Structural Synthesis

Jens Tingleff [*]   Erik Lindberg [†]

Sept 1991

Abstract

In extension of previous work [1, 2], we present an implementation of Structural Synthesis. The implementation runs on PC type microcomputers, and on Transputer based parallel computers. This implementation is faster than any previously reported.

The programs presented here, `StructSynt` and `CompSynt`, are the first complete suite of programs for structural synthesis.

1 Introduction

Structural Synthesis is a method for filter synthesis where filter structures are generated from a transfer function specification. Our work has concentrated on linear filters with resistors, capacitors and ideal operational amplifiers. Some earlier programs have been written in languages for symbolic processing [3, 4] which meant that they were simple but slow. Other programs, e.g. [2], were written in traditional programming languages which meant that they were complicated but moderately fast. Our programs [5] are written in traditional languages and make use of special bit manipulation routines which means that they are complicated but very fast.

A transfer function (the forward voltage transfer function) is considered to be defined by the quotient between two polynomials in the complex frequency s. Two transfer functions have the *same form* if the degrees, and the signs of the non-zero coefficients, are the same for the numerators of both transfer functions as well as for the denominators.

[*] Postgrad, Dept of EE, Imperial College, Exhibition Road, London SW7 2BT. e-mail: `jensting@titan.ee.ic.ac.uk`

[†] Institute of Circuit Theory and Telecommunications Technical University of Denmark

The program `StructSynt` performs the structural synthesis process proper. The purpose is to take an *initial redundant circuit* (the IRC) and a specification of the form of the transfer function, and generate subcircuits of the initial circuit. The generated circuits have transfer functions of the desired form and, optionally, minimise a user-defined costfunction (derived from the number of components of various types). The IRC is pruned to form a number of subcircuits by a *partial enumeration* process.

`CompSynt` performs a *coefficient matching* process to find a set of component values that makes the subcircuit realise the exact values of the coefficients in the desired transfer function. In other words, this program performs the work done in traditional filter design programs.

2 Using the programs

Through the use of a descriptive and flexible input data syntax, the programs are very easy to use. When using `StructSynt` and `CompSynt` for filter design, the following cycle is recommended (the steps marked * are optional)

1. Define the requested transfer function. This usually involves setting the order of the numerator and denominator polynomials, and finding the poles and zeros (or the coefficients). A program such as `ADFP` [6] can be used to define a transfer function specification from a set of design goals.
2. Find an Initial Redundant Circuit (IRC). This circuit consists of passive elements which are parallel combinations of resistors and capacitors, and active elements which are operational amplifiers. The IRC is a three-terminal circuit.

* Define an optimisation goal for the passive elements, e.g. request that the number of capacitors should be fixed, in the generated circuits.

3. Run `StructSynt` to generate subcircuits of the IRC. The subcircuits will have all the active elements from the IRC, and some (or all) of the passive elements. `StructSynt` will generate a text file describing the resulting subcircuits.

* Sort the resulting circuits from `StructSynt` to eliminate circuits that have no meaning, e.g. circuits where the op-amps do not have negative feedback.

4. Run `CompSynt` on the file generated by `StructSynt`. `CompSynt` will attempt to find a set of component values that realise the actual values of the desired transfer function. `CompSynt` allows scaling of the polynomials, as well as modification of the initial component values.

   `CompSynt` will generate a text file (for input to `ANP3` [7]) for further use.

5. Analyse the resulting circuits with `ANP3` (or any other relevant program), to evaluate the performance. If the performance is not satisfactory, it is possible to alter the requested numerical values of the transfer function and go back to step 4 (as long as the form of the transfer function is the same). This step can use any degrees of freedom, left from step 1, to optimise some performance measure.

If the performance is satisfactory, try to insert models of real elements (e.g. capacitors with finite Q, non-ideal op-amps), and re-fit components values using a program capable of handling those synthesis tasks (e.g. `LCP2` [8]).

Step 4 is quite important, because it assists the designer in the task of removing structures that cannot realise the exact values of the requested transfer function; e.g. the numerator s^2+3s+2 can be realised by a passive RC network, s^2+2s+2 cannot. While `CompSynt` does not provide a mathematical guaranty, the results provide a good background for chosing the circuits that should be analysed more thoroughly.

3 The structural synthesis process

The legal subcircuits are found by a partial enumeration process [2, 9], a special form of non-linear integer programming. We treat the specification of the desired transfer function as a set of constraints, and find all legal values. The optional optimisation is done by only accepting those legal subcircuits that minimise a certain costfunction. The legal subcircuits are termed *usable.*

The IRC is regarded as defining a discrete number space, with a binary digit in the numbers for each passive component in the IRC. Hence, the number space is defined as all values of a binary number x with $2N$ bits (with N the number of passive elements in the IRC). Each bit in the number x indicates whether a passive component is present or not in the subcircuit. The transfer function of the IRC is found by finding all trees for the directed graphs defined by the circuit [11]. Each tree is regarded as a set of bits from x. When finding the symbolic transfer function of the subcircuits, all that has to be done is to collect those trees from the IRC transfer function for which all edges are contained in the subcircuit. Regarding the trees and the subcircuits as sets, it's obvious that the "contained in" property becomes the subset property.

Partial enumeration [10] works by considering subintervals of the entire interval (defined as the values of x, from 0 to $2^{2N}-1$). For a given value x there exists a number $\tilde{x}$ which is the maximum value for which all numbers in the interval $[x,\tilde{x}-1]$ represent circuits that are subcircuits of $\tilde{x}-1$. If the subcircuit defined by $\tilde{x}-1$ does not generate all the coefficients in the requested transfer function, or if the subcircuit defined by x generates some coefficients that are requested to be zero, then the interval from x to $\tilde{x}-1$ contains no usable subcircuits. This idea of discovering that an entire interval is not usable is the core of the partial enumeration algorithm. In this way, we are able to skip a large number of values that do not generate interesting subcircuits. A trivial example of this is that it's not very interesting to examine a range of values (subcircuits) where there's no connection from input to output.

The subinterval exclusion operation can also be performed using the costfunction. The costfunction $g(x)$ is constructed of two parts, $g_1(x)$ and $g_2(x)$, $g(x) = g_1(x) - g_2(x)$. Each of the parts do not decrease when a component is added to the circuit, hence the lowest possible value of the costfunction in the interval from x to $\tilde{x}-1$ is $g_1(x) - g_2(\tilde{x}-1)$. If this value is not lower than the previously obtained minimum for the costfunction, then the interval can contain no subcircuits with lower costfunction value.

Formulated like this the algorithm to select the next value in the partial enumeration, given the minimum costfunction value so far g_{min} and the current point x, is

```
1  IF $g_{min} < g_1(x) - g_2(\tilde{x} - 1)$ THEN
       $x := \tilde{x}$
   ELSE                                          – Some values may decrease $g$
2      IF no value in $[x, \tilde{x} - 1]$ is usable THEN
           $x := \tilde{x}$
       ELSE                                      – Some values may be usable
3          IF $x$ is not usable THEN
               $x := x + 1$
           ELSE                                  – $x$ may reduce $g$
4              IF $g(x) = g_{min}$ THEN
                   store $x$ as a solution
               ELSIF $g(x) < g_{min}$ THEN
                   $g_{min} := g(x)$
                   remove old solutions, store $x$
               ENDIF
               $x := x + 1$
           ENDIF
       ENDIF
   ENDIF
```

This loop is started with the x value 0 (for which 1 is used for $\tilde{x}$), and repeated until x is greater than $2^{2N} - 1$. The value $\tilde{x}$ can be found from the formula $\tilde{x} - 1 = x \vee (x - 1)$, with $\vee$ being the bitwise OR operator.

Note that the criterion "value/interval is not usable" may be expressed by a criterion similar to the $g_{min} < g_1(x) - g_2(\tilde{x} - 1)$ criterion (this is, indeed, the basis for using Partial Enumeration to solve the problem). The function g is mirrored by a vector of functions, each of which expresses the sign of a coefficient in the transfer function. Each of these functions is formed by the difference between to functions, each of which does not decrease when a component is added to the circuit. The reason for describing the algorithm in terms of "usability" is that this describes the implementation. The costfunction, however, is actually implemented by two separate functions, so the function difference is a natural description of the costfunction considerations.

The parallel computer program is really a parallel implementation of the partial enumeration, since this process takes up the larger part (70 - 99 %) of the runtime of the `StructSynt` program. The initial tree search that finds the transfer function of the IRC is performed on a PC type computer (using a derivative of "the first part" of `StructSynt`) and the results are dumped on a text file. This text file is read by the program on the parallel computer, and the enumeration process is performed in parallel. After the enumeration, the results are written onto another text file, which is processed by a program derived from "the last part" of `StructSynt`, running on a PC.

The parallel partial enumeration [5, p 28 - 29] is performed by noting that the enumeration process is always concerned with a *sub-interval*. Hence, the algorithm may be run on any sub-interval of the entire x range, and the results will still be valid. The parallel version simply defines a series of sub-intervals, by fixing some of the most significant bits, and lets the parallel processors perform the enumeration on these sub-intervals. The results from all subintervals are then collected and sorted. Technically, the fixing of some of the most significant bits simply changes the constraints in the integer optimisation process. It turns out that the parallel version runs very efficiently, hardly losing performance due to the splitting into subproblems (there is only a very small increase in the total number of steps taken in the algorithm).

4 The coefficient matching process

`CompSynt` performs a traditional numerical optimisation process for finding a set of component values that make the actual values of the coefficients equal to the requested. The optimisation method used is a two-stage method, originating in [12] (but simplified to work with non-constrained optimisation). Initially, when we assume that the point is far from the solution point, a Levenberg-Marquardt method is used. Close to the optimum (when the steps are small), a Newton-Raphson method is used instead.

The costfunction used for this process is the sum of the squares of the differences between the requested and actual coefficient values. From `StructSynt` we get a symbolic definition of the coefficients, and hence the computations of first and second order derivatives present no problems. The ease with which the coefficients and their derivatives are computed is one of the main reasons for using coefficient matching. We do acknowledge the fact that for more normal design purposes coefficient matching is less efficient than various frequency-response sampling optimisation processes (as implemented in `LCP2` [8]).

5 Implementation issues

In this section we discuss some of the issues involved in the design of the actual program code. Hence, this section contains nothing of interest from a filter design viewpoint. We do hope, however, to demonstrate that good software engineering is necessary, even for programs that are used for research.

5.1 Execution speed

In [2], with the program `TopSynt`, it was demonstrated that partial enumeration was faster than the heuristic approach in [3]. This was probably caused by a combination of two effects: the increase in speed from using a compiled language (FORTRAN), and a faster algorithm. In writing `StructSynt` we could not improve drastically on the algorithm from `TopSynt`, but we used a more modern language, Modula-2, which brought about a nice improvement in execution speed. The main difference, from an execution time viewpoint, is the use of *sets* to implement bitvectors (used to represent trees). In `TopSynt` bitvectors were implemented by arrays of integers, which was fairly slow. In `StructSynt` we use sets to implement bitvectors, which is fast. The main operation on sets performed in the inner loop of the partial enumeration is to check for the subset property. In the two languages this becomes:

Set g to true if s_2 is a superset of (or identical to) s_1	
Modula-2	FORTRAN

`g := set1 <= set2;`	<pre> G = .TRUE.
	      DO 20 I=1,IDIM
	        IF (IS2(I)-IS1(I)) 10, 20, 20
	10 G = .FALSE.
	        GOTO 30
	20 CONTINUE
	30</pre>

The point is that in Modula-2 we are able to give a high level description of the desired action, whereas in FORTRAN we have to code the action in terms of language primitives. Hence the Modula-2 compiler has a much better change of producing the fastest possible code. (It's possible to insert varying limits for the `DO` loop in the FORTRAN code, making the program a little faster when the number of bits is much lower than the limit, but that is a trivial detail.)

One minor difference between `TopSynt` and `StructSynt` is an added feature from the partial enumeration in `TopSynt` that we did not use for `StructSynt`. In `TopSynt` there is an extra criterion for skipping single x values (as a step 3b), skipping to next x value if the number of edges (bits) is smaller then the number of edges in the trees of the IRC. We did not implement that criterion, since we felt that the partial enumeration could do a sufficiently good job on its own. Measuring the number of iterations taken in each program, it turns out that `StructSynt` takes slightly fewer steps in the enumeration loop, running the Neirynck example, than `TopSynt`.

5.2 Storage requirements

Even though the algorithms used in `TopSynt` and `StructSynt` are identical, there is a minor detail in the way transfer function trees are stored, which means that `StructSynt` only needs to store a factor 2^E trees less than `TopSynt` (with E being the number of passive edges in a tree of the IRC). This means that for a seventh order passive ladder filter `StructSynt` only needs to store 987 trees, rather than 126336! In most practical cases the saving is smaller; but for the parallel version, where each processor was limited to 256 KB of memory, the saving was essential.

This saving is achieved by noting that the IRC passive edges always consist of a parallel combination of two passive components. By definition, these two components will never be present simultaneously in a tree of the IRC. So, if we represent the edges of the IRC as untyped edges (rather than parallel combinations of specific components), and store the trees in terms of these edges, we can create the component based trees later. E.g., two edges in series, y_1 and y_2 (the parallel combination of R_1 and C_1 in series with the parallel combination of R_2 and C_2) generate the trees R_1R_2, R_1C_2, C_1R_2 and C_1C_2; these four trees can be generated from the generic description y_1y_2 with R and C elements substituted in systematically. We perform this saving only once for each edge, so if the IRC contains two parallel passive edges, we get more trees than absolutely necessary, but that's not a real problem.

5.3 Input language

Whereas the PROLOG program in [3] had the transfer function of the IRC built into the program code, `TopSynt` reads a specification of the IRC from a fixed-format text file and found the transfer function internally. For `StructSynt` we decided to go even further, in that we designed a proper input language for the program. This language allows a self-documenting description of all parameters that the user can set; i.e the IRC, the requested transfer function, the optimisation goal and the exact action to perform. As part of the input parser, we built in an extensive syntax error reporting system, so that the user gets a good idea about what to do if the input is not correct.

5.4 The parallel program `ParSynt`

We implemented the parallel partial enumeration algorithm on a Transputer based parallel computer, in the language Occam-2. As noted above the parallel program only performs the enumeration, reading the transfer function from a disk file (together with optimisation goals etc), and writes the result subcircuits to another disk file.

The Occam language does not contain any direct set operations, so the operations on trees etc. have to be coded in terms of bit manipulation primitives (that *do* exist in Occam) operating on arrays of integers (e.g. a 128 element set is code as an 4 element array of 32 bit integers). Since we were concerned with verifying the viability of using a parallel computer, speed was not the main issue anyway. Apart from using arrays of sets rather than linked lists (as in the Modula-2 code), the Occam code is remarkably similar to the Modula-2 code. In fact, the Occam code was written by handtranslating each procedure from the Modula-2 program into Occam, keeping the same structure of nested procedures, variables, etc.

5.5 The program source code

In order to spread the use of Structural Synthesis, our programs are available, with source code, for a handling charge only. Since other programmers could then reuse some of code, the quality of the source code is an important parameter in itself. The programs are written in a strongly modularised fashion, with each main operation being performed by a separate module (a source code file containing a few globally accessible procedures–variables and many local procedures–variables). By breaking the program into a series of disjoint modules we lose some performance. The program size increases because some operations are duplicated, and the speed decreases because of the less tight coupling between various data representations.

Written in this way, the main programs contain a few global variables which are passed as reference parameters to a few procedures. The procedures then become “black boxes” which operate without knowing how their input data was generated or how the output data is used. For example in `StructSynt`, the numbering of the components is different in each module that operate on components as indexes (the tree search, partial enumeration and output modules). By using a separate conversion table in each module, rather than trying to maintain one global mapping from component to number, the modules become more independent.

When the structure of the data and the program is defined according to these ideas, rather than according to some notion of maximum efficiency, the development of the program becomes far easier. When the overall specification of the data structures has been fixed, each module can be written without knowledge of the implementation of other modules. This reduces the amount of information the programmer has to remember at any moment, and hence increases productivity.

6 Results, program performance

Since the algorithm used in `StructSynt` is identical to the algorithm of `TopSynt`, it comes as no surprise that the results from the two programs are identical, with the exception that the order is reversed for the results from `StructSynt`. The results from `ParSynt` are the same as for the two other programs, with the exception that the order is more or less random.

The execution times for the Neirynck example (requesting positive coefficients for numerator and denominator) are given in the table below. Note that the PROLOG program [3] executes a different algorithm, and reaches a different result (finding fewer structures):

Program	Computer	language	time
[3]	VAX 11/750	PROLOG	70 CPU Hours
`TopSynt`	IBM Mainframe	FORTRAN IV	4 CPU Minutes
`StructSynt`	486 PC @25 MHz	Modula-2	26 CPU Seconds
`ParSynt`	16 $\times$ T414 Transputer @16 MHz	Occam-2	5 CPU seconds

Since the transfer function is found as the fully expanded sum of admittance products, i.e. as all spanning trees in the relevant graphs, the size of the transfer function will generally grow exponentially with IRC size. This will, in turn, mean that the execution time and storage requirements will grow exponentially in IRC size. Hence, Structural Synthesis is mostly usable for synthesis of small circuits. The parallel computer version will push the limit for IRC size up somewhat, but not remove the basic problem.

The `CompSynt` program runs reasonably well. Comparing to one of the few literature references [13] on coefficient matching, the algorithm in `CompSynt` takes fewer steps. Comparing, however, to a general optimising program (such as `LCP2` [8]), `CompSynt` performs worse. As the exact transfer function is used, the execution time can be assumed to increase exponentially in circuit size.

Examining the iterations from `CompSynt` closely, it turns out that the two-step method is not really necessary. For all the cases where `CompSynt` does find a solution, the initial *and* final convergence is controlled by the Marquardt method steps, not the Newton method steps.

7 Discussion and further work

Part of the purpose of the work presented here has been to encourage other researchers to look into Structural Synthesis. In order to spread the use of the programs `StructSynt`, `CompSynt` and `ParSynt`, they have been made available at no charge, including source code (covered by the GNU General Public License). In [5] there is a much more thorough discussion of the theoretical aspects, as well as a detailed description of the programs.

Since the programs are easy to use, and freely available, we expect that other researchers will be inspired to work further in the field. Among many other things to investigate, interesting aspects include: The connection between circuit structure and "good" performance; using a block-level description of the IRC, instead of a edge-based description, to work on bigger circuits and the use of various performance measures to assist in the examination of the resulting circuits.

8 Conclusion

We have described a new implementation of the Structural Synthesis procedure defined in [1, 2]. Two details have been added, the parallel computer algorithm and the compact representation of transfer functions, to make the procedure much more usable by cutting storage requirements and increasing speed.

9 Acknowledgements

Professor E.V. Sørensen of the Institute of Computer Science at the Technical University of Denmark was a great help when the work was done. Dr C. Toumazou of Imperial College kindly let Mr Tingleff work on this article (instead of all the relevant work)...

References

[1] P. Kosewski *Topological synthesis of lumped linear networks by integer non-linear programming* Proc. SSCT 74, v II, pp 271-276

[2] J. Derbosz and E. Lindberg *Topological synthesis as a stage of RC active filter design* Proc ECCTD 87 v2 pp 651-656

[3] J. Neirynck, J.C. Mouly and F. Tarpin *RC active biquad design by a prolog program* Electr. Letters vol 21 pp 708-709, Aug 1985

[4] K. Wawryn *PROLOG-based Active Filter Synthesis* Proc ECCTD 89 pp 670, 673

[5] J. Tingleff *Topological Circuit Synthesis on a Parallel Computer* M.Sc. thesis, ISBN 0105-8541, Institute of Circuit Theory and Telecommunication, Technical University of Denmark, sept -89

[6] H. Gaunholt *Users manual for the program ADFP "Amplitude and Delay Fitting Program"* Report IT-19, ISSN 105-8541, Institute of Circuit Theory and Telecommunication, Technical University of Denmark, Sept -77.

[7] R.M. Petersen *Users manual for the ANP3 program* Report IT-43, ISSN 0105-8541 Institute of Circuit Theory and Telecommunication, Technical University of Denmark.

[8] H. Gaunholt *Users Manual for the program LCP2* Report IT-29, ISSN 105-8541, Institute of Circuit Theory and Telecommunication, Technical University of Denmark.

[9] J. Denbosz *Structural Design and optimisation of RC active filters* Report IT-98 ISSN 0105-8541 Institute of Circuit Theory and Telecommunication, Technical University science Vol 15 No 2 Oct -68

[10] J.C.T. Mao and B.A Wallingford *An extension of Lawler and Bell's method of discrete optimisation with examples from capital budgeting* Management Science Vol 15 No 2 Oct -68

[11] W.K. Chen *Applied graph theory, ed 2* North-Holland 1976

[12] K. Madsen *Minimization of Non-linear Approximation Functions* Institute for Numerical Analysis, Technical University of Denmark, 1986, ISBN 87-502-0638-9

[13] R.E. Massara and J.K. Fiedler *Computer Optimisation of frequency selective networks* Proc ECCTD -74 pp 133 - 140