect α $[a, b]$	A rectangle with horizontal and vertical lines between points a and b.
text α t $[a, b]$	Text t at position a. The line between point a and b is part of the base line of the text. Actual drivers may round the angle implied by this baseline to the nearest convenient angle.

The atom declarations are as follows:

atom arc $\in$ string $\rightarrow$ float $\rightarrow$ [Coord $\times$ Coord];
atom line $\in$ string $\rightarrow$ [Coord $\times$ Coord];
atom rect $\in$ string $\rightarrow$ [Coord $\times$ Coord];
atom triangle $\in$ string $\rightarrow$ [Coord $\times$ Coord $\times$ Coord];
atom text $\in$ string $\rightarrow$ string $\rightarrow$ [Coord $\times$ Coord];

7.4.2 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 b.
trans t_x t_y $[a, b]$	Coordinate b is (t_x, t_y) a coordinate a. Can also be used to determine position of b a a.
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 b.
pcir ϕ $[a, b, c]$	Given a point b on a circle with center 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 from a. 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 though b.

The atom declarations are as follows:

atom trans $\in$ float $\rightarrow$ float$->$ [Coord $\times$ Coord];
atom xyproj $\in$ [?Coord$\times$?Coord$\times$!Coord];
atom polar $\in$ float $\rightarrow$ float$->$ [Coord $\times$ Coord];
atom pcir $\in$ float $\rightarrow$ [Coord $\times$ Coord $\times$ Coord];
atom between $\in$ float $\rightarrow$ [Coord $\times$ Coord $\times$ Coord];

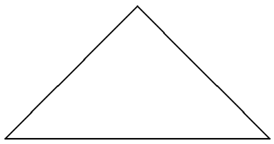
7.4.3 An example of a drawing description

basetype Coord;

atom trans $\in$ float $\rightarrow$ float $\rightarrow$ [Coord $\times$ Coord];
atom line $\in$ string $\rightarrow$ [Coord $\times$ Coord];

def atriangle $\in$ [Coord $\times$ Coord];
atriangle $[b1, b2]$ =
{
trans 10 0 $[b1, b2]$,
trans 5 5 $[b1, top]$,
line "`normal`" $[b1, b2]$,
line "`normal`" $[b1, top]$,
line "`normal`" $[b2, top]$
};

Describes:



Chapter 8

A comprehensive set of semantic functions for digital circuits

J. Vanslembrouck, ABT

8.1 Introduction

This report deals with the definition of semantics, by means of semantic or meaning functions, for digital circuits described with the hardware description language Glass . The starting point for the implementation will be the kernel abstract syntax tree produced by the language's macro-expander. Instead of describing each semantic function on its own, attention will be paid in the first place to the designer's needs at several stages in the design process. This way of description will help to situate every meaning function better in this process than a simple summation could have done. It will also make the relationships between the functions clearer.

The emphasis will be on rapid prototyping, i.e. instead of implementing algorithms for the efficient calculation of semantic aspects, the algebraic formulation of the semantics will be taken as much as possible as the implementation. This has the advantage that several semantics of the language can be defined within the timeframe of the Esprit 881 project and that the implementation should be "obviously" correct. Afterwards, the results of a more efficient algorithmic implementation can be checked against those of the prototype.

This report is organized as follows. In section 8.2, the terminology used in the rest of the paper is introduced. In section 8.3, building blocks for composing semantic functions are briefly described. Section 8.5 handles semantic functions at the logic gate level, section 8.6 describes possible semantics at the switch level and section 8.7 at the circuit level. Translation to other hardware description languages is discussed in section 8.8. The last section provides some conclusions.

8.2 Explanation of terminology

In this section the terminology which will be used in the rest of the report will be explained. Readers who are familiar with digital hardware and the FORFUN project may skip this section.

8.2.1 System semantics

Semantics (meaning) is introduced explicitly by means of a *model* [Men79], consisting of a *meaning function* m mapping elements of the set of sentences S of a language into elements of a *domain of interpretation* D, which is the set of possible meanings ($\mathrm{m} \in \mathrm{S} \rightarrow \mathrm{D}$) [Bou88].

System semantics is the description of system properties by means of semantic functions. In the model for describing a particular property, the domain of interpretation of the sentences represents the values that the property may assume.

8.2.2 Digital circuits

Digital circuits can be divided into two classes: combinational and sequential. A combinational circuit is dependent solely on the present input values for its output values. A sequential circuit's output values depend on the sequence of prior inputs (Moore machines) and on the current input values (Mealy machines). Sequential circuits thus have memory to save the state of the circuit. Sequential logic may be synchronous, which means that there is an explicit clock to synchronize the information flow, or asynchronous, where the changes propagate through the circuit without any external synchronization.

8.2.3 Levels of abstraction

(Digital) systems can be described at several *levels of abstraction*, ranging from the behavioural level to the geometric level [Mic87]. To each level corresponds a certain model of viewing the reality. The *behavioural* model is the highest level of abstraction. At this level a system is described in terms of the computations that it performs. The *register transfer level* (RTL) model or *functional* model is used to describe the flow of data and control signals within and between functional units. The circuit description is made up of building blocks such as flip-flops, registers, selectors, counters, encoders, decoders, arithmetic logic units (ALUs), and elements of similar level of complexity. A *logic* or *gate level* model describes a system as an interconnection of logic gates, while a *switch level* model describes a circuit as an interconnection of transistors. A *circuit level* model describes a circuit in terms of such things as resistances, capacitances, voltage and current sources. Finally, the *geometric level* model describes a circuit in terms of physical shapes.

At each of these levels, Glass describes a system as an interconnection of subsystems. These subsystems can again be described as an interconnection of other subsystems. This way of description is called *hierarchical*. At each of these levels, some components will be primitive (atomic), i.e. they cannot be described in terms of other components.

A system at the behavioural level can be described in Glass as a single atomic definition, giving only a definition of the system's inputs and outputs. The behaviour of the system should be described by means of a suitable behavioural oriented language. Primitive components at one level of abstraction will typically be non-primitive at a lower level. For example, a NAND gate will be a primitive component at the gate level, but will be described as an interconnection of transistors at the switch level. Systems at the logic gate level and higher are usually described using the directional subset of Glass based on the λ-calculus, while systems at lower levels, where the direction of information flow is not always obvious anymore, are described using the adirectional subset of the language which is based on the σ-calculus.

Multi-level descriptions describe some components at a higher level of abstraction than others. For example, the combinational part of a sequential circuit may be described in terms of logic gates, while its latches can be considered to be primitive (RTL) devices (although these latches are built up from logic gates).

8.2.4 Compositionality

Meaning functions are typically defined by giving a semantic rule (a partial image definition) for each syntactic rule in the grammar. Together, the semantic rules constitute the complete image definition. A meaning function is said to satisfy the principle of *compositionality* if the meaning of a composite depends only on the meanings of the constituents. Compositionality ensures the existence of useful formal properties in view of reasoning and theorem proving using the meaning functions as basic rules. Hence it can be seen as a qualitative measure for the match between language constructs and system concepts [Bou87].

In the implementation of the semantic functions described in this report, we have strived as much as possible for compositionality. Compositionality has alsot the benefit that the *hierarchy* of the Glass system descriptions remains preserved.

8.2.5 Transistor technology

From the switch level on and lower, transistor technology plays an important role for describing the semantic aspects of circuits. Because Metal Oxide Semiconductor (MOS) technology is most often used for VLSI circuits, only models concerning MOS digital circuits will be considered in this report. Two types of transistors can be distinguished. An n-transistor is a device which conducts (i.e. connects its source to its drain with a low resistance) when the gate to source voltage V_{GS} is higher than a certain threshold value V_{th} . When $V_{th} < 0$ the transistor is a *depletion mode* transistor, otherwise it is called an *enhancement mode* transistor. Depletion mode transistors are often used as pull-up or pull-down resistor (driver) in digital circuits. In this case, a short is made between the gate and the source. A p-transistor conducts when its gate to source voltage is lower than a certain value.

Various MOS design styles exist. NMOS circuits only use n-transistors (in enhancement or depletion mode). CMOS circuits use both types. The p-part network is often dual to the n-part. A distinction can be made between static and dynamic circuits. In *static* circuits, there is always a path of conducting transistors from the output of the circuit or any node in the circuit holding information to the voltage source or to the ground. In *dynamic* circuits on the other hand, information can be saved as charge on the nodes using the node's capacitance. *Pass* transistors are used to pass the information at one node to another node in the circuit. These transistors are often switched by a clock signal.

The symbols for both transistor types are given in figure 8.1. There is no real distinction between source and drain; normally the port which is connected to the highest voltage level is called the drain.

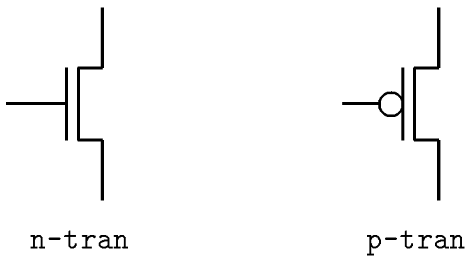


Figure 8.1: MOS transistor symbols

8.2.6 Simulation

Simulation is the imitation of the behaviour of a system on a computer. In order to imitate this behaviour, simulation employs models. A model is an imperfect replica. It must be accurate enough to imitate the behaviour of the variables of interest in the process or system being studied, but not so complex to obscure details of the variables and their relationships or so intricate that it can only be created through use of exorbitant amounts of resources [Mic87].

Simulators can be characterized as compiled or interpreted. The *compiled* simulator is created by converting a network description directly into an assembler program or program in an imperative language such as C . These programs reflect the functions and interconnections of the individual elements in the circuit. The *interpretative* simulator operates on a circuit description, without converting the network into an imperative program. The event-driven/selective-trace simulator is the best known example of this category [Mic87]. An brief overview of the history of logic simulation can be found in [Ulr86].

A fundamental idea in logic simulation is the U (or X) state. This state is used to indicate that an element output is "uninitialized" or "unknown": it is either 0 or 1. Setting an entire network first to Xs and then initializing it by driving 0s and 1s into primary inputs has become the standard method of initialization.

With the advent of new and more powerful languages, the distinction between compiled and interpreted simulation almost disappears. For example, translating a circuit description into a Miranda [Tur79] or Prolog [Clo87] script is a compilation process, but those two languages are themselves interpreted. When languages supporting concurrency, such as CCITT's SDL [Sdl88] or Ada [Ada83], are used, possibilities arise for executing only that part of the code which corresponds to the active parts in the circuit.

Register transfer level and behavioural simulators are less accurate than gate level simulators, but modeling and simulating is easier and faster. The combination of these methods with gate-level methods leads to successful *multi-level* simulation and thus to an acceptable mixture of modeling power, accuracy, and CPU speed.

Switch-level simulation was developed to solve the problem of MOS-design. What is needed is to simulate bidirectional information flow an to reconfigure bidirectional nets. Associated with this are new nodal states such as the Z (high-impedance) state, and state attributes that add strength to the ordinary logic states. These extra states are needed only for switch-level simulation, and the three older states (0, 1, X) remain sufficient for gate- and RTL-level simulation.

8.2.7 Symbolic simulation and verification

Conventional simulators, using actual values for input and state, verify the correct functionality of a logic design only for the particular data simulated. The user has to check the calculated values against the values he expected. The user is left with the difficult task of choosing data that will detect every design error and deciding when a sufficient set of tests has been applied.

To verify the correctness of a digital design in a more rigorous fashion, they can be simulated symbolically. The concept of a *symbolic simulator* used in [Bry85] resembles a conventional logic simulator whereby a description is loaded into memory, the user specifies a sequence of patterns to be applied to the inputs of the circuit, and the program models how the circuit would behave for these patterns. However, unlike a conventional simulator in which the patterns consist of only constant values (e.g. numbers or elements from enumeration types at the RTL level, 0 and 1 at the gate and switch level), and the program models only how the circuit would behave for these specific data, a symbolic simulator can accept patterns consisting of variables (as well as constants), and the program computes the functions describing the behaviour of the circuit for the set of all possible data represented by these variables. By setting some inputs to constants and some to variables, a symbolic simulator can perform conventional simulation (all inputs set to constants), symbolic analysis (all inputs set to variables), or a hybrid of the two. Furthermore, a symbolic simulator can represent the behaviour of a sequential circuit by computing the sequence of functions that would appear at each output as a function of the sequences of variables that have been applied to each input [Bry85].

Verification requires two documents: the characteristics of an implementation should be compared to those of a specification (which is believed to be correct). Verification of circuits can take place at the same level of abstraction; for example a second but faster implementation of a circuit can be verified against a first, slower, but correct reference implementation, which is considered to be the specification. More interesting is the comparison of the behaviour from an implementation at a lower level of abstraction against a specification at a higher level. Most often, this will require some conversion between these two levels. For example, states and inputs at the RTL level can be described using numbers and/or enumeration types. The system's behaviour at the gate level on the other hand is described in terms of Boolean expressions. Because the behaviour at the switch-level can also be described by Boolean expressions, a conversion between switch and logic gate level expressions is usually not necessary. However, because these Boolean expressions will often have a different form, a tautology checker will be necessary to compare both forms.

8.3 Semantic function organization

The starting point for the implementation of semantic functions is kernel Glass abstract syntax tree (AST) which is the output of the macro expander (figure 8.2)

Semantic functions can be implemented in several ways according to the spectrum *interpreted* versus *compiled*.

- Compiled functions : the AST is translated to a target language such as Miranda , Prolog, HOL, or to the input format of an existing CAD tool. The result of the function application is a text (which can be stored in a file). Therefore, the last step in the translation is always an unparsing step, preceeded by none or more transformation steps. Compiling to a target language or tool has a great advantages: the full power of languages like Miranda or Prolog or dedicated tools can be used, without the prerequisite that the user needs to know anything about the implementation of these languages or tools. The interpreta-

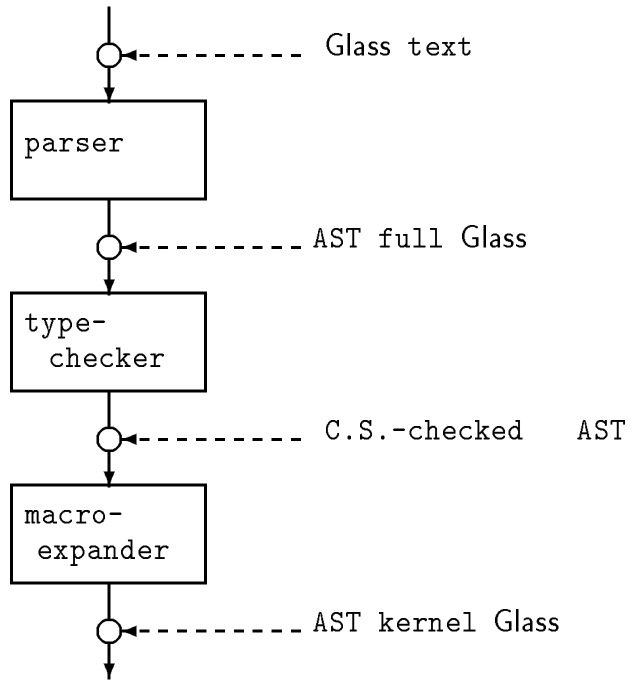


Figure 8.2: Glass front-end chain

tion is in fact postponed to the target language interpreter. The CAD tool can be considered as a special purpose interpreter for some specific application domain.

- Interpreted functions: the AST itself is taken as input by the semantic function, together with the necessary input.

- Some functions have aspects belonging to both of the above categories. The AST is taken as input to produce an intermediate data structure, such as an arrowlist or matrix. This data structure is then used in the next step, together with input information, by an algorithm to calculate the meaning of the component.

As mentioned above, the translation of the AST to a target language or data structure often requires a transformation of this tree. The transformation of the AST is split up in several steps, keeping each step simple and understandable. Transformation steps may be combined to a chain, making it possible to obtain several semantic functions using one set of building blocks. Another advantage of this approach is that an extension or refinement of the semantic function usually requires only the inclusion of one or more additional steps.

All the necessary translations from this AST to some target language or data structure have been implemented in Miranda [Tur85]. The main reasons for this decision are:

- The abstract syntax tree itself is a Miranda algebraic data structure, which can immediately be loaded by the Miranda interpreter.

- The use of algebraic data types allows the implementor of semantic functions to reason at a very high level; he/she doesn't have to worry about the details of allocation and linking of dynamic memory cells but can concentrate instead on the meaning of syntactic patterns of the language.

- As long as the abstract syntax definition Glass is not completely stable, it is recommendable to use a language which allows to write compact and readable code, in order to shorten the time for updates.

- The Miranda definitions can be seen as a formal design specification, which can be used as input for an implementation in C or Pascal afterwards, and as documentation for this code. Because there exists already a translation from the Miranda algebraic data structures to C (and Pascal) data structures [Ree89] , and because the back-end written in Miranda does not rely explicitly on the lazy evaluation mechanism of this language, it should not be a difficult task to write a Miranda to C translator which translates the Miranda form to C .

- Miranda fits in the philosophy of the project to use a functional language for describing and reasoning about hardware. Using an imperative language instead could be interpreted by outsiders as a lack of confidence in functional languages by the project group.

- Because of Miranda 's referential transparency, transformation steps do not influence each other by means of side-effects and can therefore separately be tested.

All the implemented transformation steps transform the abstract syntax tree to another tree which has the same Miranda algebraic data type definition. In general, steps can be envisaged which produce a tree with another definition, which is closer to the target language.

Not all blocks in the chain have to transform the tree. Indeed, some steps merely walk through it to collect information which will be used by blocks further in the chain. The information gathered by these blocks has much in common with synthesized attributes in attribute grammars. This information used by other blocks can be considered as inherited attributes. Many variants of these basic building blocks exist. A possible chain of blocks is illustrated in figure 8.3

In figure 8.3, block 1 only collects information in the tree and passes this information to block 2 which uses it to transform the tree. Block 3 transforms the tree and at the same time collects information which is used during transformation. For example, if the Glass description is ordered in

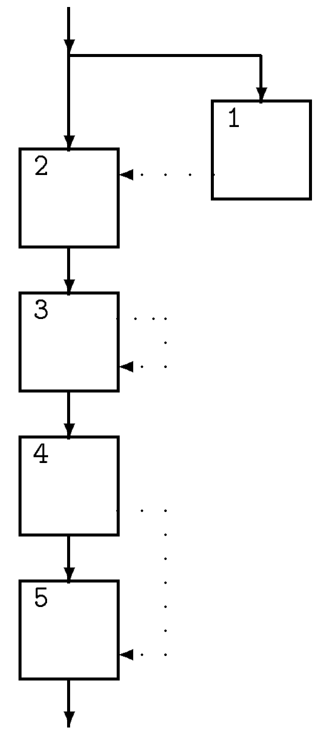


Figure 8.3: A chain of building blocks

the way that system definitions always preceed their application, information collected during the transformation of a definition can be stored and used further on when applications are encountered. One example is the removal of registers from circuits which introduces extra inputs and outputs and thus new formal connection and output arguments. Blocks 4 collects information which is passed together with the transformed tree to block 5.

The concept of building blocks has proved to be very useful in practice. The critical path semantic function (see below) was the first one implemented at Bell. Almost all of its blocks were reusable for other translations. The unparser step is of course typical for each function. For other semantic functions, some intermediate building blocks can be omitted or should be replaced by others. Because the purpose of many blocks is the transformation of the the kernel abstract syntax tree to an still simpler form, the followed strategy can be considered as a kind of configurable macro-expansion or flattening process,.

All building blocks which can be useful for several semantic functions have been placed in a separate UNIX directory, called miralib (see figure 8.4). Each semantic function (semfun.m) has its own directory. A file maininc.m is present which includes semfun.m and all necessary blocks from the miralib directory. Finally, the file main.m is itself very small and contains the top level function. The organization of the building blocks into a chain (see figure 8.3) is defined by this function.

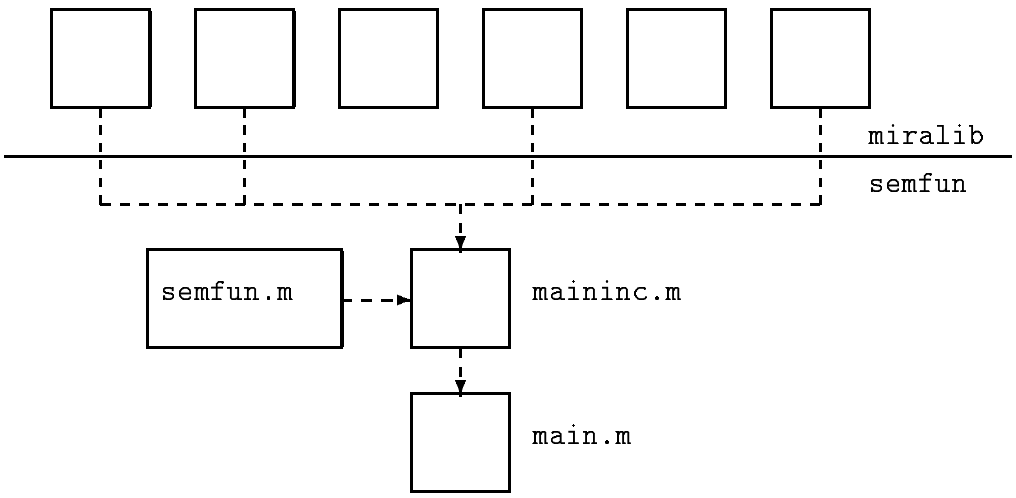


Figure 8.4: A typical semantic function organization

8.4 Semantic function overview

An overview of the implemented semantic functions is given in figure 8.5. The first column in this table gives the name of the directory where the implementation can be found. Column two refers to the section in this report where the function is described. The third column gives the name of the target language for compiled functions. It is not a coincidence that only the compiled functions preserve the hierarchy; all interpreted functions use a flattened description. Column four describes the domain of interpretation. In all cases where the Boolean set $\mathbb{B}$is used, an easy extension to a ternary set is possible. The domain *expr* stands for Boolean expressions. The last column gives a small description of the function. The simplex and multiplex semantic functions have been implemented at Bell and Sagantec (sfdig version).

8.5 Logic gate level

8.5.1 Logic level delay models

The purpose of logic gates is to perform Boolean logic operations. Therefore, the two-valued Boolean algebra is most often used to model the behaviour at the logic gate level. A third state (called U or X) [Eic65] is often added to indicate that an element output is in a temporarily unsettled or hazard state between 0 and 1.

Directory	Ref	Target	DomInt	Description
cpm	8.5.5.1	Miranda	$\mathbb{R}$	worst case delay
mossimII	8.6.4.2	Miranda	$\mathbb{B}$	switch level simulator
mult	8.5.4.1	Miranda	$\mathbb{B}^*$	multiplex behaviour
nndf	8.8.2	NNDF		translation to NNDF
prolog	8.5.3.2	Prolog	$\mathbb{B}$	simplex behaviour
senspath	8.6.7.2	(interp.)	expr	sensitized paths
sfdig	8.5.3.1	Miranda	$\mathbb{B}$	simplex behaviour
	8.5.4.1	Miranda	$\mathbb{B}^*$	multiplex behaviour
simlog	8.5.7.4	SIMLOG	$(\mathbb{B}\times\ \mathbb{R})*$	simulator input
simp	8.5.3.1	Miranda	$\mathbb{B}$	simplex behaviour
simulation	8.5.7.2	(interp.)	$(\mathbb{B}\times\ \mathbb{R})*$	event-driven simulator
symbolic	8.5.9.3	(interp.)	expr	symbolic simulator

Figure 8.5: Semantic function overview

Because gates are physical devices, more aspects than just their mathematical function are important. The most important are delay and dissipation. We will restrict ourselves to delay models. There are several kinds of delays that exist in switching devices, and there are some that do not exist but are used, nonetheless, in simulation [Mic87].

Propagation delay is a measure of the time interval between the appearance of a change on an input of a device and the appearance of the corresponding change at an output. It is measured at specified reference points on the input and output waveform. When the output waveform is changing from a low to a high level, the delay is referred to as *rise time*. A transition from high to low is called *fall time*. The *inertial* delay is the minimun amount of time during which a signal must persist at the input of a device in order to affect a change at the output.

The *transport delay* model semantics imply that a gate's output will respond faithfully to input signals no matter how briefly the stimulus may be (i.e. we assume that the gate's inertial delay equals 0) [Ayl86]. Zero and unit-delay are simple transport delays. *Zero-delay* means that the elements respond immediately to changes at their inputs. In the *unit-delay* model, we assume that all elements have the same, non-zero delay. In the *nominal delay* model, each component can be assigned a different delay. *Min/max delay* assumes that the output of a gate may reach a new state as early as the minimum delay period, and as late as the maximum delay period, with gate output set to X between the two delays [Ulr86].

8.5.2 Overview

The first semantic functions (sections 8.5.3 and 8.5.4) involve the simulation of logic circuits by translating the Glass description to a program in a declarative language. The behaviour of combinational and synchronous sequential circuits will be investigated assuming the zero delay model. These functions correspond to the *simplex* and *multiplex* semantic functions in [Bou88].

Before considering nominal delay simulation, timing analysis will be described, where critical paths in circuits will be investigated. The results of this analysis can be used to determine the maximum frequency at which inputs and clocks can change in circuits. This analysis can be seen as complementary to the simulation described above. A refinement which eliminates false paths will be described next, followed by a formulation of the test generation problem.

Delay and behavioural information together can be used to simulate circuits with much more accuracy. A translation will be made from Glass to input for an industrial event-driven simulator. A simple event-driven simulator will be described. This approach can be used to simulate asynchronous sequential circuits.

Figure 8.6 illustrates the tradeoff between behavioural and timing information.

- a). The pure Boolean behaviour of logic gates will be considered without any timing information.
- b). Only the delay of the logic gates will be taken in account. This allows for calculating critical paths.
- c). Using also behavioural information allows to eliminate false paths.
- d). Accurate simulation becomes possible when both behaviour and delay are taken in account.

8.5.3 Combinational circuits - Zero delay simulation

For combinational circuits, the designer is interested in the I/O behaviour of the hardware which should realize a Boolean expression. Simple compiled and interpreted semantic functions are possible. Compilations are possible with imperative languages such as C or Pascal as target. We will only consider translations to Miranda and Prolog.

The following Glass description will be used to illustrate the translation. The schematics of the circuit can be found in figure 8.7.

```
BASETYPE B ;

ATOM
  and :- B&B => B,
```

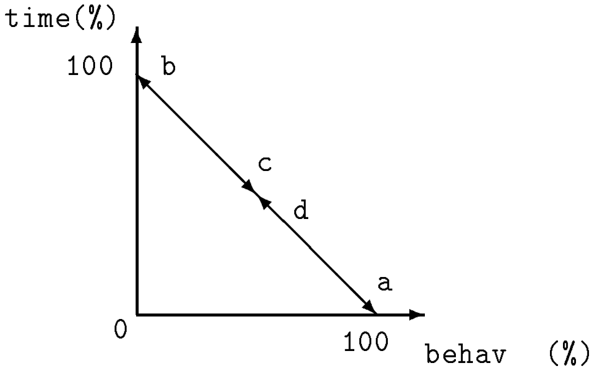


Figure 8.6: Behaviour/timing tradeoff

```
  or  :- B&B => B,
  not :- B => B ;


DEF
    imec :- B&B&B => B;
    imec [i,j,k] = sel [and[k,j], and[and[i,j],k], i] ;

DEF
    sel :- B&B&B => B;
    sel [s,a,b]  = or [and [not s, a], and [s,b]] ;
```

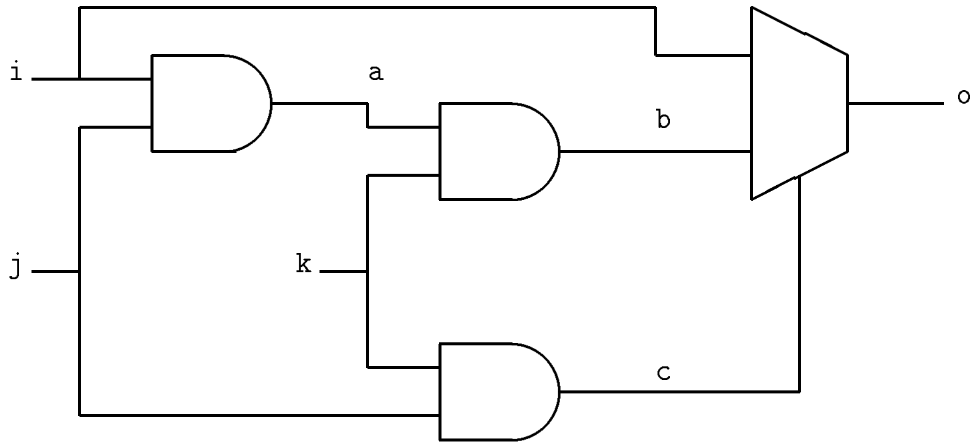


Figure 8.7: Combinational circuit example

8.5.3.1 Miranda

Because the concrete syntax for the directional subset of Glass is very much alike that of Miranda (with as consequence that the abstract syntaxes should not differ considerablu either), the translation from the Glass abstract syntax to Miranda essentially involves an unparsing step.

Translating the example Glass description leads to the following Miranda script:

```
%include "atoms_sp.m"

imec (i,j,k)
 = sel (sp_and (k,j),sp_and (sp_and (i,j),k),i)


sel (s,a,b)
 = sp_or (sp_and (sp_not s,a),sp_and (s,b))
```

The semantics of the AND gate in this model is given by

```
sp_and (B0, b ) = B0
sp_and (B1, b ) = b
```

B0 and *B1* are constants of a Miranda algebraic data type. Ternary simulation is possible by simple redefinition of the semantics of the atoms in the ternary algebra.

8.5.3.2 Prolog

Prolog does not use functions (with the separation between inputs and outputs) but relations. A translation will involve the construction of a predicate which combines inputs and outputs from Glass descriptions. Another major consequence is that nesting of function applications is (of course) impossible. A transformation step is needed which removes applications within applications from Glass descriptions and which generates names for all previously nameless connections.

The translation of the Glass description leads to the following Prolog *rules*

```
imec(I,J,K,Sel0):-
    and(K,J,And00),
    and(I,J,And0010),
    and(And0010,K,And01),
    sel(And00,And01,I,Sel0).

sel(S,A,B,Or0):-
    p_not(S,Not0000),
    and(S,B,And01),
    and(Not0000,A,And00),
```

The semantics of the AND gate is given by the following Prolog *facts*

```
and(b0,X,b0).
and(b1,X,X ).
```

Again, ternary extension is very simple. Names have been generated to denote nameless nodes in the Glass description. For example, the names *And00*, *And01*, *And0010* and *Sel0* correspond to c, b, a and o in the schematics of the *imec* circuit. The "conditions" in each description have been ordered in such way that all inputs of each component are known before calculating the output. This avoids useless backtracking if the user simulates the circuit in the forward direction (given the inputs, calculate the output). This ordering poses no problems because combinational circuits always have a directed acyclic graph (DAG) representation. Simulation in the backward direction or a mixture of both modes is also possible. In the backward direction, the user is interested in which inputs cause a certain output.

8.5.4 Synchronous sequential circuits - Zero delay simulation

In sequential circuits, memory elements are present which save information present at some internal points inside the circuit. Most often these memory elements are placed in feedback loops. A clock is present which activates the memory elements. This duration of this activation is long enough in order to pass the input of the memory element to its output, but short enough to avoid the propagation of new values which reach the memory element's inputs.

Simulating sequential circuits by applying a single signal, or by applying all possible signal combinations to each input, has little value. Indeed, the state of the circuit plays an important role and has the effect that identical inputs to the system not always lead to identical outputs. Simulation is best accomplished by applying a *stream* of signals to the circuit. Of course, only the correctness of the circuit can be verified for the applied test sequences.

In the Miranda and Prolog implementation, the (input and output) signal streams will be represented by lists. No timing information is present. The function simulating the circuit will take one value from each list, together with the old state of the circuit, for calculating the new output and the next state. Therefore, we will have to assume that corresponding elements in these lists are given at the same moment in time. In other words, if one input is changing with a much higher frequency than another, we will have to elongate the second list to "synchronize" the streams. If required, a preprocessor function can be written which, given input streams containing tuples of values and the time at which these values are applied, returns lists of equal length in which the lists have been synchronized. In a next section, we will see some methods to calculate the delay in the combinational parts of the circuit. From this delay, the maximum clock frequency can be calculated.

Simulating synchronous sequential circuits does not imply that the clock should be implicit in the description and simulation. Indeed, the clock can be considered as just another input. The crucial difference with simulating asynchronous circuits is that in these circuits, multiple transitions can occur in feedback loops. This is not the case here.

The following Glass description will be used to illustrate the translation to Miranda and Prolog. The *divide* circuit is a very simple divide-by-2 frequency divider, such that a pulse train arriving at the input *cl* will appear at the output *o* halved in frequency. The input to the circuit is the clock of the d-flipflop. The second circuit is a simple finite state automaton (fsa). An extra "input" has been added to the description of the *dff* and *fsa* systems in order to denote the initial value at the fliplfop's output. There is no other way for doing this in Glass .

```
BASETYPE B;

ATOM inv :- B => B,
     next :- B&B => B,
     out  :- B&B => B,
     dff  :- B&B => B ,
     dffc :- B&B => B ;

DEF divide :- B => B;
    divide cl = o
       where
         o = dffc [cl, inv o]
       endwhere ;

DEF fsa :- B&B => B;
    fsa [q0,is] = out [is,qs]
       where
         qs = dff [q0, next [is,qs]]
       endwhere ;
```

8.5.4.1 Miranda

Miranda is an excellent target language for the simulation of synchronous sequential circuits with feedback loops. Because of the lazy evaluation mechanism, the structure of the Glass description doesn't have to be changed to make simulation possible. However, special attention should be paid to the initial values of the registers (if these are not provided by the Glass description) to avoid so-called "black holes" caused by the absence of a "seed" to build up the signal lists in feedback loops.

The translation of the divider circuit, together with the behavioural definitions for the atoms, is given below:

```
mp_inv (a:x) = sp_inv a : mp_inv x
```

```
mp_inv []    = []

dffc (cs,is) = dffc' BX cs is

dffc' q0 (B0:cs) (o:os) = q0 : dffc' q0 cs os
dffc' q0 (B1:cs) (o:os) =  o : dffc'  o cs os
dffc' q0  cs       os   = []

divide cl
 = o
   where
     o = dffc (cl,inv o)
```

The following is the translation of the fsa, again accompanied by the necessary behavioural definitions for the atoms. The basic type B from the Glass description has been interpreted as the numeric type in Miranda . Because no explicit type definition is generated, the user remains free to interpret this basic type in the way he wants.

```
out (x:xs,y:ys) = x*y : out (xs,ys)
out (xs,ys) = []

next (x:xs,y:ys) = x+y : next (xs,ys)
next (xs,ys) = []

dff (q0,qs) = q0:qs

fsa (q0,is)
 = out (is,qs)
   where
     qs = dff (q0,next (is,qs))
```

8.5.4.2 Prolog

The translation to Prolog has not been implemented yet. The possible translation described here is based on work in [Clo87] and [Clo87]. Because Prolog does not have lazy evaluation, a different approach has to be followed. A "driver" rule is used which takes care of the circuit's state. This recursive rule drives the actual circuit.

The behaviour of a dff (in which the clock signal is explicitely considered) and inverter is described by the following rules:

```
/* dffc( D, C, Q,Qn) */
   dffc(b1,b0,b1,b1).   /* C = 0 => Qn = Q */
   dffc(b1,b0,b0,b0).
   dffc(b0,b0,b1,b1).
   dffc(b0,b0,b0,b0).
```

```
dffc(b1,b1,b1,b1).    /* C = 1 => Qn = D */
dffc(b1,b1,b0,b1).
dffc(b0,b1,b1,b0).
dffc(b0,b1,b0,b0).

inv(b0,b1).
inv(b1,b0).
```

In the definition of the flipflop, D stands for the data input, C for the clock input, Q for the present state (which equals the output of the element) and Qn for the new state. The Prolog rules for the divider circuit are given by

```
divide(In,Qo,Out) :- inv(Qo,D),
                     dff(D,In,Qo,Out).

divides([],Qo,[]).
divides([I|Is],Qo,[O|Os]) :- divide(I,Qo,O), divides(Is,O,Os).
```

The *divides* rule is the "driver" of the *divide* circuit.

The second example is again our simple fsa.

```
dffX(X,X).

next(X,Y,Z) :- Z is X+Y.
out(X,Y,Z) :- Z is X*Y.

fsa(In,Qo,Qn,Out) :- next(In,Qo,Qn),
                     dff(Qo,S),
                     out(In,Qo,Out).

fsas([],Qo,[]).
fsas([I|Is],Qo,[O|Os]) :- fsa(I,Qo,Qn,O), fsas(Is,Qn,Os).
```

8.5.5 Timing analysis

The zero delay simulator cannot predict the relative time of changes in a circuit. A nominal delay simulator (see further) can determine the relative time of occurence of signal changes on circuits elements and can explicitly identify multiple signal changes which occur within a specified interval [Mic87].

As systems grow larger, and design, simulation, test, and maintenance grow more difficult, synchronous design techniques become more attractive. The use of master clock(s) to synchronize events reduces or eliminates the need for accurate timing simulation and makes it possible to simulate the behaviour in a zero delay environment.

Zero delay simulation requires some means for computing propagation delay along signal paths to insure that they are not too short or too long.

With excessive delay, a signal will not reach its destination before the clocks pulse. If the delay is too short, the hold time requirements for a flip-flop may be violated. The slowest of all possible signal propagation paths during a machine cycle is called the *critical path* [Jou87]. The technique is called *timing verification*. Two methods for performing timing verification are *path enumeration* and *block-oriented analysis*.

Timing verification is thus a technique which aims at the determination of critical paths without performing simulation [New86]. Simulation approaches to performance analysis are pattern-dependent. Timing analysis on the other hand is pattern-independent. Notice that only the combinational subparts of a sequential circuit have to be taken in account for performing timing analysis. Examples of timing verifiers can be found in [Hit82], [McW80].

8.5.5.1 Critical path calculation

Algorithms, like that the algorithm of Ford, or PERT [Kir66] have been used to calculate the critical path and/or its associated delay (called the worst case delay). In essence, the circuit description (which is in fact a *directed acyclic graph* or DAG with vertices corresponding to nets and edges to circuit elements) is first ordered. This can be accomplished by means of a depth-first method [Aho83]. The nodes in the graph are then visited using this order. At each node, the maximum of the delays is calculated.

The approach taken in this report is compositional Several definitions are possible. In the simplest case, the atoms are defined as functions which calculate the maximum (minimum) of their input values (which correspond to the maximum (minimum) delay at these points) augmented by the delay of the component. This is the definition given in [Bou88]. More interesting definitions are possible. Not only the delay can be returned, but also the path along which this delay occurs. An intermediate form has been implemented in Miranda , in which together with the delay the input and output is given which form the begin and end point of the critical path. By using Miranda as the target language, the ordering of the components in the graph is resolved internally by the Miranda interpreter. The semantic function also accepts sequential circuits. Registers are removed from this description, leading to additional circuit inputs (register outputs) and outputs (register inputs). In this way, a combinational circuit is obtained.

The semantic function will be illustrated by means of a cascade of two simple finite state automata as described by the following Glass text.

```
BASETYPE B ;

ATOM
     and  :- B&B => B,
     or   :- B&B => B,
     xor  :- B&B => B,
```

```
     nor  :- B&B => B,
     dff  :- B => B ;

DEF
    fsa1 :- B => B;
    fsa1 i = o
      WHERE
        o = or [i, q] ;
        q = dff (nor [i,q])
      ENDWHERE ;

DEF
    fsa2 :- B => B;
    fsa2 i = o
      WHERE
        o = and [j,q] ;
        q = dff (xor [j,q]) ;
        j = fsa1 i
      ENDWHERE
```

This Glass description is translated into the following Miranda script. The function *newarrl* generates an input arrow which will be applied to the circuit. Because critical path analysis in input independent, (which in our case means that a constant input has to be applied to the circuit), the inputs are generated by the semantic function.

```
%include "aux.m"
%include "atomics.m"

i = newarrl "i"

fsa1 = pp2 (fsa1' i)
fsa1' i
 = (rf "o" o,rf "dff'fsa1'fsa11'i" fsa11)
   where
     o = s_or (i,q)
     q = newarrl "dff'fsa1'q'o"
     fsa11 = nor' (i,q)

fsa2 = pp3 (fsa2' i)
fsa2' i
 = (rf "o" o,rf "dff'fsa2'fsa21'i" fsa21,wcd31)
   where
     o = s_and (j,q)
     q = newarrl "dff'fsa2'q'o"
     fsa21 = xor' (j,q)
```

```
    (j,wcd31) = fsa1' i

pp2(a1, a2) = heading ++ pprint a1 ++ pprint a2
pp3(a1, a2, a3) = heading ++ pprint a1 ++ pprint a2 ++ pprint a3
```

Names have been generated to uniquely identify additional inputs and outputs caused by the removal of memory elements. The functions *pp2* and *pp3* are pretty-print functions for the output and are generated as well. A simple driver functions is generated for each system definition. The input for the designer is very simple; he only needs to type in the name of the system he wants the delay characteristics of. The result for the cascade of the two fsa's is given below.

```
Miranda fsa2

 The delay seen at each output from each input:
output                   delay     input

o                        5         i
o                        3         dff'fsa2'q'o
o                        5         dff'fsa1'q'o

dff'fsa2'xor1'i          7         i
dff'fsa2'xor1'i          5         dff'fsa2'q'o
dff'fsa2'xor1'i          7         dff'fsa1'q'o

dff'fsa1'nor1'i          4         i
dff'fsa1'nor1'i          4         dff'fsa1'q'o
```

The delays for the OR,AND,NOR and XOR gate are 2,3,4 and 5 time units respectively. The generated names should provide enough information to identify uniquely the intended pins. The definitions for the used atoms, as can be found in the atomics.m file, are

```
s_and (arrlst1, arrlst2) = mkarrlst "s_and" v'and (arrlst1++arrlst2)
s_or  (arrlst1, arrlst2) = mkarrlst "s_or " v'or  (arrlst1++arrlst2)
xor'  (arrlst1, arrlst2) = mkarrlst "xor"   v'xor (arrlst1++arrlst2)
nor'  (arrlst1, arrlst2) = mkarrlst "nor"   v'nor (arrlst1++arrlst2)
```

8.5.5.2 False path elimination

In the calculation of the critical path as described above, the actual behaviour of the circuit components was not taken in account. This can lead to critical paths which are not possible in reality. Consider the circuit in figure 8.7.

One critical path, as calculated with the previous semantic function, will start at input i, go through connection points a and b and ends in output o of the selector (which selects the b input if c is low, and the i input otherwise).

In order to select the b branch, c must be low, which requires j or k to be low. In order to transmit a transition at the i input to b, both j and k must be high, which contradicts with the previous statement. This critical path is called a *false path*.

To calculate the possible delays, the *sensitised* paths must be found. A sensitised path between a circuit's input and output is a path along which a transition at this input will propagate and result in a transition at its output. A false path is a path which is not sensitisable.

For a combinational circuit with n inputs, $n^*2^*2^{(n-1)}$ excitations exist in which all inputs are 0 or 1, except one which make a transition between the two logic levels. However, only a few of these combinations cause the input transition to propagate to the output. In [Ben87], two efficient algorithms are described to avoid this combinatorial explosion.

For the purposes of the FORFUN project, a different approach has been taken. Instead of applying all possible pairs of vectors (in which one element is altered in one vector) to the input and observing the output for a transition, it should pay off to apply a transition at the output (because that is what we are looking for) and calculating the input combinations responsible for this transition.

Because this output-input relation is not a function anymore, Prolog is a natural candidate for the job. The domain will not only include the logical values 0 and 1, but also the transitions $0 \rightarrow 1$ and $1 \rightarrow 0$. The semantic definitions of the atoms will be extended to include also these transitions. In a first (backward) step, a transition will be applied to the output, to produce the inputs which cause this transition. The second (forward) step uses these inputs to calculate the subcircuit outputs and the time at which they occur. This algorithm requires two semantic definitions of the atoms (one with and one without timing calculation), a translation of the Glass description of the circuit to two different forms and the generation of a "driver" rule:

```
imecr(I,J,K,O) :- sel(C,B,I,O),
                  and(K,J,C),
                  and(A,K,B),
                  and(I,J,A).

imect(I,J,K,O,It,Jt,Kt,Ot) :-
               andt(I,J,A,   It,Jt,At),
               andt(A,K,B,   At,Kt,Bt),
               andt(K,J,C,   Kt,Jt,Ct),
               selt(C,B,I,O, Ct,Bt,Ct,Ot).
imect0(I,J,K,O, Ot) :- imect(I,J,K,O,0,0,0,Ot).

try_imec :- gen_transition(O),         /*1*/
              imecr(I,J,K,O),          /*2*/
            unique_complete([I,J,K]),  /*3*/
              imect0(I,J,K,O,Ot),      /*4*/
```

```
                 out3(I,J,K,O,Ot),            /*5*/
                 fail.                        /*6*/
```

The *try_imec* driver rule first generates a transition /*1*/ , applies it to the output of the circuit /*2*/, filters the calculated input vectors where one and only one input changes value /*3*/, applies this input vector to the circuit /*4*/, prints out the result /*5*/ and backtracks /*6*/ to start the process over again. (To augment the speed of this process further by avoiding useless backtracking, all input vectors can be calculated first and stored in a data base. Afterwards, they can be read out an applied to the circuit.)

The result obtained is given below. The first columns correspond to the I,J,K and O connections. It is easy to see which transition causes the longest delay. Because the delay calculation has been split up in its components, it should be easy to reconstruct the critical path. The delay of the AND gate is 3 time units, that of the selector is 5.

```
?- try_imec.
t01 t1 t1 t01         0 +         5
t1 t1 t01 t01         0 +         3 +         5
t1 t01 t1 t01         0 +         3 +         5
t10 t1 t1 t10         0 +         5
t1 t1 t10 t10         0 +         3 +         5
t1 t10 t1 t10         0 +         3 +         3 +         5
no
```

In the example above, six different input vectors are used. Only half of them should be generated. Indeed, for each input vector V in which component V_i makes a transition, a second vector will be generated in which all components are the same, except for component V_i which makes the opposite transition. This second vector is due to the opposite transition at the output. This allows for a further optimization of the function. The proof of this reduction property is easy: applying the first input vector will generate a transition at the output; applying the second input vector will restore the original input, which should result in a transition at the output to restore the original output. Notice that the same reduction is not possible in the forward direction, because it is not always the case that the same paths will be sensitized. This is illustrated by outputs (3) and (6) of the analyzer.

The backward pass has a second application: applying a transition at the output should result in a set of input vectors in which each element should make a transition. If this is not the case for some input pins, there is no connection from this pin to the output, i.e. the input pin has no contribution to the function realised by the circuit.

To conclude this section, the definitions of *and* and *andt* are given. Because of the four-valued logic, these definitions are rather long. The order of the definitions has also its impact on the efficiency, because backtracking will be influenced. The heuristic used to order these definitions is to place input patterns causing an output transition first. If desired, only a subset of the total behaviour can be specified. This will typically generate only a subset of all possible input patterns, but this may be sufficient to find the longest possible path.

```
and(t01,t1,t01).
and(t1,t01,t01).
and(t10,t1,t10).
and(t1,t10,t10).

and(t1,t1,t1).
and(t10,t01,t0).
and(t01,t10,t0).
and(t01,t01,t01).
and(t10,t10,t10).

and(t0,X,t0).
and(X,t0,t0).
```

The definitions of *andt* provide for a simple (compositional) nominal delay simulator which unfortunately does not handle multiple transitions. The simulator can always be replaced by a more accurate one (see section 8.5.7.2). This leads to a system in which the backward pass is performed by a Prolog program and the forward pass by Miranda . A (Prolog) database can be used as interface between both passes. Both stages can independently be optimized.

```
andt(t10,t1, t10, A,B,A+3).
andt(t01,t1, t01, A,B,A+3).
andt(t1, t10,t10, A,B,B+3).
andt(t1, t01,t01, A,B,B+3).

andt(t10,t0, t0, A,B,0).
andt(t01,t0, t0, A,B,0).
andt(t0, t10,t0, A,B,0).
andt(t0, t01,t0, A,B,0).

andt(t1, t0, t0,  A,B,0).
andt(t1, t1, t1,  A,B,0).
andt(t0, t0, t0,  A,B,0).
andt(t0, t1, t0,  A,B,0).

andt(t10,t01,t0, A,B,D) :- imax2(A,B,D,3).
andt(t01,t10,t0, A,B,D) :- imax2(A,B,D,3).
andt(t01,t01,t01,A,B,D+3) :- max(A,B,D).
andt(t10,t10,t10,A,B,D+3) :- min(A,B,D).
```

The *imax2* rule assumes the inertial delay model and is defined as

```
imax2(A,A,0,N).
imax2(A,B,A+N,N) :- A > B, A - B > N, !.
imax2(A,B,0,N)   :- A > B.
imax2(A,B,B+N,N) :- A < B, B - A > N, !.
imax2(A,B,0,N)   :- A < B.
```

8.5.6 Test pattern generation

The obvious reason to *test* (already implemented) circuits is to separate good ones from faulty ones [Kir88]. The inputs of the circuit are called primary inputs. These are the only places where we can apply test patterns. The outputs from the circuits are called primary outputs. These are the only places we can observe the effects of the test. *Test pattern generation* is the task of finding a set of input patterns that will fully test the circuit. These patterns, called the test set, must cause all faulty circuits to exhibit different behaviour from good circuits at the primary outputs.

The test set must be reasonable - that is we must be able to apply it economically to all circuits produced. Testing using all possible input patterns will, of course, reveal faulty circuits but will be too expensive for large circuits with many inputs.

In practice, the *single-stuck-at-fault* model has been quite useful. This type of fault manifests itself as a node in the circuit, which permanently assumes a value of 0 or 1. It the node's value is 0, it is called a stuck-at-0 (s@0) fault; if it is a 1, it is called a stuck-at-1 (s@1) fault. We assume that only one fault is present.

Test pattern generation can therefore be reformulated as the task of generating a test set that reveals all detectable single stuck-at-faults. Consider the combinational circuit in figure 8.8. The circuit has a permanent s@0 fault at node n6. This node is called the fault location. To generate a test set, we must produce a difference at the fault location. This process is called *fault sensitizing*. In our case, we must find a pattern that produces a value of 1 at the output of gate G2. We must also propagate this difference to one or more primary outputs. This process is called *fault propagation*, which is the same as finding a sensitized path from node n6 to output n8. This problem has already been described in section 8.5.5.2

In more mathematical terms, the problem (for our circuit) can be stated as finding a solution to the equations:

$$N6(n1,n2,n3,n4) = 1$$

$$N8(n1,n2,n3,n4,1) \oplus N8(n1,n2,n3,n4,0) = 1$$

Here N6 and N8 denote the value at nodes n6 and n8 respectively. The equation for N8 also depends on the extra input value at node n6. A general

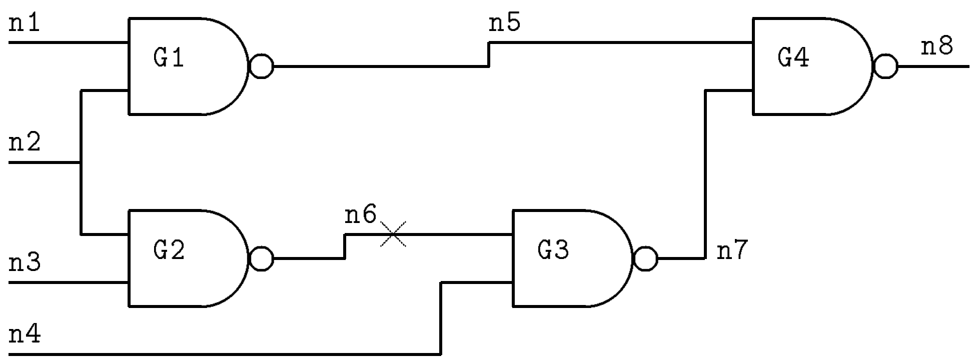


Figure 8.8: Illustration of test pattern generation

algebraic formulation can be found in [Goe81] and [Win88]. There is an immediate translation to Prolog from this mathematical formulation.

```
tpgex(I1,I2,I3,I4, I6,O6, O8) :-
   nand(I1,I2, X5),
   nand(I2,I3, O6),
   nand(I6,I4, X7),
   nand(X5,X7, O8).
```

The extra input I6 will be different for the good and faulty circuit, while output O6 must be set to the opposite value of the fault.

```
try_wind1(I1,I2,I3,I4) :-
    tpgex(I1,I2,I3,I4, t0,t1, O81),  /* faulty circuit : s@0 */
    tpgex(I1,I2,I3,I4, t1,t1, O82),  /* good circuit :   s@0 */
    O81 \= O82.
```

In the original Glass description (not included here), the node n6 has been cut, leading to an additional output (which must be driven to t1), and an additional input, which is t1 for the good circuit, but always t0 for the faulty one. For large circuits, this formulation is of course very inefficient, but for the purposes of the project, this rapid prototyping approach fits closely to the Glass philosophy.

Three of the best known algorithms for automatic test pattern generation (ATPG) are the D algorithm [Rot66], PODEM (Problem Oriented DEcision making) [Goe81] and Fan [Fuj83]. A highly readable introduction to ATPG and the three algorithms mentioned above can be found in [Kir88].

8.5.7 Asynchronous circuits - Nominal delay simulation

8.5.7.1 Introduction: event-driven/selective-trace simulation

The amount of activity within a circuit at a given time is often minimal and can terminate abruptly [Mic87]. Therefore, simulation of the entire circuit will be very time consuming. It is advantageous to simulate only those elements which are involved in signal changes.

When a signal change occurs on a primary input or the output of any circuit element, then an *event* is said to have occurred on the net driven by that primary input or element. When an event occurs at a net, then all elements driven by that net are simulated. If a signal change on the input to a device does not cause a change on the device output, then simulation is terminated along that signal path. This is called event-driven/selective-trace simulation [New86]. The term *event-driven* refers to the fact that only logic gates whose input value change are processed and the term *selective-trace* refers to the technique used to find the logic gates whose inputs have changed.

Event-driven simulation can be performed in either the nominal delay or the zero delay environment. The *nominal delay* simulator assigns delay values to logic elements based on manufacturers' recommendations or measurements with precision intruments. The *zero delay* simulator totally ignores delay time within a logic element and simply simulates the logic function performed by the element. Some simulators, in trying to strike a compromise between the two, perform a *unit delay* simulation in which each logic element is assigned a fixed delay. The nominal delay simulator can give precise simulation results but at a cost in CPU time. The zero delay simulation will usually run faster but does not indicate when events occur. The unit delay simulator lies between the other two in range of performance. It records time units during simulation, so it does more computation than zero delay simulation, but the mechanism for scheduling events in future time is simpler than for time-based simulation. However, regarding all element delays as being equal can occasionally produce inaccurate results in timing sensitive circuits and provide a false sense of security.

Event-driven nominal delay simulation makes use of the inherent delay in logic elements. However, because of the variability in the delay of individual elements, the events cannot simply be placed into a FIFO queue as they are encountered. The element presently being simulated may have an output change that occurs sooner than some elements previously scheduled and later than others. Hence, it must be scheduled so as to be processed at the right time relative to other changes. The simplest way to do so is through the use of a linear linked list. In this structure an *event notice* is used to describe an activity that must be performed and the time at which it must be performed. The notices are arranged in the order in which they are to be performed.

In order to obtain correct simulation results when an event is simulated, it is necessary that any change on an input causes a simulation using the values that existed on the other inputs at the time when the changing signal arrived at the given input. Therefore, the input change is simulated immediately. But the output value is not altered until a future time corresponding to the delay of the element. This emulates the behaviour of a logic element with finite, nonzero delay. A signal change appears at a gate input and at some future time, depending on element delay, the effects (if any) of that input change appear on the element output and then propagate to the inputs of gates in the fan-out of that gate.

If simulation does not result in a change on the output of an element, it is tempting to assume that nothing further need to be done with that element. However, it is possible for a simulation to indicate no change, but a previously scheduled change occured and has been recorded to be scheduled. For example, suppose a two-input AND gate with propagation delay of 10 ns has values (1,0) on its inputs at time t when a positive pulse of duration 3 ns reaches the second input. The simulation result at time t is a 1, which differs from the 0 presently on the output, so the event is placed on the event list to be processed at time $t + 10$. At $t + 3$, when simulating the change to 0, the simulator computes a 0 on the output, which matches the present value. Therefore, the simulator may conclude erroneously that no scheduling is required.

To solve this problem, a *schedule marker* can be used. It is used as follows: the input event is simulated, and then

- if there is an output change, schedule the change and increment the schedule marker
- if there is no output change, and the schedule marker is 0, no activity is required
- if there is no output change, and the schedule marker is > 0, an action has to be taken. This will be described below.

When an output event is processed, the schedule marker is decremented.

The third case described above requires some action. This case occurs when two or more input changes occur during the propagation time of a first change. The actions which can be taken are:

- the change is scheduled and the marker is incremented. This corresponds to the transport delay model, in which all changes, no matter how short, are propagated.
- the previously scheduled event is cancelled, the new event is discarded, and the marker is descremented. This corresponds to the inertial delay model in which the propagation delay is equal to the inertial delay of the gate. The marker will only take the values 0 and 1 in a two-valued logic and can thus be replaced by a boolean value.

- if the inertial delay is taken to be less than the propagation delay, the action to be taken depends on the length of the pulse, Consider a two-input AND gate with t_p ns propagation delay and suppose its present input values are (1,0). If it has an inertial delay of t_i ns and a pulse of duration t_g, $t_g < t_i$, appears on its lower input, then it is scheduled for a change at $t + t_p$, and eventually again at $t + t_p + t_g$. In this case, the first event should be deleted.

The first implementation which will be presented in the next section follows the mechanism described above. It is an interpreted simulator which uses a scheduler. The next sections handles compiled simulators.

8.5.7.2 An event-driven/selective-trace simulator in Miranda

A simple event-driven/selective-trace simulator has been implemented in Miranda . The approach is not compositional. The Glass abstract syntax tree representation of the circuit is input to the scheduler.

The function will be demonstrated by means of the following Glass description:

```
DEF aoa :- B&B&B => B ;
    aoa [in1,in2,s] = and[or[and[in1,in2],s],s] ;
```

Two stimuli vectors *inp1* and *inp2* are defined (see below). The first three elements in each vector are used for the initialisation of the circuit. Only the second element is necessary in fact, because 0 is taken to be the default value. The first element in each triple corresponds to the time, the second to an input node and the third to the (Boolean) value applied to this node. In *inp1*, a pulse with a duration of 4 time units is applied to the *in1* input at time 0. In *inp2* this duration has been increased to 6 units. The propagation delay of the AND and OR gate are 3 and 5 time units respectively. These times are defined in the "atomics.m" file. The circuit has been simulated in the inertial delay mode (*sw2*). The other modes are transport delay (*sw1*) and a mode with the inertial delay different from the propagation delay (*sw3*). If necessary, the user can assign initial values to internal nodes (default value = 0). We have used the default assignment by the system (hence [ ] as third argument to the *sim* function).

```
inp1 = [(-100,"in1",0),(-100,"in2",1),(-100,"s",0),
        (0,"in1",1),(4,"in1",0)]
inp2 = [(-100,"in1",0),(-100,"in2",1),(-100,"s",0),
        (0,"in1",1),(6,"in1",0)]

Miranda sim sw2 "aoa" [] inp1
      time     value          node
      -100         0          in1
      -100         1          in2
```

```
      -100        0          s
         0        1          in1
         3        1          and0000
         4        0          in1
         7        0          and0000

Miranda sim sw2 "aoa" [] inp2
      time     value         node
      -100        0          in1
      -100        1          in2
      -100        0          s
         0        1          in1
         3        1          and0000
         6        0          in1
         8        1          or00
         9        0          and0000
        14        0          or00
```

Additional names have been generated for the internal nodes of the circuit. With a little exercise, the user should be able to find which connections are meant. The simulator only lists the nodes which have effectively changed value.

An auxiliary function "display" is provided which accepts the name of a system and returns the transformed Glass description used by the simulator. In this description, not only nameless connections have received names, but the description has also been flattened. This description can be used for node initialisation purposes or for human interpretation of the simulator's output.

8.5.7.3 A compositional event-driven simulator in Miranda

The simulator described in the previous section was not compositional; the description was first flattened. An attempt to formulate a compositional model will be presented in this section. It elaborates on previous work described in [DeM86]. The definitions of this report have been rewritten to Miranda , refined and tested on some examples.

The domain of interpretation is the set of sequences of time-stamped logic events $(\mathbb{B}\times\ \mathbb{R})*$. The model defines an event (activity) directed logic level simulator. The simulator is not completely event-driven in the sense defined above. Because functions always return a value, the minimum result a function can return is the empty list (which means that none of the input signals caused an output transition). This empty list is input to the next component in the circuit, which will have to perform a minimum of calculation.

The following is the description of an AND gate according to the inertial delay model. Input transitions shorter than the propagation delay *pda* of the gate are not visible at the output.

```
andd B1 b = b
andd B0 b = B0

and1 :: [pair] -> basic -> [pair] -> basic -> [pair]  -> [pair]

and1 [] z1 [] z2 [] = []

and1 [] z1 [] z2 [(zx,zt)]
  = [(zx,zt)], zx ~= andd z1 z2
  = [],        otherwise

and1 [] z1 ((x2,t2):rest2) z2 []
  = (x2, t2+pda): and1 [] z1 rest2 x2 [], z1=B1
  =               and1 [] z1 rest2 x2 [], z1=B0

and1 [] z1 ((x2,t2):rest2) z2 [(zx,zt)]
  = (zx,zt): and1 [] z1 ((x2,t2):rest2) z2 [], zt < t2
  =          and1 [] z1 ((x2,t2):rest2) z2 [], otherwise

and1 ((x1,t1):rest1) z1 [] z2 []
  = (x1, t1+pda): and1 rest1 x1 [] z2 [], z2=B1 & x1 ~= z1
  =               and1 rest1 x1 [] z2 [], z2=B0

and1 ((x1,t1):rest1) z1 [] z2 [(zx,zt)]
  = (zx,zt):and1 ((x1,t1):rest1) z1 [] z2 [], zt < t1
  =         and1 ((x1,t1):rest1) z1 [] z2 [], otherwise

and1 ((x1,t1):rest1) z1  ((x2,t2):rest2) z2 []
  = and1 rest1 x1 ((x2,t2):rest2) z2 [(x1,t1+pda)],    t1<t2  & z2=B1
  = and1 rest1 x1 ((x2,t2):rest2) z2 [],               t1<t2  & z2=B0
  = and1 ((x1,t1):rest1) z1 rest2 x2 [(x2,t2+pda)],    t1>t2  & z1=B1
  = and1 ((x1,t1):rest1) z1 rest2 x2 [],               t1>t2  & z1=B0
  = and1 rest1 x1 rest2 x2 [(x1, t1+pda)],    t1=t2 & xor z1 z2 = B0
  = and1 rest1 x1 rest2 x2 [],                t1=t2 & xor z1 z2 = B1


and1 ((x1,t1):rest1) z1 ((x2,t2):rest2) z2 [(zx,zt)]
  = (zx,zt):and1 ((x1,t1):rest1) z1 ((x2,t2):rest2) z2 [],
                                                   zt<t1 &
                                                   zt<t2 &
                                                   zx ~= andd z1 z2
  = and1 rest1 x1 ((x2,t2):rest2) z2 [(zx,zt)],
                                                   t1<zt &
                                                   andd x1 z2 = zx
  = and1 ((x1,t1):rest1) z1 rest2 x2 [(zx,zt)],
```

```
                                                    t2<zt &
                                                    andd z1 x2 = zx
  = and1 ((x1,t1):rest1) z1 ((x2,t2):rest2) z2 [], otherwise
```

The semantic definitions have additional arguments to save the state of each input. A list argument is provided to save a scheduled output event. This list is either empty or contains one element. This event will either propagate to the output or will be cancelled in case of a spike at the input. The tail recursive definitions reflect a termination of activity in the circuit. A full examination of the definitions is left as an exercise to the reader.

The semantic definition in the way described above is really a burdon. A hard need was felt for a Miranda debugger to debug the definitions (which are highly recursive). It was sometimes difficult to predict which definition would be selected next. Updating definitions to fulfil new requirements is also not a sinecure. The presence of additional arguments makes translation from a Glass description to Miranda rather complicated. This translation has therefore not been implemented.

Finally, the event directed approach is not suitable to simulate circuits with feedback, despite the lazy evaluation mechanism of Miranda . Indeed, one value should always be present on this loop (i.e. Miranda needs at least one "active" value of the list which is built). Because this is not always the case in this approach, the unavoidable consequence is a so-called "black hole". This formulation is therefore only suited for simulating combinational circuits. The effort which has to be spent on developing completely this model does not pay off against the results which can be obtained.

8.5.7.4 Translation to SIMLOG

SIMLOG is an event-driven gate level simulator developed within ITT for the simulation of VLSI circuits [Sim1]. Translating Glass descriptions to SIMLOG offers the Glass user the opportunity to use a product quality simulator.

SIMLOG permits propagation-delay simulation at five levels. These include

- A unit delay is automatically assigned to all gates. This is the default value that is used if the user does not supply delay model information.
- The user assignes a fixed delay, which is referred to as assignable delay, to each individual gate.
- SIMLOG analyzes the circuit from the perspective of the driving gate and calculates propagation delays based on the gate loading for each signal. The propagation delay obtained by this technique is referred to as load-dependent delay.
- SIMLOG calculates the propagation delays based on data supplied from a physical layout of the circuit, on the intrinsic delay of each gate and

on the capacitance and resistance loads in the circuit. In this technique, capacitance loads and interconnection resistances extracted from a physical layout are entered into SIMLOG by the back-annotation facility and are used to modify the load-dependent propagation delays. The propagation delays developed by this technique are referred to as back-annotation delays.

- The propagation delay is calculated based on data for distributed delay processing, where each input connected to a certain output can have a different delay value.

In general, each of the five levels of delay simulation represents a more accurate simulation than the preceding levels.

8.5.8 Asynchronous sequential circuits - Ternary simulation

8.5.8.1 Introduction: delay and race models

In asynchronous sequential circuits, no clocked memory elements are present to avoid multiple transitions in feedback loops. Changes at an internal node propagate through the circuit, change the value of other nodes, and possibly again of the node itself. This can lead to circuits which do not reach a stable new state for certain inputs and old states.

Consider the flipflop-like circuit pictured in figure 8.9 and with the following Glass description:

```
DEF flipflop :- B => B ;
    flipflop x = y1
      where
        y1 = nor2 [x,y2] ;
        y2 = nor2 [x,y1]
      endwhere
```

Suppose that at times $t < 0$, the input is equal to 1. The circuit is in a stable state, with both gate outputs equal to 0. When the input changes to 0, following scenario's are possible:

- the upper nor gate changes faster from state than the lower one: the new state will be $y0 = 1$, $y1 = 0$
- the lower nor gate changes faster from state than the upper one: the new state will be $y0 = 0$, $y1 = 1$
- when both gates change state at the same time (unit-delay model), no new stable state will be reached: the outputs continue cycling between $y0 = y1 = 0$ and $y0 = y1 = 1$.

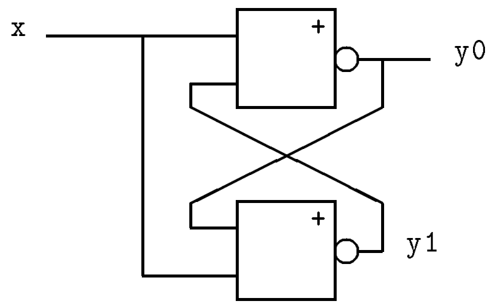


Figure 8.9: Astable flipflop

A *race* is a state in which two or more state variables are unstable - they are competing to win the race, that is, to change value. A race is said to be *critical* if the circuit can end up in different (nontransient) states depending on which state variable changed first in the race [Brz89]. The circuit described above has a critical race for the given initial state.

In [Brz89], an overview is given of several *delay* models which allow to calculate the state of a circuit and of *race* models which analyze races.

The state of a circuit can be represented by a set of feedback variables [Huf64]. These variables correspond to a set of wires with the property that cutting them would break all the loops in the circuit. This is called the *feedback-delay* model in [Brz89]. In the *gate-delay* model, the outputs of all the gates constitute the state variables [Mul59]. In the *gate- and wire-delay* model, not only gate outputs but also wire signals are used as state variables [Brz79].

The first two delay models originally used a binary race model, called the *general multiple winner* (GMW) model. In this model, delays are arbitrary and any nonempty subset of the set of unstable gates is allowed to change. Ternary models have been introduced for the analysis of races in asynchronous gate circuits [Eic65]. Eichelberger proposed a ternary simulation of a circuit using a third value, $\times$, denoting an intermediate or unknown signal. In [Brz89], a new race model is described, called the *extended multiple winner* (XMW) model. It incorporates the multiple winner aspect of the GMW model and the third value $\times$.

Both the XMW model and ternary simulation permit arbitrary delays and are therefore overly pessimistic. The XMW model is computationally intractable. However, in [Brz89] it has been shown that ternary simulation yields the same result as XMW analysis for certain delay models.

Therefore, the (two) algorithms for ternary simulation can be used. Let N be a network, a' an input vector and b the state vector, such that <a',b> is a stable total state. Let a be the new input vector. In a first pass, the algorithm calculates an "intermediate" new state y of the circuit for the total state <lub(a',a), b>. Here *lub* stands for the least upper bound of the two vectors. Because the ternary values are partially ordered as $0 < \times$ and $1 < \times$, the lub of a pair with different values is $\times$. The lub of a pair of equal values is the value itself. The lub of a vector is the vector composed of the lubs of corresponding components. In the second pass, the new state is calculated for <a,b>. The calculation of the states proceeds by succesively recalculating each state variable until convergence is obtained. We assume that the circuit can reach a new state between two input changes.

8.5.8.2 A compositional approach

The following illustrates a compositional approach, in which the algorithms have been "merged" with the original Glass description.

```
dr_flipflop::num->[num]->[num]->[[num]]
dr_flipflop x0 (x1:xs) state0
  = (take 1 state2) : dr_flipflop x1 xs state2
    where
      state2 = flipflop (inpstr2 x1 x1) state1
      state1 = flipflop (inpstr2 x0 x1) state0
dr_flipflop x0 [] state0 = []

flipflop::[num]->[num]->[num]
flipflop xs [y00, y10]
 = limit (transpose [y0s, y1s])
   where
     y0s = y00 : nor2x xs y1s
     y1s = y10 : nor2x xs y0s
```

The function *limit* (from the Miranda standard environment) is used to select the stable state. This function returns the first value in a list which is the same as its successor and can thus be used in testing for convergence. The function *flipflop* is the translation of the Glass description. Additional inputs are generated which represent the state of all gate outputs. The function returns a new state vector of which some elements represent the circuit output as defined in the Glass description.

A "driver" function should be generated which is the translation of the algorithms found in [Brz89]. The flipflop is first driven with an input which is the lub of the old and new input and the state vector of all gate outputs. Therefore, the input at time < 0 (x0) must be provided by the user. The intermediate state *state1* is then used together with the new input to calculate the final state *state2*. In the example, the first element of this state vector is selected to display the circuit's output.

8.5.8.3 Using the event-driven simulator

The event-driven simulator described in section 8.5.7.2 can also be used to analyze races in asynchronous circuits. The major difference with the binary simulation approach is that a transition of an input will be made in two steps. First, the intermediate value X is applied to a changing input, followed by the new value.

This approach has been used to analyse the following latch, in which an erroneous connection was made. This error leads to oscillations for some input sequences. Using the ternary simulation in the way described above will introduce the intermediate value X on unstable gate outputs instead.

```
BASETYPE B
DEF latch :- B&B => B ;
    latch [din,cl] = do
      where
        a   = andx [din,cl] ;
        ncl = inx cl ;
        b   = orx [ncl,do] ;  /* do should be din  !!!*/
        c   = andx [b,e] ;
        do  = norx [a,c] ;
        e   = invx do
      endwhere
```

The approach has some advantages over using the ternary simulation algorithms. First, there is no explicit relaxation computation necessary anymore: in the event-driven approach, all activity dies automatically. Second, the question of which delay model to use in order to minimize the number of states (and thus calculations) but to ensure correctness has little sense: because of the event-driven character, the maximum number of states can be used without giving any overhead.

8.5.9 Verification of synchronous sequential circuits

8.5.9.1 Introduction

One approach to verifying the equivalence of logic circuit descriptions at different levels of abstraction is exhaustive simulation [Dev88]. Unfortunately, the number of simulations required grows exponentially with the number of inputs for even a purely combinational logic circuit, and grows even faster for sequential circuits since all possible input vector sequences have to be simulated to prove equivalence. A different approach is to use *formal verification* techniques which are input pattern independent and can guarantee functional equivalence.

Many formal verification approaches have been taken to prove/disprove the equivalence of two combinational logic circuits, at the gate level and at different levels of abstraction [Don76], [Rot77], [Rot80], [Oda86]. In particular, a package of programs called PROTEUS [Wei86] incorporates several efficient algorithms for verifying combinational logic circuits and has successfully been used on circuits with a large number of gates.

Sequential circuit verification is a considerably more difficult problem in the general case when there is no correspondance between the latches (states) of the two circuits. The approaches taken to solve the sequential verification problem include the use of temporal logic [Bro85] and Prolog [Mar85]. The use of temporal logic helps for asynchronous circuits [Dil85] but is not necessary in the synchronous circuit case. Algorithms have been proposed for formally verifying the equivalence of two gate level sequential circuit descriptions with a differing number of latches using Boolean manipulation [Sup86]. However, because of the intractability of the problem, all the approaches taken so far have been restricted to small to medium sized circuits with a small amount of memory.

In [Dev88], an algorithm is presented for formally verifying the equivalence of two sequential machines, which can be described at the register transfer, state table, or logic level. The approach involves extracting the state transistion graphs (STGs) of the two finite automata - from the register level or the logic gate level circuit. These two STGs are then checked for equivalence. Given two logic level finite automata and a reset state/transfer sequence for each machine, a two-phase enumeration-simulation verification algorithm has been developed [Dev88]. The STG of one of the logic automata is enumerated using a generalized PODEM-based enumeration algorithm for sequential circuits. That is, paths from the reset state in the STG are dynamically enumerated from the first logic automaton and simulated on the second logic automaton.

We will only consider the STG extraction part in this report.

8.5.9.2 STG extraction

Extraction of the STG of the corresponding deterministic finite automata (DFA) from a logic level description consisting of gates and latches can be done in a number of ways [Dev88].

One method which has been used in this context [Sup86], [Oda86] is *symbolic Boolean simulation* of the combinational logic blocks in the circuit, which expresses each output as an algebraic function (with Boolean operations) of the inputs. Given these algebraic functions, the edge transitions between two arbitrary states can be expressed as a conjunction of these functions and possibly their complements. Unfortunately, for the combinational logic blocks with a large number of gates, the size/length of the equations becomes prohibitively large and the extraction process becomes inefficient.

Flattening the logic circuit is another alternative. Flattening involves reducing the combinational logic blocks into two-level (PLA) form. Given a truth table for these combinational logic blocks the edges in the transition table can easily be found by inspection. Flattening, however, may require exponential CPU time and memory requirements and is not viable in many cases.

The approach used in [Dev88] involves using backward justification algorithms to enumerate the ON and OFF sets of a logic function. For more details, the reader is referred to this paper. In the next section, we will describe an implementation of the first approach.

8.5.9.3 A symbolic simulation implementation

The calculation of the Boolean functions realized by the combinational logic blocks in a sequential circuit has been implemented in Miranda . The kernel abstract syntax tree is first compiled to a Miranda script. Registers are removed from the original Glass description and additional inputs (register outputs) and outputs (register inputs) are generated for each removed register. The evaluation of a Miranda definition which corresponds to a Glass system description is a set of Boolean expressions, expressing each output and new state signal in terms of the inputs and old state signals.

The combinational circuit described above will be used to illustrate the function. The produced Miranda script is:

```
%include "atomics.m"

sym'sel = pp1 (sel' (Var "s",Var "a",Var "b"))
sel(s,a,b) = pp1 (sel' (s,a,b))
sel' (s,a,b)
 = (Out "o" o)
   where
     (o) = s_or (and00,and01)
     (and00) = s_and (not0000,a)
     (not0000) = s_not (s)
     (and01) = s_and (s,b)


sym'imec = pp1 (imec' (Var "i",Var "j",Var "k"))
imec(i,j,k) = pp1 (imec' (i,j,k))
imec' (i,j,k)
 = (Out "o" o)
   where
     (o) = sel' (c,b,i)
     (c) = s_and (k,j)
     (b) = s_and (and20,k)
     (and20) = s_and (i,j)
```

For each system definition, two additional functions are generated. The following are some evaluation examples

```
Miranda sym'imec
o = (((~(k.j)).((i.j).k))+((k.j).i))
Miranda imec(B1,B1,B0)
o = 0
Miranda imec(B1,Var "x",Var "y")
o = (((~(y.x)).(x.y))+(y.x))

Miranda sym'sel
o = (((~s).a)+(s.b))
```

The first example illustrates a complete symbolic evaluation of the *imec* circuit. Therefore, the name of the system is just prefixed by "sym'". It is also possible to perform complete numeric simulations (second example) or make a mixture of the two modes (third example). The last example just gives the familiar expression for a *selector* circuit.

The translation of the two fsa's from section 8.5.5.1 is not included. Their symbolic evaluation looks like:

```
Miranda sym'fsa1
o = (i+dff'fsa1'q'o)
dff'fsa1'nor1'i = (~(i+dff'fsa1'q'o))

Miranda sym'fsa2
o = ((i+dff'fsa1'q'o).dff'fsa2'q'o)
dff'fsa2'xor1'i = (((~(i+dff'fsa1'q'o)).dff'fsa2'q'o)+
                   ((i+dff'fsa1'q'o).(~dff'fsa2'q'o)))
dff'fsa1'nor1'i = (~(i+dff'fsa1'q'o))
```

8.5.10 Verification of asynchronous sequential circuits

In [Han85],[Han2], Higher-order logic is used to specify the behaviour of hardware, to describe the implementation at the gate level, and to verify the correctness of this implementation against the specification.

The behaviour of a digital system is defined in terms of the waveforms present at its ports. Although these are often idealized as being perfect, two-valued (binary) waveforms, in reality they are analogue waveforms that are only in a binary state during certain intervals of time. Delay times are introduced to relate the waveforms at input and output ports.

In [Han85], the behaviour of an *and* gate is specified. An implementation of an *nor* gate (built from two *not* gates and one *and* gate) is described. It is verified that this implementation indeed behaves as an *nor* function. In [Han2], the behaviour of an *nand* gate is taken as starting point. The behaviour of an edge-triggered D-type flipflop is formulated, and one possible implementation is described. This correctness of this implementation is verified.

The behavioural specification of an *and* gate [Han85] is included for illustration purposes. The reader is referred to both papers for more details.

The symbol which is used for characterising the behaviour of an AND gate is

$$\text{ANDBEHAV} : dur^4 \rightarrow (wf^3 \rightarrow bool)$$

The meaning of the term

$$((\text{ANDBEHAV}\ d_1\ d_2\ d_3\ d_4\ )\ w_{in1}\ w_{in2}\ w_{out}\ )$$

is (approximately): "The three waveforms w_{in1}, w_{in2} and w_{out} exhibit the behavioural characteristics of an *and* gate, with propagation and hold times (for low inputs) of d_1 and d_2 respectively, and propagation and hold times (for high inputs) of d_3 and d_4 respectively." Figure 8.10 illustrates the predicate. The predicate is defined as:

$$\begin{array}{l}
((\text{ANDBEHAV}\ d_1\ d_2\ d_3\ d_4\ )\ w_{in1}\ w_{in2}\ w_{out} \\
\equiv \\
\forall\ \text{i} : \text{intvl}. \\
\quad\quad ((\text{G}\ d_1\ d_2\ )\ \text{i}) \\
\quad \wedge \\
\quad\quad ((\text{const i}\ c_f\ )\ w_{in1}\ ) \vee ((\text{const i}\ c_f\ )\ w_{in2}\ ) \\
\quad \rightarrow \\
\quad\quad ((\text{const}\ ((\text{Prop}\ d_1\ d_2\ )\ \text{i})\ c_f\ )\ w_{out}\ ) \\
\wedge \\
\quad\quad ((\text{G}\ d_3\ d_4\ )\ \text{i}) \\
\quad \wedge \\
\quad\quad ((\text{const i}\ c_t\ )\ w_{in1}\ ) \wedge ((\text{const i}\ c_t\ )\ w_{in2}\ ) \\
\quad \rightarrow \\
\quad\quad ((\text{const}\ ((\text{Prop}\ d_3\ d_4\ )\ \text{i})\ c_f\ )\ w_{out}\ )
\end{array}$$

Three predicates *const*, *Prop* and G are used in this definition. Each will be informally defined.

$$\text{const} : (\text{intvl} \times \text{C}) \rightarrow (\text{wf} \rightarrow \text{bool})$$

The term

$$((\text{const i c})\ \text{w}$$

means that throughout an interval of time i, the analogue waveform w assumes the digital value c.

$$\text{Prop} : (dur^2 \rightarrow (\text{intvl} \rightarrow \text{intvl})$$

The interval

$$((\text{Prop}\ d_1\ d_2\ )\ i_1)$$

is defined to be the interval which starts d_1 later than i_1 starts and finishes d_2 later than i_1 finishes.

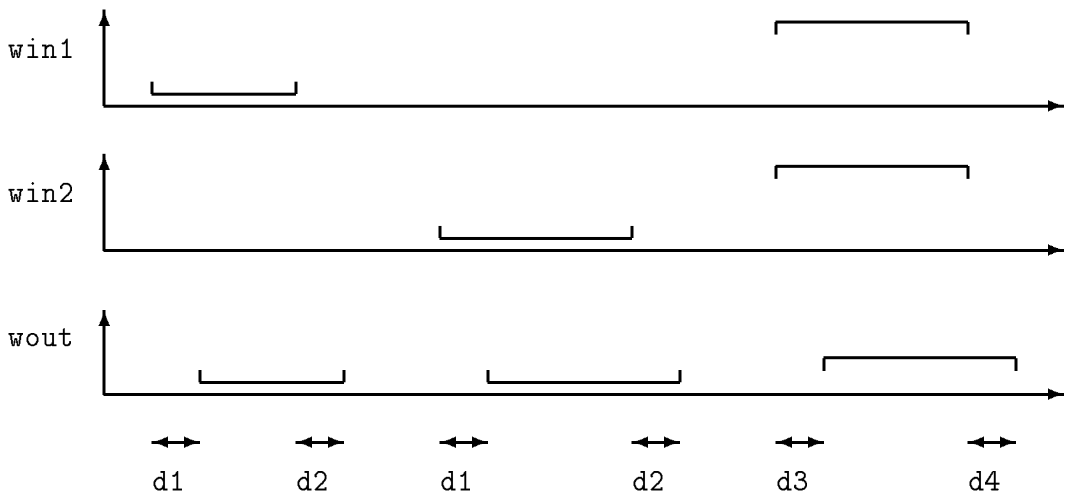


Figure 8.10: Behaviour of AND gate

G : ($dur^2 \rightarrow$ (intvl $\rightarrow$ bool)

The assertion

(($G\ d_1\ d_2$) i_1)

is defined to be true if the duration of d_2 and the duration of i_1 is not less than d_1 .

The behavioral predicate of the *and* gate reads (approximately) : "The predicate ANDBEHAV applied to a 4-tuple of durations d_1, d_2, d_3, d_4 (representing the timing parameters of the gate), in turn applied to a 3-tuple of waveforms w_{in1},w_{in2}, w_{out} (representing the waveforms at the ports of the gate) is defined as meaning that for any interval of time i, the two following assertions hold

- if the interval i is not shorter than $d_1 - d_2$, and input waveform w_{in1} is in state c_f throughout i, or input waveform w_{in2} is in state c_f throughout i, then output waveform w_{out} will be in state c_f throughout the interval starting d_1 after i starts, and finishing d_2 after i finishes, and

- if the interval i is not shorter than $d_3 - d_4$, and input waveform w_{in1} is in state c_t throughout i, and input waveform w_{in2} is in state c_t throughout i, then output waveform w_{out} will be in state c_t throughout the interval starting d_3 after i starts, and finishing d_4 after i finishes."

8.5.11 Translation to FDTs

Formal Description Techniques (FDTs), such as LOTOS [Lot88], Estelle [Est90] and SDL [Sdl88] become to find acceptance in the telecommunication world for the specification and description of telecommunication systems.

These systems are usually built of many communicating concurrent processes. Descriptions in FDTs can not only be used for simulation purposes, but many research in this area is also concentrating on verification issues. For example, it should be possible to verify liveness and free-from-deadlock properties or to verify the behavioural equivalence of two systems.

We have investigated the applicability of LOTOS and SDL for the description, simulation and verification of digital hardware descriptions. A major difference between the two languages is that the communication between processes is synchronous in LOTOS but asynchronous in SDL. Descriptions in FDTs typically fall apart in two parts. One part describes the structure of the system in terms of the interaction of the different processes. The second part describes the behaviour of each process. The first part corresponds to the structural hardware description as present in Glass (i.e. the description of a system as a connection of subsystems), the second part corresponds to the semantic definitions of atoms.

8.5.11.1 LOTOS

The behaviour of an AND-gate in LOTOS may look like the following:

```
process and' [a,b,c] (z1,z2:Bool) : noexit :=
     a?x1:Bool ;
       (    [z2 = 1] ->
               (    [z1 and z2 /= x1] -> c!x1 ; and' [a,b,c] (x1,z2)
                [] [z1 and z2 = x1]  ->        and' [a,b,c] (x1,z2)
               )
        [] [z2 = 0] ->         and' [a,b,c] (x1,z2)
       )
  [] b?x2:Bool ;
       (    [z1 = 1] ->
               (    [z1 and z2 /= x2] -> c!x2 ; and' [a,b,c] (z1,x2)
                [] [z1 and z2 = x2]  ->        and' [a,b,c] (z1,x2)
               )
        [] [z1 = 0] ->                         and' [a,b,c] (z1,x2)
       )
endproc
```

The AND has three *gates* (*a,b,c*) and accepts a boolean value on its first two gates (a?x1:Bool, b?x2:Bool) which are inputs. Gate *c* is the output node (c!x1 or c!x2). The variables *z1* and *z2* are used to save the state on gate *a* and *b* respectively. If e.g. a value is accepted on gate *a*, this value will be stored in variable *x1*. If the other input gate *b* has a high value (stored in *z2*), then the value of *x1* will be transmitted on gate *c* and the process recursively repeats itself. The semi-colon is the sequencing operator.

The description does not state at what time the output *c* will take a new value after an input occurrence at gates *a* or *b*. We only know that this output will change at some time in the future. The description also tells that after an input event, the next event will surely be an output event (if an output transition has to take place).

LOTOS does not allow to describe the relative time between events. This makes the language unsuitable for simulating hardware in the nominal delay mode. With time-stamped inputs as described above with Miranda , it should be possible to specify more accurately the behaviour of logic gates. The length of the LOTOS description will comparable with that of the compostional Miranda approach in section 8.5.7.3. Recently, a timed calculus has been proposed for LOTOS [Que89]. This calculus should provide the basis for specifying and describing precisely timed systems.

The relationship between components of a system have to be stated in terms of *parallel composition* operators. The circuit "or [and [a,b], and [c,d]]" could look in LOTOS as

```
(and [a,b,and1out] ||| and [c,d,and2out])
         |[and1out,and2out]|
     or [and1out, and2out, orout]
```

which says that the execution of the AND gates is purely interleaved, while both of these gates have to synchronize with the OR gate at ports *and1out* and *and2out*.

8.5.11.2 SDL

In SDL, the behaviour of the AND gate could look as

```
PROCESS _and1 (0,1)    ;
  FPAR
    p1, p2 PID;
  DCL
    z1, z2  BOOLEAN := FALSE ,
  START ;
    NEXTSTATE _and;
  STATE _and;
    INPUT _0  ;
      DECISION (z1 and z2 = TRUE);
        (TRUE):
          OUTPUT _0;
        (FALSE):
      ENDDECISION ;
      DECISION SENDER = p1;
        (FALSE):
          TASK z2 := FALSE;
        (TRUE):
          TASK z1 := FALSE;
      ENDDECISION ;
```

```
      NEXTSTATE -;

    INPUT _1 ;
      DECISION SENDER = p1;
        (FALSE):
           DECISION z1;
             (TRUE):
               OUTPUT _1;
             (FALSE):
           ENDDECISION ;
          TASK v2 := TRUE;
        (TRUE):
           DECISION z2;
             (TRUE):
               OUTPUT _1;
             (FALSE):
           ENDDECISION ;
          TASK v1 := TRUE;
      ENDDECISION ;
      NEXTSTATE -;
  ENDSTATE ;
ENDPROCESS _and1 ;
```

Two signals (_0 and _1) can be received by the logic gate. One way to distinguish between the input ports is to check the process identity of the sender. The "aoa" circuit above has been described in SDL in order to simulate its behaviour using the inertial delay model. Unfortunately, due to a bug in our current SDL code generation tool, this has not been possible at present.

8.6 Switch level

8.6.1 Introduction

The use of the Boolean gate model to describe the behaviour of integrated circuits using MOS transistors has been widely recognized to be quite inadequate [Rao85], [Bry81], [Byr1], [Ter83], [Haj1], [Dum83]. In the Boolean gate model, a circuit consists of a set of logic gates connected by unidirectional memoryless wires. The logic gates compute Boolean functions of their input signals and transmit these values along the wires to the inputs of other gates. Each gate input has a unique signal source. Information is only stored in the feedback paths of sequential circuits.

Some MOS pass transistor networks, however, can implement combinational logic in ways that resemble relay contact networks more closely than conventional logic gate networks [Bry87], [Tsa83]. Dynamic memories using MOS devices can store information without feedback paths by exploiting the capacitance of the wires (interconnection region) and the gates of the transistors attached to them. A variety of bus structures can provide multidirectional, multipoint communication. Thus, MOS circuits consist of bidirectional switching elements connected by bidirectional wires with memory due to the interconnect and device capacitances and, hence, cannot be modeled accurately by Boolean gate-level simulators [Rao87].

Recently, models have been developed for describing the behaviour of circuits at the MOS transistor level. At this level, a circuit is modeled as a network of transistor switches connecting a set of charge storage nodes. Node voltages are represented by discrete logic levels, and electrical behaviour is modeled in a highly simplified way. Switch-level algorithms have been applied to such tasks as logic and fault simulation, formal hardware verification, timing analysis, and automatic test program generation [Bry87].

A new class of digital simulators has recently emerged specifically for the digital simulation of MOS VLSI circuits. These switch-level simulators [Rao85], [Bry80],[Bry84], [Ter83], [Haj83] model a MOS circuit as a set of nodes of transistor switches. These simulators can handle a variety of MOS configurations such as logic gates, pass transistors, busses, and static and dynamic memory. The dynamics of the circuit is usually modeled by suitably delaying transistors between discrete logic levels. Some simulators, such as MOSSYM [Bry80], EXPRESS [Haj83], and several others [Bry84], [Byr1], operate in the so-called unit-delay mode, where all the logic gates or functional units are assumed to have the same delay. Thus, unit-delay simulators can verify only the static-state behaviour or the logic functionality of the digital circuit. Some other switch-level simulators [Dum83], [Nha80] allow assignable multiple delays to provide some timing information. Switch level simulators such as RSIM [Ter83] and MOSTIM [Rao85] could be used on large MOS digital circuits to provide accurate timing information during the simulation itself. RSIM models a MOS transistor as a linear resistor and computes an equivalent RC time constant to estimate the transitional delays. MOSTIM uses a more accurate approach involving the use of a delay operator [Rao85], [Rao83] to delay transitions through a functional unit in the MOS network [Rao87].

8.6.2 Overview

We will start with some simple switch-level models which can be used to investigate the behaviour of CMOS circuits without pass transistors. This will be followed by a well-known model from R.E.Bryant. A simulator using this model will be described. After the description of timing analysis at the switch-level, we will turn to verification of switch-level circuits.

8.6.3 Simple switch-level models

The simplest model A two valued logic will be used to model the transistors. The formal definitions are :

ntran (g,s,d) $\equiv$ (g $\supset$ (s = d))
ptran (g,s,d) $\equiv$ ($\neg$ g $\subset$ (s = d))

This simple model can be used for circuits where no pull-up or pull-down and pass transistors are present. In [Gor8a] and [Lar88], this model is used to describe and verify a bus driver and a carry propagate adder structure.

A Threshold Switching Model of Transistors The same simple model is given in [Mel88]. In this paper, however, a second model is given which at least partly captures threshold switching behaviour. It is based on an abstract data type that has exactly three distinct values: Hi, Lo, and X. Such an abstract data type can be represented in higher order logic by defining a logic type *tri* specified by:

tri := Hi | Lo | X

Using *tri*, the behaviour of an n-type transistor can be specified by the following definition:

Ntran(g,s,d) $\equiv$ (g=Hi $\supset$ ((s=Lo) = (d=Lo))

It follows from this definition that when the gate g has value Hi and the source s has value Lo (i.e. the gate-to-source voltage is large) then the drain d must also have value Lo. Thus the logic level Lo is transmitted unchanged through the transistor. But when both g and s have the value Hi then the value at d may be either Hi or X; the predicate Ntran is satisfied in both cases. This reflects the fact that a logic level Hi can be degraded to an "error" value X when it is transmitted through the transistor.

The behaviour of a p-type transistor can be specified using *tri* as follows:

Ptran(g,s,d) $\equiv$ (g=Lo $\supset$ ((s=Hi) = (d=Hi))

Again, the value X is used to model the possibility of a logic level being degraded as it is transmitted through a transistor. In this case, when g and s are Lo the value of d can be either Lo or X, reflecting the fact that the logic level is only imperfectly transmitted through a p-type transistor [Mel88].

A switch level model based on four value logic In [Joy88], yet another model is presented. The behavioral primitives are based on a four value logic with values for high impedance Zz and error Er in addition to Lo and high Hi. The values are ordered to form a lattice with a top Er and a bottom Zz. The least upper bound function based on this ordering is given by

$\vdash$ x $\sqcup$ y = ((x = y) $\Rightarrow$ x | (z = Zz) $\Rightarrow$ y | (y = Zz) $\Rightarrow$ x | Er)

The definitions of Ptran, Ntran and Pullup are:

Ntran (g,i,o) = (o = (g=Hi) $\Rightarrow$ i | ((g=Lo) $\Rightarrow$ Zz | Er))
Ptran (g,i,o) = (o = (g=Lo) $\Rightarrow$ i | ((g=Hi) $\Rightarrow$ Zz | Er))
PullUp (i,o) = (o = ((i=Zz) $\Rightarrow$ Hi | i))

8.6.3.1 Compilation to Prolog

The c-mod subcomponent of the adder in [Lar88] will be used for illustration purposes. The Glass description is given by:

```
BASETYPE B

TYPE tran = [B&B&B]

ATOM
     ntran :- [B&B&B],
     ptran :- [B&B&B]

DEF cmods :- [B&B&B&B];
    cmods [Cin,A,B,Ncout] =
       { ntran [A,P1,b0],
         ntran [B,P1,b0],
         ntran [Cin,Ncout,P1],
         ntran [A,Ncout,P2],
         ntran [B,P2,b0],
         ptran [A,b1,P3],
         ptran [B,b1,P3],
         ptran [Cin,P3,Ncout],
         ptran [A,P4,Ncout],
         ptran [B,b1,P4]
       }
```

In this description, b1 and b0 stand for Boolean logic values and the Prolog convention for identifiers (constants starting with lower case, variables with upper case) has been used.

Because of the bidirectional nature of transistors, a relational language like Prolog can be used as target language for the semantic function. The meaning of the atoms *ntran* and *ptran* in the simple switch model is given by:

ntran(b1,A,A).
ntran(b0,A,B).

ptran(b0,A,A).
ptran(b1,A,B).

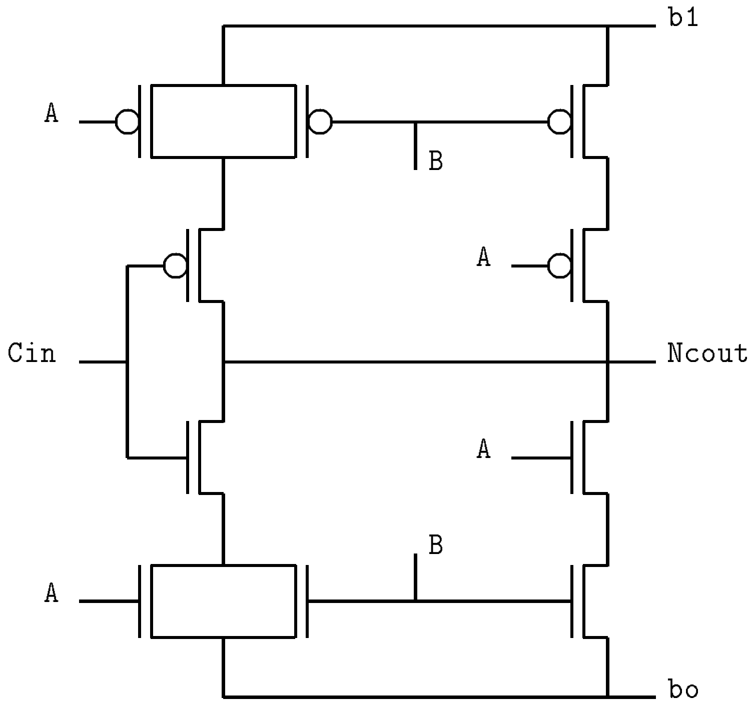


Figure 8.11: The c-mod module

The same semantic function as for the simplex behavioural model (Prolog version) can be used for this purpose. A Prolog program is generated from the Glass description to simulate the behaviour.

The produced Prolog code is:

```
cmod(Cin,A,B,Ncout) :-
  ntran(A,P1,b0),
  ntran(B,P1,b0),
  ntran(Cin,Ncout,P1),
  ntran(A,Ncout,P2),
  ntran(B,P2,b0),
  ptran(A,b1,P3),
  ptran(B,b1,P3),
  ptran(Cin,P3,Ncout),
  ptran(A,P4,Ncout),
  ptran(B,b1,P4).
```

8.6.3.2 Compilation to logic

Compilation to the language used in [Lar88] has not been implemented but should be very simple, as is illustrated by the following description which has been taken from this paper:

```
(=(cmod Cin A B Ncout)
  (there-exist ((boolean P1 P2 P3 P4))
    (and (ntran A P1 0)
         (ntran B P1 0)
         (ntran Cin Ncout P1)
         (ntran A Ncout P2)
         (ntran B P2 0)
         (ptran A 1 P3)
         (ptran B 1 P3)
         (ptran Cin P3 Ncout)
         (ptran A P4 Ncout)
         (ptran B 1 P4))))
```

8.6.3.3 Compilation to temporal logic

In [Mos85], temporal logic is used to describe and reason about hardware. The definitions of the n-type and p-type transistors is given by:

$$\text{ntran(S,A,B)} \equiv \Box([S = 1] \supset [A = B])$$
$$\text{ptran(S,A,B)} \equiv \Box([S = 0] \supset [A = B])$$

Here $\Box$ stands for "always". A CMOS inverter is then described by the predicate:

$$\text{cmos\_inverter(X,Y)} \equiv [\text{ntran(X,Y,0)} \wedge \text{ptran(X,Y,1)}]$$

The correctness of the inverter is given by the following property:

$$\models \text{cmos\_inverter(X,Y)} \supset (Y \approx \ominus X)$$

where $\ominus$ stands for the logic negation.

8.6.4 The Bryant switch-level model

8.6.4.1 Description

The following brief review of the switch-level model [Bry84] is taken from [Ree87]. According to this model, a MOS circuit is composed of wires (*nodes*) and transistors (*switches*).

Nodes have a *type*, a *size*, and a *value*. A node type is either *input* (driven by an external voltage source), or *storage* (i.e, non-input). A storage node's size is proportional to its capacitance, and is approximated by a small integer in the range $\kappa_1..\kappa_{max}$. An input node's size is represented by an integer ω, which is greater than any capacitive node size. The set of values for nodes is {0,1,X}, where 0 and 1 are digital values representing low and high voltage, respectively, and X represents an unknown or error (nondigital) voltage.

Each transistor has a *type* (depletion, n-type enhancement, p-type enhancement) and a *strength*. The strength of a transistor is inversely proportional to its resistance, and is approximated by a small integer in the range $\gamma_1..\gamma_{max}$. Transistors are bidirectional, and either turned off, potentially conducting, or definitely conducting (depending on the transistor type and gate node value). Transistors also have three terminals associated with them, which are the *gate*, *source*, and *drain* nodes.

The transistor strengths and node sizes are arranged in the sequence

$$\lambda < \kappa_1 < .. < \kappa_{max} < \gamma_1 < .. < \gamma_{max} < \omega$$

This allows the development of simple algorithms for switch-level simulation.

The purpose of switch-level simulation is to determine the behaviour of the circuit, termed the *steady-state response*, as a function of the circuit state and the values of the inputs. For this purpose, the circuit may be represented by a *switch graph*, in which the vertices V represent the nodes of the circuit, and the edges E represent the conducting transistors. A path in the switch graph is directed from a *root* node to a *destination* node. The *strength of a path* is the minimum of the root node size and the weakest edge (i.e., transistor strength) along the path. A *definite* path contains only edges representing definitely conduction transistors, while a *potential* path contains edges representing definitely and/or potentially conducting transistors. A definite path will *block* a potential path of lesser strength if the destination node of the definite path is contained in the potential path; a potential path which is not blocked is called a *contributing* path.

Let M be the set of roots of contributing paths to a node n. Then the steady-state response at node n is defined to be the least upper bound of the values of the nodes in M, where the least upper bound is defined on a semi-lattice with structure $0 < \mathrm{X}$ and $1 < \mathrm{X}$.

The basic unit of computation in switch-level simulation is the *partition* (also called a component), which is a group of nodes that are connected by conducting transistors. Input nodes may belong to multiple partitions, and isolate them from one another. Partitions are ordered for solution of their steady-state behavior based on the structure of the circuit. If the circuit has feedback, the *unit delay* assumption controls the sequence in which partitions are considered for solution.

While the switch-level model has important limitations, it is accurate enough to predict the logical behavior of digital MOS circuits fairly well, and is much faster than circuit-level simulation.

For more details on this model, the reader is referred to [Bry84]. The following is the algebraic formulation of the solution

$$\begin{aligned}
&\mathrm{q} = \mathrm{s} + \mathrm{G1} * \mathrm{q} \\
&\mathrm{u} = [\mathrm{up}\ (\mathrm{s,y}) + (\mathrm{G1} + \mathrm{GX}) * \mathrm{u}] \sim \mathrm{q} \\
&\mathrm{d} = [\mathrm{down}\ (\mathrm{s,y}) + (\mathrm{G1} + \mathrm{GX}) * \mathrm{d}] \sim \mathrm{q}
\end{aligned}$$

where

+ takes the maximum of its arguments
- takes the minimum of its arguments

$\sim$ is defined as
a $\sim$ b = a, a $\geq$ b
= λ, a $<$ b
these three operators have been extended to vectors and matrices

y is the state vector
y i is the state of node n i in the start state
s is a vector where s i is the size of node i
G1 is a matrix where G1 i j is the strength
of the strongest 1-edge between node i and node j
GX is a matrix where G1 i j is the strength
of the strongest X-edge between node i and node j
up (s,y) is a vector where for each node n i :
up (s,y) = s, y = 1 or X
= λ, y = 0
down (s,y) is a vector where for each node n i:
down (s,y) = s, y = 0 or X
= λ, y = 0

The matrices *G1* and *GX* depend on the inputs for the circuit, but in the general case also on the present state *y*, because internal nodes can be connected to the gates of transistors in the circuit. The new steady state *y'* response is given by:

y' i = 1, d i = λ
= 0, u i = λ
= X, otherwise

The computation of the new steady state *y'*, given the old state *y*, uses a number of iterations. First, *G1* and *GX* are calculated from the state *y* and the inputs of the circuits. The vectors *up* and *down* are calculated from *y* and *s*. The vector *q* is computed and the result is used for the calculation of *u* and *d*. These vectors are then used to calculate the first estimate for *y'*. This value is used in the next iteration. The iterative loop is left when the new calculated state is the same as the old one.

The matrices *G1* and *GX* are sparse, In [Bry84], an algorithm is given for solving fixed-point equations for sparse systems. This solution has been implemented in the MOSSYM II simulator. In [Bar88], another implementation (called SLS) based on the Bryant switch level model is presented.

8.6.4.2 A switch-level simulator in Miranda

A switch-level simulator based on the algebraic formulation in [Bry84] has been implemented in Miranda . The same example as in [Bry84] will be used to illustrate the results. The circuit is an NMOS inverter followed by a pass transistor and has the following Glass description. The schematics can be found in figure 8.12.

```
BASETYPE B ;

ATOM ntran :- STRING -> [B&B&B],
     dtran :- STRING -> [B&B&B],
     node  :- STRING -> [B] ;

DEF inverter :- [B&B&B] ;
    inverter [in,pass,b]
      = {ntran "Y1" [a,vdd,a],
         ntran "Y2" [in,a,vss],
         ntran "Y2" [pass,a,b],
         node "K1" a,
         node "K1" b,
         node "W" vdd,
         node "W" vss
        }
```

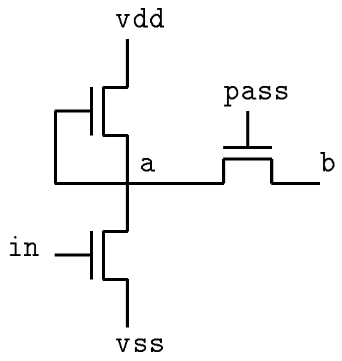


Figure 8.12: NMOS inverter with pass transistor

The inverter has a 1 on its input and the pass transistor an X on its gate. All storage nodes have size κ_1. The size of the depletion load transistor (between *vdd* and a) is γ_1, the size of the other two transistors is γ_2. Two initial state vectors *y1* and *y2* are considered. In the first case, the initial state of node a is 1 and of node b is 0. In the second case, node b has an 1 or X as initial state. The results are:

```
y1 = [("vdd",S1),("vss",S0),("a",S1),("b",S0)]
y2 = [("vdd",S1),("vss",S0),("a",S1),("b",SX)]
```

```
yin = [("in",S1),("pass",SX)]

Miranda mossim "inverter" y1 yin
[("a",S0),("b",S0),("vdd",S1),("vss",S0)]
Miranda mossim "inverter" y2 yin
[("a",S0),("b",SX),("vdd",S1),("vss",S0)]
```

Here, *y1* and *y2* are the initial states of the storage nodes, while *yin* is the input vector.

Although the state vectors and input vectors have the same syntax, the user has to give them separately. This opens the possibility for a simulation with streams of input vectors in which the new calculated state is used together with the next vector from this stream. This involves only a small extension of the implemented semantic function.

8.6.5 Timing analysis

This section follows the lines of [Jou87]. Several representations for the delay through a transistor block have been proposed. These different representations can be summarized as a scalar (i.e., single RC time constant), an interval (i.e., waveform bounds), or a distribution.

In the simplest model, a single RC time constant is calculated for each node. Without pass transistors and resistive interconnects (e.g., as in most gate arrays), a single resistor and capacitor are sufficient. A driver (pull-up or pull-down structure) can be modeled as a resistor and the node capacitance is easily obtained from circuit extractors. For circuits with resistors or pass transistors, more complicated models are needed. In TV [Jou83], the Penfield-Rubenstein model [Pen81] is used. This approach basically multiplies each capacitance by the resistance it sees coming from the driver. Two simplifications are used in practice. First, only the waveform estimate and not the bounds are computed. Second, only one path is assumed open through a tree of pass transistors. Both these simplications have worked very well in practice. A good discussion of different methods for modeling delay as a single RC time constant, including methods taking into account difference in slope input, has been given by Ousterhout [Ous84].

The reader is referred to [Jou87] for more details and discussion of the other models.

8.6.6 The Bryant model for symbolic analysis

Bryant [Bry85] has proposed a method of symbolic simulation which he calls MOSSYM. The major points of this method are outlined below. The goal of symbolic simulation is to compute the steady-state solution for all storage nodes in the circuit in terms of systems of Boolean equations.

Let the value of a node be a Boolean function of symbolic variables, and encode this value into two distinct parts, termed *state0* and *state1*. This encoding allows to convert equations using ternary variables to systems of equations using binary (Boolean) variables. *state0* represents all the conditions under which a node is connected to a source of low voltage, while *state1* represents the conditions under which a node is connected to a node of high voltage. The digital node values 0, 1, and X may be encoded as follows:

digital value	state0	state1
0	1	0
1	0	1
X	1	1

Let the initial value of a node n be denoted by *state0(n)* and *state1(n)*. The solution for a node n is denoted by *newstate0(n)* and *newstate1(n)*. Let *pot(t)* and *def(t)* be the conditions under which a transistor t with gate terminal n is potentially (in state 1 or X) or definitely (in state 1) conducting depending on the state of its gate, respectively, as defined below:

transistor type	pot(t)	def(t)
n	state1(n)	¬ state0(n)
p	state0(n)	¬ state1(n)
depletion	1	1

The formulation of the steady-state response in terms of systems of Boolean equations (using the concepts defined above) can be found in [Bry87]. A short description of the MOSSYM algorithm can be found in [Ree87]. An introduction to the solution of systems of Boolean equations can be found in [Bry87].

8.6.7 Simplified models for symbolic analysis and verification

8.6.7.1 Simplifications

The equations in [Bry87] are very long. Bryant's model is applicable to many design styles and allows nodes with different sizes and transistors with different strengths.

Several simpler (and thus easier to implement) models have been presented in the literature. These restrictive models require the use of stringent design rules. Following simplications are sometimes possible:

- In the simplest model (valid for combinational static CMOS circuits), we can abstract from node capacitances and transistor strengths.

- In a little more complicated model, we can assume that there is only one storage node in each path from the output node to the voltage source or ground

- Next, we can assume that there is more than one key node in each channel-connected subnetwork.

These models are explained in [Brz89]. In these models, all storage nodes are assumed to have the same size. Furthermore, no account is taken of transistors with different strengths. On the other hand, the models take account of the fact that a P-transistor conducts a 1-signal well, but a 0-signal rather poorly. Similarly, an N-transistor conducts a 0 well, and a 1 poorly. For this reason, a distinction is made between P- and N-paths.

8.6.7.2 A semantic function for calculating path expressions

The simplest model has been used for the implementation of a semantic function, which calculates path expressions. The *path expression* indicates the logic conditions for which there is a conducting path between two nodes *source* and *target* [Bol89]. The approach will be illustrated by means of the *bridge* circuit in figure 8.13 and the *c-mod* element previously described. The Glass description of the *bridge* circuit is given by

```
BASETYPE B ;

ATOM
     ntran :- [B&B&B],
     ptran :- [B&B&B];

DEF bridge :- [B&B&B&B&B & B&B];
    bridge [a,b,c,d,e, s,t] =
      { ntran [a,s,u],
        ntran [b,s,v],
        ptran [c,u,v],
        ntran [d,u,t],
        ntran [e,v,t]
      } ;
```

Two examples of the calculation of path expressions for the *bridge* are

```
Miranda pathex "bridge" "s" "t"
d.a + d.~c.b + e.b + e.~c.a
Miranda pathex "bridge" "u" "v"
b.a + ~c + e.d
```

The function realized by the *c-mod* element can be derived by calculating the path expression between the high voltage value *b1* and the output *Ncout*.

```
Miranda pathex "cmods" "b1" "Ncout"
~Cin.~A + ~Cin.~B + ~A.~B
```

This function should be the negation of the path expression between the low voltage value *b0* and *Ncout*.

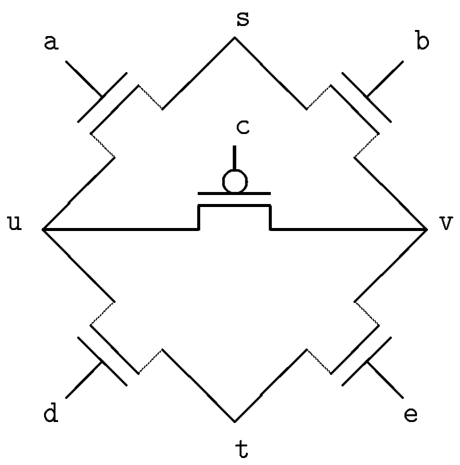


Figure 8.13: bridge circuit

```
Miranda pathex "cmods" "b0" "Ncout"
Cin.A + Cin.B + A.B
```

This expression corresponds to the "majority" function familiar from the adder circuit. The *c-mod* circuit thus realises the negation of this function. For a well design circuit, it should be impossible to have a path between *b1* and *b0*, i.e. the following expression should be equal to False:

```
Miranda pathex "cmods" "b1" "b0"
A.Cin.~Cin.~A + A.Cin.~Cin.~B + A.Cin.~A.~B + B.Cin.~Cin.~A +
B.Cin.~Cin.~B + B.Cin.~A.~B + B.A.~Cin.~A + B.A.~Cin.~B + B.A.~A.~B
```

The implemented semantic function is also able to handle hierarchical descriptions. These are flattened "on the fly". With the present implementation, the function doesn't return the path expression as a Boolean expression of only the external inputs; internal gate variables still appear in the expression. Of course, it is always possible to ask for their path expression. Further manipulation of the Boolean expressions is also left to more specialized tools.

8.6.8 Verification systems for MOS digital circuits

The model relevant for describing a MOS circuit depends heavily on the design rules used in the implementation of the circuit. Because many design styles for MOS circuits exist (e.g. NMOS, CMOS, NORA-CMOS, C2MOS), systems for verifying the design and for detection of the violation of design rules become very important.

In [Bol89], an expert verification system for digital, synchronous MOS circuits, called DIALOG, is presented. A knowledge base is used to store the rules belonging to a certain design style. A description of the circuit is then checked against these rules.

The rule for recognizing a two-input NAND gate in CMOS in the DIALOG system looks as follows:

```
(DEFRULE two-input-nand()
 (* IF
   ((property pmos m1)
    (relation ionodes m1 vdd)
    (relation ionodes out m1)
    (relation ionodes m2 out)
    (property pmos m2)
    (relation ionodes vdd m2)
    (relation ionodes m3 out)
    (property nmos m3)
    (relation ionodes k m3)
    (relation ionodes m4 k)
    (property nmos m4)
    (relation ionodes gnd m4)
    (relation innodes in1 m1)
    (relation innodes in1 m3)
    (relation innodes in2 m2)
    (relation innodes in2 m4))
  * THEN ((...))))
```

A function to generate this rule from a Glass description such as

```
TYPE tran = [B&B&B];

DEF par :- tran -> tran -> [B&B&B&B];
    par a b [in1,in2,c1,c2] =
      { a [in1,c1,c2],
        b [in2,c1,c2]
      } ;

DEF ser :- tran -> tran -> [B&B&B&B];
    ser a b [in1,in2,c1,c2] =
      { a [in1,c1,c0],
        b [in2,c0,c2]
      } ;

DEF nand :- [B&B&B] ;
    nand [in1,in2,out] =
      { par ptran ptran [in1,in2,vdd,out],
        ser ntran ntran [in1,in2,out,gnd],
        Vdd vdd,
        Gnd gnd
```

```
    } ;
```

should not be difficult to implement. Combinators have been used to clarify the structural relationship between components. The schematics can be found in figure 8.14.

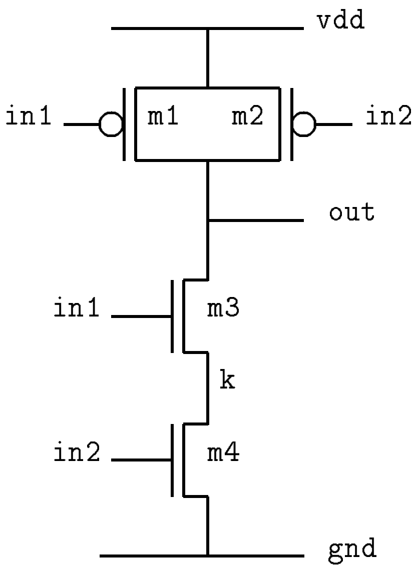


Figure 8.14: CMOS NAND gate

A similar approach is proposed by the VERA program (VERification Assistant) [Kos88]. The system can be used to analyse as well as manipulate electrical circuit designs. It can be used to check the presence of facts, to check the absence of a given set of errors, to modify a network description and to gain statistics on a design. The NAND gate in the VERA language looks like

```
NAND:
   (Terminal-names   (in1     in2     out)
    Terminal-classes (nand-in nand-in nand-out)
    Terminal-types   (input   input   output)
    Network
      ((mos m1   vdd in1 out   vdd ? ? pmos)
       (mos m2   vdd in2 out   vdd ? ? pmos)
       (mos m3   out in1 k     gnd ? ? nmos)
       (mos m4   k   in2 gnd   gnd ? ? nmos))
    Global-nodes (vdd gnd)
    Fan-out 3
   )
```

Again, it should be possible to interface Glass with the VERA tool.

8.6.9 A compositional switch-level model

Bryant's switch level model is not compositional. For this reason, no behavioural definition for transistors has (can) be given. A compositional model has been introduced by Winskel in [Win87]. In this model the semantic definitions for the transistors become very long, however.

$$
\begin{aligned}
&\mathrm{ntran}(\alpha, \beta, \gamma)\\
&\quad = \{\sigma \in Sta[\alpha, \beta, \gamma] \mid\\
&\qquad\qquad I\alpha = 0 \wedge I\beta = 0 \wedge I\gamma = 0 \wedge\\
&\qquad\qquad \gamma \mid\mid \alpha \wedge \gamma \mid\mid \beta \wedge (\alpha \mid\mid \beta \vee \alpha \sim \beta) \wedge\\
&\qquad\qquad (V\gamma = H \rightarrow \alpha \sim \beta) \wedge (V\gamma = L \rightarrow \alpha \mid\mid \beta)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{ptran}(\alpha, \beta, \gamma)\\
&\quad = \{\sigma \in Sta[\alpha, \beta, \gamma] \mid\\
&\qquad\qquad I\alpha = 0 \wedge I\beta = 0 \wedge I\gamma = 0 \wedge\\
&\qquad\qquad \gamma \mid\mid \alpha \wedge \gamma \mid\mid \beta \wedge (\alpha \mid\mid \beta \vee \alpha \sim \beta) \wedge\\
&\qquad\qquad (V\gamma = L \rightarrow \alpha \sim \beta) \wedge (V\gamma = H \rightarrow \alpha \mid\mid \beta)\}
\end{aligned}
$$

Here, Sta is the set of static configurations on the points α, β, γ. A static configuration σ of sort Λ (with Λ the sort of points) is a 4-tuple

$< \mathrm{S}, \mathrm{V}, \mathrm{I}, \succ >$

where

$\mathrm{S} \in \Lambda \rightarrow \mathbf{S}$ (the strength function)
$\mathrm{V} \in \Lambda \rightarrow \mathbf{V}$ (the value function)
$\mathrm{I} \in \Lambda \rightarrow \mathbf{V}$ (the internal value function) and
$\succ$ is a reflexive, transitive relation on Λ (the flow relation)

which satisfies

(1) $\alpha \succ \beta \rightarrow S\alpha \geq S\beta$
(2) $\alpha \succ \beta \wedge S\alpha = S\beta \rightarrow \beta \succ \alpha$
(3) $\alpha \succ \beta \wedge S\beta \in K \rightarrow S\alpha = S\beta$
(4) $S\alpha = 0 <-> V\alpha = 0$
(5) $I\alpha \leq V\alpha$
(6) $\alpha \succ \beta \rightarrow I\alpha \leq I\beta \wedge V\alpha \leq V\beta$

$\alpha \sim \beta \equiv \alpha \succ \beta \wedge \beta \sim \alpha$
$\alpha \mid\mid \beta \equiv \neg(\alpha \succ \beta) \wedge \neg(\beta \succ \alpha)$

For the definition of the composition of static configurations, the reader is referred to [Win87]. The model is mainly used for verification.

8.7 Circuit level

8.7.1 Electrical-logic simulation

Circuit simulators, such as SPICE2 [Nag75] and ASTAP [Wee83], have been successfully used for the design and performance evaluation of integrated circuits, since they provide very accurate output waveforms. However, it is almost prohibitive to use these simulators for the analysis of large integrated circuits, due to long computer run times [Kim89]. Techniques have been used to reduce the required CPU time for large circuits, but circuit simulators are still much slower than logic simulators. One of the major differences between circuit simulation and logic simulation are the models used to describe the elements being analyzed. In circuit simulation, individual transistors are modeled using complex analytical expressions. In logic simulation, the circuits are modeled as logic gates, which are groups of transistors, or by single transistors modeled as ideal switches.

In [Kim89], a new form of circuit modeling and analysis, called Electrical-Logic (ELogic) is presented. In a conventional circuit modeling and simulation approach, a timestep is selected and the voltages (and currents) corresponding to the new time are computed. The key idea of ELogic is to model a technology in terms of a set of discrete, possible nonuniform, states of network variables (such as voltages and currents) and to solve for the time required to make a transition between adjacent states. The implementation uses transistor level circuit models and provides a continuous speed-precision tradeoff between circuit level and logic/switch level.

Elogic is defined as a circuit modeling and analysis technique which solves for the amount of time required for a network variable to make a particular transition as follows.

- A set of discrete states of network variables $\{ S^i \mid i \in 1..N_s \}$ is defined. The network variables can be charges, fluxes, voltages, and currents.

- Network variables are only allowed to make a transition from one state to an adjacent state:

  $$S^i \rightarrow S^j, j = i \pm 1$$

- The first time t_{n+1} at which the next state of the variables (S^{i+1} or S^{i-1}) is achieved is computed, rather than solving for the network variables x at a given time point:

  $$t_{n+1} = \min \{ t \mid x(t) = S^j, j = i \pm 1, t > t_n, \text{ where } x(t_n) = S^i \}$$

- The number and values of the states of the variables can be varied between analyses and between portions of a circuit in the same analysis.

ELogic has been implemented in three forms, Elogic-1, Elogic-2, and Elogic-3. Discrete node-voltages are chosen as the network variables. The input to a circuit consists of a set of state changes at discrete time points. This set of state changes and corresponding times is called a *waveform fragment* and is used as an input to the finite-state-machine model of the device. These small edge transitions are moved through the network in an event-driven, selective-trace manner. If a node voltage is computed to lie between two adjacent states at some steady-state value, its value must be rounded to the nearest ELogic state. Therefore the *precision* of an Elogic waveform is determined by the number and values of the voltage states defined for the simulation. As more states are used, the waveform fragments that represent activity at a node will contain more timepoints and, as a result, solving for the corresponding output fragment will take longer. Hence a direct trade off between precision and efficiency is achieved.

8.7.2 Analog simulation and analysis tools

At the lowest level, digital circuit exhibit an analog behaviour. The reader is referred to the work of C. van Reeuwijk on this subject [Ree89].

8.8 Compilation to other HDLs

8.8.1 Motivation

Glass is not the first hardware description language. In industrial environments, other HDLs have been in use for years and complete tool chains have been centered around these languages. The easiest way to benefit from these toolkits is to provide a translation from Glass to the HDL. This way of working has also the advantage the Glass can be introduced in a smooth way in an existing environment: instead of throwing away existing tools and methods, a new language is just added to the environment. This will inevitably lead to a comparison between the "native" HDL and Glass .

8.8.2 Compilation to NNDF

The compilation of Glass to the Neutral Network Description Format (NNDF) used at Alcatel Bell is independant of any level of abstraction or model. NNDF is a file format definition for neutral interfaces between tools requiring the transfer of the network description data of a (V)LSI chip [Lov83]. NNDF is a netlist format which is normally generated by a schematic capture tool. NNDF has been used to generate the input for several tools. Conversion to this format offers the possibility to integrate Glass in an easy way in the CAD tool chain. The unparser of the semantic function has been written in half a day in Miranda .

8.8.3 Compilation to VHDL

VHDL [Lip86] has become the standard hardware description language. It allows three ways of description: structural (description of a system as an interconnection of subsystems), dataflow and behavioural. The traditional way to obtain a structural description is by using schematic entry and schematic capture tools. Logic synthesis tools automatically synthesize structural descriptions from behavioural ones. Little attention has therefore been spent to the readability of most structural description formalisms, which is also true for VHDL. Because this is not the case for Glass , this language can be considered as a useful alternative for many structural description languages. The translation of Glass to the structural subset of VHDL has not been implemented yet, but should not be a hard exercise, at least not for kernel Glass . Conversion of recursively defined macro's will be a harder task because recursion will have to be replaced by iteration.

8.9 Discussion

The Glass kernel abstract syntax has proved to be an excellent starting point for the implementation of semantic functions. Because of the reusability of the transformation blocks, interpreted functions for which an algebraic solution is available in the literature can be implemented in about three days in Miranda . Unparsers for compiled functions can be implemented in half a day. Some points which deserves more attention for a product quality implementation are now discussed.

The transformation process has been completely implemented in Miranda and is therefore rather slow. However, because none of the transformations explicitly rely on the lazy evaluation mechanism, there should be a straightforward mapping to imperative languages such as C .

An attempt has been made to use compositional approaches and to preserve as much as possible the hierarchical description. In some cases, such as for delay calculation, the compositional formulation posed no problems and was even easier to implement than a non-compositional one. Other attempts, however, were less successful and led to very long semantic definitions. This was especially true for the event-driven simulation and the Winskel switch-level model.

Many semantic functions require a "tailored" description. For example, the switch-level simulator requires information concerning the strength of transistors or the size of nodes. The semantic function assumes that this information can be found at certain places in the description and uses only a limited set of values. The type-checker does not (and should not) check these descriptions, but it would be advisable if for some semantic functions an additional check is performed after type-checking.

Another point which deserves more attention is the user interface. Simulators typically require lots of input and produce lots of output. A graphical interface to provide input or interpret the output is very important. This interface code can even be much larger than the semantic function itself. If possible, as much code as possible should be generated from the Glass description. Tuple types such as B^n can be used to produce I/O functions which accept and return integer values.

The semantic definitions of the atoms are now spread over the directories of the semantic functions. If a new atomic device is introduced, its definitions should typically be included in all directories which use a model applicable to the device. An alternative solution is to group all atom definitions in one file (or perhaps even better a data base) such that the addition or modification of a definition should only be made at one place.

Appendix A

List of participants

Catholic University of Nijmegen (KUN) Faculty of Mathematics and Computer Science Toernooiveld 1 6525 ED Nijmegen tel: 080-612217 fax: 080-553450 email: germaine@cs.kun.nl	Prof. Dr. Ir. R.T. Boute Ir. H. Oolman Drs. M. Seutter Drs. E. Voss
Sagantec Europe B.V. (SAG) Software Division Postbus 2102 5600 CC Eindhoven tel: 040-642342 fax: 040-4155555 email: simon@sagan.nl	Ir. J. Baurdoux Ir. R. Doesborg Ir. S. J. Klaver J.P. Cramer
Delft University of Technology (TUD) Department of Electronics Mekelweg 4 2628 CD Delft tel: 015-781099 fax: 015-785922 email: reeuwijk@dutentb.nl	Prof. Dr. Ir. Davidse Dr. Ir. E.H. Nordholt Ir. C. van Reeuwijk Shr. v. Maaren
Bell Telephone (ABT) Research Center Francis Wellesplein 1 B-2018 Antwerpen, Belgium tel: 09-32-3-2404011 fax: 09-32-3-2379880 email: btmcrc@kulcs	Ir. W. de Kinder Ir. J. de Man Ir. J. Vanslembrouck

Appendix B

Implementation language equivalents

B.1 Equivalents for section 3.2

```
Basetype E;

Atom
   not :- E => E;

Def
   same :- E => E;
   same x = not (not x);

Atom
   and :- E & E => E,
   nand :- E & E => E,
   or :- E & E => E,
   xor :- E & E => E;

Def
   select :- E & E & E => E;
   select [sel, a, b] =
      or [and [not s, a], and [s, b]];

Def
   halfadder :- E & E => E & E;
   halfadder [x,y] = [xor [x,y], and [x,y]];

Def
   ornot :- E & E => E;
   ornot [a,b] = not (or [a,b]);
```

```
Def
   ornot :- E & E => E;
   ornot = %[a,b].not (or [a,b]);

Def
   SetOnce :- E => E;
   SetOnce i = o where o = or [i,o] endwhere

Def
   RSFF :- E^2 => E^2;
   RSFF [R', S'] = [q, q']
      where
         q = nand [S',q'];
         q' = nand [R',q]
      endwhere;

Def
   my_xor :- E & E => E;
   my_xor [x,y] = nand [nand [b,x], nand [b,y]]
      where b = nand [x,y] endwhere;
```

B.2 Equivalents for section 3.4

```
Basetype E;

Atom
   R :- [E & E],
   C :- [E & E],
   NPN :- [E & E & E];

Def
   RCnet :- [E & E & E];
   RCnet [a,b,c] = { R [a,c], C [b,c] };

Def
   RCnet2 :- [E & E & E];
   RCnet [a,d,x] = { R [a,x], C [y,d], *[x,y] };

Atom
   Plus12V :- [E],
   Gnd :- [E];
```

```
Def
   VoltageFollower :- [E & E];
   VoltageFollower [in, out] =
      { NPN [out, in, plus],
        R [out, gnd],
        Plus12V plus,
        Gnd gnd };

Atom
   Monoflop :- [?E & E & E & !E];

Def
   delay :- E => E;
   delay [in,out] = { Monoflop [in, x, y, out], C [x,y] };

Atom
   Nenh :- [E & E & E],
   Penh :- [E & E & E];

Atom
   Vdd :- [E],
   Gnd :- [E];

Def
   CMosInvertor :- E => E;
   CMosInvertor [in, out] =
      { Penh [in, plus, out],
        Nenh [, gnd, out],
        Vdd plus,
        Gnd gnd };
```

B.3 Equivalents for section 3.5

```
Basetype E;

Atom
   divide_by_two :- E => E;

Mac
   triple :- E => E -> E => E;
   triple A in = A (A (A in));
```

```
Def
   divide_by_8 :- E => E;
   divide_by_8 in = triple divide_by_two in;

Mac
   replicate :- Int -> E => E -> E => E;
   replicate n A in = n=0 -> in;
                      A (replicate (n-1) A in);

Def
   divide_by_1024 :- E => E;
   divide_by_1024 in = replicate 10 divide_by_two in;
```

```
Atom
   adc :- E & E & E => E & E;

Mac
   nbitsadder :- Int -> E^n & E^n & E => E & E^n;
   nbitsadder 0 [[],[],c] = [c,[]];
   nbitsadder n [a:as, b:bs, cin] = [cout, s:ss]
      where
         [v,s] = adc [a,b,cin];
         [cout,ss] = nbitsadder (n-1) [as,bs,v]
      endwhere;

Def
   Fourbitsadder :- E^4 & E^4 & E => E & E^4;
   Fourbitsadder [as,bs,cin] = nbitsadder 4 [as,bs,cin];

Atom
   and5 :- E^5 => E,
   and4 :- E^4 => E,
   and3 :- E^3 => E,
   and2 :- E & E => E,
   nand :- E & E => E,
   nand3 :- E & E & E => E,
   ornot :- E & E => E,
   or :- E & E => E,
   jkff :- E & E => E,
   buf :- E => E;
```

```
Def
   counter :- E^8 => E^5;
   counter [d3,d2,d1,d0,nload,nclear,enap,enat] = [carry,q3,q2,q1,q0]
      where
         carry = and5 [q3,q2,q1,q0,enat];
         q3 = counterstage [d3, preset, and4 [q2,q1,q0, en], nclr];
         q2 = counterstage [d2, preset, and3 [q1,q0,en], nclr];
         q1 = counterstage [d1, preset, and2 [q0,en], nclr];
         q0 = counterstage [d0, preset, en, nclr];
         preset = ornot [nload, nclr];
         nclr = buf nclear;
         en = and2 [enap,enat];
         Mac
            counterstage :- E^4 => E;
            counterstage [d, preset, toggle, nclr] =
               jkff [and2 [ja, ea], and2 [ka, ea]]
               where
                  ja = nand [ka, preset];
                  ka = nand3 [d, preset, nclr];
                  ea = or [toggle, preset]
               endwhere;
      endwhere;
```

Bibliography

[Abt1] N.2 documentation at Alcatel Bell.

[Abt2] Verilog documentation at Alcatel Bell.

[Ada83] "Ada Programming Language",
ANSI/MIL-STD-1815A, Jan. 22, 1983.

[Aho83] A.V.Aho, J.E.Hopcroft, J.D.Ullman,
Data Structures and Algorithms,
Addison-Wesley Publication, Reading, Massachusetts, 1983.

[Ayl86] J.H.Aylor, R.Waxman and C.Scarratt,
"VHDL - Feature Description and Analysis",
IEEE Design & Test of Computers,
April 1986, pp.17-27.

[Bac78] J.Backus,
"Can Programming Be Liberated from the von Neumann Style?"
Comm. of the ACM, Vol. 21. No.8, Aug. 1978, pp.613-641.

[Bar88] Z.Barzilai, D.K.Beece, L.M.Huisman, V.S.Iyengar, and G.M.Silberman,
"SLS - A Fast Switch-Level Simulator",
IEEE Transactions on Computer-Aided Design,
Vol. 7, No. 8, August 1988, pp. 838-849.

[Bel71] C.G.Bell and A.Newell,
Computer Structures: Readings and Examples,
McGraw-Hill, New York, N.Y., 1971.

[Ben87] J.Benkoski, E.Vanden Meersch, L.Claesen, and H.De Man,
"Efficient Algorithms for Solving the False Path Problem in Timing Verification",
Proceedings IEEE Conference on Computer-Aided Design,
ICCAD-87, Santa Clara CA, November 9-12, 1987, pp. 44-47.

[Bir88] R.S. Bird, P.Wadler,
Introduction to Functional Programming, Prentice-Hall (UK), Hemel Hampstead, 1988.

[Bol89] I.Bolsens, *Electrical Correctness Verification of MOS Digital Circuits using Expert System and Symbolic Analysis Techniques*, Ph.D. Thesis, KULeuven, May 1989.

[Bou86] R.T. Boute, "System semantics and formal circuit description", *IEEE Trans. Circuits and Systems, CAS-33, 12*, December 1986, pp 1219-1231.

[Bou88] R.T. Boute, "Systems Semantics: Principles, Applications, and Implementation." *ACM Transactions on Programmimg Languages and Systems*, Vol. 10, No 1, Jan 1988, pp 118-155

[Bou90] R.T. Boute, "On the formal description of non-computational objects", in: G.David, R. Boute, B. Shriver, eds., *Declarative Systems*, North Holland, 1990

[Bou87] R.T.Boute, "Application of functional formalisms", Esprit report E881/b13/KUN/RTB/8703, Mar. 1987.

[Bro85] M.Browne, E.Clarke, D.Dill, and B.Mishra, "Automatic verification of sequential circuits using temporal logic", *Tech. Rep. CMU, Pittsburgh*, PA-CS-85-100, 1985.

[Bur75] W.H. Burge, *Recursive programming techniques*, Addison-Wesley, Reading, Mass., 1975.

[Bur81] J.R. Burstall et al., "HOPE: an experimental applicative language", *Report CSR-62-90*, University of EdinBurgh, February 1981.

[Bry80] R.E.Bryant, "An algorithm for MOS logic simulation", LAMBDA Magazine, vol.1, no.3, 1980, pp. 46-53.

[Bry84] R.E.Bryant, "A switch-level model and simulator for MOS digital systems", *IEEE Transactions on Computers*, vol C-33, No.2, Feb. 1984, pp. 160-177.

[Bry85] R.E.Bryant,
"Symbolic verification of MOS circuits",
in *Proc. of the 1985 Chapel Hill Conf. on VLSI*
1985, pp. 419-438.

[Bry87] R.E.Bryant,
"A survey of switch-level algorithms",
IEEE Design and Test,
Aug. 1987, pp. 26-40.

[Bry81] R.E.Bryant,
"A switch-level simulation model for integrated logic circuits",
Ph.D. thesis, MIT/LCS/TR-259, MIT, Cambridge, Mar. 1981.

[Bry87] R.E.Bryant,
"Algoritmic Aspects of Symbolic Switch Network Analysis",
IEEE Transactions on Computer-Aided Design,
Vol. CAD-6, No.4, July 1987, pp. 618-633.

[Bry87] R.E.Bryant,
"Boolean Analysis of MOS Circuits",
IEEE Transactions on Computer-Aided Design,
Vol. CAD-6, No.4, July 1987, pp. 634-649.

[Brz89] J.A.Brzozowski and C-J.Seger,
"A Unified Framework for Race Analysis of Asynchronous Networks",
Journal of the Association for Computing Machinery,
Vol.36, No.1, Jan. 1989, pp. 20-45.

[Brz79] J.A.Brzozowski and M.Yoeli,
"On a ternary model of gate networks",
IEEE Transactions on Computers,
Vol.C-28, No.3, March 1979, pp. 178-183.

[Byr1] R.H.Byrd, G.D.Hachtel, M.R.Lightner, and M.H.Heydemann,
"Switch-level simulation: Models, theory and algorithms",
in *Advances in Computer-Aided Engineering Design*,
Greenwich CT.

[Clo87] W.F.Clocksin and C.S.Mellish,
Programming in Prolog,
Springer-Verlag Berlin Heidelberg 1987.

[Clo87] W.F.Clocksin,
"A Prolog Primer",
Byte,
Aug. 1987, pp. 147-158.

[Clo87] W.F.Clocksin,
"Logic Programming and Digital Circuit Analysis",
J. Logic Programming 4, no.1 (March 1987), pp.59-82.

[DeM86] J.De Man,
"The description of sequential digital systems by means of a functional language",
Esprit report E881/d1/BTM/JdM/8611, Nov. 1986.

[Dev88] S.Devadas, H.Tony Ma, and A.R.Newton,
"On the Verification of Sequential Machines at Differing Levels of Abstraction",
IEEE Transactions on Computer-Aided Design,
Vol.7, No.6, June 1988, pp. 713-722.

[Dil85] D.Dill and E.Clarke,
"Automatic verification of asynchronous ciruits using temporal logic",
in *Proc. of the 1985 Chapel Hill Conf. on VLSI*

[Don76] W.E.Donath, and H.Ofek,
"Automatic identification of equivalence points for Boolean logic verification",
IBM Tech. Disclosure Bull.,
Vol.18, Jan. 1976.

[Dum83] D.Dumlogol, H.J.De Man, P.Stevens, and G.G.Schrooten,
"Local relaxation algorithms for event driven simulation of MOS networks including assignable delay modelling",
IEEE Transactions on Computer-Aided Design,
Vol.CAD-2, No.3. July 1983, pp. 193-202.

[Eic65] E.B.Eichelberger,
"Hazard detection in combinational and sequential switching circuits",
IBM J. Res. Devel. 9,
March 1965, pp. 90-99.

[Est90] *Estelle - A formal description technique based on an extended state transition model*,
IS 9074.

[Fuj83] H.Fujiwara and T.Shimono,
"On the Acceleration of Test Generation Algorithms",
IEEE Transactions on Computers,
Dec. 1983, pp. 1137-1144.

[Goe81] P.Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits",
IEEE Transactions on Computers,
Vol.C-30, No.3, March 1981, pp.215-222.

[Gor8a] M.Gordon, P.Loewenstein, and M.Shahaf,
"Formal verification of a cell library",
in *Applied Formal Methods For Correct VLSI Design*,
Proceedings of the IMEC-IFIP International Workshop,
Houthalen, Belgium, 13-16 Nov. 1989, pp. 659-667.

[Haj1] I.N.Hajj and D.G.Saab,
"Logic and fault simulation of MOS circuits based on symbolic expression generation",
IEEE Transactions on Circuits and Systems.

[Haj83] I.N.Hajj and D.G.Saab,
"Symbolic logic simulation of MOS circuits",
in *Proc. IEEE Int. Symp. Circuits and Systems*,
(Newport Beach, CA), May 1983, pp. 246-249.

[Han85] F.K.Hanna and N.Daeche,
"Specification and Verification using Higher-Order Logic",
Proceedings of the 7th International Conference of Computer Hardware Design Languages,
Tokyo, Japan, 1985.

[Han2] F.K.Hanna and N.Daeche,
"Specification and Verification Using Higher-Order Logic: A Case Study".

[Hil78] F.J.Hill and G.R.Peterson,
Digital Systems: Hardware Organization and Design,
John Wiley & Sons, New York, N.Y., 1978.

[Hil81] F.J.Hill et al.,
"Structural Specification with a Procedural Hardware Description Language",
IEEE Trans. Computers,
Vol C-30, No.2, Feb. 1981, pp.157-161.

[Hit82] R.B.Hitchcock, Sr., G.L.Smitch, and D.D.Cheng,
"Timing analysis of computer hardware",
IBM J. Res. Develop.,
Vol.26, No.1, Jan.1982, pp. 100-116.

[Huf64] D.A.Huffman,
"The synthesis of sequential switching circuits",

In *Sequential Machines: Selected Papers*,
E.F.Moore ed. Allison-Wesley, Reading, Mass., 1964, pp. 3-62.

[Ive62] K.Iverson,
A Programming Language,
John Wiley & Sons, New York, N.Y., 1962.

[Jou87] N.P.Jouppi,
"Timing Analysis and Performance Improvement of MOS VLSI Designs",
IEEE Transactions on Computer-Aided Design,
Vol. CAD-6, No. 4, July 1987, pp. 650-665.

[Jou83] N.P.Jouppi,
"TV: An nMOS Timing Analyzer",
Third Caltech Conference of Very Large Scale Integration,
1983, pp. 71-85.

[Joy88] J.Joyce,
"Generic Structures in the Formal Specification and Verification of Digital Circuits",
in *The Fusion of Hardware Design and Verification*,
Proceedings of the IFIP WG 10.2 Working Conference,
Glasgow, 3-6 July 1988, pp. 50-74.

[Kim89] Y.H.Kim, S.H.Hwang, and A.R.Newton,
"Electric-Logic Simulation and Its Applications",
IEEE Transactions on Computer-Aided Design,
Vol.8, No.1, Jan. 1989, pp. 8-22.

[Kir88] T.Kirkland and M.R.Mercer,
"Algorithms for Automatic Test Pattern Generation",
IEEE Design & Test of Computers,
June 1988, pp. 43-55.

[Kir66] T.I.Kirkpatrick, and N.R.Clark,
"PERT as an Aid to Logic Design",
IBM J. Res. Develep.,
Vol.10, No.2, March 1966, pp. 135-141.

[Knu67] D.E. Knuth,
"Semantics of context-free languages",
Mathematical Systems Theory 2,, 2,
1967, pp. 127-145.

[Knu73] D.E. Knuth,
The Art of Computer Programming,
Vol 1: Fundamental Algorithms,
Addison-Wesley, Reading, Mass., 1973.

[Kos88] A.P.Kostelijk and G.G.Schrooten,
"VERA, a Rule Based Verification Assistant for VLSI Circuit Design",
Proc. ESPRIT Technical Week,
Brussels, 1988, pp. 263-268.

[Lan66] P.J. Landin,
"The next 700 programming languages",
Comm. ACM 9, 3
March 1966, pp. 157-166.

[Lar88] T.Larsson,
"Hardware Verification based on Algebraic Manipulation and Partial Evaluation",
in *The Fusion of Hardware Design and Verification,*
Proceedings of the IFIP WG 10.2 Working Conference,
Glasgow, 3-6 July 1988, pp. 228-249.

[Lie83] K.J.Lieberherr and S.E.Knudsen,
"Zeus: A Hardware Description Language for VLSI",
20th Design Automation Conf.,
Miami, Florida, June 1983, pp.17-23.

[Lie84] K.J.Lieberherr,
"Report on the Hardware Description Language Zeus",
GTE Laboratories, Inc., Feb. 1 1984 version.

[Lin80] C.H. Lindsey, S.G. van der Meulen,
Informal introduction to ALGOL 68",
Revised ed., North Holland, Amsterdam, 1980

[Lip86] R.Lipsett, E.Marschner, M.Shahdad,
"VHDL - The Language",
IEEE Design & Test of Computers,
April 1986, pp.28-41.

[Lot88] *LOTOS - A formal description technique based on the temporal ordering of observational behaviour,*
IS 8807.

[Lov83] G.J.Lovitt,
"Formal Definition for Neutral Network Description Format (NNDF)",
STC document, July 83.

[Mai82] L.I.Maissel and D.L.Ostapko,
"Interactive Design Language: A Unified Approach to Hardware Simulation, Synthesis, and Documentation",

19th Design Automation Conf.,
Las Vegas, Nev., June 1982, pp.193-201.

[Mai83] L.I.Maissel and R.L.Phoenix,
"IDL (Interactive Design Language) Features and Philosophy",
IEEE Int'l Conf. Computer Design,
Port Chester, N.Y., Sept. 1983, pp. 667-669.

[Mai84] L.I.Maissel and H.Ofek,
"Hardware Design and Description Languages in IBM",
IBM J. Research and Development,
Vol. 26, No.5, Sept. 1984, pp.557-563.

[Mar85] F.Maruyama and M.Fujita,
"Hardware verification",
IEEE Computer,
Vol.18, No.2, Feb. 1985, pp. 22-32.

[McW80] T.M.McWilliams,
"Verification of Timing Constraints in Large Digital Systems",
Proc. IEEE Int. Symp. on Circ. and Syst.,
Houston, Texas, April 1980, pp. 415-423.

[Mee86] L.G.T. Meertens, "Algorithmics — towards programming as a mathematical activity",
in: J.W. de Bakker et al., eds.,
Mathematics and Computer Science,
CWI Monographs, Vol 1, North Holland, 1986, pp 289-334.

[Mee87] L.G.T. Meertens,
"An Abstracto reader prepared for IFIP WG 2.1",
Note CS-N8702,
Centre for Mathematics and Computer Science, Amsterdam, April 1987.

[Mel88] T.Melham,
"Using Recursive Types to Reason about Hardware in Higher Order Logic",
in *The Fusion of Hardware Design and Verification*,
Proceedings of the IFIP WG 10.2 Working Conference,
Glasgow, 3-6 July 1988, pp.26-49.

[Men79] E.Mendelson,
Introduction to Mathematical Logic (2d. ed.),
Van Nostrand Reinhold, New York 1979.

[Mic87] A.Miczo,
Digital Logic Testing and Simulation,
John Wiley & Sons, Inc, 1987.

[Mor82] J.D.Morison, N.E.Peeling and T.L.Thorp,
"ELLA: A Hardware Description Language",
IEEE Int'l Conf Circuits and Computers,
New York, 1982, pp.604-607.

[Mos85] B. Moszkowski,
"A Temporal Logic for Multilevel Reasoning about Hardware",
Computer,
Vol.18, No.2, Feb 1985, pp.10-19.

[Mul59] D.E.Muller and W.S.Bartky,
"A theory of asynchronous circuits".
In *Proceedings of an International Symposium on the Theory of Switching*,
Annals of the Computation Laboratory of Harvard University,
vol.29, Harvard University Press, Cambridge, Mass., 1959, pp. 204-243.

[New86] A.R.Newton and A.L.Sangiovanni-Vincentelli,
"Computer-Aided Design for VLSI Circuits",
IEEE Computer,
April 1986, pp. 38-60.

[Nag75] W.Nagel,
"SPICE2: A computer program to simulate semiconductor circuits",
Ph.D.Dissertation. UCB/ERL M75/520,
Univ. California, Berkeley, May 1975.

[Nha80] H.N.Nham and A.K.Bose,
"A multiple delay simulator for MOS LSI circuits",
in *Proc. 17th Design Automation Conference*,
June 1980, pp. 610-617.

[Oda86] G.Odawara, M.Tomita, O.Okuzawa, and T.Ohta,
"A logic verifier based on Boolean comparison",
in *Proc. 23rd Design Automation Conference*,
June 1986.

[Ous84] J.K.Ousterhout,
"Switch-level delay models for digital MOS VLSI",
in *Proc. 21th Design Automat. Conf.*,
June 1984, pp.542-548.

[Pas85] W.G.Paseman,
"Data flow concepts speed simulation in CAE systems",
Computer Design, Jan.1985, pp. 131-140.

[Pen81] P.Penfield Jr. and J.Rubenstein,
"Signal delay in RC tree network",
in *Proc. 18th Design Automat. Conf.*,
June 1981, pp.613-617.

[Pre82] G.W.Preston,
"Technical Briefings at IDA Summer Study on a VHSIC Hardware Design and Description Language",
IDA Record Document D-2, Institute for Defense Analyses, Alexandria, Va.,
April 1982.

[Que89] J.Quemada, A.Azcorra, and D.Frutos,
"A Timed Calculus for LOTOS",
in *Proc. FORTE '89*,
5-8 Dec. 1989, Vancouver, pp.245-264.

[Rao85] V.B.Rao and T.N.Trick,
"Switch-level timing simulation of MOS VLSI circuits",
in *Proc. Int. Symp. Circuits and Systems*,
(Kyoto, Japan), June 1985, pp. 229-232.

[Rao85] V.B.Rao,
"Switch-level timing simulation of MOS VLSI circuits",
Ph.D. thesis, Univ. of Illinois at Urbana-Champaign", Jan. 1985.

[Rao83] V.B.Rao, T.N.Trick, and I.N.Hajj,
"A table-driven delay operator approach to timing simulation of MOS VLSI circuits",
in *Proc. IEEE Int. Conf. Computer Design*
(New York), Nov. 1983, pp.445-448.

[Rao87] V.B.Rao and T.N.Trick,
"Network Partitioning and Ordering for MOS VLSI Circuits",
IEEE Transactions on Computer-Aided Design,
Vol. CAD-6, No.1, Jan. 1987, pp. 128-144.

[Ree89] C.van Reeuwijk,
"tm: a code generator for stuctured data interfaces (draft)",
Esprit report E881/b38/TUD/CvR/8905, May 1989.

[Ree89] C.van Reeuwijk,
"Writing semantic functions for analog system descriptions",
Esprit report E881/an5/TUD/CvR/8905, May 1989.

[Ree90] C.van Reeuwijk,
"Glass Environment implementation and maintenance",
Esprit report E881/BF2/TUD/CvR/9005, May 1990.

[Ree87] D.Reeves and M.J.Irwin,
"Fast Methods for Switch-Level Verification of MOS Circuits",
IEEE Transactions on Computer-Aided Design,
Vol. CAD-6, No.5, Sep. 1987, pp. 766-779.

[Rot66] J.P.Roth,
"Diagnosis of Automata Failures: A Calculus and a Method",
IBM J.Research and Development,
July 1966, pp. 278-291.

[Rot77] J.P.Roth,
"Hardware verification",
IEEE Transactions on Computers,
Vol. C-26, 1977, pp. 1292-1294.

[Rot80] J.P.Roth,
Computer Hardware Testing and Verification,
Potomac, MD: Computer Science Press, 1980.

[Sdl88] CCITT Recommendation Z.100:
Specification and Description Language SDL,
AP IX-35, 1988. Annexes A-F to Z.100, COM X-R16,R25-29,R42.

[Seu89] M. Seutter
"Revised Report on the language Glass"
Esprit report E881/b??/KUN/MS/8911, November 1989

[Sha85] M.Shahdad, R.Lipsett, E.Marschner, K.Sheehan, H.Cohen, R.Waxman, D.Ackley,
"VHSIC Hardware Description Language",
Computer,
Vol. 18, No. 2, Feb 1985, pp.94-103.

[She83] M.Sheeran,
muFP, an Algebraic VLSI Design Language,
Ph.D. Thesis, Programming Research Group, University of Oxford, 1983.

[Sim1] "SII-SIMLOG: Users guide",
ITT document 653 JNGY xxxx-PW 001

[Sup86] K.J.Supowit and S.J.Friedman,
"A new method for verifying sequential circuits",
in *Proc. 23rd Design Automation Conference*,
June 1986.

[Suz85] N.Suzuki,
"Concurrent Prolog as an Efficient VLSI Design Language",

Computer,
Vol. 18, No. 2, Feb 1985, pp.33-40.

[Ter83] C.J.Terman,
"Simulation tools for digital LSI design",
Ph.D thesis, Dept. Electrical Engineering and Computer Science",
MIT, Cambridge, 1983.

[Ter83] C.J.Terman.
"RSIM - A logic-level timing simulator",
in *Proc. IEEE Int. Conf. Computer Design*,
(New York), Nov. 1983, pp. 437-440.

[Tex83] "VHSIC Phase 1 Program HDL/Design Simulator User's Manual",
Texas Instruments, Dallas, Texas, Nov. 1983.

[Tsa83] M.Y.Tsai, "Pass transistor networks in MOS technology:
Synthesis, performance, and testing",
in *Proc. IEEE Int. Symp. Circuits and Systems*,
(Newport Beach, CA), May 1983, pp. 509-512.

[Tur85] D.A.Turner,
"Miranda: A non-strict functional language with polymorphic types",
Functional Programming Languages and Computer Architecture,
Lecture Notes in Computer Science, Vol.201,
Springer-Verlag, Berlin, 1985, pp 1-16.

[Tur79] D.A. Turner,
Sasl language Manual (rev. ed.)
University of Kent, Kent, England, 1979.

[Ulr86] E.Ulrich and I.Suetsugu,
"Techniques for Logic and Fault Simulation",
VLSI Systems Design,
Oct. 1986, pp. 68-81.

[Wee83] W.Weeks, A.Jimenez, G.Mahoney, D.Mehta, H.Qassemzadeh, and T.R.Scott,
"Algorithms for ASTAP - A network analysis program",
IEEE Trans. Circuit Theory,
Vol.CT-20, Nov. 1983, pp. 628-634.

[Wei86] "Logic verification and test generation for VLSI circuits",
Ph.D. Dissertation, U.C.Berkeley, Sept. 1986.

[Wes85] N.Weste and K.Eshraghian,
Principles of CMOS VLSI Design: A Systems Perspective,
Addison-Wesley, 1985.

[Wij76] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker,
Revised Report on the Algorithmic Language ALGOL 68,
Springer-Verlag, Berlin, 1976.

[Win88] T.Windeatt, "Handling complexity of test generation using a constraint satisfaction approach",
Report University of Surrey, June 1988.

[Win87] G.Winskel,
"Models and logic of MOS circuits",
Logic of Programming and Calculi of Discrete Design,
NATO ASI Series, Series F, Computer and Systems Sciences,
Vol.36 (Springer-Verlag, Berlin, 1987), pp. 367-413.

[Wir82] N.Wirth,
Hades: A Notation for the Description of Hardware,
ETH Zurich, Aug. 1982.

[Wir82] N.Wirth
Programming in Modula-2,
Springer-Verlag, New York, N.Y., 1982.

Contents