Translating Programs into Delay-Insensitive Circuits

Jo C. Ebergen
Dept. of Computer Science and Mathematics
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven

1. Introduction

In 1938 Claude E. Shannon wrote his seminal paper [23] entitled 'A Symbolic Analysis of Relay and Switching Circuits'. He demonstrated that Boolean algebra could be used elegantly in the design of switching circuits. The idea was to specify a circuit by a set of Boolean equations, to manipulate these equations by means of a calculus, and to realize this specification by a connection of basic elements. The result was that only a few basic elements, or even one element such as the 2-input NAND gate, suffice to synthesize any switching function specified by a set of Boolean equations. Shannon's idea proved to be very fertile and out of it grew a complete theory, called switching theory, which is used by most circuit designers nowadays.

In the thesis [5] a method is presented for designing *delay-insensitive circuits.* (Operationally speaking, a delay-insensitive circuit is a connection of basic elements whose functional operation is insensitive to delays in wires or elements.) The principal idea of this method is similar to that in Shannon's paper: to design a circuit as a connection of basic elements and to construct this connection with the aid of a formalism. The method of constructing such a circuit, as described in [5], is by translating programs satisfying a certain syntax. The result of such a translation is a delay-insensitive connection of elements chosen from a finite set of basic elements. Moreover, this translation has the property that the number of basic elements in the connection is proportional to the length of the program. Furthermore, in [5] a rigorous formalization is given of what it means for such a connection to be delay-insensitive.

In this paper[1] we briefly describe some of the history of designing delay-

1. The research reported in this paper was carried out while the author was working at CWI in Amsterdam.

insensitive circuits and some of the reasons why we would like to design delay-insensitive circuits. By means of an example we convey the idea of designing delay-insensitive circuits. We conclude with an outline of the method described in [5].

2. Some history

Delay-insensitive circuits are a special type of circuits. We briefly describe their origins and how they are related to other types of circuits and design techniques. The most common distinction usually made between types of circuits is the distinction between *synchronous circuits* and *asynchronous circuits.*

Synchronous circuits are circuits that perform their (sequential) computations based on the successive pulses of the clock. From the time of the first computer designs many designers have chosen to build a computer with synchronous circuits. In [25] Alan Turing, one of the first computer designers, has motivated this choice as follows:

> We might say that the clock enables us to introduce a discreteness into time, so that time for some purposes can be regarded as a succession of instants instead of a continuous flow. A digital machine must essentially deal with discrete objects, and in the case of the ACE (Automatic Computing Engine) this is made possible by the use of a clock. All other digital computing machines except for human and other brains that I know of do the same. One can think up ways of avoiding it, but they are very awkward.

In the past fifty years many techniques for the design of synchronous circuits have been developed and are described by means of switching theory [11, 15]. The correctness of synchronous systems relies on the bounds of delays in elements and wires. The satisfaction of these delay requirements cannot be guaranteed under all circumstances, and for this reason problems can crop up in the design of synchronous systems. (Some of these problems are described in the next section.) In order to avoid these problems interest arose in the design of circuits without a clock. Such circuits have generally been called *asynchronous* circuits.

The design of asynchronous circuits has always been and still is a difficult subject. Several techniques for the design of such circuits have been developed and are discussed in, for example, [11, 15, 28]. For special types of such circuits formalizations and other design techniques have been proposed and discussed. David E. Muller has given a formalization of a type of circuits which he coined by the name of *speed-independent* circuits. An account of this formalization is given in [16].

From a design discipline that was applied in the Macromodules project [3, 4] at Washington University in St. Louis, the concept of a special type of circuit evolved which was given the name *delay-insensitive* circuit. It was realized that a proper formalization of this concept was needed in order to specify and design such circuits in a well-defined manner. A formalization of the

concept of a delay-insensitive circuit was later given in [26]. For the design and specification of delay-insensitive circuits several methods have been developed based on, for example, Petri Nets and techniques derived from switching theory [17].

Another name that is frequently used in the design of asynchronous circuits is *self-timed systems.* This name was introduced by C. L. Seitz in [22] in order to describe a method of system design without making any reference to timing except in the design of the self-timed elements.

Recently, Alain Martin has proposed some interesting and promising design techniques for circuits of which the functional operation is unaffected by delays in elements or wires [12, 13]. His techniques are based on the compilation of CSP-like programs into connections of basic elements. The techniques presented in [5] exhibit a similarity with the techniques applied by Alain Martin.

3. Why delay-insensitive circuits ?

The reasons for designing delay-insensitive systems are manifold. One reason why there has always been an interest in asynchronous systems is that synchronous systems tend to reflect a *worst-case* behavior, while asynchronous systems tend to reflect an *average-case* behavior. A synchronous system is divided into several parts, each of which performs a specific computation. At a certain clock pulse, input data are sent to each of these parts and at the next clock pulse the output data, i.e. the results of the computations, are sampled and sent to the next parts. The correct operation of such an organization is established by making the clock period larger than the worst-case delay for any subcomputation. Accordingly, this worst-case behavior may be disadvantageous in comparison with the average-case behavior of asynchronous systems.

Another more important reason for designing delay-insensitive systems is the so-called *glitch phenomenon.* A glitch is the occurrence of metastable behavior in circuits. Any computer circuit that has a number of stable states also has metastable states. When such a circuit gets into a metastable state, it can remain there for an indefinite period of time before it resolves into a stable state. For example, it may stay in the metastable state for a period larger than the clock period. Consequently, when a glitch occurs in a synchronous system, erroneous data may be sampled at the time of the clock pulses. In a delay-insensitive system it does not matter whether a glitch occurs: the computation is delayed until the metastable behavior has disappeared and the element has resolved into a stable state. One frequent cause for glitches are, for example, the asynchronous communications between independently clocked parts of a system.

The first mention of the glitch problem appears to date back to 1952 (cf. [1]). The first publication of experimental results of the glitch problem and a broad recognition of the fundamental nature of the problem came only after 1973 [2, 8] due to the pioneering work on this phenomenon at the Washington University in St. Louis.

A third reason is due to the effects of *scaling.* This phenomenon became

prominent with the advent of integrated circuit technology. Because of the improvements of this technology, circuits could be made smaller and smaller. It turned out, however, that if all characteristic dimensions of a circuit are scaled down by a certain factor, including the clock period, delays in long wires do not scale down proportional to the clock period [13, 21]. As a consequence, some VLSI designs when scaled down may no longer work properly anymore, because delays for some computations have become larger than the clock period. Delay-insensitive systems do not have to suffer from this phenomenon if the basic elements are chosen small enough so that the effects of scaling are negligible with respect to the functional behavior of these elements [24].

A fourth reason is the clear separation between functional and physical correctness concerns that can be applied in the design of delay-insensitive systems. The correctness of the behavior of basic elements is proved by means of physical principles only. The correctness of the behavior of connections of basic elements is proved by mathematical principles only. Thus, it is in the design of the basic elements only that considerations with respect to delays in wires play a role. In the design of a connection of basic elements no reference to delays in wires or elements is made. This does not hold for synchronous systems where the functional correctness of a circuit also depends on timing considerations. For example, for a synchronous system one has to calculate the worst-case delay for each part of the system and for any computation in order to satisfy the requirement that this delay must be smaller than the clock period.

As a last reason, we believe that the translation of parallel programs into delay-insensitive circuits offers a number of advantages compared to the translation of parallel programs into synchronous systems. In [5] a method is presented with which the synchronization and communication between parallel parts of a system can be programmed and realized in a natural way.

4. An example

In order to get an idea of designing delay-insensitive circuits we describe in an informal way a small example. Consider the modulo-3 counter specified by the following communication behavior. The modulo-3 counter has three communication actions: one input, denoted by $a?$, and two outputs, denoted by $p!$ and $q!$. The communication behavior is an alternation of inputs and outputs, starting with an input. The outputs depend on the inputs as follows. After the n-th input, where $n>0$, if $n \bmod 3 \neq 0$, then output $q!$ is produced, else output $p!$ is produced. This behavior is expressed in the following program, or so-called *command*,

$$E0 = \mathbf{pref}[a?;q!;a?;q!;a?;p!].$$

Here, $[E]$ denotes repetition of the enclosed behavior E and $E1;E2$ denotes concatenation of $E1$ and $E2$. The notation $\mathbf{pref}\,E$ denotes the prefix-closure of the behavior E, i.e. if the string of symbols (also called *trace*) $a?q!a?q!a?p!$ is a possible behavior of E, then also each prefix of this trace is a possible

behavior of **pref** E.

The following physical interpretation may be associated with the symbols. With each symbol corresponds a terminal of the circuit and with each occurrence of that symbol in a trace corresponds a voltage transition (either a high-going or a low-going transition) in that terminal. Voltage transitions corresponding to inputs are caused by the environment of the circuit; voltage transitions corresponding to outputs are caused by the circuit itself.

In the same way the basic TOGGLE and XOR component can be specified as given in Figure 1.

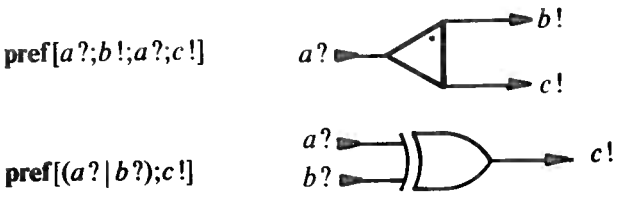

FIGURE 1

The first component is the TOGGLE component and can be considered as a modulo-2 counter. The second component is an XOR component and has the following repetitive behavior. First, the environment provides either an input $a?$ or an input $b?$ (the | separates the alternatives), and then the component produces an output $c!$. After the environment has received an output $c!$ it may produce a new input again, and so on. (Notice that the behaviors of components are specified as orderings of events instead of as logical functions.)

We emphasize that all specifications must be understood as prescriptions for the behavior of the component *and* environment. Consequently, in constructing a decomposition for the modulo-3 counter $E0$ we assume that the environment satisfies the prescribed behavior in $E0$, i.e. the environment provides new inputs $a?$ only when an output has been received. Under this assumption the modulo-3 counter can be decomposed as depicted in Figure 2.

FIGURE 2

Notice that in the decomposition the prescription for the environment of every basic component is not violated. Without much difficulty we can convince ourselves that the functional behavior of this decomposition is unaffected by delays in connection wires or in basic elements. In other words, we could say that the modulo-3 counter is realized by a delay-insensitive connection of

basic elements. Knowing how to construct a modulo-3 counter the reader may try, as an exercise, to construct a modulo-17 counter from TOGGLE and XOR components. (There exist several solutions, some more efficient than others.)

5. Outline of the method

The method presented in [5] for designing delay-insensitive circuits is briefly described as follows. An abstraction of a circuit is called a *component*; components are specified by programs written in a notation based on *trace theory*. Trace theory was inspired by Hoare's CSP [6, 7] and developed by a number of people at the University of Technology in Eindhoven. It has proven to be a good tool in reasoning about parallel computations [18, 19, 24, 9] and, in particular, about delay-insensitive circuits [10, 20, 21, 26, 27].

The programs are called *commands* and can be considered as an extension of the notation for regular expressions. Any component represented by a command can also be represented by a regular expression, i.e. it is also a *regular* component. The notation for commands, however, allows for a more concise representation of a component due to the additional programming primitives in this notation. These extra programming primitives include operations to express parallelism, tail recursion (for representing finite state machines), and projection (for introducing internal symbols).

Based on trace theory the concepts of *decomposition* of a component and of *delay-insensitivity* are formalized. The decomposition of a component is intended to represent the realization of that component by means of a connection of circuits. Several theorems are presented that are helpful in finding decompositions of a component. Delay-insensitivity is formalized by the definition of *DI decomposition*. A DI decomposition represents a realization of a component by means of a *delay-insensitive* connection of circuits. In order to link decomposition and DI decomposition, the definition of a DI component is introduced. Operationally speaking a DI component represents a circuit for which the communication between circuit and environment takes place in a delay-insensitive way. (It turns out that the definition of a DI component is equivalent with Udding's formalization of a delay-insensitive circuit.) By means of the definition of a DI component one of the fundamental theorems in the thesis can be formulated as follows: DI decomposition and decomposition are equivalent if all components involved are DI components.

This theorem is applied as follows to the example described in the previous section. We showed, informally, that the modulo-3 counter can be decomposed into TOGGLE and XOR components. Furthermore, we have that the TOGGLE component, XOR component, and modulo-3 counter $E0$ are DI components. Consequently, it follows by the above mentioned theorem that the decomposition of Figure 2 forms a DI decomposition of the modulo-3 counter.

Because of the above mentioned theorem, it is important to have techniques to recognize DI components. For this purpose a number of so-called *DI grammars* are developed, i.e. grammars for which any command generated by these

grammars represents a (regular) DI component.

Based on these grammars syntax-directed translations of commands into DI decompositions of components represented by these commands are developed. With these grammars the language $\mathfrak{L}_4$ of commands is defined. It is shown that any regular DI component represented by a command in the language $\mathfrak{L}_4$ can be decomposed in a syntax-directed way into the finite set $\mathbb{B}$ of basic DI components and so-called *CAL components.* CAL components are also DI components. Consequently, since all components involved are DI components, the decomposition into these components is, by the above theorem, also a DI decomposition.

The set of all CAL components is, however, not finite. In order to show that a decomposition into a finite basis of components exists, two decompositions of CAL components are discussed: one decomposition into the finite basis $\mathbb{B}0$ and one decomposition into the finite basis $\mathbb{B}1$. The decomposition of CAL components into the basis $\mathbb{B}1$ is in general *not* a DI decomposition, since not every component in $\mathbb{B}1$ is a DI component. This decomposition, however, is in general simpler than the decomposition into $\mathbb{B}0$ and can be realized in a simple way if so-called *isochronic forks* are used in the realization. The decomposition of CAL components into the basis $\mathbb{B}0$ is an interesting but difficult subject. Since every component in $\mathbb{B}0$ is a DI component, every decomposition into $\mathbb{B}0$ is therefore also a DI decomposition. In [5] a general procedure for the decomposition of CAL components into the basis $\mathbb{B}0$ is described, which is conjectured to be correct

The complete decomposition method can be described as a syntax-directed translation of commands in $\mathfrak{L}_4$ into commands of the basic components in $\mathbb{B}0$ or $\mathbb{B}1$. Consequently, the decomposition method is a constructive method and can be completely automated: as soon as we have a specification of a component expressed as a command in $\mathfrak{L}_4$ we can find mechanically a decomposition of this component into $\mathbb{B}0$ or $\mathbb{B}1$. Moreover, it is shown that the result of the complete decomposition of any component expressed in $\mathfrak{L}_4$ can be linear in the length of the command, i.e. the number of basic elements in the resulting connection is proportional to the length of the command.

Although many regular DI components can be expressed in the language $\mathfrak{L}_4$, which is the starting point of the translation method, probably not every regular DI component can be expressed in this way. Nevertheless, it is also shown that for any regular DI component there exists a decomposition into components expressed in $\mathfrak{L}_4$, which can then be translated by the method presented.

6. Concluding remarks

The research described in [5] has been fascinating and many-sided. It includes, for example, aspects of

- Language design: which programming primitives do we include in the language in order to be able to present a clear and concise program for a component?
- Programming methodology: do there exist techniques to design programs

from specifications for components in the language of commands?
- Translation techniques: how do we translate programs into connections of basic elements?
- Syntax and semantics: how we can satisfy semantic properties (like a DI component) by imposing syntactic requirements on programs?
- VLSI design: what physical constraints must be met in order to realize the circuit designs obtained in the VLSI medium?

In the thesis the aim of delay-insensitive design has been pursued as far as possible, i.e. correctness arguments based on delay-assumptions have been postponed as far as possible, in order to see what sort of designs such a pursuit would lead to. In this approach our first concern has been the correctness of the designs and only in the second place have we addressed their efficiency. Accordingly, although the number of basic components is already proportional to the length of the program, still many optimizations are possible in translating programs into delay-insensitive circuits.

REFERENCES

1. T.J. CHANEY, *A Comprehensive Bibliography on Synchronizers and Arbiters*, Technical Memorandum No. 306C, Institute for Biomedical Computing, Washington University, St. Louis.
2. T.J. CHANEY, C.E. MOLNAR (1973). Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers C-22*, 421-422.
3. W.A. CLARK (1967). Macromodular computer systems. *Proceedings of the Spring Joint Computer Conference*, AFIPS.
4. W.A. CLARK, C.E. MOLNAR (1974). Macromodular computer systems. R. STACY, B. WAXMAN (eds.). *Computers in Biomedical Research, Vol. IV*, Academic Press, New York.
5. JO C. EBERGEN (1987). *Translating Programs into Delay-Insensitive Circuits*, Ph. D. Thesis, Eindhoven University of Technology.
6. C.A.R. HOARE (1978). Communicating sequential processes. *Communications of the ACM 21*, 666-677.
7. C.A.R. HOARE (1985). *Communicating Sequential Processes*, Prentice-Hall.
8. M. HURTADO (1975). *Dynamic Structure and Performance of Asymptotically Bistable Systems*, D. Sc. Dissertation, Washington University, St. Louis.
9. ANNE KALDEWAIJ (1986). *A Formalism for Concurrent Processes*, Ph.D. Thesis, Eindhoven University of Technology.
10. ANNE KALDEWAIJ (1987). The translation of processes into circuits. J.W. DE BAKKER, A.J. NIJMAN, P.C. TRELEAVEN (eds.). *Proceedings PARLE, Parallel Architectures and Languages Europe, Vol. I*, Springer LNCS, 195-213.
11. ZVI KOHAVI (1970). *Switching and Finite Automata Theory*, McGraw-Hill.
12. ALAIN J. MARTIN (1985). The design of a self-timed circuit for distributed mutual exclusion. H. FUCHS (ed.). *Proceedings 1985 Chapel Hill Conference on VLSI*, Computer Science Press, 247-260.

13. ALAIN J. MARTIN (1986). Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing 1,* 226-234.
14. CARVER MEAD, MARTIN REM (1982). Minimum propagation delays in VLSI. *IEEE Journal of Solid-State Circuits SC-17,* 773-775.
15. R.E. MILLER (1965). *Switching Theory,* Wiley.
16. R.E. MILLER, Chapter 10 in: [15].
17. C.E. MOLNAR, T.P. FANG, F.U. ROSENBERGER (1985). Synthesis of delay-insensitive modules. H. FUCHS (ed.). *Proceedings 1985 Chapel Hill Conference on VLSI,* Computer Science Press, 67-86.
18. MARTIN REM (1985). Concurrent computations and VLSI circuits. M. BROY (eds.). *Control Flow and Data Flow: Concepts of Distributed Computing,* Springer-Verlag, 399-437.
19. MARTIN REM (1987). Trace theory and systolic computations. J.W. DE BAKKER, A.J. NIJMAN, P.C. TRELEAVEN (eds.). *Proceedings PARLE, Parallel Architectures and Languages Europe, Vol. I,* Springer LNCS, 14-34.
20. HUUB M.J.L. SCHOLS (1985). *A Formalisation of the Foam Rubber Wrapper Principle,* Master's Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology.
21. HUUB M.J.L. SCHOLS, TOM VERHOEFF (1985). *Delay-Insensitive Directed Trace Structures Satisfy the Foam Rubber Wrapper Postulate,* Computing Science Notes 85/04, Department of Mathematics and Computing Science, Eindhoven University of Technology.
22. C.L. SEITZ (1980). System Timing. CARVER MEAD, LYNN CONWAY. *Introduction to VLSI Systems,* Addison-Wesley, 218-262.
23. CLAUDE E. SHANNON (1938). A symbolic analysis of relay and switching circuits. *Trans. AIEE, 57,* 713-723.
24. JAN L.A. VAN DE SNEPSCHEUT (1985). *Trace Theory and VLSI Design,* LNCS 200, Springer-Verlag.
25. ALAN M. TURING (1986). Lecture to the London Mathematical Society on 20 February 1947. B.E. CARPENTAR, R.W. DORAN (eds.). *Charles Babbage Institute Reprint Series for the History of Computing, Vol. 10,* MIT Press.
26. JAN TIJMEN UDDING (1984). *Classification and Composition of Delay-Insensitive Circuits,* Ph.D. Thesis, Eindhoven University of Technology.
27. JAN TIJMEN UDDING (1986). A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing 1,* 197-204.
28. S.H. UNGER (1969). *Asynchronous Sequential Switching Circuits,* Wiley Interscience.