PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 35, NO. 3, PP. 221-234 (1991)

FORMAL HARDWARE DESCRIPTION USING GLASS[1]

M. SEUTTER

Dept. of Computer Science
Faculty of Mathematics and Informatics
University of Nijmegen, The Netherlands

Received Aug. 29, 1990.

Abstract

The principle of systems semantics is discussed. The language *Glass* based upon this principle is described and its usage for hardware description. The implementation of the description environment for this language developed in Esprit project 881 will be presented.

1 Introduction

The aim of Esprit project 881, 'Forfun' (*For*mal description of arbitrary systems by means of *fun*ctional languages), was a feasibility study of the implementability of the principle of systems semantics (BOUTE). In the course of the project a system description language for digital and analogue systems was designed called *Glass* (*G*eneral *la*nguage for *s*ystems *s*emantics). Furthermore an implementation of a prototype user environment, called *Glue* (*Gl*ass *u*ser *e*nvironment) supporting the language and the semantic functions expressing the various interpretations of the language has been made.

In Section 2 of this article we will discuss the principle of systems semantics. Section 3 will describe the language *Glass* and its usage in hardware design. In Section 4 we will look at the implementation of the environment and the set of semantic functions thus far developed. Finally we will draw some conclusions in the last Section.

2 Systems Semantics

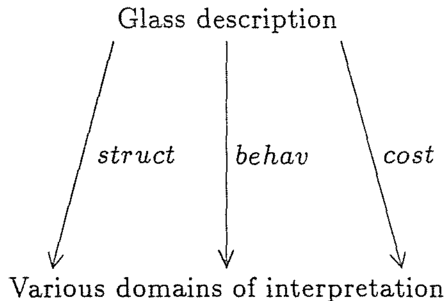
VLSI design is a very complex activity. One deals with thousands of transistors, nets, pins, etc. One therefore would like to describe hardware formally and then verify such a design in an automated way.

[1]Research supported by CEC as Esprit project 881, 'Forfun'

Hardware has many aspects such as behaviour, structure, reliability, etc. Within one aspect one might even discern several subaspects. (For instance regarding behaviour: one may be interested in the static behaviour of a logic circuit, expressed by Boolean functions, or in the electric behaviour determined by the loading, etc.)

Current hardware description languages often have just one 'a priori' semantics, describing only one aspect, so that several languages and descriptions are necessary to describe a system in all of its aspects. Most of these languages are also deficient in that they describe a system in an algorithmic way, whereas most systems are essentially not algorithmic. Such an algorithmic description of a system is only suitable for simulation, which again deals with just one aspect of the system. The non-algorithmic aspects must then be expressed in a different way (often in natural language).

In Esprit project 881, Forfun, we try to cope with these problems in the following way: we 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:



Consider for instance the following description of a dataselector:
Def

$$select \in E \times E \times E \Rightarrow E;$$

$$select\ \langle sel, a, b \rangle = or\ \langle and\ \langle not\ s, a \rangle,\ and\ \langle s, b \rangle \rangle;$$

If we apply the semantic function *struct* to this definition we obtain its structural interpretation, represented by its schematic in *Fig. 1.*

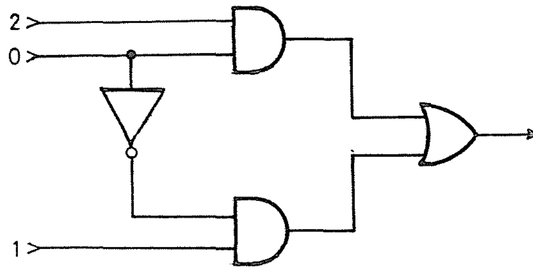


Fig. 1. Structural interpretation of select

Applying the semantic function *cost* to this description will yield the cost of that description according to some cost model. (Taking the number of gates in a description as model we would obtain 4).

In the same way, if we apply the semantic function *simplex* (truth table semantics) to this description we would derive a function of type $\mathcal{B}^3 \rightarrow \mathcal{B}$ describing its static (truth table) behaviour:

$$(\textit{simplex select})\ (0,1,0) = 1$$
$$(\textit{simplex select})\ (1,0,0) = 0$$

Or, if we wish to have a more detailed view of the above description we apply the semantic function *dtme* (Discrete timing model using Eichelberger algebra) which then yields a function of type $(\mathcal{N} \rightarrow \mathcal{E})^3 \rightarrow \mathcal{N} \rightarrow \mathcal{E}$ (with $\mathcal{E} = \{0, 1, x\}$) expressing (part of) its dynamic behaviour.

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*. For each of the atoms the semantic function writer has to code the desired meaning into the semantic function.

3 The Language Glass

3.1 Basic Ideas

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 we will use only one basetype E, indicating 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 (often 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.

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. In this article we will mainly focus on directional systems. However, the reader is warned that the language *Glass* is more general.

3.2 Directional Systems

Consider the following description, introducing some well known primitive components:

```
Basetype E;
Atom
```

$not \in E \Rightarrow E,$

$and \in E \times E \Rightarrow E,$

$nand \in E \times E \Rightarrow E,$

$or \in E \times E \Rightarrow E,$

$xor \in E \times E \Rightarrow E;$

The fat arrow '$\Rightarrow$' in the type specifications denotes the directionality of the interface. 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. The Cartesian product '$\times$' indicates the bundling of terminals into compound ones.

Let us introduce a composite system description:

Def

$select \in E \times E \times E \Rightarrow E;$

$select\ \langle sel, a, b\rangle = or\ \langle and\ \langle not\ s, a\rangle,\ and\ \langle s, b\rangle\rangle;$

Composite system descriptions consist of two parts, namely the declaration of the external interface and its decomposition into subsystems. In this decomposition iuxtaposition denotes system application. 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 an input. The angle brackets '$\langle$' and '$\rangle$' are used to form tuples of terminals. These may be used in formal arguments as well as in expressions:

Def

$halfadder \in E \times E \Rightarrow E \times E;$

$halfadder\ \langle x, y\rangle = \langle xor\langle x, y\rangle,\ and\ \langle x, y\rangle\rangle;$

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

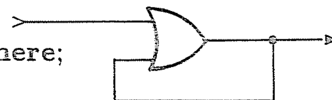
In *Glass feedback* and also *fanout* (that is the connection of several inputs to one output) may be described by a **where**-*clause*, with which you may introduce *local* definitions. Consider for instance the following definition:

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*:

Def

$RSFF \in E^2 \Rightarrow E^2;$

$RSFF \langle R', S' \rangle = \langle q, q' \rangle$

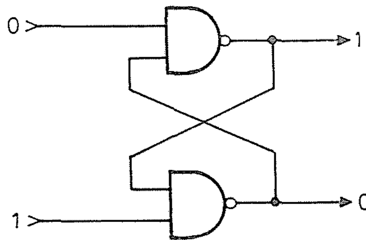
where

$q = nand \langle S', q' \rangle;$

$q' = nand \langle R', q \rangle$

endwhere;

whose structural interpretation can be represented by this schematic:



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

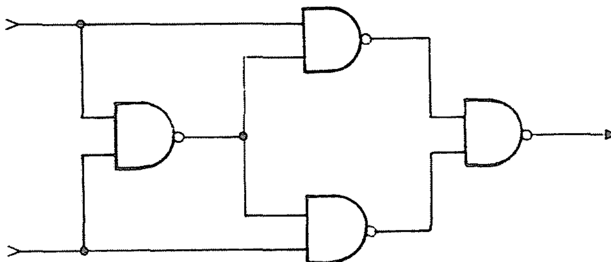
Def

$my_xor \in E \times E \Rightarrow E;$

$my_xor \langle x, y \rangle = nand \langle nand \langle b, x \rangle, nand \langle b, y \rangle \rangle$

where $b = nand \langle x, y \rangle$ **endwhere**;

whose structural interpretation is represented by this schematic:



3.3 Adirectional and Hybrid 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 cannot 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 the 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:

Atom

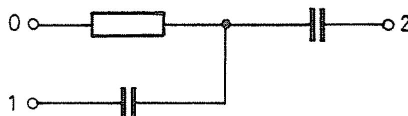
$R \in [E \times E]$,
$C \in [E \times E]$;

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.* 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).

Def

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

with the following structural interpretation:



Of course, circuits exist that are only partially adirectional, that is, some of its terminals are 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 an 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:

Atom

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

As one might wish to combine such circuits with directional ones there must exist an equivalence between the types of directional systems and the types of systems that are adirectional but have only a directional interface. In *Glass* the type $U \Rightarrow V$ is equivalent to the type $[?U \times !V]$. Thus $E^2 \Rightarrow E$ is equivalent to $[?(E^2) \times !E]$. With these concepts we can describe systems that are directional to the outside world but internally are decomposed in adirectional components. A nice example of this can be seen in the following description of an invertor in CMos technology:

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

$CMosInvertor \in E \Rightarrow E;$

$CMosInvertor \ \langle in, out \rangle =$

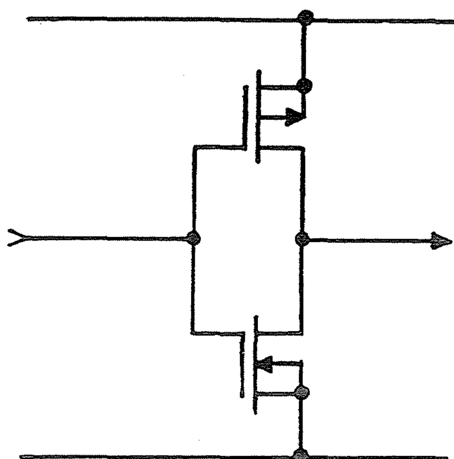
$\{Penh \ \langle in, plus, out \rangle,$

$Nenh \ \langle in, gnd, out \rangle,$

$Vdd \ plus,$

$Gnd \ gnd\};$

which has the following structural interpretation:



Remark how supplies can be introduced as atomic components having only one adirectional terminal.

3.4 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.

The language contains a macro expansion mechanism, with which you can describe regular systems. A macro describes only how to generate all the descriptions that make up such a regular circuit. When using *Glass*, the macro expander in the *Glass* 'describing environment' replaces all applications of macros by their expansions. Remark also that a macro does not have a meaning nor can a semantic function ascribe a meaning to it.

With $U \rightarrow V$ we denote the set of all functions with U as domain and V as codomain. 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. When reading type constructions containing both type constructors, the $\Rightarrow$ has the higher precedence.

Consider the next example:

Atom

$divide_by_two \in E \Rightarrow E$;

Mac

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

Def

$divide_by_8 \in E \Rightarrow E;$
$divide_by_8 \; in = triple \; divide_by_two \; in;$

Macro expansion will lead to a cascade of three *divide_by_two* circuits.

The language 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 an integer type parameter, which may then be used to specify the sizes of the constituents in the rest of the type. Thus one may specify

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

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

In order to enable case distinction in an 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.

Using pattern matching we may now be able to specify an adder chain:

Atom

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

Mac

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

Def

$Fourbitsadder \in E^4 \times E^4 \times E \Rightarrow E \times E^4;$
$Fourbitsadder \; \langle as, bs, cin\rangle = nbitsadder \; 4 \; \langle as, bs, cin\rangle.$

Calculation may be used to steer macro expansion. In fact *Glass* is a full functional language containing special syntax to describe systems and system types.

However, constructs that are the result of macro expansion are required to be systems, or in other words, the resulting expressions after macro expansion *must* have a structural interpretation. The set of all *Glass* descriptions resulting from macro expansion is called *kernel Glass*.

3.5 Designing in Glass

When designing systems in *Glass* one can discern several major design methodologies: *top down*, *horizontal* and *vertical*.

With the top down methodology one refines a description stepwise from an abstract top level description to a more detailed low level description. One decomposes this top level system description into functional blocks, these functional blocks into logical building blocks which in their turn may be decomposed into systems built with gates.

With the horizontal methodology we mean altering an existing description in such a way that only one aspect changes (decreasing cost, silicon area, power consumption, etc.) while keeping the other aspects (behaviour) the same. This methodology is typically used for optimization of the design.

With the vertical methodology one refines a description by redefining the atomic components as compound ones using lower level atoms as building blocks. In this way one can inspect several aspects of a system at a much lower level (by appropriate semantic functions). One example of this is checking if the behaviour of a circuit at gate level is matched by its behaviour at switch level.

4 The Environment

The *Glass* user environment consists of a 5 pass front end which translates and expands *Glass* text into appropriate kernel *Glass* abstract syntax trees (asts), a number of semantic functions, for the digital as well as for the analogue domain, and a menu driven shell which hides the user from bare UNIX commands.

A more detailed picture of the front end is presented in *Fig. 2* First the C preprocessor is used to perform inclusions of other *Glass* files. Then the parser translates the text into a full *Glass* ast. This is then partially checked by the context sensitive analysis done by the type checker. This check can only be partial because parametrized types only get their fixed size after macro expansion. If no errors are detected thus far, macro expansion will expand all macro applications. After that a last type check is performed, checking the sizes of all terminals in the description. Finally the description is stored on disk using a text representation of the ast of the description, inspired by *Miranda* abstract datastructures.

The parser and macro expander are written in *Glammar*, which is a variant of EAG (WATT, 1974) (MEIJER, 1986). The type checker and size check are currently written in *C*, although their first versions were written

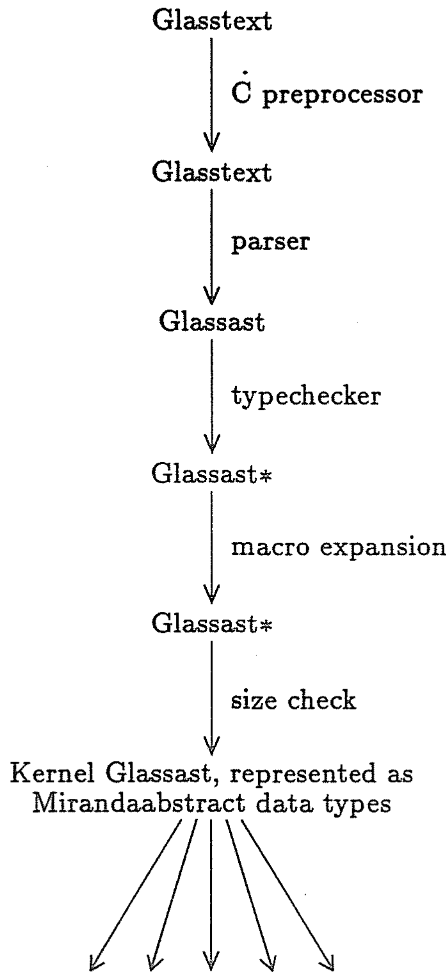


Fig. 2 The front end of the environment

in *Pascal*. The semantic functions are written in several languages: *C*, *Miranda*, *Prolog*, *Pascal*, and others. This variety of programming languages caused severe portability problems during the course of the project. Moreover during the course of the project the actual form and representation of the datastructures was not stable.

A tool therefore was developed, called *Tm* (VON REEUWIJK, 1989). This takes a datastructure definition file and a template file as input and generates data structure definitions plus access, building, loading, saving and traversion routines for one of the languages used in the project, steered

by the template file. Input of the datastructures of a description to a semantic function is coded by one call only to the generated routines thereby shortening the development time of that semantic function.

During the project semantic functions were developed in three kinds: The first kind translates the kernel *Glass* datastructures directly into the desired aspect of the system. An example of these are simple cost functions. The second translates the datastructures into computer programs expressing for instance behaviour in terms of the semantics of that computer language. These then may be compiled and run. (This is in fact simulation). The third kind converts the datastructures to input for already existing analysis and simulation tools.

The following semantic functions have been developed until now:

- for the digital domain:
  - simplex: truth table semantics
  - multiplex: stream semantics
  - uflat: flattener for directional systems
  - uflat2: same, but flattens to a list of atom applications
  - dtm: discrete timing model
  - dtme: discrete timing model, Eichelberger variant
  - discev: discrete event modelling
  - mossimII: switch level simulation
  - senspath: sensitized paths
  - cpm: worst case delay
- for the analogue domain:
  - kgflat: flattener for adirectional systems
  - kgspice: conversion to *Spice*
  - kganp: conversion to *Anp3* (Analysis of zeroes and poles)
  - kggnl: conversion to netlist format
  - gldraw: conversion to graphics language for schematics

5 Conclusions and Future Work

A language for systematic hardware description has been designed based upon the principle of systems semantics. A 'describing environment' has been implemented as well as a number of semantic functions.

Our future research will continue into two directions:

- We will try to enhance the set of semantic functions with even more sophisticated functions. One may think of functions handling hybrid descriptions like pass transistors, transmission gates, etc., which are frequently encountered in VLSI design. A further field of research would be the development of general timing models for the digital as

well as for the hybrid field. Another interesting research area is the generation of layout out of *Glass* descriptions.

- *Glass* can only be used for formal hardware *description*. We would like to have synthesis tools that generate *Glass* descriptions out of a desired description of the circuits' behaviour, which then may be verified using the semantic functions. These synthesis tools together with the existing environment would constitute a good environment for doing VLSI design.

References

BOUTE, R. T. (1988): Systems Semantics: Principles, Applications, and Implementation. *ACM Transactions on Programming Languages and Systems*, Vol. 10, No 1, Jan 1988, pp. 118-155.

MEIJER, H. (1986): Programmar, a Translator Generator, PhD Thesis, Catholic University of Nijmegen, 1986.

VAN REEUWIJK, C. (1989): tm: a Code Generator for Structured Data Interfaces (draft), Esprit report E881/b38/TUD/CvR/8905, May 1989.

WATT, D. A. (1974): Analysis – Oriented to-Level Grammars, PhD Thesis, University of Glasgow, 1974.

Address:

M. SEUTTER
Dept. of Computer Science
Faculty of Mathematics and Informatics
University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands