CZECH TECHNICAL UNIVERSITY
Faculty of Electrical Engineering
Department of Electromagnetic Field

Evolutionary Design of Microwave Circuits

Diploma Thesis

Tomáš Müller

Supervisor: Prof. Ing. Zbyněk Škvor, CSc.

May 2003

Acknowledgements

Many thanks to Prof. Ing. Zbyněk Škvor, CSc. for patient and systematic guidance in creation of this diploma thesis.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I claim that I have developed this diploma thesis individually and exclusively with the use of the cited sources. I agree with lending of this work.

Prague, 23rd May, 2003 Tomáš Müller

Abstract

Evolutionary and genetic algorithms are powerful multi-purpose optimization tools that model the principles of evolution. In this thesis an evolutionary (single-member population), local search algorithm for designing microwave circuits is presented and implemented. The input of this algorithm is a number of input/outout ports, available circuit elements and desired characteristics (S-parameters) on a discrete set of frequencies. The output of the algorithm is a circuit (described as a connection list with evaluated circuit elements), which meets the required characteristics. Several practical results are also included in this work. The thesis certainly prooves that evolutionary algorithms are capable to construct circuit topology so that the design goals are met.

Keywords

computer aided design, microwave circuits, evolutionary algorithms, local search, neighbourhood search, simulated annealing, stochastic hill climbing

Table of Contents

0. INTRODUCTION 1
1. EVOLUTION 2
1.1. Genetic Algorithms 2
1.2. Evolutionary (single individual population, local search) algorithms 4
1.2.1. Hill-Climbing 5
1.2.2. Hill-Climbing-Random-Walk 6
1.2.3. Tabu-Search 6
1.2.4. Simulated-Anealing 7
1.2.5. Stochastic Hill-Climbing 8
1.2.6. Local Search 9
1.3. Conclusion 9
2. THE ALGORITHM PROPOSAL 10
2.1. Algorithm Requirements 10
2.2. Circuit Representation 10
2.3. Evaluation of Circuits 12
2.4. Circuit's Neighbourhood 14
2.5. Initial Configuration 17
2.6. Algorithm 17
2.6.1. Operation Selection 18
2.6.2. Acceptance Criterion 19
2.7. Algorithm Improvements 19
2.7.1. Local (Parameter) Optimization 19
2.7.2. Backtracking 20
2.7.3. RMS Error Estimation 22
2.8. Conclusion 25
3. IMPLEMENTATION ISSUES 26
3.1. Circuit Elements 26
3.2. Circuit Representation 27
3.3. Operations 28
3.4. Frequency Requirements 29
3.5. Analysis 31
3.6. Design 32
3.7. Conclusion 32
4. PRACTICAL RESULTS 34
4.1. A Band Pass Filter 34
4.1.1. First Execution 35
4.1.2. Second Execution 36
4.2. An Amplifier 38
4.2.1. First Execution 38
4.2.2. Second Execution 39
4.2.3. Third Execution 40

4.3. A Frequency Splitter 42
4.4. Conclusion 45
5. CONCLUSION 46
6. REFERENCES 47
APPENDIX A USER DOCUMENTATION ..... A - 1
A.1. Prerequisites ..... A - 1
A.2. Installation ..... A - 1
A.3. Usage ..... A - 1
A.4. Input File ..... A - 1
A.4.1. Section CONFIGURATION ..... A - 1
A.4.2. Section LINKS ..... A - 4
A.4.3. Section NODES ..... A - 4
A.4.4. Section OPERATIONS ..... A - 4
A.4.5. Section OPT ..... A - 6
A.5. Output Files ..... A - 7
APPENDIX B PROGRAM EXTENSION EXAMPLES ..... B - 1
B.1. New Circuit Element Implementation ..... B - 1
B.2. New Circuit Operation Implementation ..... B - 3
APPENDIX C CD-ROM CONTENT ..... C - 1

0. Introduction

Evolutionary and genetic algorithms are powerful multi-purpose optimization tools that model the principles of evolution. They are often capable of finding globally optimal solution even in the most complex search spaces.

The mission of this thesis is to propose and implement an evolutionary (single-member population) algorithm for designing microwave circuits. The main goal is to present a possibility of usage of evolutionary algorithms in computer aided design of microwave circuits, not to produce some universal designing product.

The computer aided design of microwave circuits represents tools for design, evaluation and optimization of microwave circuits. Nowadays, optimization in the microwave circuit design means finding circuit element values for a fixed circuit topology, where the circuit topology can be e.g. an input parameter. A different approach is presented in this work: the design of a microwave circuit is left completely to the evolutionary (design and optimization) algorithm. Input parameters include design goals (e.g. required frequency response), number of circuit ports and a set of available circuit elements. The evolutionary algorithm then sets up different circuit topologies, and the topology that satisfies goals is an output parameter, together with circuit element values.

The thesis is organized in several chapters. In the first chapter, the general principles and methods of evolutionary algorithms are described, with the main emphasis on their usage in the circuit design. In the second chapter the algorithm for the design of microwave circuits is proposed. The implementation of the proposed algorithm in Java is outlined in the following third chapter. The fourth chapter contains some practical results achieved with the implemented algorithm on some basic assignments. There are also several appendixes. A user documentation of the implemented program is included in appendix A. In appendix B, some examples of extensions of the implemented algorithm are shown. Finally, appendix C contains an organization structure of the CD-ROM, which is included in the thesis.

1. Evolution

Evolutionary (genetic) algorithms are powerful general purpose optimization tools which model the principles of evolution [1, 11, 12]. They are often capable of finding globally optimal solution even in the most complex search spaces. They operate on a population of members which are selected according to their quality and then used as the basis for a new generation of members found by combining (crossover) and/or altering (mutating) current individuals.

1.1. Genetic Algorithms

A *genetic algorithm* [11, 12] starts by generating an initial population (a set of initial solution candidates, members). Basically, each member is represented by a set of genes, encoded as a bit string with the fixed length.

The algorithm works in iterations, evolutions of a population. In each evolution step, a new population is produced from the previous one. There are several methods how to produce a new population. Most of them use three different operators:

- selection of two parents (e.g. according to some evaluation of parents)
- crossover of parents (e.g. combining of their genes)
- mutation of the arose individual (e.g. by swapping several genes)

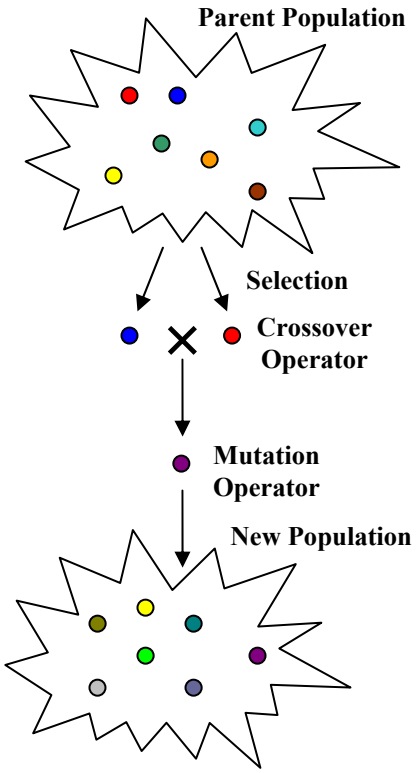


Figure 1.1. An evolution step

An example of a new population creation is presented in Figure 1.1. It starts with an empty generation. Until this new generation has the demanded number of members: Two members are selected (via selection operator) from the parent population and combined (via crossover operator). The resultant individual is then altered (via mutation operator) and placed into the new population.

The algorithm stops when there is a suitable solution found. It means that there is a member in the new population which fits the input requirements. Otherwise, the new population becomes the parent population for a new evolution step.

Another version of the genetic algorithm can for example work with the crossover of two arbitrary (randomly selected) members from the parent population, where only those created individuals that are good enough are placed into the new population. The selection is moved after crossover and mutation operator in this example.

Typically, the number of members in a population is constant during the search. However, there are also approaches in which the number of population members differs during the search (e.g. according to their quality) [8].

The selection of the two members from the parent population can be based on the possibility of evaluation of these candidates. Whether it is possible to evaluate each member, which means that there is a mapping function of each member to a number according to its quality. Or whether we can only compare two members and say which of them is better. In the first case, the probability of a selection of a candidate can be guided according to its evaluation. For example:

$$P_a = eval(a) \Big/ \sum_{s \in M} eval(s)$$

Figure 1.2. Probability of acceptance according to the evaluation

where a is a member from population M and *eval* is the evaluation function (a better solution corresponds to a higher number). This approach is called *roulette wheel selection*, see [8] for details. In the second case, where we can only mutually compare two members, the *tournament selection* can be used [8]. In such case, the selection is made in several rounds. In the first round, two members are selected randomly (with uniform probability) and the better one is used for the second round. In each next round, one member is taken randomly and compared to the previous winner (previous better solution). After several rounds, the winner is returned as the selected element. Another possibility of organizing the tournament is based on a binary tree, where in each round, the number of candidates (from the initial randomly selected subset of the population) is divided by two.

Often, the crossover operator is the most crucial step in the design of a genetic algorithm. Traditionally, the search mechanism has been domain independent. That means the crossover and mutation operators have no knowledge of the problem and what a good solution would be. They are working with members represented as binary strings, combining and mutating their genes. For example, the crossover operator can be designed as splitting of the genes into the two parts (e.g. genes at odd or even position in the bit string) and combination these parts from the respective parents (e.g. odd bits from the first parent, even bits from the second, as shown in Figure 1.3). The mutation operator can be than designed as a switch of randomly selected fixed number of genes (e.g. three genes are swapped in Figure 1.4).

```
10011110111011101
01011110100111011
----------------
1101111101100 1001
```

Figure 1.3. Crossover operator example

```
1101111101100 1101
----------------
1101011101000 11101
```

Figure 1.4. Mutation operator example

The disadvantage of this approach is that the given solution (e.g. circuit represented by the bit string) is often much different from both parent solutions. In many cases [8, 12], it was empirically shown that the use of domain dependent operators is much more efficient in practice. Such operators are trying to obtain a solution which is better than both its parent solutions, e.g. a solution which takes the best "features" from both parents. In some literature [2], such algorithms are called memetic algorithms, because they are working with bigger entities than genes (called memes).

1.2. Evolutionary (single individual population, local search) algorithms

Because of the computational demands and high complicatedness of design of crossover operator for the microwave circuits (members of a population), we will focus on (in some sense extreme) case of the genetic algorithm, where the population has always cardinality equal to one. In each evolution step, the algorithm takes the previous (parent) solution and by its mutation it creates other (children) solutions from it. Finally, one of the generated solutions is picked up to be the parent for the next iteration step.

There are naming differences: Such algorithms are sometimes called evolutionary (to distinguish them from genetic algorithms, which work with bigger population). But sometimes, the expression evolutionary algorithm is used for the class of all algorithms, which mimics the process of evolution (genetic algorithms belong to this class of evolutionary algorithms).

This single-member population approach can also be considered as a variation of the local search algorithms. These algorithms also work in iterations, with one infeasible solution. In each iteration step, neighbour solutions from the previous solution are generated and evaluated. At the end of the iteration, one neighbour solution is selected into the next iteration. The purpose of these algorithms is simple: They are repeatedly trying either to decrease the number of inconsistencies in the solution and/or to get closer and closer to some optimal one.

Recently, there have been developed many kinds of the local (neighbour) search algorithms (e.g. hill climbing, steepest descant, min-conflict, genet) and their improvements (e.g. random walk, tabu-list, simulated annealing). We will discuss some of these algorithms in the following subsections.

Genetic algorithms with a single member population and local search algorithms work in iterations, always with one configuration (or state, solution, solution candidate). In the design of microwave circuits, such configuration will be a microwave circuit (both with topology and values of all circuit elements). Each such solution should have an evaluation value and a set of neighbour solutions defined. For us, value of a circuit can express its difference from the required, searching solution. The neighbour solutions can be defined as a circuits derived from the previous one by applying a mutation operator. Such mutation can be for example a change of a single circuit element, or some circuit value or a simple change of circuit topology.

1.2.1. Hill-Climbing

Hill-climbing (or *steepest-descendant*) is probably the most famous algorithm of local search [7, 8]. The idea of hill-climbing is following:

1. start at randomly generated configuration
2. move to the neighbour with the best evaluation value
3. if a strict local-minimum is reached (there is no better neighbour) then restart at another randomly generated configuration
4. if sufficient (required) configuration is found then return solution
5. otherwise repeat to step 2

The hill-climbing procedure repeats local steps to neighbourhood till the solution is found or the time exceeds the limit for the computation. Visibly, the algorithm does not guarantee to find the best solution but it can return any "solution" at each time (the longer time, the better solution). Therefore, this algorithm (and all other algorithms in this section) belongs to the group of *anytime algorithms* that are able to return some "reasonable" solution at each time. The name of the algorithm, hill-climbing, is derived from its original principle when a maximum was searched by climbing – increasing the evaluation value (note, that are descending now because of looking for the minimum).

In the algorithm presented in Figure 1.5, the parameter *max_flips* is used to limit the maximal number of moves between restarts which helps to leave non-strict local minimum. The resultant solution has to have the evaluated value below the *limit*.

```
procedure HC(max_moves, limit)
    restart: s <- random initial configuration;
    for j:=1 to max_moves do
        if eval(s)<=limit then return s;
        if s is a strict local minimum then
            go to restart;
        else
            s <- neighbour with smallest evaluation value;
        end if
    end for
    go to restart;
end HC
```

Figure 1.5. Hill Climbing algorithm

Remark that the hill-climbing algorithm has to explore all neighbours of the current state before choosing the move. In general, this can take a lot of time. To avoid exploring all neighbours of the current configuration some alternatives were proposed to find the next move.

For example, randomized hill-climbing are repeatedly choosing the neighbour randomly, until a better neighbour than the current solution is found. This neighbour is choosen for the following iteration.

Another approach is to select neighbours acording to some heuristics, some ordering of neighbours according to their quality. Again, until the first better neighbour than the current state is found.

1.2.2. Hill-Climbing-Random-Walk

Because the pure hill-climbing algorithm cannot go beyond a local-minimum, some noise strategies were introduced here. Among them, the *random-walk* strategy becomes one of the most popular [6]. For a given configuration, the random-walk strategy picks randomly a neighbour with probability *p*, and applies the hill-climbing strategy (e.g. selection of the best neighbour) with probability *1-p*. See Figure 1.6.

```
procedure HCRW(max_moves, limit, p)
    s <- random initial configuration;
    nb_moves <- 0;
    while eval(s)>limit & nb_moves<max_moves do
        if probability p verified then
            s <- randomly chosen neighbour;
        else
            s <- neighbour with smallest evaluation value;
        end if
        nb_moves <- nb_moves + 1;
    end while
    return s;
end HCRW
```

Figure 1.6. Hill Climbing Random Walk algorithm

This algorithm is controlled by the random probability *p*. It should be clear that the value of this parameter has a big influence on the performance of the algorithm. The preliminary studies determined the following feasible ranges of parameter values $0.02 <= p <= 0.1$.

1.2.3. Tabu-Search

Tabu search [3, 8] is another method to avoid cycling and getting trapped in local minimum. It is based on the notion of *tabu list*, which is a special short term memory that maintains a selective history, composed of previously encountered configurations or more generally pertinent attributes of such configurations (the change of the configuration, e.g. a pair <circuit_element, circuit_value> when there is only moves changing a value of an arbitrary circuit element – no topology changes). A simple TS strategy consist in preventing configurations of tabu list from being recognised for the next *k* iterations (*k*, called tabu tenure, is the size of tabu list). Such a strategy prevents Tabu from being trapped in short term cycling and allows the search process to go beyond local optima.

Tabu restrictions may be overridden under certain conditions, called *aspiration criteria*. Aspiration criteria define rules that govern whether next configuration is considered as a possible move even it is tabu. One widely used aspiration criterion consists of removing a tabu classification from a move when the move leads to a solution better than that obtained so far. See Figure 1.7.

Again, the performance of Tabu Search is greatly influenced by the size of tabu list *tl*. Preliminary studies determined the following feasible range of parameter values $10 <= tl <= 35$.

```
procedure TS(max_moves, limit)
    s <- random initial configuration;
    nb_moves <- 0;
    initialise randomly the tabu list;
    while eval(s)>limit & nb_moves<max_moves do
       choose a move, e.g. <element, value> with the best
          performance among the non-tabu moves and the moves
          satisfying the aspiration criteria;
       introduce the move, e.g. <element, value> in the tabu list;
       remove the oldest move from the tabu list;
       apply the move to the s;
       nb_moves <- nb_moves+1;
    end while
    return s;
end TS
```

Figure 1.7. Tabu Search algorithm

1.2.4. Simulated-Aneling

Another meta-heuristic method allowing escape from a local minimum is called *simulated annealing* [5, 8]. The algorithm is derived from the physical process of cooling of a metal in a thermal bath. Quality of the output material is reached the way the material is alternately heated and cooled.

Similarly to the algorithms described above, this algorithm gradually goes from one configuration to another by repeatedly selecting one of the neighbour configuration randomly. If the solution has lower evaluation value, it is automatically chosen to the next iteration. But, if the randomly selected neighbour has higher evaluation value, it also has a chance to be chosen, according to the following probability

$$p = e^{eval(s)-eval(s')/T}$$

Figure 1.8. Probability of acceptance for simulated annealing

where s is the previous solution, s' is the randomly selected neighbour solution and T denotes the actual temperature. The algorithm is gradually changing this temperature according to some cooling schedule, which is a part of the algorithm's configuration. While the temperature is decreasing, the probability of choosing the worst neighbour is decreasing as well. So, the whole process is based on the selection of the cooling schedule. This schedule can be also chosen for a particular problem during the execution, e.g. to avoid premature cool down. The algorithm is shown in Figure 1.9.

```
procedure SA(max_moves, limit)
    s <- random initial configuration;
    for j:=1 to max_moves do
        if eval(s)<=limit then return s;
        s‘ <- random neighbour of s;
        if (eval(s')>eval(s)) or
           (random(0,1)<exp[(eval(s)-eval(s'))/T]) then s=s';
        T = g(T,j);
    end for
    return s;
end SA
```

Figure 1.9. Simulated Annealing

1.2.5. Stochastic Hill-Climbing

An algorithm similar to the simulated annealing is the algorithm called *stochastic hill-climbing* (sometimes also called simulated-annealing) [8]. The main difference is that every randomly selected neighbour (even better) is selected according to some probability. The traditional probability of acceptance of a neighbour solution is

$$p = \frac{1}{1 + e^{eval(s') - eval(s) / T}}$$

Figure 1.10. Probability of acceptance for stochastic hill climbing

where again s is the previous (parent) solution, s' is the randomly selected neighbour and T is the current temperature. The function *eval* denotes the solution's evaluated value. See the following Figure 1.11.

```
procedure SHC(max_moves, limit)
    s <- random initial configuration;
    for j:=1 to max_moves do
        if eval(s)<=limit then return s;
        s' <- random neighbour of s;
        if (random(0,1)<1/(1+exp[(eval(s')-eval(s))/T]))
            then s=s';
        T = g(T,j);
    end for
    return s;
end SHC
```

Figure 1.11. Stochastic Hill-Climbing

The following example table (Figure 1.12.) shows the probability of acceptance for the better neighbour solution (the difference of evaluated values is equal to -13) according to the chosen temperature T. At the beginning, when the temperature is high, the probability of acceptance goes to one half. During the search, while the temperature is decreased, the probability of acceptance of a better neighbour solution goes near to one.

T	$e^{-13/T}$	***P***
1	2e-7	1.00
5	0.0743	0.93
10	0.2725	0.78
20	0.52	0.66
50	0.77	0.56
10e10	0.9999	0.500..

Figure 1.12. Probability of acceptance of a solution for eval(s')-eval(s) equals to -13 according to temperature T

The limit case is when the temperature limitary goes to zero. The better neighbour solution is always accepted (probability goes to one) and the worse neighbour solution is always declined (probability goes to zero).

1.2.6. Local Search

All above algorithms are based on the common idea known under the notion *local search* [8]. In local search, an initial configuration is generated and the algorithm moves from the current configuration to neighbourhood configurations until a solution (decision problems) or a good solution (optimization problems) is found or the resources available are exhausted. This idea is expressed in the following general local search algorithm that enables implementation of many particular local search algorithms via definitions of specific procedures. See Figure 1.13.

```
procedure LS(max_tries, max_moves, limit)
    s <- random initial configuration;
    for i:=1 to max_tries while Gcondition do
        for j:=1 to max_moves while Lcondition do
            if eval(s)<=limit then return s;
            select n in neighbourhood(s);
            if acceptable(n) then s<-n;
        end for
        s <- restartState(s);
    end for
    return s;
end LS
```

Figure 1.13. Local Search algorithm scheme

1.3. Conclusion

In this chapter, various basic approaches which mimic an evolution of a single individual have been described. In the following chapter, we will propose an evolutionary algorithm for design of microwave circuits on the basis of this knowledge.

Remark that not all of the above presented methods are suitable for an evolution of a single circuit. For example, it is either impossible or at least not effective to evaluate all the neighbours of a circuit, because of the non-discrete character of circuit elements' values. For the same reason (large or infinity cardinality of the circuit's neighbourhood), the tabu-list technique will be also inefficient – a local optimum can also have excessively large neighbourhood.

2. The Algorithm Proposal

In this chapter, we will at first summarize the requirements for the evolutionary design algorithm. Next, we will discuss the representation of a circuit, which will be our individual configuration during the evolutionary search. Then, we will propose an evaluation of circuits and we will define the circuit's neighbourhood. Finally, we will describe an evolutionary algorithm handling our requirements and propose several improvements to it.

2.1. Algorithm Requirements

As the input, we have the requirements for the described microwave circuit, namely:

- the number of circuit ports,
- design goals, e.g. frequency response or in more general any requirements derivable for the S-parameters over the given finite set of frequencies,
- a set of available circuit elements

The goal of the algorithm is to find a circuit which will meet the above mentioned requirements. The output of the algorithm should be a structure of the resultant circuit (topology) and values of all used circuit elements. We will discuss some other requirements (e.g. planarity of the resultant circuit) later.

As prerequisites, we can assume that we have a (3^{rd} party) tool which can provide us an analysis of the given circuit. In the algorithm implementation, the program MIDE [9, 10] is used (the possibility of using program MIDE was a part of the assignment of this work). This program can calculate the S-parameters for the given circuit on the given frequency or on a finite set of frequencies. But this tool also makes some demands on the algorithm, namely:

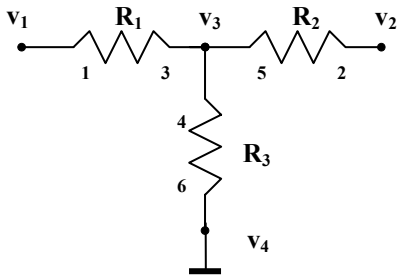
- We has to use circuit elements compatible with (known by) the analysis tool.
- Our circuit representation has to be the same as or easily transformable to the representation understandable by the analysis tool (this structure is called *net-list* in MIDE, see [9, 10] for details).
- The analysis tool has to provide us some API, so that we can repeatedly use it (with no user interaction required).
- There can be some limitation of the size of the circuit demanded by the analysis tool.

2.2. Circuit Representation

There are several demands on the representation of a microwave circuit, which we will use. At first, the circuit representation has to be able to describe an

arbitrary circuit (its topology, circuit elements and values of their parameters) we can obtain during the design process. Next, it has to be able to define a set of neighbour circuits to the given circuit representation and also enumerate representations from this neighbourhood. And finally, the representation has to be convertible to the representation required by the analysis tool.

Therefore, the circuit is represented as a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, with vertices $\mathbf{V} = \{v_1, v_2, \cdots v_n\}$ and edges $\mathbf{E} = \{e_1, e_2, \cdots e_m\} \subseteq \mathbf{V}^2$ between these vertices. Every edge represents an element (resistor, capacitor, inductor, micro-stripe line, …), every vertex represents an input/output gate, a connection between one to four elements (ideal or non-ideal) or an element with one or more than two ports (e.g. short end, open end, bounded wiring, …). The conversion between this representation and the net-list used by program MIDE is straightforward. An example is shown in the following figures:



Vertices $\mathbf{V} = \{v_1, v_2, v_3, v_4\}$:
- v_1 … input/output port
- v_2 … input/output port
- v_3 … 3-port ideal connection
- v_4 … short end

Edges $\mathbf{E} = \{(v_1, v_3), (v_3, v_2), (v_3, v_4)\}$:
resistors $\mathbf{R}_1, \mathbf{R}_2, \mathbf{R}_3$ respectively

Figure 2.1. Circuit representation

```
BLO circuit 1 2
  RES 1 3 R=R1          edge (v1,v3) .. resistor R1
  ITEE 3 4 5            vertex v3
  RES 4 6 R=R3          edge (v3,v4) .. resistor R3
  SHORT 6               vertex v4
  RES 5 2 R=R2          edge (v3,v2) .. resistor R2
END
```

Figure 2.2. Respective net-list for the circuit from Figure 2.1

To enable a direct connection of two elements represented by vertices (e.g. Figure 2.1. without resistor $\mathbf{R}_3$ - short end connected directly to vertex v_3), it is not strictly required that an edge has a circuit element assigned. In the implementation, such edge has assigned a special element called *zero element*. Such edges are not included in the resultant net-list structure. Zero elements are also used for vertices with ideal connection of two edges.

The above described representation of a circuit gives us a possibility to represent an arbitrary circuit and it is easily transformable to the net-list representation (implemented by simple listing of all edges and vertices, where each edge has two unique numbers assigned, one for each end). The representation of a circuit as a graph gives us also a huge possibility to define

several kinds of its neighbours, e.g. by simple graph operations, for example by adding an edge or splitting an edge into two edges connected with a new vertex.

Remark that according to the net-list structure, input and output ports can be connected exactly to one circuit element and each port (denoted by its number) can be shared by exactly two different circuit elements – it has to occur in the net-list exactly twice. There are special elements for connection of more than two ports (e.g. ITEE for ideal connection of three ports – see Figure 2.1 and 2.2). This behaviour has to be propagated into the representation, namely:

- Input/output vertices can have only one edge.
- Each inner vertex can contain at least one and at most four edges (because there is no connection element for more than four ports in MIDE).
- Graph has to be connected. It means that there has to be a path between each two different vertices in the graph.
- Each vertex and edge has to have an appropriate circuit element associated. And such element has to have the correct number of ports (two for the edge, number of edges connected a to vertex for the vertex). There are two special elements: An element denoting that the vertex is an input/output port (appropriate number of such elements has to be used) and a zero element as described above.
- Each circuit element has to have appropriate parameters assigned: In dependence of the type of an element, it can contain zero, one or more parameters with assigned values of respective type and from respective bounded domain (interval). For example, a resistor contains one parameter, its resistance (measured in ohms) from the interval of values `<0.001;155555>`. There is also an implicit value defined, e.g. for the resistor it is 50 ohms.

2.3. Evaluation of Circuits

As it is mentioned above, the external analysis tool MIDE is used for the evaluation of a circuit on the given frequency or on the given finite set of frequencies. Such evaluation gives us a square matrix of S-parameters for each frequency, which represents S-parameters between each pair of input/output ports. From such evaluation we would like to compute one real number denoting the quality (or closeness) of the circuit to the input requirements. The resultant function is called *objective function*.

At first, the set of frequencies which will be used in analysis can be derived from the requirements. Remark that our requirements are defined as requirements derivable from S-parameters (e.g. frequency response) on the given finite set of frequencies. For example, we can require the transmission of a filter between 1GHz and 3GHz step 100MHz (it means on frequency 1GHz, 1.1GHz, 1.2GHz, … 3GHz) to be below -25dB.

So, we have requirements on the given frequencies and computed characteristics for the given circuit. Then we can compute the root mean square (RMS) error from differences between these results and requirements, as follows:

$$\sqrt{\frac{1}{|\mathbf{F}|}\sum_{f\in\mathbf{F}}\left(\frac{1}{|\mathbf{R}(f)|}\sum_{R\in\mathbf{R}(f)}\left(diff_R\left(S_f\right)\right)^2\right)}$$

Figure 2.3. Definition of RMS error

where $\mathbf{F}=\{f_1,f_2,\cdots f_n\}$ is a finite set of given frequencies, $\mathbf{R}(f)$ denotes the set of requirements on the given frequency, S_f are the computed S-parameters for the frequency f and $diff_R\left(S_f\right)$ denotes the difference between the computed S-parameters S_f and the requirements R, so that the difference is zero, if the requirement is met. For example, in case of a requirement expressed as an inequality, e.g. transmission to be below -25dB, the difference is zero if the transmission is bellow the given limit. Otherwise it expresses the difference of required and measured transmission in decibels.

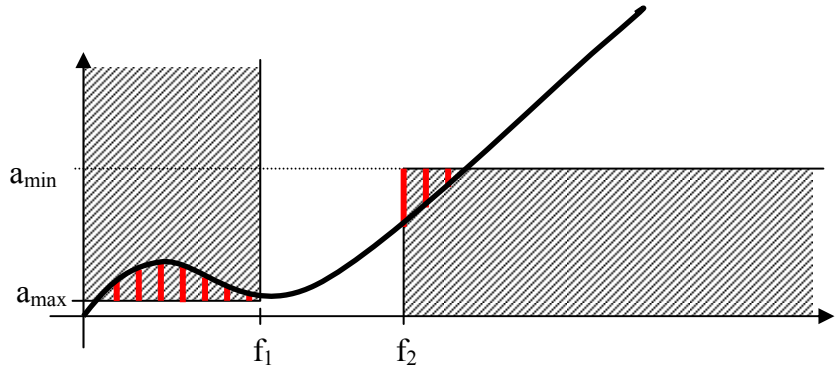


Figure. 2.4. RMS error for a low-pass filter:
only differences in shaded areas are considered

An example of evaluation is indicated in the above Figure 2.4 for a low-pass filter. The requirements are denoted by the shaded rectangles, the transmission of the given circuit is denoted by the thick black line. The resultant RMS error is computed as the root mean square of the differences, marked as vertical red lines below respective above the computed characteristics of the given filter.

As follows from the above description, we want to minimize this RMS error. The wanted circuit is found when this objective function is equal to zero.

Remark that the requirements (and the differences) can be also weighted. It gives us for example a possibility to put some stress to some key frequencies (e.g. f_1 and f_2 in Figure 2.3). This can help the algorithm to reach the optimum faster. We can also set a limit for the RMS error, which says that we do not require an optimal solution (with all requirements met - error equal to zero) but a solution with the RMS error below this limit. For example, a circuit with error below 1 dB can be sufficient for us. Such solution can be much easier to find than the optimal one (with RMS error equal to zero).

2.4. Circuit's Neighbourhood

As described in the previous chapter, in each evolution step, a solution (or configuration) is derived somehow from the previous solution. This can be also seen as the selection of an element from the solution's neighbour solutions. In this paragraph we will discuss a neighbour circuit creation from the previous circuit by applying one of the "circuit" operation. Most of these operations are deducted from the graph representation of a circuit described above.

Let's define a circuit neighbourhood as a set of circuits created by applying of one of the below listed operations. An operation is applied only if the resultant circuit is consistent – it is connected, there is no vertex with more than four edges and every vertex has an appropriate circuit element associated (with the correct number of ports).

The operations are:

O1. **A parameter of some circuit element is changed.** (e.g. resistor's resistance is changed).

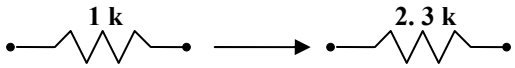


Figure 2.5. A parameter of some element is changed

O2. **A circuit element associated to a vertex or to an edge is replaced by another suitable one** (with the same number of ports, e.g. a resistor is changed to a capacitor).

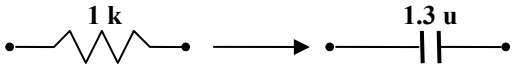


Figure 2.6. A circuit element is replaced by another one

O3. **An edge is added between two suitable vertices** (with the number of connected edges below four, not i/o vertices). Elements of respective vertices are changed – the number of ports is increased by one. An arbitrary suitable circuit element (2-port) is associated with the created edge.

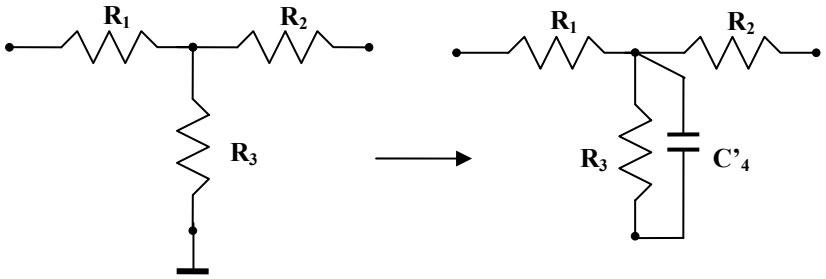


Figure 2.7. An edge is added between two vertices

O4. **An edge is removed.** Elements of appropriate vertices are changed – the number of ports is decreased by one. The graph has to remain connected.

Figure 2.8. An edge is removed

O5. **A vertex** (with two or more edges) **is split up into two vertices and an edge is added between them**. Suitable circuit elements are assigned to the created vertices and the created edge.

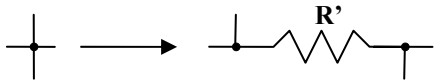


Figure 2.9. A vertex is split up to two vertices

O6. **A single-edge vertex is created and connected with an edge to a suitable vertex** (with less then four connections, not i/o vertex). Suitable elements are assigned to the changed vertex and to the created edge and vertex.

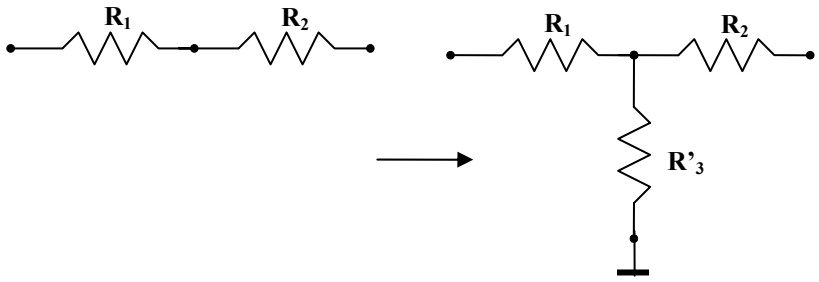


Figure 2.10. A vertex is created and connected to a suitable vertex

O7. **An arbitrary edge is split into two edges and a vertex is added between them.** Suitable elements are assigned to the created edge and vertex.

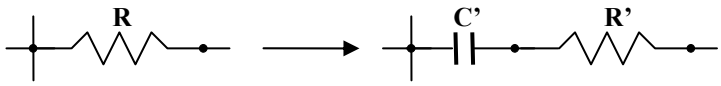


Figure 2.11. An edge is split into two edges with an edge between

O8. **A vertex with one edge** (not i/o vertex) **is removed. The appropriate edge is also removed.** The element on the other side of the removed edge is changed.

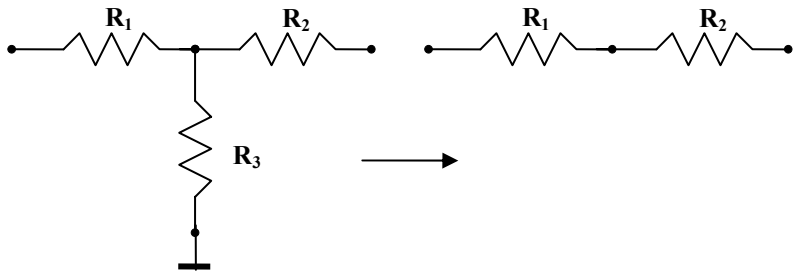


Figure 2.12. A single-edge vertex is removed

O9. **A vertex with two edges is removed. The appropriate edges are joined into the one edge.** A suitable element is assigned to the created edge (e.g. an element from one of the removed edges is used).

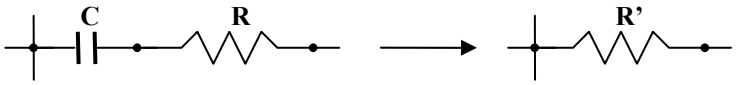


Figure 2.13. A vertex with two edges is removed.

O10. **Two suitable vertices connected with an edge are joined into one vertex.** The edge between the previous vertices is removed. Suitable elements are assigned to the created edge and vertex.

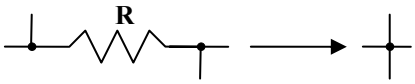


Figure 2.14. Two connected vertices are joined

There are also many other possibilities what operations to use, for example the above operations can be adopted not to violate planarity of the circuit (e.g. operation O3 can break the planarity if it remains defined as above). There can also be other circuit limitations introduced, e.g. for the maximum number of elements, elements of given type or physical feasibility of the resultant circuit.

Each circuit element which can be used has a defined set of its parameters with minimal and maximal bounds and implicit values. When an edge or a vertex is created or changed, an appropriate element is selected randomly (or according to some heuristics). The element's parameters are selected from the log-normal distribution with the mean value equal to the parameter's implicit value. When a parameter is changed (the operation O1 in the above list), a new value is selected again from the log-normal distribution, but with mean value equal to the parameter's previous value.

Remark that there are also operations for simplifying the graph (e.g. by removing an edge or a vertex). It gives us a possibility to return (e.g. from some local-minima) in local search algorithm.

Also remark that the neighbour of a circuit can be extremely large even if the parameters of all circuit elements will have finite domains (obtained by fixed

accuracy of the parameters). Only for adding an edge we have approximately m^2 possibilities how to select the vertices (m is number of vertices), exactly:

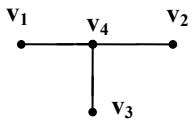
$$|\mathbf{V}'| \cdot (|\mathbf{V}'| - 1)$$

Figure 2.15. Number of possibilities to add an edge

where $\mathbf{V}'$ is the set of the suitable vertices (all vertices except the vertices with assigned i/o element and vertices with four edges connected). Next, for such selection of suitable vertices, we have also many possibilities what element to select to the created edge and what values to assign to the parameters of the selected circuit element.

2.5. Initial Configuration

Till now, we have defined input requirements, a representation of an arbitrary microwave circuit and its evaluation. We have also described circuit's neighbour solutions. But for the design of an algorithm, there is still one thing missing. We have to define the initial configuration, the circuit which we will start from. This task is rather easy, we can for example create a graph with directly connected input/output vertices. Some inner vertices can occur, as shown in Figure 2.16 for 3 input/output ports.



Input/output vertices: $\mathbf{V}_{io} = \{v_1, v_2, v_3\}$

```
BLO CIRCUIT 1 2 3
  ITEE 1 2 3
END
```

Figure 2.16. Initial 3-port circuit and the appropriate net-list

2.6. Algorithm

Now, we are ready to propose an algorithm for the evolutionary design of microwave circuits. This algorithm works with single member population and it is derived from the local search technique. An initial proposal of such algorithm is shown in the below Figure 2.17.

```
procedure EDMC-LS(max_iters)
    s <- initial circuit; //initial solution
    err <- eval(s);       //RMS error of the initial solution
    i <- 0;               //iterations counter
    while i<max_iters end err>0 do
        op <- select_operation();     //operation selection
        s' <- apply_operation(s,op);  //operation application
        err' <- eval(s');             //new RMS error
        if acceptable(err,err') then  //accept change
            s <- s';
            err <- err';
        end;
        i <- i + 1;                   //increase iteration counter
    end while
    return s;
end EDMC-LS
```

Figure 2.17. Local search algorithm

The algorithm works in iterations. It starts from the initial solution, which is a simple circuit with only input/output ports connected (as described in section 2.5. and Figure 2.16).

In each iteration step, an operation is selected somehow and it is applied to the current solution. If the altered solution is acceptable (e.g. it is better then the current one), it becomes the current solution for the next iteration. Otherwise, it is refused.

The iteration process stops when the requested solution (with RMS error equal to zero, or less then the given limit) or the maximum number of iterations is reached. In case of the reach of the maximum number of iterations, the algorithm can

- **return fail** – requested solution was not found
- **return the best solution ever found** – the solution with the lowest evaluation (RMS error). Remark, that in case of random walk, simulated annealing or stochastic hill climbing, the best solution need not be the current solution from the last iteration. The algorithm can sometimes accept worse solution to the next iteration.
- **restart the search from the beginning** (or maybe from some previous memorized solution). This makes sense only when the above described algorithm is somehow randomized: The selection of an operation (and its operands – edges, vertices, circuit elements, their parameters) and/or solution acceptance is not deterministic.

The above described algorithm is parameterized by two functions: The first one is the selection of an operation to be applied on a current circuit solution. The second one is the acceptance criterion for an altered circuit.

2.6.1. Operation Selection

For the operation selection criterion, we propose the operations to be selected randomly, but with respect to their weights. This gives us a possibility to determine the frequency of use of each operation. So, for example, the modification of a single parameter or circuit element can be more frequent than an insertion of an edge. This way, we can also drive the speed of magnification of the circuit's complexity. These weights can also be changed during the search, e.g. according to the current solution, its complexity and evaluation or according to some statistics of successfulness of the previous operation and operands selections. After an operation is selected, there has to be also its operands selected, namely:

- appropriate vertices and/or edges, required for the operation (e.g. a suitable pair of vertices for adding an edge)
- appropriate circuit elements for new or altered edges and vertices (e.g. an element for a new edge and for the two altered vertices when there an edge should be added)
- parameters of the new or altered circuit elements

These operands can be selected also randomly, uniformly or with respecting some priorities from the set of suitable vertices, edges and circuit elements.

We already proposed (in the section 2.4) to select parameters of a circuit element from the log-normal distribution respecting the parameter bounds and its previous (in case of altering the parameter, see operation O1) or its implicit value (in case of new circuit element). The distribution will have the mean value equal to the previous or implicit value. When a value outside the bounds is selected, the appropriate maximum or minimum bound value is returned. The reason of choosing the log-normal distribution is based on the idea of selecting a value not too different from the previous or implicit value. For example, when there is a resistor with implicit resistance 50 ohms, we want the most of the new resistance values to be selected from the interval <0.5,5000> ohms – plus/minus two orders.

2.6.2. Acceptance Criterion

There were several types of acceptance criteria described in the first chapter of this thesis. We can accept **only better circuits**, introduce **random walk** technology (besides accepting only better circuits, we accept an arbitrary solution with some small probability), **simulated annealing** (probability of acceptance of a worse solution depends on the difference, how much worse the solution is) and **stochastic hill climbing** (probability of acceptance of any solution depends on its quality according to the previous one). We propose to use either simulated annealing or stochastic hill climbing in the implementation because we need to escape a local minimum, e.g. when there is a long sequence of worse neighbour solutions generated – to try select one of them. But we do not want to select a too bad neigbour either.

We use only a simple schedule of the temperature in the simulated annealing techniques: the temperature is linearly decreasing from an initial temperature to some minimal (almost zero) temperature. After this minimum is reached, the temperature remains constant for the rest of the search.

2.7. Algorithm Improvements

Because the above described algorithm was unable to find a solution in reasonable time for many even simple toy instances, in the following paragraphs we will propose several improvements, developed for the better efficiency and stability of the algorithm.

2.7.1. Local (Parameter) Optimization

Because the selection of an operation and its operands can be rather expensive and because it will probably fail many times only just because the wrong parameters of the created or altered circuit elements are selected, we propose to try to change these parameters several times before the operation is totally rejected.

For example, when there is a need to decrease a single parameter of one element: Once this parameter (and appropriate operation which do this) is selected, we can improve the efficiency if the change of the parameter is tried several times. It can be also sometimes accepted during this, e.g. when the change

of the parameter goes to the right direction (circuit's evaluation value decreases). This can be seen as a simple optimization of the selection of the new or altered elements' parameters. Improved algorithm is shown in Figure 2.18.

```
procedure EDMC-LSO(max_iters, max_tries)
    s <- initial circuit; //initial solution
    err <- eval(s);       //RMS error of the initial solution
    i <- 0;               //iterations counter
    while i<max_iters and err>0 do
        op <- select_operation();     //operation selection
        s' <- apply_operation(s,op);  //operation application
        err' <- eval(s');             //new RMS error
        for t:=1 to max_tries do
            if acceptable(err,err') then  //accept change
                s <- s'; err <- err';
            end if
            s' <- alter_parameters(s,op);  //modify solution
            err' <- eval(s');              //new RMS error
        end for
        i <- i + 1;                   //increase iteration counter
    end while
    return s;
end EDMC-LSO
```

Figure 2.18. Local search algorithm with local optimization

2.7.2. Backtracking

Because the algorithm can struggle in some local optimal solution where all neighbour solutions are worse, there has to be a technique to get out of this point. There are two basic methods. The first one restarts the search process after a local optimum (or the maximum number of iterations) is reached. The second one tries to avoid staying in some local optimum point by randomization of the search (e.g. random walk techniques).

Empirical results have shown that the use of only the second technique (simulated annealing and hill climbing was tested) can sometimes lead to a circuit which is very hard or almost impossible to improve (e.g. the circuit is too complex).

So, we proposed to use also the first method, but we do not restart from the beginning of the search. After a certain amount of iterations, the solution is memorized and compared to the previous memorized one. If the solution is not much better (e.g. the RMS error of the solution is not 10% or more lower than the RMS error of the previous solution), the algorithm jumps back to the previous memorized solution. Each such memorized solution has also included a counter of these back jumps to it. If this counter exceeds some value, the algorithm does not jump back to it, but to its memorized ancestor. In some sense, this approach mimics a backtracking search.

An example of such behaviour is shown in Figure 2.19. After each *N* iterations are applied, the solution is compared to the previous memorized one or to the initial one when there is no previous memorized solution – called *choice point*, denoted by a circle. When the solution is not much batter (e.g. evaluation of a solution is not less then the evaluation of the previous memorized (or initial) solution decreased by 10%) the solution is rejected (denoted by cross) and the

search continues again from the previous memorized (or initial) solution. The amount of executions the search from memorized solution is limited to four in the below example. When this number is reached the memorized solution is also rejected (denoted by crossed circle) and the search starts from the previous memorized one etc. Improved algorithm is shown in Figure 2.20.

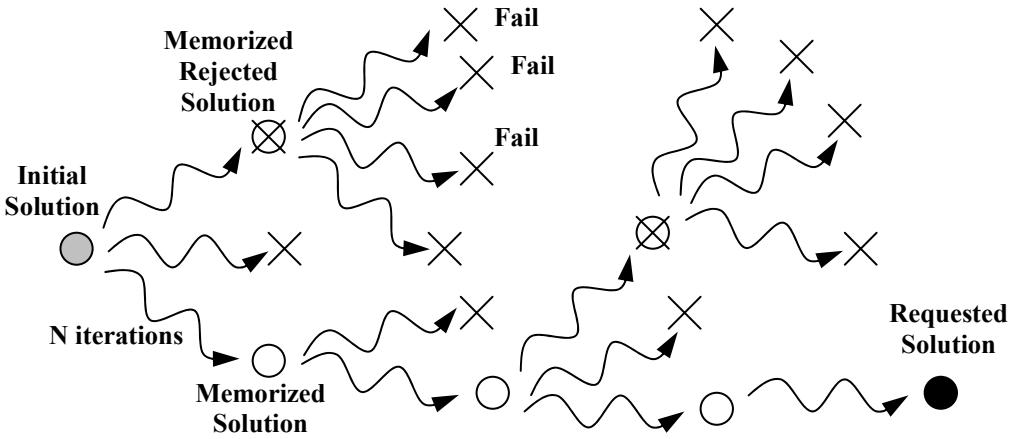


Figure 2.19. Memorizing solutions, backtrack approach

```
procedure EDMC-LSOB(max_iters, max_tries, max_backs)
    s <- initial circuit; //initial solution
    err <- eval(s);       //RMS error of the initial solution
    b <- 0; back[b].s <- s;       //memorize initial solution
    back[b].err <- err;
    back[b].backs <- 0;   //number of returns to the solution
    while b>0 or backs[b]<max_backs do
        i <- 0;               //iterations counter
        while i<max_iters and err>0 do
            op <- select_operation();     //operation selection
            s' <- apply_operation(s,op);  //operation application
            err' <- eval(s');             //new RMS error
            for t:=1 to max_tries do
                if acceptable(err,err') then  //accept change
                    s <- s'; err <- err';
                end if
                s' <- alter_parameters(s,op);  //modify solution
                err' <- eval(s');              //new RMS error
            end for
            i <- i + 1;               //increase iteration counter
        end while
        if (err=0) then return s;      //return solution if found
        if (err<back[b].err*0.9) then    //new choice point
            b <- b + 1;
            back[b].s <- s;       //memorize solution
            back[b].err <- err;
            back[b].backs <- 0;
        end else                  //back to previous choice point
            s <- back[b].s;       //return to memorized solution
            err <- back[b].err;
            back[b].backs <- back[b].backs + 1; //increase counter
        end if
```

(continues on the next page)

```
        while b>0 and back[b].backs>max_backs do
            b <- b - 1;          //maximum returns is reached
            s <- back[b].s;      //return to previous memorized sol.
            err <- back[b].err;
            back[b].backs <- back[b].backs + 1; //increase counter
        end while
    end while
    return fail;  //or best solution ever found
end EDMC-LSOB
```

Figure 2.20. Local search with optimization and backtrack approach

2.7.3. RMS Error Estimation

According to the tests made on the above proposed algorithm (see chapter 4 for details), most time of the program's computation (at least 85%) was spend in the analysis of the circuit. Therefore, in this section we propose a technique which will estimate the result of the suggested operation. The technique will use the knowledge acquired from the previous application of the appropriate operation. According to this estimation of improvement (or degradation) of the current solution a probability of direct rejection of the proposed operation will be computed.

```
procedure EDMC-LSOBE(max_iters, max_tries, max_backs)
    s <- initial circuit; //initial solution
    err <- eval(s);       //RMS error of the initial solution
    b <- 0; back[b].s <- s;       //memorize initial solution
    back[b].err <- err;
    back[b].backs <- 0;   //number of returns to the solution
    stat <- {};           //statistics (empty or some default)
    while b>0 or backs[b]<max_backs do
        i <- 0;               //iterations counter
        while i<max_iters and err>0 do
            op <- select_operation();     //operation selection
            prjc <- estimate_reject_prob(stat, op);//pr. rejection
            if (random(0,1) > prjc) then //otherwise reject
                s' <- apply_operation(s,op);  //op. application
                err' <- eval(s');             //new RMS error
                update_stat(stat,op,err'-err); //update statist.
                for t:=1 to max_tries do
                    if acceptable(err,err') then  //accept change
                        s <- s'; err <- err';
                    end if
                    s' <- alter_parameters(s,op);  //modify sol.
                    err' <- eval(s');              //new RMS error
                    update_stat(stat,op,err'-err);
                end for
            end if
            i <- i + 1;               //increase iteration counter
        end while
        if (err=0) then return s;      //return solution if found
        if (err<back[b].err*0.9) then    //new choice point
            b <- b + 1;
            back[b].s <- s;       //memorize solution
            back[b].err <- err;
            back[b].backs <- 0;
        end else                  //back to previous choice point
```

(continues on the next page)

```
            s <- back[b].s;       //return to memorized solution
            err <- back[b].err;
            back[b].backs <- back[b].backs + 1; //increase counter
        end if
        while b>0 and back[b].backs>max_backs do
            b <- b - 1;         //maximum returns is reached
            s <- back[b].s;     //return to previous memorized sol.
            err <- back[b].err;
            back[b].backs <- back[b].backs + 1; //increase counter
        end while
    end while
    return fail;  //or best solution ever found
end EDMC-LSOBE
```

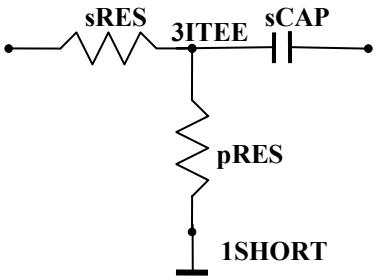
Figure 2.21. Local search with optimization, backtrack and RMS estimation

The adopted algorithm is shown in Figure 2.21. There is a new entity introduced in the algorithm: a statistics of "successfulness" of the applied operations so far. This statistics is updated after each evaluation of a solution (circuit) and it takes the difference of latest and previous evaluation and the applied operation (or some classification of this operation) into account. According to this statistics, after an operation is selected, the probability of direct rejection of the operation is computed. If the probability is then met the operation is refused and the algorithm continue with the another operation selection in the next iteration. Otherwise, the algorithm continues as before – with applying the operation, evaluation of the new circuit, computing whether the change is acceptable and so on. We assume that the probability of rejection (because it is only some estimation) will never be equal to one, e.g. we can limit it to 0.9.

Because we cannot compute the rejection probability only from the operation (it is too inaccurate) or from the operation and from all the operands (there will be very probably no exactly the same operation selected before) we propose a simple classification of an operation and operands. The rejection probability is then computed from the previous achieved differences of operations from the same class. The idea is, to describe each operation with a short string (descriptor). This also helps us to hash the operation classes in accordance with this string. This descriptor is also proposed to be easily readable.

Circuit elements are classified by their names in the net-list (e.g. RES, IND, MISTR, ITEE). In case that an element is assigned to a vertex, there is also a prefix '1', '2', '3' or '4' added according to the number of edges connected to the vertex. Otherwise, when an element is assigned to an edge, there is also a prefix 'p', 's' or 'c' included in the classification, according to the edge's location in the circuit:

- **s:** If this edge is removed, the graph becomes disconnected (graph is divided into the two parts with no edge between them) and both parts contain at least one input/output port.
- **p:** If this edge is removed, the graph become disconnected, but all input/output ports are located in only one part. (See Figure 2.22.)
- **c:** If this edge is removed, the graph remains connected. The edge creates a circle in the graph.



2.22. Circuit element classification

The classification of an operation consists of above described classifications of used elements (created, removed and modified) in the scheme expressing the behaviour of the operation. If there is an edge or a vertex with no (zero) element assigned, the edge or vertex is classified with the name "z" (zero element assignment). In case of the operation O1 (which changes only a single parameter of a selected element), the element classification is followed by the change direction (+ for increasing the parameter, - for decreasing). This direction mark can be repeated several times according to the extent of a change – one mark for a difference less then 25% (from the previous value), two marks for a difference less then 50%, three if less then 75% and four otherwise. Some examples are shown in Figure 2.23.

```
O1:  pRES R-                    -- one parameter is changed
     sMISTR L+++
     pMISTR W++
O2:  sMISTR -> sRES             -- one circuit element is chanded
     1SHORT -> 1OPEN
O3:  1SHORT 3ITEE -> z-cRES-4ICROS -- an edge is added
O4:  3ITEE-sRES-z -> z 1OPEN       -- an edge is removed
O5:  4ICROS -> 3ITEE-cMISTR-3ITEE  -- a node is split to two nodes
     4ICROS -> z-cMISTR-4ICROS
     3ITEE -> 3ITEE-sMISTR-z
O6:  z -> 3ITEE-pRES-1SHORT        -- a vertex w. an edge is added
     3ITEE -> 4ICROS-pMISTR-1OPEN
O7:  sRES -> sRES-z-sCAP           -- an edge is split to two edg.
O8:  4ICROS-pMISTR-1OPEN -> 3ITEE  -- a vertex w. 1 edge is remov.
O9:  pRES-z-pCAP -> pRES           -- 2 edges are joined into 1
     sRES-z-sRES -> sRES           -- a vertex w. 2 edges is rem.
O10: 3ITEE-sRES-z -> 3ITEE         -- 2 connected vertices are
     3ITEE-sRES-3ITEE -> 4ICROSS   --   joined into one
```

Figure 2.23. Examples of operation classes (descriptors)

So, the probability of rejection of a selected operation (with also selected operands) is counted according to the successfulness (differences `err'-err`) of the previous evaluated operations of the same class (operations with the same descriptor). We can for example use the mean value of the previous computed differences for estimation of the new difference. The probability of acceptance can be counted from such estimated differences according to the formula used in simulated annealing (Figure 1.8.) or stochastic hill climbing (Figure 1.10), respective one minus probability of acceptance, which is computed there. We can

also use variances of these differences for estimation of the variance of the estimated difference. If we are unable to compute the rejection probability because the set of the similar previous evaluated operations is not large enough, we can set this probability to zero.

This approach can be also adopted to be used as guidance for the operation selection.

2.8. Conclusion

We have proposed a promising algorithm based on single member evolution and local search techniques which is able to solve many interesting assignments. We also proposed several improvements to this algorithm.

In the following chapter, we will discuss some implementation aspects of this algorithm and in the fourth chapter we will present some results achieved with this algorithm.

3. Implementation Issues

In this chapter, we will discuss some implementation issues of the above proposed and described algorithm. The algorithm is written in JAVA (JSDK 1.4.1). Complete documentation of all implemented classes is generated in HTML format (via JavaDoc) and included on the CD which is a part of this thesis.

The whole implementation is divided into several packages (see Figure 3.1.). We will discuss these packages in the following paragraphs.

Package	**Description**
edmc.elements	Circuit elements
edmc.circuits	Representation of a circuit (graph)
edmc.operations	Operations over circuits
edmc.frequency	Input (frequency) requirements
edmc.analysis	Analysis – connection to the program MIDE
edmc.design	Circuit design – implementation of the algorithm
edmc.util	Supporting classes

Figure 3.1. Packages

3.1. Circuit Elements

Each circuit element which can be used in the design has an appropriate element in the MIDE's net-list and also an appropriate class implemented. Such class has to be extended from the abstract class *edmc.elements.AbstractElement*. It has to have several methods implemented:

Method	**Description**
constructor	Creates an element, assigns values to all parameters
getNrGates()	Returns the number of ports of the element (e.g. 2 for a resistor)
getNrParams()	Returns the number of parameters of the element (e.g. 1 for a resistor – its resistance)
getParameterBounds(i)	Returns i-th parameter‘s bounds
getName()	Returns the name of the element in MIDE
getParameter(i)	Returns the value of the i-th parameter
setParameter(i,val)	Sets the value of the i-th parameter
toString(int[] nodes)	Returns the net-list representation of the element (as a string)
getWidthParam(int i)	For micro-stripes: returns the number of the parameters which represents width for the i-th port (or -1 otherwise, e.g. if there is no width)

Figure 3.2. Element's API

Probably the most interesting part of the API is the method *toString(int[])*. It has one parameter – the array of numbers – which are assigned to the appropriate ports of the element. The result is a string, an appropriate line of the net-list representation. An example implementation of this method for resistor is shown in Figure 3.3. (attribute *iR* represents the resistance parameter)

```
public String toString(int[] nodes) {
    return "RES "+nodes[0]+" "+nodes[1]+
           " R="+ToolBox.formatValue(iR)+"ohm";
}
```

Figure 3.3. An example implementation of toString(int[]) for resistor

There is also an API to propagate micro-stripe's width between connected elements. This can be done via the method *getWidthParam*(*int)*. But, because there is a bug in the analysis tool, which is used in this work, micro-stripe connection elements (e.g. MKR4, MTE4) cannot be used anyway (MIDE always fails with some error).

All access to the element (e.g all changes of parameters) is made through this API. So, a new element can be easily added to the system, just appropriate element's class has to be written. For more information, see appendix B, section B.1.

3.2. Circuit Representation

A circuit is represented by a graph, implemented with classes in *edmc.circuits* package, namely:

Class	Description
Vertex	Representation of a vertex
Edge	Representation of an edge
Graph	Representation of a graph, API for basic graph operations (e.g. add or remove a vertex or an edge) included
Circuit	Representation of a circuit: a graph plus circuit operations (e.g. printing a net-list)
SerialNumberFactory	Generator of unique numbers for vertices and edges (e.g. to be used in net-list)

Figure 3.3. Package edmc.circuits

Basically a graph is represented by vertices and edges (connections between vertices). Besides the traditional graph-based functionality (e.g. to enumerate vertices connected to a given vertex), there is also information and API for the assignment of a circuit element to a vertex or an edge and to maintenance unique serial numbers for vertices (one per vertex, e.g. for comparison of two vertices) and for edges (two per edge, one for each end – to be used in the net-list representation). Class *Graph* provides the basic graph functionality, for example for counting and enumerating all edges and vertices, to add or remove an edge or a vertex, to check whether a vertex or an edge can be removed (graph has to

remain connected) and last but not least for cloning a graph (used e.g. for memorizing the previous configuration). Class *Circuit* provides the circuit-based functionality, it creates an initial circuit, it determines whether a vertex is an input/output port or not, it collects information about available circuit elements, it prints net-list representation and enumerates edge and vertex classification (e.g. sRES string). There is also API for cloning circuits. The following Figure 3.4 shows an example of creating a circuit. The resultant net-list representation is presented in Figure 3.5.

```
Circuit c = new Circuit(2,"test"); // 2 i/o ports, name is test

//drop the initial edge between i/o ports
c.getGraph().dropEdge(c.getIO(0),c.getIO(1));

Vertex v1 = c.getGraph().createVertex(); // new vertices
Vertex v2 = c.getGraph().createVertex();
Vertex v3 = c.getGraph().createVertex();

Edge ei1 = c.getGraph().createEdge(c.getIO(0),v1); //new edges
Edge e1o = c.getGraph().createEdge(v1, c.getIO(1));
Edge e12 = c.getGraph().createEdge(v1,v2);
Edge e13 = c.getGraph().createEdge(v1,v3);

v1.setElement(new ICROS()); //assign elements to vertices
v2.setElement(new OPEN());
v3.setElement(new SHORT());

//assign elements to edges - set appropriate parameters
ei1.setElement(new RES());ei1.getElement().setParameter(0,75.0);
e1o.setElement(new CAP());e1o.getElement().setParameter(0,1e-6);
e12.setElement(new IND());e12.getElement().setParameter(0,2.3e-9);
e13.setElement(new RES());e13.getElement().setParameter(0,1e3);

//print resultant net-list representation
c.printNetlist(System.out);
```

Figure 3.4. A circuit example

```
BLO test 1 4
RES 1 2 R=75 ohm
CAP 3 4 C=1 miF
ICROS 2 5 3 7
IND 5 6 L=2.3 nH
RES 7 8 R=1000 ohm
OPEN 6
SHORT 8
END
```

Figure 3.5. A circuit example – resultant net-list

3.3. Operations

Available circuit operations are located in the package *edmc.operations*. Each circuit operation has to be extended from abstract class *edmc.operations. CircuitOperation*. There has to be following four methods implemented:

Method	Description
boolean process()	Selects operation‘s operands, applies the operation to the current circuit and returns true. If there are no suitable operands (e.g. no edge can be removed) it returns false (operation was not applied).
boolean redo()	Local optimization: it changes parameters affected by previous call of method process (or redo, when the changes are accepted). (redo can be called several times)
good()	Called when changes made by process or redo are accepted.
toString()	Returns operation’s classification (descriptor).

Figure 3.6. Operation’s API

So, in each iteration step, the current circuit is memorized first. An operation is selected and the appropriate method *process* is called then. If it returns false, nothing was changed and the algorithm returns to the selection of an operation. Otherwise, the changed circuit is evaluated and if the local optimization is used (as described in section 2.7.1) the method *redo* is called several times (to optimize parameters changed by method *process*). If the changed circuit is accepted, method *good* is called (can be called after *process* method and each *redo*). This affects the mean values of the distribution used for the selection of parameters’ values (previous accepted values are used). The last accepted circuit is used for the next iteration.

3.4. Frequency Requirements

Classes for representation of the input requirements (on the given frequncy) are located in the package *edmc.frequency*. These classes also provide computation of the resultant RMS error from the values computed in analysis. There are classes:

Class	Description
FrequencyElement	Interface for frequency requirements.
Frequency	Requirements on a single frequency.
FrequencyStep	Requirements on a set of frequencies given by minimal, maximal frequency and a difference between two frequencies from this set.
FrequencyLog	Requirements on a logarithmically spread set of frequencies – given by minimal, maximal and the number of frequencies.
FrequencySet	Set of input requirements.

Figure 3.7. Package edmc.requency

Each requirement (described by class *Frequency*, *FrequencyStep* or *FrequencyLog*) is an implementation of the interface *FrequencyElement* and provides API for enumeration its frequencies (class *Frequency* presents a set of one requirement) and a requirement on these frequencies. All these requirements are placed together in the class *FrequencySet* which represents all the input

requirements. This class also provides API for enumeration of all used frequencies (with no duplicities) needed by the analysis, conversion of requirements from input strings and for computation of the RMS error based on the requirements and the results from the analysis.

A requirement consists from a parameter (e.g. transmission of a circuit), an operation (e.g. less then), a value (e.g. -25 dB) and a weight of the requirement (default weight is 1.0). A parameter consists from a type and from the row and column of the S-matrix (denoted by type, character 'S', row and column - e.g. transmission from first to second gate of a circuit expressed in dB is denoted by DS21). Available types are denoted in Figure 3.8 and available operations are denoted in Figure 3.9. An example of requirements is shown in Figure 3.10.

Type	Description
A	Magnitude
F	Phase
D	Magnitude in dB
R	Real part
I	Imaginary part
G	Operational transmission (S21*S21)
K	Rollet stability factor

Figure 3.8. Types of requirement parameters

Operation	Description
?	No requirement
=	Equal
>	Greater than
>=	Greater or equal
<	Less than
<=	Less or equal

Figure 3.9. Requirement operations

```
OPT
  1.0 GHz DS21 < -25
  1.2 GHz DS21 < -25 w 5
  STEP 100 MHz 250 MHz 10 MHz AS21 > 0.9
  STEP 100 MHz 250 MHz 10 MHz DS11 < -33 w 2.3
  LOG 1 GHz 29 GHz 12 RS12 > 0
END
```

Figure 3.10. Requirements example

So, a single requirement is described as a line in the OPT block, it consists of frequency specification: single frequency, linearly distributed set of frequencies described by STEP and their minimum, maximum and distance between them (e.g. 100, 110, 120 MHz in the above example) or logarithmically distributed set of frequencies described by LOG and their minimum, maximum and their amount (plus one – according to the specification in MIDE) (e.g. thirteen frequencies 1, 1.324, … 29 GHz in the above example).

3.5. Analysis

Classes providing connection to external analysis tool (program MIDE) are located in the package *edmc.analysis*. There are two classes, first (called *MideAnalysis*) calls the MIDE analysis for the given circuit and the second (called *SParam*) is used for returning resultant S-parameters on the given frequency. It also provides conversion of S-parameters to other parameters types described in Figure 3.8.

Because, the program MIDE is used as an external program, the parameters are handed over as a single file, with two blocks – demanded frequencies and a circuit represented as net-list. An example of such file is shown in Figure 3.11.

```
FREQ
STEP 500 MHz 900 MHz 100 MHz
STEP 995 MHz 1005 MHz 1000 kHz
STEP 1100 MHz 2400 MHz 100 MHz
STEP 2495 MHz 2505 MHz 1000 kHz
STEP 2600 MHz 3000 MHz 100 MHz
END
BLO edmc 1 3 5
IND 1 2 L=0.00175 pH
MCPL 3 500 4 501 W=1.71058 mm L=28.78127 mm S=110 mim
SHORT 500
SHORT 501
MCPL 5 502 6 503 W=540 mim L=81.99079 mm S=2.16089 mm
SHORT 502
OPEN 503
ICROS 6 16 7 22
CAP 15 16 C=3.3766 pF
MISTR 7 8 W=1.34809 mm L=110.52228 mm
IND 21 22 L=15.24331 nH
MCPL 8 9 11 504 W=540 mim L=2.10562 mm S=298.95314 mim
OPEN 504
MGAP 9 10 W=770.18605 mim S=62.03038 mim
IND 11 12 L=25.90359 nH
ITEE 12 39 13
RES 39 40 R=1852.58228 ohm
CAP 13 14 C=2.25181 pF
ICROS 18 27 19 31
MISTR 14 18 W=2 mm L=51.83293 mm
CAP 27 28 C=0.52194 pF
MISTR 19 10 W=540 mim L=15.31472 mm
CAP 31 32 C=3.98469 pF
ITEE 15 34 32
MISTR 33 34 W=540 mim L=19.00889 mm
ICROS 4 23 21 30
MCPL 23 505 24 506 W=2 mm L=55.14562 mm S=500 mm
SHORT 505
OPEN 506
MGAP 29 30 W=2 mm S=665.71218 mim
CAP 26 29 C=8.03912E-16 F
MCPL 24 25 28 507 W=875.24938 mim L=2.16024 mm S=420.94364 mm
SHORT 507
IND 25 26 L=41.92279 nH
ITEE 2 35 33
MISTR 35 36 W=540 mim L=5.11045 mm
OPEN 36
OPEN 40
END
```

Figure 3.11. A file for MIDE analysis

Result from the MIDE program is another file, with a table in CSV format (coma separated text file, e.g. readable by Microsoft Excel) with the computer S-parameters on the given frequencies. Example is shown in Figure 3.12. If the analysis fails (from any reason), the altered solution is automatically rejected.

Freq	S11[re]	S11[im]	S12[re]	S12[im]	S13[re]	S13[im]	...
1	-1,55E-01	3,04E-01	-4,75E-02	2,69E-02	-1,20E-01	4,71E-02	
2	-9,56E-01	1,49E-01	1,88E-02	2,24E-02	1,87E-02	7,86E-03	
3	-5,56E-01	7,97E-01	3,71E-03	5,75E-03	3,25E-03	1,05E-03	
4	8,62E-01	1,95E-01	-1,75E-04	-6,25E-03	-1,64E-03	-1,76E-03	
5	-2,36E-01	8,49E-01	-1,28E-01	1,69E-01	4,25E-03	6,68E-02	
6	1,78E-01	1,47E-01	6,04E-01	1,78E-01	1,38E-01	-2,73E-02	
...							...

Figure 3.12. Resultant S-parameters from MIDE

3.6. Design

Design is provided by classes located in package *edmc.design*. Beside the main class *Designer* which does the search, there are also plenty of classes to provide selection of vertices, edges, elements and operations. In the current implementation, most (all except the selection of an operator) them selects one of the available element randomly with the uniform probability. But there is a nice API for change these selections. Operation is also selected randomly, but individual operations can have different probability, according to their weights (Figure 3.13), where O is the set of available operations. These weights can be also changed during the search (e.g. increased by one when the operation is accepted).

$$P_{op} = w_{op} \Big/ \sum_{o \in O} w_o$$

Figure 3.13. Selection probability of a operation

The design algorithm (class *Designer*) is implemented according to the chapter two, with all improvements proposed there.

In this package (*edmc.design*), there is also a class (called *InputFile*) for reading input file (with requirements and designer settings) and a class for providing the rejection probability (called *Prediction*) according to the section 2.7.3.

3.7. Conclusion

In this chapter, we have described some interesting issues of the implementation of the above presented algorithm. The main advantage of this implementation is the great possibility to extend the implementation for new circuit elements, frequency requirements, circuit operations, selection heuristics and also extension of the algorithm. Different tools for analysis can also be added.

This chapter is also mentioned as the starting point before reading javadoc documentation included on the CD, which provides the complete description of each implemented class.

The user documentation, with describes usage of the resultant program, structure of input files etc. is placed in the appendix A of this thesis. Some examples of how to extend the implementation are located in the appendix B.

4. Practical Results

In this chapter, we present some practical results of the algorithm. Our main goal here is to prove the applicability of single member population evolutionary algorithms in design of microwave circuits. All measurements were made on Pentium III 1GHz, 512 MB RAM, with java JSDK 1.4.1.

4.1. A Band Pass Filter

First example is a band pass filter with the following design goals:

- Between 1 GHz and 3 GHz transmission below -25 dB
- Passband transmission above -1.5 dB between 5.5 GHz and 6.5 GHz
- Stopband attenuation between 9.225 GHz and 12 GHz below -35 dB

The appropriate design input file, which was used, is presented in Figure 4.1.

```
*EDMC Input File – band pass filter
*Designer configuration
CONFIGURATION
 NR_GATES 2
 FREQ_MIN 1e9
 FREQ_MAX 12e9
 FREQ_STEP 50e6
END
*Available circuit elements for edges
LINKS
 MISTR
 MGAP
END
*Available circuit elements for vertices
NODES
 ICROS
 ITEE
 SHORT
 OPEN
 MCPL
END
*Available operations – names of appropriate classes + weight
OPERATIONS
 LinkParamModifier 50
 NodeParamModifier 50
 LinkElementChanger 20
 NodeElementChanger 10
 SplitLink 6
 SplitLinkT 4
 SplitNode 6
 AddLink 4
 RemoveLink 5
 JoinNodes 5
 RemoveNode 10
 RemoveNode2 7
END
```

(continues on the next page)

```
*Input requirements
OPT
 STEP 1     GHz 3     GHz 250 MHz DS21 < -25.0
 STEP 5.5   GHz 6.5   GHz  50 MHz DS21 >  -1.5 w 10
 STEP 9.225 GHz 12    GHz 250 MHz DS21 < -35.0
END
```

Figure 4.1. Band pass filter input file

In the following subsections, we will described results obtained from the implemented program, for two different executions.

4.1.1. First Execution

A complete solution (with RMS error equal to zero) was found after 4058 seconds. Program MIDE was called 48513 times and its execution takes 88% of all time spent. Resultant characteristics are given on the following graph (Figure. 4.2) and the resultant circuit is presented in Figure 4.3. (in the net-list language).

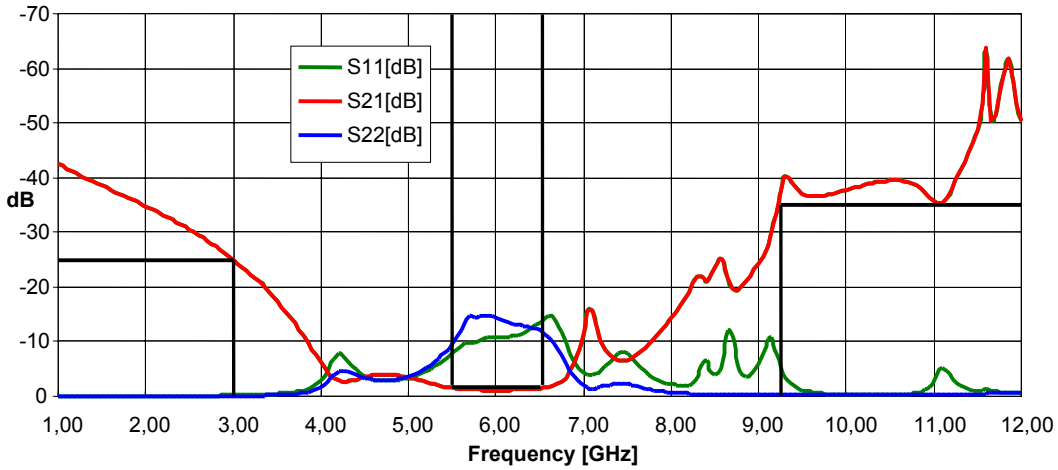


Figure 4.2. Characteristics of the resultant circuit

```
BLO edmc 7 32
MISTR 7 8 W=540 mim L=8.71104 mm
MISTR 31 32 W=540 mim L=8.02935 mm
ICROS 5 13 2 3
MCPL 5 500 6 501 W=1.95273 mm L=2.50405 mm S=652.99094 mim
SHORT 500
OPEN 501
MCPL 13 502 14 503 W=1.42652 mm L=1.09886 mm S=500 mm
SHORT 502
SHORT 503
MISTR 1 2 W=540 mim L=10.68541 mm
MISTR 3 4 W=540 mim L=810.41299 mim
ITEE 6 17 9
MCPL 17 504 18 505 W=540 mim L=1.77733 mm S=110 mim
OPEN 504
OPEN 505
MCPL 9 506 10 507 W=2 mm L=1.80125 mm S=8.36805 mm
SHORT 506
```

(continues on the next page)

```
SHORT 507
ITEE 10 49 24
MCPL 49 508 50 509 W=2 mm L=1.06373 mm S=121.10919 mm
OPEN 508
OPEN 509
MGAP 23 24 W=2 mm S=182.2038 mim
ICROS 8 18 1 16
MCPL 15 510 16 511 W=2 mm L=3.02606 mm S=110 mim
SHORT 510
SHORT 511
ICROS 14 41 15 33
MISTR 41 29 W=540 mim L=2.53002 mm
MCPL 33 512 34 513 W=2 mm L=2.09088 mm S=670.71981 mim
OPEN 512
OPEN 513
ICROS 4 31 23 26
MCPL 25 514 26 515 W=919.93726 mim L=6.49818 mm S=5.8395 mm
SHORT 514
OPEN 515
MCPL 34 516 35 517 W=2 mm L=6.97535 mm S=52.7003 mm
OPEN 516
OPEN 517
MCPL 35 518 11 519 W=665.94771 mim L=11.08678 mm S=1.85916 mm
OPEN 518
SHORT 519
MGAP 29 30 W=2 mm S=526.22862 mim
SHORT 12
MGAP 11 12 W=540 mim S=10.1 mim
SHORT 50
MGAP 30 25 W=2 mm S=526.22862 mim
END
```

Figure 4.3. Resultant circuit

4.1.2. Second Execution

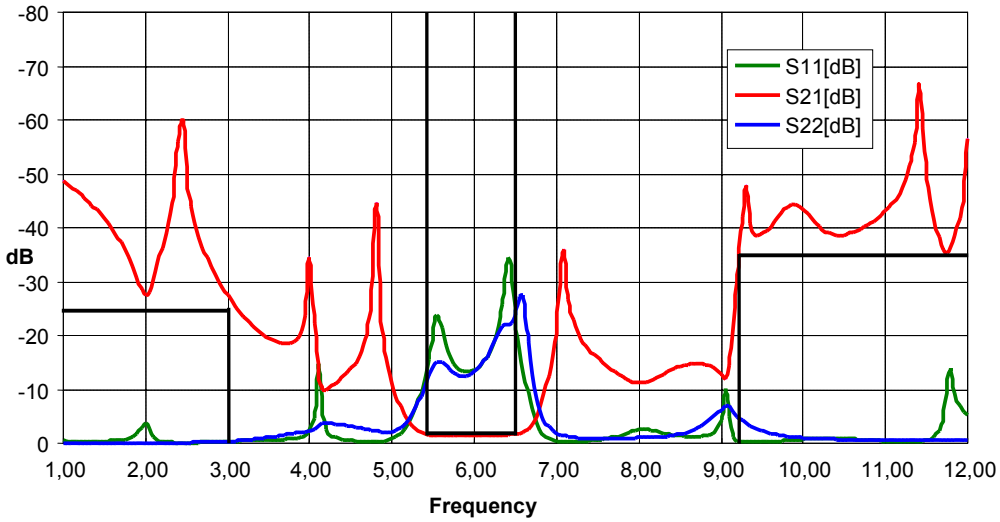


Figure 4.4. Characteristics of the resultant circuit – second execution

In the second execution, a complete solution (with RMS error equal to zero) was found after 5110 seconds. Program MIDE was called 88690 times and its execution takes 90% of all time spent. Resultant characteristics are given on the

following graph (Figure. 4.4) and the resultant circuit is presented in Figure 4.5. and 4.6.

```
BLO edmc 1 6
MISTR 1 2 W=1.56204 mm L=5.93313 mm
MISTR 5 6 W=2 mm L=3.87198 mm
ITEE 3 17 10
MCPL 3 500 4 501 W=759.77503 mim L=6.56124 mm S=126.19163 mim
OPEN 500
OPEN 501
MISTR 17 18 W=552.14427 mim L=924.06968 mim
MISTR 9 10 W=540 mim L=8.65577 mm
ICROS 4 13 7 21
MCPL 13 502 14 503 W=540 mim L=4.1677 mm S=110 mim
OPEN 502
OPEN 503
MISTR 7 8 W=2 mm L=2.16245 mm
MISTR 21 22 W=540 mim L=3.69405 mm
OPEN 8
ITEE 9 2 11
MISTR 11 12 W=1.67296 mm L=23.11804 mm
SHORT 12
ITEE 14 15 5
MISTR 15 16 W=822.68662 mim L=2 mm
SHORT 16
OPEN 18
MISTR 22 24 W=2 mm L=2.10017 mm
OPEN 24
END
```

Figure 4.5. Resultant circuit – second execution

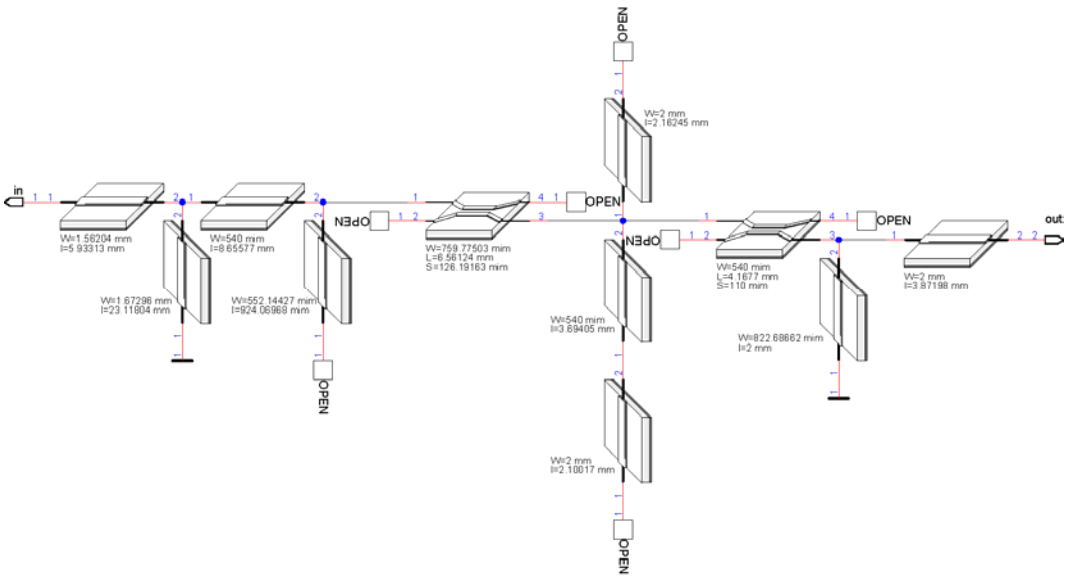


Figure 4.6. Resultant circuit – second execution

4.2. An Amplifier

Second example is an amplifier with the following requirements:

- Transmission at 1.7 GHz is above 17 dB
- Reflection both on input and output gate is less then -40 dB at frequency 1.7 GHz
- Use of a MGF1801 transistor

The appropriate design input file, which was used, is presented in Figure 4.7.

```
*EDMC Input File - amplifier
*Designer configuration
CONFIGURATION
 NR_GATES 2
 FREQ_MIN 1.5e9
 FREQ_MAX 1.9e9
 FREQ_STEP 5e6
END
*Available circuit elements for edges
LINKS
 MISTR
 RES
 2PORT data\\Mgf1801
END
*Available circuit elements for vertices
NODES
 ICROS
 ITEE
 SHORT
END
*Available circuit operations - names of appropriate classes
OPERATIONS
 LinkParamModifier 50
 LinkElementChanger 20
 SplitLink 6
 SplitLinkT 4
 SplitNode 6
 AddLink 4
 AddNode 4
 RemoveLink 10
 JoinNodes 10
 RemoveNode 20
 RemoveNode2 15
END
*Input requirements
OPT
 STEP 1.7   GHz 1.7   GHz 1   MHz DS21 > 17.0
 STEP 1.7   GHz 1.7   GHz 1   MHz AS11 < 0.01
 STEP 1.7   GHz 1.7   GHz 1   MHz AS22 < 0.01
END
```

Figure 4.7. Amplifier input file

4.2.1. First Execution

A complete solution (with RMS error equal to zero) was found after 163 seconds. Program MIDE was called 5396 times and its execution takes 84% of all

time spent. Transistor MGF1801 was used thee times. Resultant characteristics are given on the following graph (Figure. 4.8) and the resultant circuit is presented in Figure 4.9.

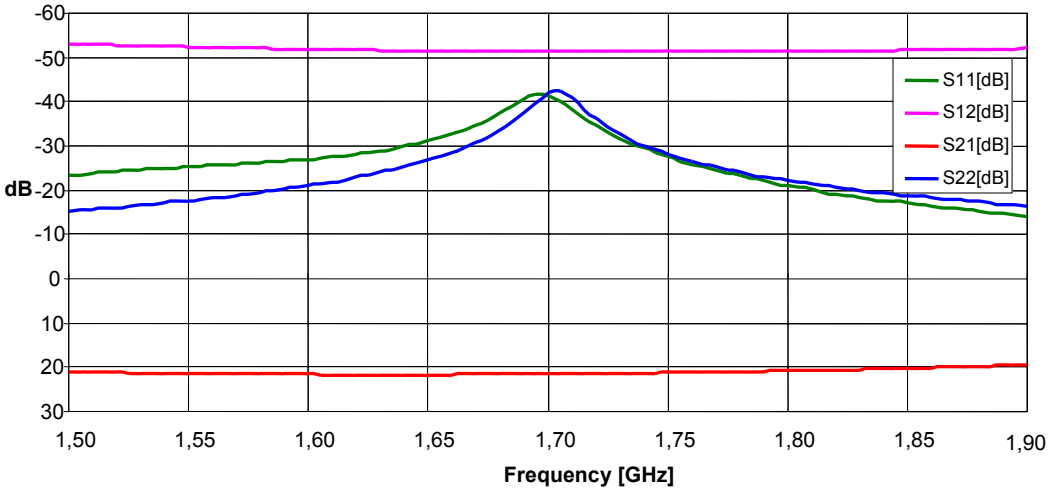


Figure 4.8. Characteristics of the resultant circuit

```
BLO edmc 3 6
MISTR 3 4 W=540 mim L=45.64546 mm
RES 5 6 R=9.01785 ohm
ICROS 1 7 12 25
2PORT 1 2 data\\Mgf1801.spr
RES 7 8 R=521.31844 ohm
2PORT 11 12 data\\Mgf1801.spr
MISTR 25 26 W=1.52669 mm L=15.28378 mm
SHORT 26
ITEE 4 8 21
RES 21 22 R=12.13988 ohm
ITEE 2 15 5
MISTR 15 16 W=1.26851 mm L=31.1886 mm
RES 17 9 R=218.32565 ohm
2PORT 9 10 data\\Mgf1801.spr
ICROS 17 16 19 23
RES 19 20 R=83.7481 ohm
MISTR 23 24 W=2 mm L=14.99323 mm
SHORT 20
ITEE 22 13 11
RES 13 14 R=1002.3585 ohm
SHORT 24
SHORT 10
SHORT 14
END
```

Figure 4.9. Resultant circuit

4.2.2. Second Execution

A complete solution (with RMS error equal to zero) was found after 150 seconds. Program MIDE was called 5306 times and its execution takes 85% of all time spent. Transistor MGF1801 was used two times. Resultant characteristics are

given on the following graph (Figure. 4.10) and the resultant circuit is presented in Figure 4.11 and 4.12.

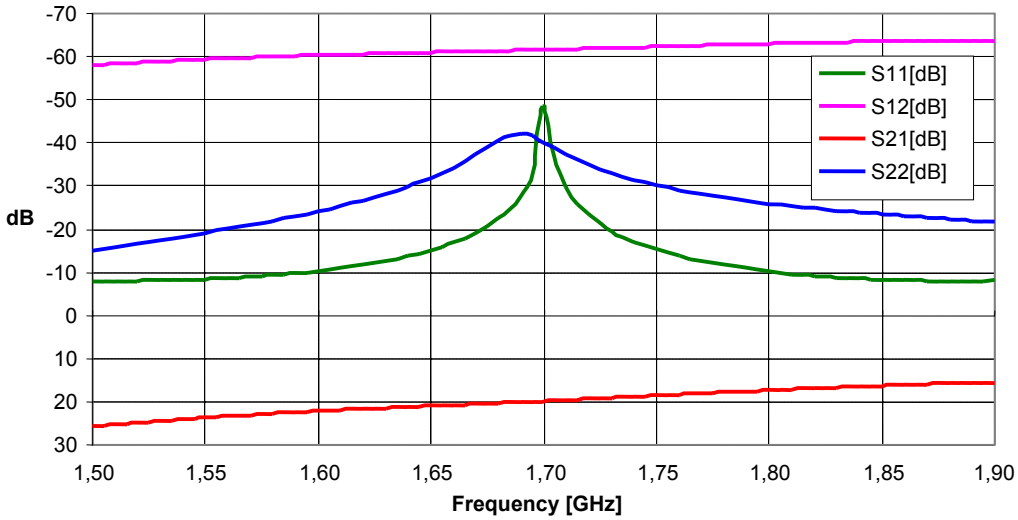


Figure 4.10. Characteristics of the resultant circuit – second execution

```
BLO edmc 5 18
MISTR 5 6 W=2 mm L=156.08503 mm
MISTR 17 18 W=540 mim L=1.09148 mm
2PORT 8 14 data\\Mgf1801.spr
2PORT 10 3 data\\Mgf1801.spr
ITEE 6 9 11
RES 9 10 R=4.37169 ohm
RES 11 12 R=44.55863 ohm
SHORT 12
MISTR 3 4 W=2 mm L=11.66162 mm
MISTR 4 8 W=540 mim L=17.78792 mm
ITEE 14 1 17
RES 1 2 R=100.19728 ohm
ITEE 2 25 15
RES 25 26 R=34.57923 kohm
MISTR 15 16 W=540 mim L=8.17067 mm
MISTR 16 20 W=2 mm L=1.69509 mm
SHORT 20
SHORT 26
END
```

Figure 4.11. Resultant circuit – second execution

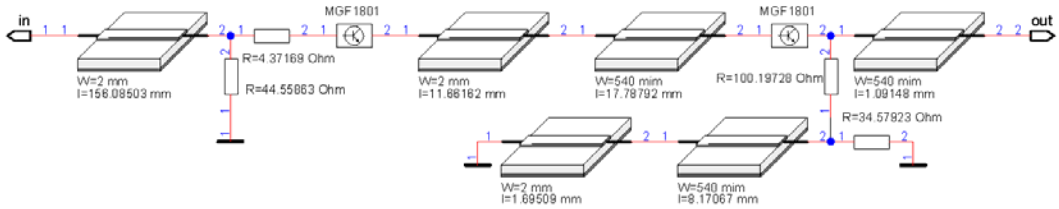


Figure 4.12. Resultant circuit – second execution

4.2.3. Third Execution

In the third execution, we want to present an interesting feature of the implemented algorithm. If we enlarge the set of available circuit elements, the

resultant circuit becomes much larger and the program will spend more time in searching for it. For the third execution, we added elements MGAP, IND and CAP to available edge (link) elements and elements OPEN and MCPL to available vertex (node) elements.

A complete solution (with RMS error equal to zero) was found after 1220 seconds. Program MIDE was called 28794 times and its execution takes 70% of all time spent. Transistor MGF1801 was used six times. Resultant characteristics are given on the following graph (Figure. 4.13) and the resultant circuit is presented in Figure 4.14.

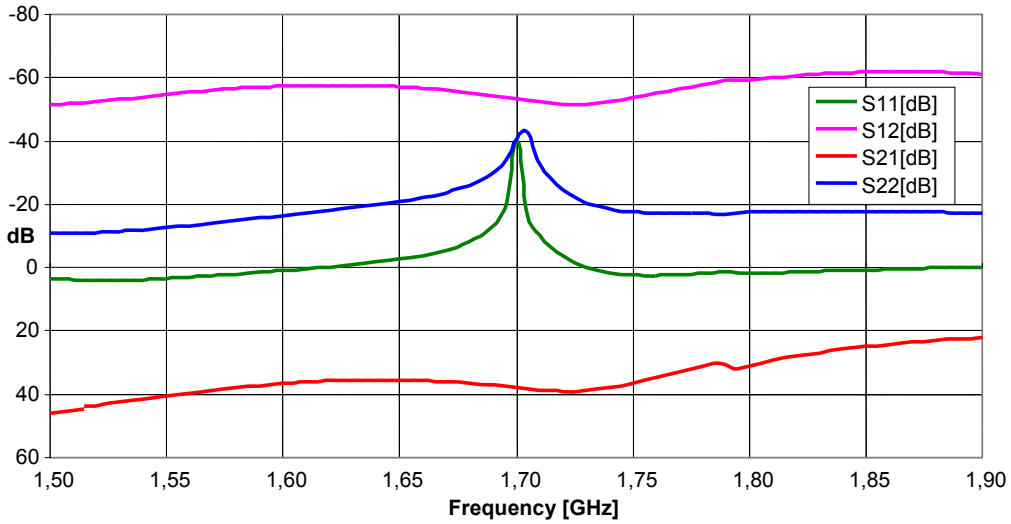


Figure 4.13. Characteristics of the resultant circuit – third execution

```
BLO edmc 1 4
IND 1 2 L=6.59921 nH
2PORT 3 4 data\\Mgf1801.spr
ICROS 8 11 9 45
2PORT 7 8 data\\Mgf1801.spr
IND 11 12 L=31.97994 nH
MISTR 9 10 W=2 mm L=186.23889 mm
MGAP 45 46 W=2 mm S=28.35029 mim
2PORT 13 7 data\\Mgf1801.spr
ICROS 2 22 13 15
RES 15 16 R=161.14212 ohm
ITEE 3 37 20
IND 37 38 L=142.51702 nH
IND 19 20 L=16.47499 nH
MCPL 17 6 25 500 W=540 mim L=18.35206 mm S=500 mm
SHORT 500
MISTR 17 18 W=2 mm L=150.3482 mm
MGAP 5 6 W=540 mim S=10.1 mim
MGAP 25 26 W=1.8122 mm S=10.1 mim
ITEE 18 55 32
MGAP 55 56 W=2 mm S=123.98318 mim
MISTR 31 32 W=989.83331 mim L=2.8251 mm
ITEE 10 53 19
CAP 53 54 C=3.09664 nF
```

(continues on the next page)

```
ITEE 21 49 24
IND 21 22 L=63.08765 nH
CAP 49 50 C=1.24495E-18 F
2PORT 23 24 data\\Mgf1801.spr
ITEE 26 39 23
CAP 39 40 C=1.00000E-22 F
ICROS 12 27 5 59
CAP 59 60 C=0.01562 pF
MCPL 16 35 31 501 W=866.32719 mim L=2.23132 mm S=16.50215 mm
OPEN 501
MCPL 38 33 58 502 W=540 mim L=1 mm S=26.92026 mm
OPEN 502
MGAP 33 34 W=2 mm S=182.07513 mim
2PORT 48 58 data\\Mgf1801.spr
ITEE 29 54 43
IND 29 30 L=526.81286 nH
RES 27 28 R=5941.60814 ohm
MGAP 35 28 W=540 mim S=28.98025 mim
SHORT 42
CAP 34 42 C=0.00423 pF
SHORT 44
IND 43 44 L=469.52831 nH
SHORT 52
IND 51 52 L=7.17647 miH
SHORT 30
MCPL 46 47 51 503 W=540 mim L=22.61353 mm S=500 mm
OPEN 503
2PORT 47 48 data\\Mgf1801.spr
SHORT 50
SHORT 40
SHORT 60
SHORT 56
END
```

Figure 4.14. Resultant circuit – third execution

4.3. A Frequency Splitter

Third example is a frequency splitter circuit with 3 ports with the following requirements:

- Transmission from first port to second port:
  - Stopband attenuation below 700 MHz transmission below -25 dB
  - Passband transmission above -2 dB between 990MHz and 1010 MHz
  - Stopband attenuation after 1.3 GHz below -25 dB
- Transmission from first port to third port:
  - Stopband attenuation below 2.2 GHz transmission below -25 dB
  - Passband transmission above -2 dB between 2.49 GHz and 2.51 GHz
  - Stopband attenuation after 2.8 GHz below -25 dB

The appropriate design input file, which was used, is presented in Figure 4.15.

```
*EDMC Input File - frequency splitter
*Designer configuration
CONFIGURATION
 NR_GATES 3
END
*Available circuit elements for edges
LINKS
 MISTR
 RES
 MCPL_OPEN_OPEN
 MCPL_SHORT_SHORT
 MCPL_OPEN_SHORT
 MCPL_SHORT_OPEN
END
*Available circuit elements for vertices
NODES
 ICROS
 ITEE
 SHORT
 OPEN
END
*Available circuit operations - names of appropriate classes
OPERATIONS
 LinkParamModifier 50
 NodeParamModifier 50
 LinkElementChanger 20
 NodeElementChanger 10
 SplitLink 6
 SplitLinkT 4
 SplitNode 6
 AddLink 4
 RemoveLink 5
 JoinNodes 5
 RemoveNode 10
 RemoveNode2 7
END
*Input requirements
OPT
 STEP 0.5   GHz 0.7   GHz 100 MHz DS21 < -25.0
 STEP 0.990 GHz 1.010 GHz   1 MHz DS21 > -2.0 w 3
 STEP 1.3   GHz 3     GHz 100 MHz DS21 < -25.0
 STEP 0.5   GHZ 2.2   GHz 100 MHz DS31 < -25.0
 STEP 2.490 GHz 2.510 GHz   1 MHz DS31 > -2.0 w 3
 STEP 2.8   GHz 3     GHz 100 MHz DS31 < -25.0
END
```

Figure 4.15. Amplifier input file

A complete solution (with RMS error equal to zero) was found after 1049 seconds. Program MIDE was called 7276 times and its execution takes 89% of all time spent. Resultant characteristics are given on the following graph (Figure. 4.16) and the resultant circuit is presented in Figure 4.17. (in the net-list language). Errors in stopbands (e.g. on frequency 1.65 GHz) are caused by frequency step 100 MHz used in the requirements (e.g. it forces transmission to be less then -25 dB only on frequencies 1.6GHz and 1.7GHz).

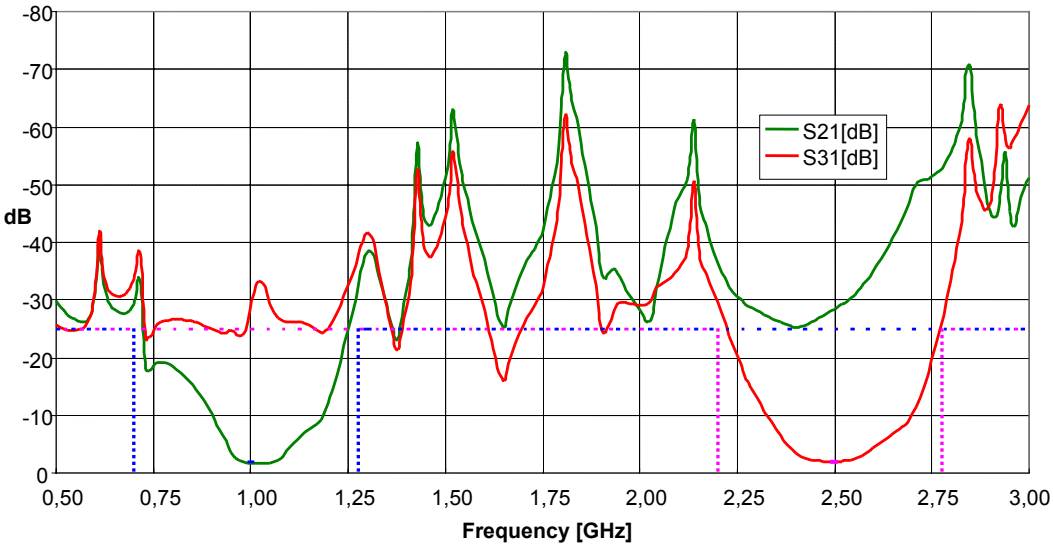


Figure 4.16. Characteristics of the resultant circuit – frequency splitter

```
BLO edmc 1 3 5
MISTR 1 2 W=569.51105 mim L=19.23556 mm
MISTR 3 4 W=540 mim L=3.43969 mm
MISTR 5 6 W=2 mm L=11.80003 mm
ICROS 7 22 16 28
MISTR 7 8 W=2 mm L=38.8069 mm
MISTR 21 22 W=582.2626 mim L=59.00661 mm
MISTR 15 16 W=540 mim L=25.19255 mm
MISTR 27 28 W=2 mm L=52.95338 mm
ICROS 8 39 11 51
RES 39 40 R=14.98205 ohm
MISTR 11 12 W=1.15274 mm L=8.89615 mm
RES 51 52 R=5816.53144 ohm
ICROS 2 25 13 9
MISTR 25 26 W=2 mm L=78.09236 mm
MCPL 13 500 14 501 W=2 mm L=62.26594 mm S=18.7913 mm
SHORT 500
OPEN 501
MISTR 9 10 W=725.85307 mim L=16.4946 mm
ITEE 12 49 19
RES 49 50 R=11.72978 kohm
RES 19 20 R=10.32206 kohm
SHORT 14
ITEE 6 17 15
MCPL 17 502 18 503 W=1.05499 mm L=27.1887 mm S=26.76051 mm
OPEN 502
OPEN 503
MCPL 18 504 34 505 W=540 mim L=1 mm S=323.87823 mim
SHORT 504
SHORT 505
ICROS 4 23 21 43
MCPL 23 506 24 507 W=1.98613 mm L=10.15956 mm S=3.19237 mm
SHORT 506
OPEN 507
MCPL 43 508 44 509 W=540 mim L=29.18606 mm S=1.01856 mm
```

(continues on the next page)

```
SHORT 508
SHORT 509
ITEE 24 31 41
MCPL 31 510 32 511 W=540 mim L=9.21812 mm S=641.62831 mim
SHORT 510
OPEN 511
MCPL 41 512 40 513 W=540 mim L=1 mm S=2.05224 mm
OPEN 512
OPEN 513
SHORT 26
ICROS 10 29 27 36
MCPL 29 514 30 515 W=2 mm L=30.23749 mm S=901.65432 mim
OPEN 514
SHORT 515
MISTR 35 36 W=1.64023 mm L=35.27847 mm
MCPL 30 516 38 517 W=540 mim L=5.13426 mm S=1.6458 mm
SHORT 516
OPEN 517
ITEE 35 48 45
MCPL 34 518 48 519 W=540 mim L=1 mm S=323.87823 mim
SHORT 518
SHORT 519
MCPL 45 520 46 521 W=2 mm L=28.14705 mm S=2.81992 mm
OPEN 520
SHORT 521
ITEE 38 54 55
MISTR 52 54 W=540 mim L=21.8578 mm
OPEN 20
OPEN 46
MCPL 55 522 32 523 W=540 mim L=18.70358 mm S=112.07285 mim
SHORT 522
OPEN 523
SHORT 44
OPEN 50
END
```

Figure 4.17. Resultant circuit – frequency splitter

4.4. Conclusion

As it is shown in this chapter, the implemented evolutionary algorithm is able to solve many interesting assignments. On the above presented problems, the program has always been able to find a complete solution, with RMS error equal to zero, if there is enough time for execution. But, the results significantly vary one execution from another. The dependency of the results and needed time for execution on the input parameters, especially on the set of available circuit elements, is also visible.

For the proposed improvements, the first two of them (local optimization and backtrack) seem to be very helpful in the design. Without these improvements, the algorithm was much less stable: sometimes it was unable to find a solution even in enormous amount of time. The third improvement seems to be useless – there is no visible improvement of the speed, stability or quality of solutions when this improvement is used. The results presented in this chapter were made with the first two improvements enabled.

5. Conclusion

We have presented a promising evolutionary algorithm for microwave circuit design. The main advantage of this algorithm is to point out a possibility of use of evolutionary algorithms in the area of circuit design. The implemented program can give solutions on rather complicated requirements without any knowledge of traditional design strategies and algorithms.

In the first chapter, the general principles and methods of evolutionary algorithms were described. In the second chapter the algorithm for the design of microwave circuits was proposed and the implementation of the proposed algorithm in Java was outlined in the third chapter. The fourth chapter contains some practical results achieved with the implemented algorithm on three different assignments.

6. References

[1] T. Bäck, F. Hoffmeister, H. P. Schwefel. *A survey of evolution strategies.* Proc. of the Fourth International Conference on Genetic Algorithms and their Applications, San Diego, California, USA, 1991.

[2] E. K. Burke, A. J. Smith. *A memetic algorithm for the maintenance scheduling problem*. In Proceedings of the International Conference on Neural Information Processing and Intelligent Information Systems, volume 1, pages 469--472. Springer, 1997

[3] P. Galinier and J. K. Hao. *Tabu search for maximal constraint satisfaction problems*. In Proceedings of CP'97, G. Smolka ed., LNCS 1330, pp. 196-208, Schloss Hagenberg, Austria, Springer, 1997.

[4] W. D. Harvey. *Nonsystematic Backtracking Search*. PhD thesis, Stanford University, 1995

[5] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. *Optimization by Simulated Annealing*. Science, Number 4598, pp. 671-680, 1983

[6] J.-M. Labat, L. Mynard, *Oscillation, Heuristic Ordering and Pruning in Neighborhood Search*. In Proceedings of CP'97, G. Smolka ed., LNCS 1330, pp. 506-518, Schloss Hagenberg, Austria, Springer, 1997.

[7] K. Marriot, P. J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998

[8] Z. Michalewicz, D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2000

[9] Z. Škvor, K. Hoffmann, J. Tomandl, Z.Medek. *Computer aided design of radiofrequency and microwave circuits.* Radioengineering, 2 (1993) 1, pp. 2-5

[10] Z. Škvor. *CAD pro VF techniku.* Skriptum ČVUT, 1999

[11] D. Whitley. *A Genetic Algorithm Tutorial.* Technical report, Computer Science Department, Colorado State University, 1997.

[12] A. H. Wright. *Genetic algorithms for real parameter optimization.* Foundations of Genetic Algorithms, pages 205--218. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[13] *The Java Tutorial*, http://java.sun.com/docs/books/tutorial/index.html

[14] *JSDK (Java Software Development Kit) 1.4.1 Documentation*, http://java.sun.com/j2se/1.4.1/docs/api/index.html

Main part of this diploma thesis, describing the evolutionary algorithm and some of its practical results, was submitted to the COMITE'03 conference. Initial work, the evolutionary algorithm for design sequence coaxial filters was presented at CAD&CAE workshop.

[15] T. Müller, Z. Škvor. *Evolutionary Design of Microwave Circuits*. Submitted to COMITE'03 conference, Pardubice, September 2003

[16] T. Müller. *Evoluční návrh filtru.* CAD&CAE workshop, Katedra elektromagnetického pole, FEL ČVUT, Praha, September 2002

Appendix A User Documentation

A.1. Prerequisites

Java Runtime Environment (JRE) or Java Software Development Kit (JSDK) version at least 1.4.0 has to be installed on the target machine. An installation kit of JRE 1.4.1 is included on the CD-ROM, file *extra\java\j2re-1_4_1_02-windows-i586-i.exe*.

A.2. Installation

Because the program needs to write some temporrary files during the design of microwave circuits, it cannot be executed directly from the CD-ROM. The program is located in the folder *edmc* on the CD-ROM. This folder should be copied from CD-ROM to hard drive before execution (e.g. *c:\edmc*).

Before the first execition, file *edmc.bat* has to be adopted according to the JRE or JSDK installation. Environment variable JAVA_HOME has to be set. For example, if you have Java installed in folder c:\j2sdk1.4.1, change fourth line of *edmc.bat* to:

```
SET JAVA_HOME=c:\j2sdk1.4.1
```

Figure A.1. Set JAVA_HOME environment variable

A.3. Usage

When the program is installed, it can be executed via *edmc.bat*. It has three parameters:

```
edmc.bat input.edmc output.cad output.csv
```

Figure A.2. Program usage

First parameter denotes the input file (with the configuration and requirements, see section A4). Output circuit is stored in the file denoted by the second parameter. Characteristics of the output circuit are stored in the file denoted by the third parameter. During the search, the best ever found solution is stored in these output files.

A.4. Input File

Input file is a text file. It contains several sections started by section name and finished by keyword END. Empty lines or lines starting with star character are ignored.

A.4.1. Section CONFIGURATION

Initial program configuration is stored in this section. It has simple structure, each line represents one parameter. It consists from the parameter name followed

by space and the value. A simple example with only a number of input/output ports (equal to two) is shown in Figure A.3. Redundant spaces are ignored.

```
CONFIGURATION
  NR_GATES 2
END
```

Figure A.3. Section CONFIGURATION

The following table (Figure A.4) describes all available parameters, which can be set and their default values.

Parameter	**Default Value**	**Description**
NR_GATES	2	Number of input/output ports minimum is 2, maximum is 4
FREQ_MIN	*Minimal used frequency*	Minimum frequency for the output characteristics
FREQ_MAX	*Maximal used frequency*	Maximum frequency for the output characteristics
FREQ_STEP	*0.01e9*	Frequency step for the output characteristics
ACCEPT_TYPE	*3*	Type of acceptance criterion 0 .. only better solution is accepted 1 .. random walk 2 .. simulated annealing 3 .. stochastic hill climbing
ACCEPT_PROB	*0.005*	Random walk: probability of acceptance a random step
ACCEPT_TMAX	*1.0*	Simulate annealing or stoch. hill climb.: Initial (maximal) temperature
ACCEPT_TMIN	*1e-4*	Simulate annealing or stoch. hill climb.: Minimum temperature
ACCEPT_ITERS	*100*	Simulate annealing or stoch. hill climb.: First number of iterations: $T = T_{\max} - \left((T_{\max} - T_{\min})/N_{iters}\right) \cdot i$, otherwise $T = T_{\min}$
BT_MAX_DEPTH	*1000*	Backtrack improvement: Maximal number of choice points
BT_ACCEPT	*0.01*	Backtrack improvement: Required solution improvement (in percentage) to be accepted (percentage).
BT_MAX_ITERS	*30*	Backtrack improvement: Number of iteration between choice points

(continues on the next page)

BT_MAX_CHOICES	*10*	Backtrack improvement: Maximum number of choices (back jumps) for a single choice point
MAX_RESTARTS	*7*	Maximum number of restarts (when no solution is found) – restarted search starts from best solution ever found
OPT_MAX_TRY	*10*	Local parameter optimization: Number of tries
OPT_ACCEPT	*0.001*	Local parameter optimization: Required improvement to increase number of tries
OPT_ADD_TRY	*25*	Local parameter optimization: When a solution is accepted and better enough during local optimization – number of additional tries
MAX_RMS	*2.0*	Current solution limitation: If the RMS error of the new solution is greater than MAX_RMS multiplied by RMS error of the best solution ever found – it is not accepted
BT_MAX_RMS	*1.2*	Backtrack improvement: Limitation for new choice points – if a solution is much worse than the best solution found (BT_MAX_RMS times), it is not accepted
PREJ_USE	*0*	RMS Error Estimation: 0 .. feature disabled, 1 .. feature enabled
PREJ_MAX	*0.9*	RMS Error Estimation: Maximum rejection probability
PREJ_MIN_PREV	*3*	RMS Error Estimation: Minimal required number of operations with the same descriptor in statistics for estimation.
PREJ_TYPE	*2*	RMS Error Estimation: Type of rejection criterion 0 .. worse solution is rejected 1 .. random walk (worse solution) 2 .. simulated annealing 3 .. stochastic hill climbing
PREJ_RW	*0.005*	RMS Error Estimation: Random walk: probability of acceptance of worse solution
PREJ_TEMP	*10.0*	RMS Error Estimation: Temperature for simulated anneal. or stoch. hill clim. (remain constant during the search)

Table A.4. Available configuration parameters

A.4.2. Section LINKS

This section describes a set of available elements for edges, see example A.5. Currently implemented available elements are: CAP, IND, RES, MISTR, MGAP, 2PORT, OSTUB, PPRC, PSRC, RST4, SESTO, SESTS, SPRC, SSRC, SSTUB. An element, listed in the section NODES can be also used, if it has two ports. Element 2PORT represents arbitrary element with two ports described by S-parameters table (see folder *edmc\data*).

```
LINKS
  RES
  2PORT data\\Mgf1801.spr
  my.elements.MYCAP
END
```

Figure A.5. Section LINKS

If the element is not included in the package *edmc.elements*, its location has to be denoted in the list (in java notation, e.g. *my.elements.MYCAP*, where *MYCAP* is the name of the class representing such element and *my.elements* is the package, where the class *MYCAP* is located)

A.4.3. Section NODES

This section describes a set of available elements for vertices, see example A.6. Currently implemented available circuit elements are: OPEN, SHORT, ITEE, ICROS, ICIR, LOAD, MCPL. There are also several special elements based on MCPL element with the fourth port opened (MCPL_OPEN) or shortened (MCPL_SHORT) and with the second and the fourth element opened or shortened (MCPL_OPEN_OPEN, MCPL_OPEN_SHORT, MCPL_SHORT _OPEN and MCPL_SHORT_SHORT). Also arbitrary element listed in section LINKS can be used (e.g. MGAP).

```
NODES
  SHORT
  ITEE
  ICROS
  MGAP
END
```

Figure A.6. Section NODES

If the element is not included in the package *edmc.elements*, its location has to be denoted in the list (java notation, e.g. *my.elements.MYCAP*, where *MYCAP* is the name of the class representation and *my.elements* is the package, where the class *MYCAP* is located)

A.4.4. Section OPERATIONS

This section describes operations, which can be used and their initial weights. Available operations are listed in the following table (Figure A.7).

Operation	Description
LinkParamModifier	O1, changes a single parameter of an element assigned to the arbitrary edge
NodeParamModifier	O1, changes a single parameter of an element assigned to the arbitrary vertex
LinkElementChanger	O2, changes an element assignment of an arbitrary edge
NodeElementChanger	O2, changes an element assignment of an arbitrary vertex
AddLink	O3, adds an edge between two suitable vertices
RemoveLink	O4, removes suitable edge (graph has to stay connected)
SplitNode	O5, splits a suitable vertex into two vertices connected with an edge
AddNode	O6, add a new vertex with an edge and connect it to a suitable vertex
SplitLink	O7, splits an edge into two connected edges
SplitLinkT	O7+O6, splits an edge into two connected edges, connects another new vertex to the created vertex
RemoveNode	O8, removes a vertex with degree 1 and the appropriate edge
RemoveNode2	O9, removes a vertex with degree 2, join the appropriate edges into one
JoinNodes	O10, joins two suitable vertices connected by an edge into the one vertex

Figure A.7. Available circuit operations

An example of the section OPERATIONS is shown in Figure A.8. If there is an operation not contained in *edmc.operations* package, name with package has to be use, e.g. *my.operations.MyOperation*)

```
OPERATIONS
 LinkParamModifier 50
 NodeParamModifier 50
 LinkElementChanger 20
 NodeElementChanger 10
 SplitLink 6
 SplitLinkT 4
 SplitNode 6
 AddLink 4
 AddNode 4
 RemoveLink 5
 JoinNodes 5
 RemoveNode 10
 RemoveNode2 7
END
```

Figure A.8. Section OPERATIONS

A.4.5. Section OPT

Section OPT describes the input requirements of the designed microwave circuit. There are three types of requirements available: a requirement on a single frequency, on a linearly distributed set of frequencies (keyword STEP) and on a logarithmically distributed set of frequencies (keyword OPT).

A requirement consists from a parameter (e.g. transmission of a circuit), an operation (e.g. less then), a value (e.g. -25 dB) and a weight of the requirement (default weight is 1.0). A parameter consists from a type and from the row and column of the S-matrix (denoted by type, character 'S', row and column - e.g. transmission from first to second gate of a circuit expressed in dB is denoted by DS21). Available types are denoted in Figure A.9 and available operations are denoted in Figure A.10. An example of requirements is shown in Figure A.11.

Type	Description
A	Magnitude
F	Phase
D	Magnitude in dB
R	Real part
I	Imaginary part
G	Operational transmission (S21*S21)
K	Rollet stability factor

Figure A.9. Types of requirement parameters

Operation	Description
?	No requirement
=	Equal
>	Greater than
>=	Greater or equal
<	Less than
<=	Less or equal

Figure A.10. Requirement operations

```
OPT
  1.0 GHz DS21 < -25
  1.2 GHz DS21 < -25 w 5
  STEP 100 MHz 250 MHz 10 MHz AS21 > 0.9
  STEP 100 MHz 250 MHz 10 MHz DS11 < -33 w 2.3
  LOG 1 GHz 29 GHz 12 RS12 > 0
END
```

Figure A.11. Requirements example

So, a single requirement is described as a line in the OPT block, it consists of frequency specification: single frequency, linearly distributed set of frequencies described by STEP and their minimum, maximum and distance between them (e.g. 100, 110, 120 MHz in the above example) or logarithmically distributed set of frequencies described by LOG and their minimum, maximum and their amount

(plus one – according to the specification in MIDE) (e.g. thirteen frequencies 1, 1.324, … 29 GHz in the above example).

A.5. Output Files

There are two output files: a CAD file with a description of the resultant circuit in the net-list representation (see example in Figure A.12) and the CSV file (table) with the characteristics of the resultant circuit (see example in Figure A.13).

```
*Created: Wed Mar 26 17:36:52 CET 2003
*RMS Error: 0.00000E00 dB
*Total time: 163.07 s
*MIDE Analysis time: 138.00 s (84.62%), called 5396x (one
execution aprox. 25.57 ms)

*Requirements:
*  STEP 1700 MHz 1700 MHz 1000 kHz DS21 > 17.0
*  STEP 1700 MHz 1700 MHz 1000 kHz AS11 < 0.01
*  STEP 1700 MHz 1700 MHz 1000 kHz AS22 < 0.01

* 81 frequenencies:
FREQ
STEP 1500 MHz 1900 MHz 5 MHz
END

BLO edmc 3 6
MISTR 3 4 W=540 mim L=45.64546 mm
RES 5 6 R=9.01785 ohm
ICROS 1 7 12 25
2PORT 1 2 data\\Mgf1801.spr
RES 7 8 R=521.31844 ohm
2PORT 11 12 data\\Mgf1801.spr
MISTR 25 26 W=1.52669 mm L=15.28378 mm
SHORT 26
ITEE 4 8 21
RES 21 22 R=12.13988 ohm
ITEE 2 15 5
MISTR 15 16 W=1.26851 mm L=31.1886 mm
RES 17 9 R=218.32565 ohm
2PORT 9 10 data\\Mgf1801.spr
ICROS 17 16 19 23
RES 19 20 R=83.7481 ohm
MISTR 23 24 W=2 mm L=14.99323 mm
SHORT 20
ITEE 22 13 11
RES 13 14 R=1002.3585 ohm
SHORT 24
SHORT 10
SHORT 14
END
```

Figure A.12 Example output CAD file

freq[Hz]	DS11[dB]	DS12[dB]	DS21[dB]	DS22[dB]
1,500E+09	3,354207	-51,5210	45,86762	-9,83428
1,505E+09	3,695656	-51,6891	45,58129	-9,80518
1,510E+09	3,936849	-51,9098	45,24109	-9,83215
1,515E+09	4,086431	-52,1783	44,85206	-9,91719
1,520E+09	4,154911	-52,4870	44,42170	-10,0578
1,525E+09	4,153712	-52,8272	43,95897	-10,2490
...	...	...	...	...

Figure A.12 Example output CSV file

Appendix B Program Extension Examples

In this appendix, two examples of creating a new circuit element and a new circuit operation are presented.

B.1. New Circuit Element Implementation

Each circuit element, which can be used in the design, has appropriate element in the MIDE's net-list and also appropriate class implemented. Such class has to be extended from the abstract class *edmc.elements.AbstractElement*. It has to have several methods implemented:

Method	**Description**
constructor	Creates an element, assign values to all parameters.
getNrGates()	Returns number of ports of the element (e.g. 2 for resistor)
getNrParams()	Returns number of parameters of the element (e.g. 1 for resistor – its resistance)
getParameterBounds(i)	Returns i-th parameter‘s bounds
getName()	Returns name of the element in MIDE
getParameter(i)	Returns value of the i-th parameter
setParameter(i,val)	Sets value of the i-th parameter
toString(int[] nodes)	Return net-list representation of the element (as a string)
getWidthParam(int i)	For micro-stripes: returns the number of the parameter which represents width for the i-th port (or -1 otherwise – e.g. if there is no width)

Figure B.1. Element's API

Probably the most interesting part of the API is the method *toString(int[])*. It has one parameter – the array of numbers – which are assigned to the appropriate ports of the element. Result is a string, appropriate line of the net-list representation.

The following example B.2 shows a complete implementation of the SPRC element.

```
/** Circuit element SPRC */
package edmc.elements;

import edmc.util.*;

public class SPRC extends AbstractElement {
  /** Parameters bounds - consit from name, unit, min, max
    * and implicit value */
  protected static ParameterBounds BOUND_IND
       = new ParameterBounds("L","H",1e-19,1400,2e-9);
  protected static ParameterBounds BOUND_CAP
       = new ParameterBounds("C","F",1e-33,1e-3,1e-13);
  protected static ParameterBounds BOUND_RES
       = new ParameterBounds("R","ohm",1e-17,5e34,1.1e34);

  /** Parameters L, R, C */
  double iL,iR,iC;

  /** Constructor */
  public SPRC() {
      iR = ToolBox.random(getParameterBounds(0));
      iL = ToolBox.random(getParameterBounds(1));
      iC = ToolBox.random(getParameterBounds(2));
  }

  /** Number of ports */
  public static int getNrGates() { return 2;}

  /** Number of parameters */
  public static int getNrParams() { return 3;}

  /** Returns bounds of i-th parameter */
  public static ParameterBounds getParameterBounds(int i) {
      switch (i) {
          case 0 : return BOUND_RES;
          case 1 : return BOUND_IND;
          case 2 : return BOUND_CAP;
          default : return null;
      }
  }

  /** Returns width parameter for i-th port ... -1 not used */
  public static int getWidthParam(int node) {return -1;}

  /** Returns name */
  public String getName() {
      return "SPRC";
  }

  /** Returns net-list representation */
  public String toString(int[] nodes) {
      return getName()+" "+nodes[0]+" "+nodes[1]+
             " R="+ToolBox.formatValue(iR)+"ohm"+
             " L="+ToolBox.formatValue(iL)+"H"+
             " C="+ToolBox.formatValue(iC)+"F";
  }
```

(continues on the next page)

```
  /** Return string representation - for debug purposes */
  public String toString() {
      return getName()+" R="+ToolBox.formatValue(iR)+"ohm"+
                        " L="+ToolBox.formatValue(iL)+"H"+
                        " C="+ToolBox.formatValue(iC)+"F";
    }

  /** Returns value of i-th parameter */
  public double getParameter(int i) {
      switch (i) {
          case 0 : return iR;
          case 1 : return iL;
          case 2 : return iC;
          default: return 0;
      }
  }

  /** Sets i-th parameter */
  public void setParameter(int i, double value) {
      switch (i) {
          case 0 : iR=value; return;
          case 1 : iL=value; return;
          case 2 : iC=value; return;
      }
  }
}
```

Figure B.2. SPRC circuit element representation

B.2. New Circuit Operation Implementation

Available circuit operation are located in the package *edmc.operations*, each circuit operation has to be extended from abstract class *edmc.operations. CircuitOperation*. There has to be following four methods implemented:

Method	Description
boolean process()	Selects operation‘s operands, applies the operation to the current circuit and returns true. If there are no suitable operands (e.g. no edge can be removed) it returns false (operation was not applied).
boolean redo()	Local optimization: it changes parameters affected by previous call of method process (or redo, when the changes are accepted). (redo can be called several times)
good()	Called when changes made by process or redo are accepted.
toString()	Returns operation’s classification (descriptor)

Figure B.3. Operation’s API

So, in each iteration step, the current circuit is memorized first. An operation is selected and the appropriate method *process* is called then. If it returns false, nothing was changed and the algorithm returns to the selection of an operation. Otherwise, the changed circuit is evaluated and if the local optimization is used (as described in section 2.7.1) the method *redo* is called several times – parameters changed by method *process* are optimized. If the changed circuit is accepted, method *good* is called (can be called after *process* method and each *redo*). This affects the mean values of the distribution used the selection of

parameters' values (previous accepted values are used). Last accepted circuit or the previous one (if there was no circuit accepted) is used for the next iteration.

The following example B.4 shows a complete implementation of the O6 operation: **A single-edge vertex is created and connected with an edge to suitable vertex** (with less then four connections, not i/o vertex). Suitable elements are assigned to the changed vertex and to the created edge and vertex.

```
/** Operation O6: A single-edge vertex is created and connected
  * with an edge to suitable vertex (with less then four
  * connections, not i/o vertex). */
package edmc.operations;

import edmc.circuits.*;
import edmc.design.*;
import edmc.util.*;
import java.util.*;

public class AddNode extends CircuitOperation {
  /** For classification descriptor - selected vertex */
  private String iFromStr;
  /** Selected suitable vertex*/
  private Vertex iSelectedVertex;
  /** New created vertex */
  private Vertex iNewVertex;
  /** New edge between selected and new vertex */
  private Edge iNewEdge;

  /** Constructor */
  public AddNode(Designer designer) {
    super(designer);
  }

  /** perform operation, returns false of operation was not made*/
  public boolean process() {
    //Select a suitable vertex
    Vector suitableVertices = new Vector();
    for (Enumeration e=getCircuit().getGraph().getVertices();
        e.hasMoreElements();) {
            Vertex v = (Vertex)e.nextElement();
            if (v.countEdges()<4 && !getCircuit().isIO(v))
                  suitableVertices.addElement(v);
        }
    if (suitableVertices.size()<1) return false; //no suitable v.

    iSelectedVertex = (Vertex)getDesigner().getVertexSelector().
                        select(suitableVertices);
    if (iSelectedVertex==null) return false; //no vertex selected

    iFromStr = iSelectedVertex.getName(getCircuit());
```

(continues on the next page)

```
    // select classes for selected vertex and new edge & vertex
    Class selectedVertexClass =
       (Class)((NodeElementSelector)getDesigner().
         getNodeElementSelector()).
         select(iSelectedVertex.countEdges()+1);
    Class newVertexClass =
        (Class)((NodeElementSelector)getDesigner().
          getNodeElementSelector()).select(1);
    Class newEdgeClass =
        (Class)getDesigner().getLinkElementSelector().select();
    if (selectedVertexClass==null || newVertexClass==null ||
        newEdgeClass==null) return false; //element not selected

    // alter graph
    iNewVertex = getCircuit().getGraph().createVertex();
    iNewEdge = getCircuit().getGraph().
         createEdge(iSelectedVertex,iNewVertex);

    // create & assign elements
    iSelectedVertex.setElement(
          Circuit.createElement(selectedVertexClass));
    iNewVertex.setElement(
           Circuit.createElement(newVertexClass));
    iNewEdge.setElement(
           Circuit.createElement(newEdgeClass));

    return true;
  }

  /** alter modification - alter another elements */
  public boolean redo() {
    // select classes for selected vertex and new edge & vertex
    Class selectedVertexClass =
       (Class)((NodeElementSelector)getDesigner().
         getNodeElementSelector()).
         select(iSelectedVertex.countEdges());
    Class newVertexClass =
        (Class)((NodeElementSelector)getDesigner().
          getNodeElementSelector()).select(1);
    Class newEdgeClass =
        (Class)getDesigner().getLinkElementSelector().select();
    if (selectedVertexClass==null || newVertexClass==null ||
        newEdgeClass==null) return false; //element not selected

    // recreate & reassign elements
    iSelectedVertex.setElement(
          Circuit.createElement(selectedVertexClass));
    iNewVertex.setElement(
           Circuit.createElement(newVertexClass));
    iNewEdge.setElement(
           Circuit.createElement(newEdgeClass));

    return true;
  }

  /** redo or process accepted notification */
  public void good() {}
```

(continues on the next page)

```
  /** returns operation classification */
  public String toString() {
    return iFromStr+" -> "+
      iSelectedVertex.getName(getCircuit())+"-"+
      iNewEdge.getName(getCircuit())+"-"+
      iNewVertex.getName(getCircuit());
  }
}
```

Figure B.4. Implementation of operation O6 – class AddLink

Appendix C CD-ROM Content

This diploma thesis includes a CD-ROM with electronic form of this thesis, implemented program, program documentation and source codes and several examples. The papers [15], [16] are also included.

Folder or file	**Content**
\thesis\thesis.pdf	This diploma thesis in PDF format
\thesis\commite03.pdf	Paper [15], *"Evolutionary design of Microwave Circuits"*
\thesis\cadcea02.pdf	Paper [16], "*Evoluční návrh filtru"*
\edmc	Implemented program
\examples\amplifier	Example: amplifier
\examples\filter	Example: pass band filter
\examples\splitter	Example: frequency splitter
\src\src.zip	Program source codes
\doc\index.html	JavaDoc (source) documentation
\extra\java	Java Runtime Environment 1.4.1
\extra\acrobat	Acrobat Reader 5

Figure C.1. Included CD-ROM Content