An Extensible TCAD Optimization Framework Combining Gradient Based and Genetic Optimizers

Clemens Heitzinger[a] and Siegfried Selberherr[b]

[a,b] Institute for Microelectronics, TU Wien, Gußhausstraße 27–29, Vienna, Austria

ABSTRACT

Our Simulation Environment for Semiconductor Technology Analysis (SIESTA) is a flexible, user programmable tool for optimization and inverse modeling of semiconductor devices. It is easily customizable through an interactive, object-oriented and functional scripting language. Dynamic load balancing enables to take advantage of a cluster of hosts with minimal requirements on the software infrastructure. Our approach combines the advantages of gradient based and evolutionary algorithm optimizers into one framework. Gradient based optimizers are well-suited for finding local extrema. Evolutionary algorithm optimizers add the capability of finding global extrema and thus make unattended optimizations without guessing starting values possible. Experiments can be interactively set up and tested. Bindings for the most common simulation tools are provided, and new bindings can easily be integrated taking advantage of the object-oriented and functional design. Results of experiments are saved in an object database and can be interactively retrieved as starting points for further computations or for visualizations. The user may impose arbitrary constraints (as functions defined on the parameter space) on the set in which solutions are searched. Evaluating the constraints before any simulation tool is called and getting rid of useless combinations of parameter values saves computation time and eliminates the risk of the simulation tools being called with input values that might lead to unforeseen behavior. The combination of gradient based and evolutionary algorithm optimizers enables many new optimization strategies and includes convenient handling of results.

Keywords: TCAD, optimization, evolutionary computation, genetic optimization, gradient based optimization, inverse modeling

1. INTRODUCTION

Technology CAD (TCAD) tools like SIESTA[1,2] have been successfully used for optimizing semiconductor devices[3] and for inverse modeling.[4] Although SIESTA proved to be a valuable tool and several interesting results[5,6] were achieved using it, it did not, at that time, provide any global optimizer, but two gradient based optimizers.[7] The most recent advances include new, global optimizers, combine these two approaches to optimization, and improve on the flexibility and extensibility.

Over the years, it has been recognized that a successful TCAD optimization framework has to meet the following criteria.

1. The ability to execute simulation tools on a number of computers in the local network, and to schedule the execution of these simulation tools in reaction to changes in this network. For example, in a heterogeneous network that is not solely dedicated to executing simulation tools, the question how the tasks have to be scheduled so that the overall execution time is minimized is not trivial. Furthermore, software and hardware failures have to be taken into account.

2. Stability. This property is crucial for a program that usually runs for several days and has to deal with all kinds of software failure.

3. Extensibility. A TCAD framework has to deal with various programs,[8] various data formats[9] and combinations thereof. Evaluating the goal function (i.e., the function to be optimized) often entails several calls to simulation tools. Because of the abundance of possible goal functions, a framework has to provide a flexible extension language which enables the user to succintly describe the desired goal function.

Further author information: (send correspondence to C. Heitzinger)
C. Heitzinger: E-mail: heitzinger@iue.tuwien.ac.at
S. Selberherr: E-mail: selberherr@iue.tuwien.ac.at

4. Specialized Optimizers. The evaluation of the goal function is usually very expensive: times range from about a minute to one hour or more for process simulations on current hardware. Strategies for finding global extrema of computationally very expensive functions are needed. The respective advantages and disadvantages of gradient based optimizers and evolutionary computation will be discussed in Sect. 4.

5. Finding a suitable starting value is often the most difficult and time consuming task when using a gradient based optimizer. Hence global optimizers which do not need a starting value sufficiently near to the global extremum are called for.

It should be noted that the goal function of an optimization in TCAD analysis may not even be a function in the mathematical sense. Simulation tools like MINIMOS[10,11,12] provide modes of operation which are not deterministic, i.e., the same input may lead to slightly different results. In the case of MINIMOS this is due to preconditioning which depends on the elapsed simulation time. Furthermore, it happens in practice that simulation tools do not converge for certain inputs, or yield results only after consuming exceptionally long computation time.

2. DESIGN AND IMPLEMENTATION

SIESTA was originally implemented in a dialect of Lisp[13,14] called xlisp.[15] Because of new developments in language design and implementation since xlisp was written, the choice of a suitable base language for SIESTA was thought over. In order to meet the requirements described in Sect. 1, we posed the following demands on a suitable base language.

1. It has to provide an interface to the underlying operating system and network.
2. There has to be one (preferably more) stable and well supported implementation.
3. It has to support multiprocessing (or multithreading).
4. An extension language is necessary in order to provide the required flexibility.
5. The implementation has to be able to load additional code at run time and to enable the user to execute commands interactively.
6. It should be well designed and preferably standardized.

After evaluating several languages, namely C, C++, Common Lisp, Java, Perl, Python, Scheme, and Tcl, we decided to use Common Lisp[16] with multiprocessing support. In addition to fulfilling all of our requirements, it provides the following features which helped reducing the implementation time.

1. Common Lisp supports the paradigms of functional programming and of object oriented programming.
2. All language constructs are available at run time.
3. Several implementations on all major platforms are available and all of these provide compilers and interactive listeners.
4. A powerful macro system makes Common Lisp a very extensible programming language.
5. Its condition system and operators like `unwind-protect` and `ignore-errors` contribute to stability and robustness.
6. Common Lisp is an ANSI standard.

SIESTA runs on UNIX platforms, since UNIX provides good support for executing commands on remote computers and distributing files in a cluster of computers. Apart from these requirements, SIESTA is platform independent.

The requirements on the software infrastructure installed on the cluster of computers to be used in an experiment have been reduced to a minimum. Early versions of SIESTA required that the user's home directory is visible on all computers of the cluster and relied on NFS (network file system). NFS, however, is a source of problems since it does

not provide sufficient synchronization between the state of files on different computers. Files are synchronized only after a pause which leads to problems when the result file of a simulation tool exists on a client, but is not seen on the computer where SIESTA runs. Furthermore, it is not possible to request a synchronization between two computers manually.

Several solutions for that problem were tried, yet none worked satisfactorily. One attempt was to wait a certain amount of time (up to thirty seconds) after a simulation tool had finished. Because of this limitation of NFS, we decided to dispense with it and use `rcp` or `scp` instead.

In order to run SIESTA, the following programs have to be installed on a cluster.

- One of these possiblities for communication has to be chosen:
  - `rsh`, `rcp`, and `rshd`. These are standard on UNIX, but not secure.
  - `ssh`, `scp`, and `sshd`. These are not found on every UNIX system by default, but are secure.

  Of course these programs have to be set up such that no password entry is needed for each individual login, but only once per session.
- Standard UNIX commands like `kill`, `mkdir`, `rm`, `top`, and `uptime`.
- Finally it goes without saying that the simulation tools to be used have to be installed. Licenses are managed as described in Sect. 3.6.

3. SEAL (SIESTA EXTENSION AND APPLICATION LANGUAGE)

To outline the capabilities of SIESTA, we give short summaries of some of the most important concepts of its extension language. Some examples of its use are given in order to show what the building blocks of more complex experiments look like.

3.1. Parallelization

Because of the need for parallelizing several evalutions of the goal function and thus the simulation tools, we chose to extend the language with a macro called `parallel`. It takes an arbitrary number of expressions as input and returns a list of results after the evalution of all the input forms (which is done in several processes) has finished.

Example.

```
?  (parallel (sleep 1) (sleep 1) (sleep 1))
→  (NIL NIL NIL)
```

This took one second, not three.

```
?  (parallel (sleep 1)
             (parallel (sleep 1)
                       (parallel (sleep 1) (sleep 1))))
→  (NIL (NIL (NIL NIL)))
```

This took one second as well.

The `parallel` macro and its sister function `p-apply` are an important building block of SIESTA and may of course be called by the user.

3.2. Persistent Object Storage

In order to facilitate the configuration and exchange of data (e.g., optimization results, populations from evolutionary algorithm optimizers) between different invocations of SIESTA, we implemented a simple persistent object storage. Using two functions for writing objects to and reading objects from a file, all of SIESTA's internal data-structures can be stored and retrieved in a text based format. A human readable format was chosen in order to enable inspection and changes with ordinary text editors.

The question how to define equality for any two given objects is closely related to storing and copying objects. Of course, the equality relation has to be reflexive, symmetric, and transitive. Furthermore, the equality relation should not be affected by storing and retrieving objects. Since objects usually contain references to other objects, there is no general copy operator which does the right thing in every application (deep copy vs. shallow copy). Thus the copy operator and the question whether a copied object and the original object should be considered equal depend on the problem at hand.

In SIESTA objects with id's, named objects, and appropriate equality and copying operators are defined and can be extended in an object oriented way. In addition to Common Lisp's four equality operators, we provide the equality operator `samep` and specialized copy operators. These operators work with the persistant object storage and SIESTA's data-structures as expected.

3.3. Setting Up Experiments

In this section we show how SIESTA is typically used. When starting up, it reads its initialization file in which the hosts to be used are defined via `define-all-hosts`. The information about the hosts will be used later by the task manager which schedules the execution of the various simulation tools. Hosts may later be enabled and disabled interactively by the user. Furthermore, for every host a function can be provided which decides if a host is usable right now; this is useful when certain hosts must not be used at certain times of day.

Commands are entered in any number of interactive listeners. After loading a file containing the definition of an experiment, the `run` command starts an optimization run. The value returned by `run` can be stored in a file, although the automatically generated log file contains all the results and information about the progress of the optimization. The result can also be used as a starting value for the next run.

Experiments are defined using `define-experiment`. The definition of an experiment consists of an optional description, the list of the free variables, their interval and their default values, the list of user variables which enable sophisticated customized setups, the goal function, the default value of the goal function to be used when no attempt was successful, the constraint function, and the configuration of one or more optimizers.

States correspond to points in the search space or to individuals of the population in the language of evolutionary computation. A state consists of a list of all (free) variables and their respective values, the experiment it belongs to and—after evaluation—the value of the goal function. States can be manipulated with the `make-state`, `copy-state`, `with-state-vars`, `setq-in-copied-state`, and `show-state` operators.

Setting up the evaluation function is usually the hardest part of defining an experiment. `test-run` can be used to evaluate the goal function on the default values interactively and to see if it works satisfactorily.

The following simple example shows the important steps when setting up new experiments.

Example.

```
?  (defun sum-of-all-free-vars (state)
     (reduce #'+ (free-vars state) :key #'value))
→  SUM-OF-ALL-FREE-VARS


?  (define-experiment :e1 (genetic-experiment)
     :description "A simple test experiment.  Optimizes the sum of all free variables."
     :vars '((var1 (interval 0 10) :default 5)
             (var2 (interval 0 10) :default 5)
             (var3 (interval 0 10) :default 5 :free nil))
     :evaluation-function 'sum-of-all-free-vars
     :last-resort 0
```

```
      :optimizer-configuration (make-genopt-configuration :e1
                                                          :minimize-or-maximize :maximize
                                                          :algorithm "GASteadyStateGA"
                                                          :crossover "TwoPointCrossover"
                                                          :population-size 50
                                                          :number-of-generations 12))
→  (#<GENETIC-EXPERIMENT :E1>)

 ?  (test-run :e1)
→  10

 ?  (sum-of-all-free-vars (setq-in-copied-state (make-state :e1) var1 1 var2 2))
→  3

 ?  (run :e1)
    Several lines deleted.
→  ========== Generation 12 ==========
→  Number of requests evaluated:  426
→  Best:  #<REQUEST (id 637) (score 20.0) (vars ((VAR1 10.0) (VAR2 10.0)))>
→  #<REQUEST (id 637) (score 20.0) (vars ((VAR1 10.0) (VAR2 10.0)))>
```

3.4. Calling Simulation Tools

Since several types of software (and hardware) failures may occur when running simulation tools, especially in a networked environment, we extended the base language with a macro called `with-retries`. Its calling signature is `(number-of-tries &body default-forms) &body body`. `with-retries` executes its `body` until no error was raised, but at most `number-of-tries` times. Upon success the result values are those of the last form in `body`, otherwise the values returned by `default-forms`.

The first example tries to execute the body of the call three times and succeeds. The body in the second example is tried three times as well, but call to `error` defeats any success. The same happens in the third example; after three tries the default forms, here a call to `p-norm`, are executed and the resulting values are returned.

Example.

```
 ?  (with-retries (3)
      (p-norm (vector 1 2 3)))
→  3.7416575

 ?  (with-retries (3)
      (error "Foo!"))
→  NIL

 ?  (with-retries (3
                   (p-norm (vector 1 2 3)))
      (error "Foo!"))
→  3.7416575
```

Among the inputs to the call of a simulation tool there is always one state. Calling simulation tools through the preferred interface entails the construction of an instance of class `task`, or a subclass thereof. Tasks contain all the information about calling a simulation tool and returning the required data. While subclasses and methods

specialized to certain simulation tools are provided, the class hierarchy starting at `task` can also be specialized by the user, as well as the methods acting on tasks, e.g., the `execute` method.

The task manager schedules the execution of the tasks. In order to execute a task, it looks for reachable hosts (i.e., hosts whose load average can be retrieved) that are not disabled and that are not too busy. If such a host is available, the task is run on the host which currently provides the most computational resources. Otherwise, it waits until a host becomes available.

The following is an example of calling MINIMOS and extracting a certain vector from a result file. Here `tm-exec` constructs an instance of `minimos-task` and executes it using the task manager. MINIMOS is run at nice level 10 and it will consume at most 120 seconds of cpu time. The input state is constructed from the default values for a certain experiment (we assume this experiment has already been set up). Variables will be substituted in the given input deck (cf. Sect. 3.7) file. The input pif file[17] (containing the device structure) is given as well, and the last option means we request a curve file as result. After the task has finished, we parse the resulting curve file with `parse-curve-file` and return one of its column vectors. Other simulation tools are called similarly.

Example.

```
? (with-scratch-directory (dir)
    (find-column-vector-named
     (parse-curve-file
      (output-crv
       (tm-exec 'minimos-task
                :nice 10
                :max-cpu-time 120
                :input-state (make-state :e12)
                :input-deck #f"~/work/experiment-12/nmos.ipd"
                :input-pif #f"~/work/experiment-12/nmos.pif"
                :output-crv (make-unique-file "crv" :directory dir :pre "crv-"))))
     "Id"))
→ #(-9.203809e-9)
```

3.5. Inverse Modeling

Many models in TCAD applications contain free parameters which depend on properties of the device material and have to be calibrated using measurements. Usually vectors of measured values are fit to characteric curves of the device in question.

It is not obvious which goal function should be used in an inverse modeling experiment where the distance between two vectors (where one is constant) is to be minimized. To facilitate experimentation and allow users to construct a suitable goal function, we provide the `log10`, `euclidean-norm`, `p-metric`, `p-norm`, `append-vectors` and `relative-error` functions.

Example.

```
? (mapcar #'log10 (list 1e10 1e20 1e30))
→ (10.0 20.0 30.0)


? (p-norm (list 1 2 3))
→ 3.7416575


? (mapcar (lambda (p)
            (p-norm (vector 1 2 3) p))
          (list 1/4 1/3 1/2 1 2 3 4))
→ (150.97025 50.7422 17.19151 6 3.7416575 3.3019273 3.1463463)
```

```
 ?  (append-vectors (vector) (vector 1) (vector 2 3) (vector 4 5 6))
→  #(1 2 3 4 5 6)
```

3.6. License Management

When using commercial simulation tools, the number of available licenses for a certain program is often limited. Thus we have to provide a way to ensure that at any point in time only a certain user prescribed amount of licenses of such programs are in use. Users can call simulation tools not only by using the predefined functions of the framework, but also from self written programs like shell scripts which are in turn called from within the framework. A suitable license management scheme has to take this into account.

In order to make our license management scheme meet these needs it works independently from the predefined functions of the framework. The following steps are necessary to use it. Whenever the number of requested licenses exceeds the number of available ones, certain processes have to wait until the required number becomes available. In order to show how many licenses are currently in use, the command `show-licenses` can be used.

1. Define the names of the licenses and how many of each may be used simultaneously. This is accomplished with `define-licenses` and is usually done in your SIESTA configuration file.
2. When using a license, wrap the code into `with-locked-licenses`.

Example.

```
 ?  (define-licenses
      (:dios :total-number 5)
      (:foo :total-number 7))
→  (#<LICENSE :Dios> #<LICENSE :Foo>)
```

The following locks one license.

```
 ?  (with-locked-licenses (:dios)
      (show-licenses))
→  Dios     Total:  5    In use now:  1
   Foo      Total:  7    In use now:  0
```

The following locks four licenses total.

```
 ?  (with-locked-licenses (:dios)
      (with-locked-licenses (:foo 3)
        (show-licenses)))
→  Dios     Total:  5    In use now:  1
   Foo      Total:  7    In use now:  3
```

Users expressed interest in varying the number of available licenses while an optimization is running. This need frequently arises in a setting where people want to reserve one or two licenses for interactive work at certain times, but want all of them to be used, e.g., at night time. For simply changing the number of totally available licenses the function `set-number-of-licenses` can be used in an idle listener.

Example.

```
 ?  (set-number-of-licenses :dios 3)
→  3
```

Later, increase the number of licenses to use again.

```
 ?  (set-number-of-licenses :dios 5)
→  5
```

3.7. Input Deck Handling

Nearly all simulation tools use a text file for configuration. The configuration files of MINIMOS are called input deck[12] files, and we will use this term for all simulation tools. SIESTA generates these files by substituting the values of the variables of a state in template files. If a template file contains a string `<(foo)>` and the value of `foo` in the current state is, e.g., `1.23`, the string will be replaced with `1.23`. This applies to free and user variables of a state.

In case the use of `<(` and `)>` leads to collisions in the input deck file of some simulation tool, the begin and end marker can be changed.

4. GRADIENT BASED OPTIMIZERS AND EVOLUTIONARY COMPUTATION

During the last three or four decades there has been increasing interest in optimization algorithms which work similar to processes found in nature. The methods of genetic algorithms[18,19,20] and evolutionary strategies,[21] although having different roots, have converged and are now commonly known under the name of evolutionary computation. A journal of the same name[22] has been published since 1993.

A brief outline of evolutionary algorithms is as follows. They work with sets (or populations) of potential solutions. Starting from a random population, each individual is assigned a score via a goal function. In the selection step, certain individuals are chosen to proliferate and form a new population. Operators (e.g., mutation) are applied to the individuals of the new population with prescribed probability, and the population is evaluated again. New generations are formed in this way until a termination condition is fulfilled. Obviously, many alternatives for every step in this algorithm exist and have been described and discussed in many publications.[20] Although the Schema Theorem[20] and similar theorems explain how evolutionary algorithms work in a quantifiable way, and many special algorithms have been intensively studied, no consistent theory of evolutionary computation exists to date and the question of which evolutionary algorithm to choose for a given problem can only be answered by experimentation and experience. Nevertheless evolutionary algorithms proved to be very general and valuable tools. While domain specific optimizers typically perform better than their general evolutionary algorithm counterpart, evolutionary algorithms can easily be adapted to the problem at hand and are usable whenever the lack of detailed knowledge about the goal function prohibits developing a domain specific optimization algorithm.

Although evolutionary computation is a well established optimization technique today, its application to TCAD analysis has been limited. Reasons are certainly the need for lots of computational resources and the requirements outlined in Sect. 1. While most research in evolutionary computation has been done on relatively cheap to evaluate goal functions, the optimization of semiconductor devices has to cope with a relatively limited number of evaluations.

The most important difference from the usual practice of evolutionary algorithm optimizers is that runs are usually finished before a common termination condition like "95% of the population are identical" is fulfilled.

In the following we discuss the advantages and disadvantages of gradient based and evolutionary algorithm optimizers and show why the combination of both is worthwhile. An optimization run with an evolutionary algorithm optimizer usually yields a set, or population, of nearly optimal solutions. The best of these is used as the starting point for a run with a gradient based optimizer, thus bringing together global and local optimization methods. Other combinations are also possible, for example: starting populations can be constructed manually; other states than the best in a population may yield better final results because they lie closer to the global optimum; the configuration of an optimizer can be changed and the computation restarted with the latest population or starting point.

4.1. Advantages and Disadvantages of Gradient Based Optimizers

Most importantly, gradient based optimizers are hill climbing algorithms and therefore local optimization techniques. Although very sophisticated algorithms[23] have been developed, they all depend on a suitable starting point. In practice, finding this starting point has been found to be the major hurdle when trying to do unattended, automatic optimizations. Typically finding such a point and shortening the parameter intervals so that the goal function can actually be evaluted requires several tries and can easily take several days.

When increasing the number of variables, the number of evaluations increases as well. While goal functions with few variables are feasible, optimizations with about 20 variables are usually impractical. Evolutionary algorithms do not suffer as much from this effect.

4.2. Advantages and Disadvantages of Evolutionary Algorithm Optimizers

Evolutionary algorithm optimizers are global optimization methods and scale well to higher dimensional problems. They are robust with respect to noisy evaluation functions, and the handling of evalution functions which do not yield a sensible result in given period of time is straightforward.

The algorithms can easily be adjusted to the problem at hand. Almost any aspect of the algorithm may be changed and customized. On the other hand, although lots of research has been done on which evolutionary algorithm is best suited for a given problem, this question has not been answered satisfactorily. Although the standard values usually provide reasonably good performance, different configurations may give better results. Furthermore, premature convergence to a local extremum may result from adverse configuration and not yield (a point near) the global extremum.

5. AVAILABLE OPTIMIZERS

The following brief overview lists all optimizers currently available in SIESTA, namely two gradient based[23] and two stochastic global ones.

5.1. Genopt

The interface to GAlib,[24] a C++ library for genetic optimization, is called genopt. It provides standard selection, crossover, mutation, scaling, and termination methods.[20]

For our experiments we mainly use the following setup, because it provides good results in an acceptable amount of computation time. Since all parameters are reals chosen from intervals, we represent them as floating point numbers, and not as binary vectors as favoured in early genetic optimization. We use a mutation operator which adds a random number from a normal distribution, more precisely, $x \in [a, b]$ is changed to $\min(\max(N(x, \sigma), a), b)$, where σ depends on the length of the interval.

As crossover operators we use two point and uniform crossover. Most populations consist of about 40 to 50 individuals. Some optimization tasks allow us to evaluate about 20 generations per hour, which amounts to roughly 500 generations per day. Typical runs last for two or three days.

Constraints handling is done using the popular penalty method, i.e., the scores of states which do not fulfill given constraints, which are defined as an arbitrary function, are increased by prescribed amounts.

5.2. Siman

Simulated annealing[25,26] was invented by Kirkpatrick in 1982 and is a modified version of hill climbing. Starting from a random point in the search space, a random move is made. If this move yields a better point, it is accepted. If it yields a worse point, it is accepted only with a certain probability $p(t)$ which depends on the time t. The function $p(t)$ is initially close to 1, but gradually reduces towards 0 in analogy to the cooling of a solid. Hence initially any moves are accepted, but as the temperature reduces, the probability of accepting a negative move is lowered. Negative moves are essential sometimes if local maxima are to be escaped, but obviously too many negative moves will simply lead away from an extremum. Versions like fast re-annealing, adaptive annealing and parallel annealing have been developed. In our framework we provide an interface to an implementation[27] by Lester Ingber.

5.3. Donopt

This gradient based optimizer[1,7] minimizes a scalar value and supports equality and inequality constraints. It is based on donlp2[28,29] by Peter Spellucci.

5.4. Lmmin

The Levenberg-Marquardt algorithm[30] is an efficient method to solve nonlinear least squares problems, and is therefore well suited for inverse modeling tasks. SIESTA provides an interface to the implementation found in the MINPACK[31,32] project.

The parameter values are chosen from prescribed intervals. However, arbitrary constraints are not supported by this optimizer. The step size used for the gradient computation and a tolerance value acting as termination criterion can be adjusted.

6. CONCLUSION

SIESTA has been used for several optimizations of real world devices on a cluster of a dozen workstations with 18 cpus and has proven to be very robust and to yield good results. The combination of gradient based optimizers and evolutionary computation allows to take advantage of the benefits of both approaches. While the default configurations of the optimizers provide reasonably good performance, lots of aspects of an optimization run can be customized. The output of one or more optimization runs can be combined and used as input for the next run. This interoperability allows for interesting combinations of optimizers, comparisons of their performance, and specialization to the problem at hand.

ACKNOWLEDGMENTS

The authors acknowledge support from the "Christian Doppler Forschungsgesellschaft", Vienna, Austria. Many fruitful discussions with Thomas Binder and Robert Klima regarding extensions and new features were valuable and always appreciated.

REFERENCES

1. R. Strasser, *Rigorous TCAD Investigations on Semiconductor Fabrication Technology*. Dissertation, Technische Universität Wien, 1999. http://www.iue.tuwien.ac.at/diss/strasser/diss-new/diss.html.
2. R. Plasun, M. Stockinger, R. Strasser, and S. Selberherr, "Simulation based optimization environment and its application to semiconductor devices," in *Intl. Conf. on Applied Modelling and Simulation*, pp. 313–316, (Honolulu, Hawaii, USA), Aug. 1998.
3. R. Plasun, C. Pichler, T. Simlinger, and S. Selberherr, "Optimization tasks in technology CAD," in *9th European Simulation Symposium*, W. Hahn and A. Lehmann, eds., pp. 445–449, Society for Computer Simulation International, (Passau, Germany), Oct. 1997.
4. R. Strasser, R. Plasun, and S. Selberherr, "Practical inverse modeling with SIESTA," in *Simulation of Semiconductor Processes and Devices*, pp. 91–94, (Kyoto, Japan), Sept. 1999.
5. M. Stockinger, *Optimization of Ultra-Low-Power CMOS Transistors*. Dissertation, Technische Universität Wien, 2000.
6. M. Stockinger, A. Wild, and S. Selberherr, "Closed-loop MOSFET doping profile optimization for portable systems," in *Proc. 2nd Intl. Conf. on Modeling and Simulation of Microsystems*, pp. 411–414, (San Juan, Puerto Rico, USA), Apr. 1999.
7. R. Plasun, *Optimization of VLSI Semiconductor Devices*. Dissertation, Technische Universität Wien, 1999. http://www.iue.tuwien.ac.at/diss/plasun/diss-new/diss.html.
8. J. Daniell and S. Director, "An object oriented approach to CAD tool control," *IEEE Trans.Computer-Aided Design* **10**, pp. 698–713, June 1991.
9. T. Binder and S. Selberherr, "Object-oriented design patterns for process flow simulations." to be published in Proc. 4th Annual IASTED International Conference on Software Engineering and Applications, Las Vegas 2000, Nov. 2000.
10. S. Selberherr, W. Fichtner, and H. Pötzl, "MINIMOS - a program package to facilitate mos device design and analysis," in *Numerical Analysis of Semiconductor Devices and Integrated Circuits*, B. Browne and J. Miller, eds., vol. I, pp. 275–279, Boole Press, (Dublin), 1979.
11. T. Simlinger, H. Kosina, M. Rottinger, and S. Selberherr, "MINIMOS-NT: A generic simulator for complex semiconductor devices," in *25th European Solid State Device Research Conference*, H. de Graaff and H. van Kranenburg, eds., pp. 83–86, Editions Frontieres, (Gif-sur-Yvette Cedex, France), 1995.
12. T. Binder, K. Dragosits, T. Grasser, R. Klima, M. Knaipp, H. Kosina, R. Mlekus, V. Palankovski, M. Rottinger, G. Schrom, S. Selberherr, and M. Stockinger, *MINIMOS-NT User's Guide*. Institut für Mikroelektronik, 1998.
13. J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine (part I)," *Communications ACM* **3**, pp. 184–195, Apr. 1960.
14. J. McCarthy, *Lisp 1.5 Programmer's Manual*, MIT Press, 1962.
15. D. Betz, *XLISP: An Object-Oriented Lisp, Version 2.1*. Apple, Peterborough, New Hampshire, USA, 1989.
16. P. Graham, *ANSI Common Lisp*, Prentice Hall, New Jersey, 1996.

17. F. Fasching, C. Fischer, S. Selberherr, H. Stippel, W. Tuppa, and H. Read, "A PIF implementation for TCAD purposes," in *Simulation of Semiconductor Devices and Processes*, W. Fichtner and D. Aemmer, eds., vol. 4, pp. 477–482, Hartung-Gorre, (Konstanz), 1991.
18. D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.
19. J. Holland, *Adaption in Natural and Artificial Systems*, MIT Press, 1975.
20. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, Berlin, 1996.
21. I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, Frommann-Holzboog Verlag, Stuttgart, 1973.
22. K. D. Jong, ed., *Evolutionary Computation*, journal published by MIT Press, 1993.
23. C. Kelley, *Iterative Methods for Optimization*, SIAM, Philadelphia, 1999.
24. M. Wall, "Galib: A C++ genetic algorithm library." `http://lancet.mit.edu/ga/`, 1994.
25. D. Beasley, D. Bull, and R. Martin, "An overview of genetic algorithms: Part 2, research topics," *University Computing* **15**, pp. 170–181, 1993.
26. R. Rutenbar, "Simulated annealing algorithms: an overview," *IEEE Circuits & Devices* , pp. 19–26, Jan. 1989.
27. L. Ingber, "Adaptive simulated annealing." `http://www.ingber.com/#ASA-CODE`, 1993.
28. P. Spellucci, "donlp2 users guide." part of the netlib project, 1995.
29. P. Spellucci, "Solving general convex QP problems via an exact quadratic augmented lagrangian with bound constraints." `http://www.mathematik.th-darmstadt.de/ags/ag8/spellucci`, June 1996.
30. D. Marquardt, "An algorithm for the estimation of nonlinear parameters," *Soc. Ind. Appl. Maths. J.* **11**, pp. 431–441, 1963.
31. J. Moré, B. Garbow, and K. Hillstrom, "Users guide for minpack-1," 1980. Argonne National Laboratory Report ANL-80-74, Argone, IL.
32. J. Moré, D. Sorensen, K. Hillstrom, and B. Garbow, *The MINPACK Project*, Sources and Development of Mathematical Software, Prentice-Hall, Englewood Clifs, NJ, 1984.