This is the authors’ final peer reviewed (post print) version of the item published as:


Available from Deakin Research Online:

http://hdl.handle.net/10536/DRO/DU:30029504

Reproduced with the kind permission of the copyright owner.

Copyright : 1987, Institution of Engineers
1 Introduction

Systems characterised by general graphs are very flexible because for a given application they offer a great number of degrees of freedom and consequently a great possibility of choice to the designer. This can become a drawback in practice when myriads of cases have to be examined analytically and most of them rejected for a variety of heterogeneous reasons.

For such design problems, it appears that logic programming languages are ideally suited because they can implement the heterogeneous rules which describe the desired system in a natural way. An application of the Prolog language to the design of active RC biquad circuits investigated by Mouly in his doctoral thesis [1] (see also [2], [3]) illustrates the advantages of such a rule-based approach.

The solution proceeds in two stages: firstly, the class of objects (specific circuits in this case) is defined algebraically, and secondly the properties of the desired system are stated as constraints or selection rules. The constraints correspond to "goals" in Prolog and a statement of these goals and the search domain constitute the essence of a Prolog program. Prolog incorporates an automatic search algorithm which examines all candidates for solutions which simultaneously satisfy all the goals.

However, there are two main drawbacks to the approach. Firstly, the desired properties of the solution system are frequently expressed algebraically, and the basic tools for algebraic manipulation in Prolog have not been developed, or are not yet widely available. Mouly mimicked the necessary algebra with character manipulation operations. The second impediment in applying Prolog to the design of large systems is the slowness of the standard implementations of the language. Mouly and Neirynck [2], [3] reported that many hours of CPU time were required to solve the active biquad problem on a Vax 11/782 and other machines.

Although we admit to some initial scepticism, we tried to code the same problem in Turbo-Prolog, a non-standard variant of the Prolog family which produces programs in compiled form. We were surprised to find that, after some effort spent in re-coding the problem (however without any essential change in the method itself), the problem required only 15 min on a PC/ AT clone with results identical to those in [3]. This encouraged us to apply the approach to the realisation of a digital system.
2 Digital Transfer Function Realisation

A suitable representation of digital networks for this purpose is the signal flow graph [4]. The network is represented as a set of nodes (adders) connected by directional branches. The branches may represent delay elements or constant multipliers. The signal flow graph may be interpreted equally as a hardware block diagram or as an algorithm definition for software implementation.

The most general flow graph includes all possible branches between node pairs. A specific choice for each branch element, namely delay element, constant multiplier or no connection (i.e. zero multiplier), has a corresponding transfer function $H(z)$. The realisation task is, given an $H(z)$, to find a choice of branch elements so that the transfer function has the prescribed form. Additionally, other constraints on the flow diagram may be desired, for example that the number of delay elements be the minimum (a canonic realisation) and that delay-free loops should not occur in the network.

This can be stated as a set of selection rules for a solution network: given a general flow graph, find a set of branch elements such that

1. the network transfer function has the prescribed form,
2. the number of delays is minimum,
3. delay-free loops do not occur,

and any other application-specific constraints.

As an experiment we attempted the realisation of the general biquadratic transfer function

$$H(z) = \frac{a_2 z^{-2} + a_1 z^{-1} + a_0}{b_2 z^{-2} + b_1 z^{-1} + 1}$$

(1)

using a 4-node flow graph without self-loops and with fixed input and output nodes. This graph (Fig 1) has 12 possible directional branches (which could be a unit delay, constant multiplier or null) giving $3^{12} = 531441$ candidates. The equation describing the transfer function of this system in terms of the 12 branch values was computed using the algebraic manipulation package Reduce.

![Fig 1. General 4-node signal flow graph without self-loops.](image)

The solution constraints chosen were those (1)-(3) given above and the problem was coded in Turbo-Prolog using the same character manipulation method for handling the algebraic constraints as used by Mouly. With these rules, the program reported 20 solutions after a run time of about 12 min on a PC/AT clone.
On inspection, 3 of these were rejected because they allowed only real poles, a further 5 because they had poles outside the unit circle, and 2 were rejected because they did not have sufficient degrees of freedom to implement the general biquadratic function (1). This examination was done manually, but in retrospect it could have been incorporated into the Prolog program as additional selection rules.

The remaining 10 solutions fell into 2 groups of 5 which were transposed duals of each other. One of these groups of 5 flow graphs is shown in Fig 2. The set comprises a parent graph (a) with one excess degree of freedom, from which 3 graphs (b)-(d) are obtained by deleting a constant multiplier branch, plus one related graph (e). Graph (b) is the well-known Direct Form II realisation.

![Fig 2. 4-node flow graphs realising the biquadratic transfer function. Branches marked k and z represent constant multipliers and unit delays respectively.](image)

3 Comments
Although this is a simple example it suggests the potential of the method for more advanced digital system design. The implementation of more powerful algebraic...
manipulation tools in Prolog (see for example [5]) should make coding for this type of problem simpler and more efficient. Finally, although Turbo-Prolog has some serious deviations from the standard logic programming languages (in particular, its adherence to strict argument typing), it nevertheless appears suited to a cheap hands-on experiment in the use of logic programming for system design.

4 References