An object-oriented simulation-optimization interface

R. Le Riche[1,*], J. Gaudin[2] and J. Besson[3]

[1] Ecole des Mines de St. Etienne / CNRS URA1884,
158 cours Fauriel, 42023 St. Etienne Cedex, France
[*] correspondig author, leriche@emse.fr
[2] EADS CCR, 12, rue Pasteur, 92152 Suresnes Cedex, France
[3] Ecole des Mines de Paris / CNRS UMR7633,
Centre des Matériaux, BP87, 91003 Evry Cedex, France

published in Computers & Structures, Elsevier publ., vol.81, no.17, June 2003, pp. 1689 – 1701.

Keywords: Object oriented programming; Optimization; Composite design; Software engineering; Process optimization; Multiple objective optimization

Abstract

Progress in the field of structural optimization naturally leads to an increasing number of structural models and optimization algorithms that need to be considered for design. Software architecture is of central importance in the ability to account for the complex links tying new structural models and optimizers. An object-oriented programming pattern for interfacing simulation and optimization codes is described in this article. The concepts of optimization variable, criteria, optimizers and simulation environment are the building blocks of the pattern. The resulting interface is logical, flexible and extensive. It encompasses constrained single or multiple objective formulations with continuous, discrete or mixed design variables. Applications are given for composite laminate design.

1 Introduction

As research in the field of structural modelling and optimization progresses, design needs to account for an increasing number of interdependent models and allow diverse problem formulations. The architecture of the computer program that implements structural models and optimization algorithms has therefore become crucial in dealing with such increasing complexity. Deliberate programming is a stepping stone for effectively capitalizing knowledge so that creativity in design is not limited by technical book-keeping.

This article describes a central part of any structural optimization program, the interface between simulation and optimization sub-programs. The interface is presented as an object oriented programming pattern. Unlike sequential programming languages (FORTRAN, C, BASIC, PASCAL, ...), object oriented languages like C++ ([1, 2]) provide ways to express the solutions to a problem directly and

concisely, by creating new relevant types and their functionalities (objects). Object oriented programming patterns ([3, 4]) go one step further: they describe how different objects work together to fulfill a task.

In the field of mechanics, object oriented patterns have mainly covered finite element programming ([5, 6, 7, 8]). The need for object oriented analysis in advanced optimization strategies has been clearly stated and solutions proposed in [9] and [10]. The text starts with a review of simulation-optimization interfaces. Next, a pattern for an internal interface is proposed, which consists of assigning responsibilities to new programming types related to optimization variables, criteria, algorithm and simulation environment. The resulting program is versatile as it allows constrained single or multiple objective formulations with continuous, discrete or mixed design variables. Applications are given for composite laminate design. A coupled process-structure problem and a coupled material selection-structure problem are solved.

2 Interfacing models and optimizers

Simulation programs and optimizers can be interfaced externally or internally.

2.1 External interfaces

The external interface is composed of at least two executable programs, the optimizer and the simulation. The optimizer calls the simulation (system call). Both modules communicate through files. It is illustrated in Figure 1, where x denotes the design variables, f the objective function(s), and g the constraint(s). Such implementation is typical when optimization is not planned beforehand and the simulation source files cannot be modified. It also occurs when resorting to an optimization package (such as [11] or [10]) separated from the simulation software. It usually requires writing a translator subprogram. The translator changes the vector of design variables into an input file that is read by the simulation. Reciprocally, the translator may have to interprete simulation output files in terms of objective function(s) and constraint(s).

The advantage of external interfaces is that, except for the translator, optimizer and simulation implementations are fully decoupled. The external interface has one important drawback: the simulation is entirely repeated at each evaluation of the optimization criteria, including loading of data, even if the design variables that are changed do not require it. Suppose for example that the optimization aims at choosing the material of a mechanical part. It is a waste of computer resources to read the geometry of the part at each evaluation, since it has not changed. Moreover, optimizer and simulation should be able to exchange information other than x, f and g: it is possible that certain combinations of design variables translate into an impossible simulation (consider rigid body motion of a mechanical component for example). The translator can of course read optional messages from the simulation, but this will typically involve tedious file parsing. It is worth mentioning that some software using external interfaces propose versatile ergonomic translators. The DAKOTA program ([10]) provides object oriented utility classes, called `IOFilter`, to implement the translator. Another example is given by the "template mechanism" of [12], sketched in Figure 2. In the template mechanism, input files are copied to template files (by adding `.tmpl` extension), and numerical values that are design variables are changed to identifiers (a `?` followed by a string). The translator, at each evaluation, copies the template files, remove the `.tmpl` extension, and replaces the variables identifiers by a numerical value.

2.2 Internal interfaces

Other implementations, where simulation and optimizer are embedded in the same program have internal interfaces. This is the case of the vast majority of analysis and optimization packages. With internal interfaces, design variables, optimization criteria (objective function and constraints) and optimization algorithms can typically be chosen from a pre-programmed list. For Computer Aided Design software (e.g. [13, 14]) geometrical parameters already exist and are used as design variables. Some software allows both continuous and discrete optimization variables [15]. Results of analyses, such as, in finite elements, displacements, stresses, eigenfrequencies, buckling loads, ..., make up the list of possible optimization criteria. In some implementations ([12, 16]), analytical functions of optimization criteria can define new optimization criteria. With internal interfaces, simulation data are put in memory once and for all and updated only when necessary. Great effort is often put into computing, analytically or semi-analytically, sensitivities of optimization criteria with respect to design variables for use with mathematical programming algorithms ([17, 16]).

To the authors'knowledge, no commercial software cumulates all the above possibilities. The object oriented internal interface proposed next makes it easier to accommodate many optimization approaches while keeping the program manageable.

3 An object oriented simulation-optimization interface

3.1 General programming style

The upcoming interface description uses notions of object oriented programming, pseudo-C++, where many aspects of the language are omitted, and class diagram notation ([3]). Basic utility classes like templatized containers (`LIST<>`), mathematical classes (`VECTOR`), input-output classes (`ASCII_FILE`) are considered self-explanatory and referred to without further explanations (which can be found in [6]). References to general object oriented patterns from [3] are made and corresponding definitions are reproduced.

From a general programming point of view, systematic use of object composition through pointers to other objects is made. This choice is carefully argued in [6], and briefly summarized now.

Object composition is a natural way of implementing the is-made-of relationship between objects. Complemented by single inheritance for representing the is-a relationship, it provides a versatile programming scheme. In most cases, it allows one to avoid, as should be the case, multiple inheritance. In Figure 3 for example, `OPTIMIZER` is seen to create and be made of ($\diamond\!\!\longrightarrow\!\!\bullet$ arrows) one to many optimization variables, `OPT_VARIABLES`. `OPTIMIZER` also has a pointer to a `SIMULATION_ENV` (simple arrow). Object composition is nicely enriched by the abstract factory technique (see Appendix A or [6]), which is a refined way of instantiating pointers at run time.

3.2 Interface implementation

The ingredients of every optimization problem formulation, optimization algorithms, variables and criteria, constitute the base classes of the interface, `OPTIMIZER`, `OPT_VARIABLE`, and `CRITERION`, respectively. Not surprisingly, the DAKOTA project for advanced engineering optimization ([10]) has generated equivalent classes. An additional class proposed here is `SIMULATION_ENV`, which contains the simulation environment. A global view of the interface is given in Figure 3. The role of the base classes is described hereafter. Note that the architecture of the interface, in particular

systematic use of object composition, added to the abstract factory pattern permits the addition of optimization variables, criteria, and algorithms in a non-intrusive way.

OPTIMIZER has two goals. The first is to encapsulate optimization algorithms, the second is to provide storage and functions shared by many specific optimizers.

OPTIMIZER is a family of algorithms that change simulation parameters. It is the base class of MONOEVOLUTION and MULTIEVOLUTION, a single and multiple criterion evolutionary algorithm ([18, 19]), respectively, of GBNM, a globalized and bounded Nelder-Mead algorithm ([20]), and of ENUMERATOR, an algorithm that performs parameter study in the space of design variables. OPTIMIZER encapsulates optimization algorithms in the pure abstract function OPTIMIZER::optimize(), along with the function that reads their parameters, OPTIMIZER::read_convergence(...). The inheritance tree of optimizers is given in Figure 4. This first aspect of OPTIMIZER makes it a strategy pattern: the strategy pattern intends to "define a family of algorithms, encapsulate them, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it", from [3].

The other responsibility of OPTIMIZER is to create design variables, variables, and to keep pointers to the continuous and discrete variables, cont_variables and disc_variables. OPTIMIZER also provides utility functions for all optimizers, such as of_monocriterion, constraints_monocriterion, and of_multicriterion, that calculate the objective function and constraints of mono-criterion optimizers, and the objective functions of multi-criteria optimizers, respectively.

OPT_VARIABLE is the base class of continuous and discrete design variables, CONT_OPT_VARIABLE and DISC_OPT_VARIABLE, respectively.

Its first, obvious, responsibilities are to factorize data and functions common to many optimization variables, which is effectively done by following the mathematical nature of the variables. A continuous variable has a reference value for normalization purpose, minimum and maximum bounds, and accompanying utility functions (norm(), unnorm(),...). A discrete optimization variable has a list of possible values. Figure 5 shows the inheritance tree of optimization variables.

The second purpose of OPT_VARIABLE in the interface is to define how a design variable affects the simulation module. The pure abstract function update_simulation_env() is the definition. The simulation program is accessed through a pointer to SIMULATION_ENV, env. In composite design, for example, a continuous ply angle design variable, CONT_PLY_ANGLE, can alter the stacking sequence through env and change a given ply orientation in the CONT_PLY_ANGLE::update_simulation_env() function. In terms of object oriented patterns, OPT_VARIABLE is an adapter between the OPTIMIZER and the SIMULATION_ENV classes: an adapter creates "a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces", from [3]. OPT_VARIABLE has a virtual verification function, which uses the env pointer and checks that the right simulation modules have been loaded. For example, an injection pressure variable, CRTM_PRESSURE, checks in verification that an injection module has been loaded in SIMULATION_ENV. Figure 5 shows the optimization variables related to composite laminate design that are implemented. Ply orientations, ply material, and number of plies are three kinds of ply variables. They derive from both OPT_VARIABLE and PLY_VARIABLE, which have methods of calculating ply positions.

CRITERION is the base class of optimization criteria, that serve both as objective functions, $\min_x f(x)$, or constraints, $g(x) \leq 0$. CRITERION's primary purpose is to provide a method that fetches needed quantities in SIMULATION_ENV to calculate a specific optimization criterion, hence the pure abstract `double calculate()` function (see Figure 3). Like OPT_VARIABLE, CRITERION is an adapter between OPTIMIZER and SIMULATION_ENV. It contains a number, `ref_value`, that is used for norming the criterion. For example, the longitudinal strain criterion, EPSX_CRITERION, is written $|\varepsilon_x|/\varepsilon_x^{\text{ref}} - 1$, or,

```
double EPSX_CRITERION::calculate()
{  VECTOR v = env->get_result("epsx");
   return fabs( v[0]/ref_value-1. ; }
```

The function `double calculate_unnormed_crit()` calculates a criterion without normalization (for example it returns ε_x in EPSX_CRITERION). 45 criteria have been implemented to date in relation to composite design. They are summarized in Table 1. Like OPT_VARIABLE, CRITERION has a virtual verification function that checks compatibility between a specific criterion and modules loaded in SIMULATION_ENV. For example, it is necessary to make sure that Monte Carlo analysis is executed when a criterion based on standard deviations (`dev_...`) is optimized.

SIMULATION_ENV stands for simulation environment. This class has two goals.

Firstly, SIMULATION_ENV decouples optimization algorithms, OPTIMIZER, from the simulation models by providing a simple interface to the complex, constantly evolving subsystems of the simulation. Parameterized access to all classes making up the simulation is given by the function `VECTOR SIMULATION_ENV::get_result(STRING )`, which extracts data from the simulation and `void SIMULATION_ENV::update_model(STRING, VECTOR )`, which sets data in the simulation. This is made possible by having all simulation classes related to optimization variables and criteria derive from a common IOPARAMETER class:

```
class IOPARAMETER  { ...
  public :
    virtual VECTOR get_result(STRING );
    virtual void update_model(STRING,VECTOR);
    static LIST<STRING> criteria_names, variables_names;
    static LIST<IOPARAMETER*> criteria_objects,
                                        variables_objects;
... };
```

The lists of keywords, `criteria_names` and `variables_names`, and the corresponding lists of pointers to objects, `criteria_objects` and `variables_objects`, are updated by the constructors and destructors of simulation classes deriving from IOPARAMETER. These lists are used in `VECTOR SIMULATION_ENV::get_result(STRING )` and `void SIMULATION_ENV::update_model(STRING, VECTOR )` functions for reaching objects based on the STRING argument. In the mechanical applications considered here, the fundamental simulation classes are the structure, STRUC_TYPE, and the mechanical problem, MECH_SYST. As can be seen in Figure 3, SIMULATION_ENV contains a pointer to a structure and a list of pointers to mechanical problems. Plates or cylinders are examples of

structures. MECH_SYST can be instantiated as process (Compaction Resin Transfer Molding), static (which includes failure) or buckling analyses. A pair of pointers to STRUC_TYPE and MECH_SYST makes a complete simulation formulation. SIMULATION_ENV provides a default view of the possible simulations. In terms of object-oriented patterns, SIMULATION_ENV is a facade ([3]).

Second, SIMULATION_ENV manages memory and computations. The void SIMULATION_ENV::read(ASCII_FILE &) function loads in memory, once and for all at the beginning of the optimization, most of the data describing the problem. This is more efficient than re-loading simulation data at each optimizer iteration as external interfaces do (see section 2.1). Here, at each optimizer iteration, SIMULATION_ENV is minimally updated by OPT_VARIABLE variables which call the void SIMULATION_ENV::update_model(STRING,...) family of functions. Simulation module executions are, typically, the most costly part of an optimization. At each optimization iteration, the simulation module is run once in void SIMULATION_ENV::execute_simulation(). Optimization criteria are calculated afterwards as simulation post-processing computations. SIMULATION_ENV is in charge of optimization criteria calculation using the list of pointers to criteria all_criteria, and the family of functions VECTOR SIMULATION_ENV::get_result(STRING).

Figure 6 shows a sample of the simulation-optimization interface code with the calculations of the objective function and constraints for a monocriterion optimizer. In this code sample, optimization variables and criteria come into play. The calculations rely on void OPTIMIZER::run_simulation(LIST<int>) which first checks that design variables have changed (successive calls to the objective function and the constraints do no require repeated simulations), then, if necessary, updates the simulation environment according to the design variables and executes the simulation and, finally, calculates the optimization criteria.

4 Applications to composite laminate design

Composite laminate design is prone to complex optimization approaches (see [21]) because of the number and the mathematical diversity of parameters that can be changed: the material in each layer (discrete variable), the number of layers (discrete), the orientation of the fibers in each layers (continuous or discrete), process parameters (e.g., injection pressure, which is continuous), ... The variables implemented are shown in Figure 5. Various types of simulations can be performed : compaction and resin transfer module [22], static, failure, buckling and reliability analyses ([23]). The 45 optimization criteria of Table 1 call on these simulations. Beyond the wealth of variables and criteria, couplings between phenomena make composite design both difficult and interesting. An important example is the fiber orientations which have an effect both on the injection and on the structural properties of the laminate. It should also be noted that composite design criteria are often multimodal in terms of the variables, so that global optimization methods are recommended. The complex links between composite laminate analysis and optimization have fostered the development of the aforementioned simulation-optimization interface (which underlies the *LAMKIT* software, [24]).

The following examples illustrate how the proposed interface enriches composite laminate design formulations. To show how the interface facilitates solving difficult problems, the applications given are deliberately complex: the first example is a coupled process, structure, reliability optimization, the second is a multiple objective optimization with material and structural variables. Each problem is tackled through the successive application of different optimizers.

4.1 Optimization input files

Optimization input files closely match the programming of the interface. An example is given in Figure 7. The file starts with `****optimize` followed by the name of an optimizer (`enumerator`, `evolution`, `multievolution`, or `gbnm`) and finishes at `****return`. They are composed of three parts, associated to the base classes of the interface. `***variables` announces the description of the variables, `OPT_VARIABLE`. `***criteria` and `***constraint` start the input of the objective functions and constraints, respectively, both fields instantiating objects derived from `CRITERION`. Finally, `***convergence` precedes the list of parameters associated to the particular optimizer named after `****optimize`.

Optimization variables are dynamically and independently created after reading `**` followed by the variable name. Variables have 2 input fields, the one common to all variables of the same type (continuous or discrete) and announced either by `*continuous` or by `*discrete`, the other specific to the variable and following `*specific`. All continuous variables have a name, minimum, maximum, initial and reference values. All discrete variables have a name, an ordered list of possible values, and an initial value. As an example of specific variable data field, laminate ply variables (deriving from `PLY_VARIABLE`, cf. Figure 5), need a layer position and a laminate type (general, symmetric, or balanced and symmetric).

Optimization criteria keywords and definitions are listed in Table 1. Optimization criteria can represent an objective function f, in which case they follow the `***criteria` card. The reference value of the criterion, which is a real number, comes after the criterion keyword. When optimization criteria are constraints, they follow the `***constraint` card. Two numbers are input after the constraint keyword, the first is the initial value of the associated penalty parameter, the second is the reference value of the criterion.

The list of parameters controlling each optimizer comes after the `***convergence` card. The specific parameters and their meaning, which involves a description of each optimizer, is beyond the scope of this article.

4.2 Coupled process, structure, reliability optimization

The design of a laminate produced by compaction and resin transfer molding ([22]) is considered. The resin is injected on the side of the laminate at an imposed pressure. Compaction starts once 80% of the mold has been filled at a rate of $10^{-4}\ m/s$. One wishes to minimize the total processing time, T_{crtm}, while keeping the injection pressure, P_{crtm}, below $2.10^5\ Pa$ (to prevent the fiber preform from being damaged), and guaranteeing a minimum transverse stiffness, $E_y \geq 40.10^9\ Pa$, with a certain dispersion on E_y, $S(E_y) \leq 8.10^8\ Pa$. Other properties of the material are: $E_1 = 230.\ GPa$, $E_2 = 14.4\ GPa$, $\nu_{12} = 0.32$, $G_{12} = G_{13} = 4.9\ GPa$, the ply thickness is $0.001\ m$, the viscosity of the resin is taken as constant and equal to $0.16\ Pa.s$, the permeability along and transverse to the fiber are $8.93\ 10^{-10}$ and $4.93\ 10^{-10}\ m^2$, respectively. For the reliability analysis, each laminate property is considered as a Normal random variable whose mean is its nominal value and whose standard deviation is 2% of the nominal value. The standard deviation of the criteria is estimated by Monte Carlo analysis with 1000 independent samples.

Figure 7 shows the optimization file. The problem has 5 continuous design variables, the injection pressure, `cont_crtm_pressure`, and 4 continuous ply angles, `cont_ply_angle`. The laminate is balanced and symmetric. Before actually optimizing the laminate composition and process, it is usually desirable to gain some insight into the effect of the design variables on the optimization criteria. A fake optimizer, called `enumerator`, is used to perform the needed parameter study.

Enumerator search the design space following, by default, a linear trajectory between the lower and upper bound of each variable in `no_increments` steps. With the `exhaustive_search` option, it samples the hypercube encompassing the design variables in `no_increments`$^{\text{no. of variables}}$ steps. Figure 8 presents the results of the execution of the enumerator optimizer, i.e., the variation of the criteria T_{crtm}, P_{crtm}, E_y, and $S(E_y)$ as a function of the variables, the injection pressure and the ply angles. Unlike injection pressure, ply angles have an effect both on the process, T_{crtm} and P_{crtm}, and the final structural properties, E_y, and $S(E_y)$. Such graphs shows extremal values of the criteria, but all combinations of variables cannot be explored because of the exponentially increasing price of an exhaustive search.

To obtain an optimal design, one resorts to a real optimizer, the globalized and bounded Nelder-Mead algorithm proposed in [20]. The optimization input file is the same as in Figure 7, except that the keyword `gbnm` replaces `enumerator` and other convergence parameters are set. The optimum solution found after 500 analyses is: injection pressure = $2.\ 10^5\ Pa$, stacking sequence = $(\pm 9.8^\circ / \pm 13.8^\circ / \pm 7.2^\circ / \pm 87.3^\circ)$. The plate is processed in $45.5\ sec$, the maximum pressure during injection and compaction is $2.\ 10^5\ Pa$, and the transverse Young's modulus is $E_y = 6.78 \pm 0.08\ 10^{10}\ Pa$. Upper bounds on the Kuhn and Tucker multipliers[1] are also obtained with this optimizer. They are $\lambda_P = 32.4$, $\lambda_{Ey} = 25.9$ and $\lambda_{S(Ey)} = 54.5$, for the pressure, transverse stiffness, and transverse stiffness standard deviation constraints, respectively. These multipliers, in accordance with the constraints at the optimum, show that the transverse stiffness constraint is the least critical and could probably be relaxed.

4.3 Multiple and single objective structural optimization with material selection

At an early stage in the design process of a structural component, one may wish to have an unbiased insight into the effect of a material choice on the various criteria considered. In such a case, a multiple objective problem formulation is appropriate because it aims at describing all possible compromises between the design criteria. The set of compromises is called Pareto set. A design belongs to the Pareto set if there is no other design with all criteria better. On the contrary, a single objective constrained optimization problem formulation implies emphasising the design criterion chosen as the objective function while limiting values are more or less arbitrarily set for the other criteria. The most general analysis when many criteria are looked after is therefore the multiple objective optimization. Numerically however, the multiple objective is more expensive than the single objective formulation. In a reasonable number of analyses (20000 here), one can only expect to obtain an approximation of the Pareto front. The most promising designs in the Pareto front should then be fine tuned through single objective optimization.

The process of starting with an imperfect multiple objective optimization, selecting interesting regions of the approximated Pareto front and finishing with single objective optimization is readily carried out with the proposed software architecture. The optimization input file is reproduced in Figure 9. Design variables are the material, the fiber orientation and the number of each layer. The material (`ranked_matl` token) can be chosen independently in each layer. In the current example, the material is glass-epoxy (GE), high resistance carbon-epoxy (CEHR) or high module carbon-epoxy (CEHM) (cf. Table 2). The number of layers (`nb_ply` token) is the number of adjacent copies of each layer. Fiber orientations (`disc_ply_angle` token) are, unlike the previous example, discrete and taken among 0°, $\pm 45^\circ$ and 90°. The laminate remains symmetric and balanced during the optimization

[1] Since they are upper bounds, the usual relation $\bar{\lambda} g(x^*) = 0$ does not stand.

thanks to the instruction `lam_type balanced_sym` which makes a ply variable affect two adjacent plies with opposite signs and the two symmetrical plies. Design criteria are the thickness, first ply failure due to strains in the material principal directions, Young's moduli in the x and y directions, and buckling. More details concerning the definitions of the design criteria are given in Table 1 and section 4.1. The plate is $35.\ cm$ long, $25.\ cm$ wide, subjected to a compressive longitudinal load $N_x = -5.10^6\ N/m$ and a tensile lateral load $N_y = 2.857.10^4\ N/m$. The multiple objective algorithm used in the example is the Niched-Pareto evolutionary method ([19]). The "niching" part of this optimizer requires counting neighbors in the criteria space, an aspect of the method which is sensitive to criteria normalization (see the numbers following criteria names in Figure 9).

The solution to this multiple objective problem is a Pareto front in 6 dimensions (the number of criteria). A projection of the estimated Pareto front and the initial population on the $(thickness, buckling)$ criteria plane is shown in Figure 10. The plot shows that the approximated Pareto front carries more designs that strike optimal compromises between $thickness$ and $buckling$ than the initial population does. In particular, thick buckling resistant designs that were unlikely in the initial population (they are created by the cumulated choice of glass-epoxy and a large number of layers for many variables) are much better sampled in the Pareto front. However, the approximated Pareto front, which contains about 800 designs, is too complex to enable any physical interpretation. An algorithm, activated as an optimizer with the `postpareto` keyword, permits selecting the points of the Pareto front according to their criteria values. Designs that do not buckle and that do not have failure and stiffness criteria above 0.5 are selected. The 10 selected designs are summarized in Table 3. All the selected designs but the 6^{th} use the possibility of mixing materials to achieve better compromises. The last design is the only one that neither buckles nor fails in the Table.

This "safe design" is now improved by running a final single objective constrained evolutionary search ([18, 25]). The values of the safe design criteria are transformed into constraints, apart from the thickness which becomes the objective function. The optimization input file is shown in Figure 11. Criteria become constraints simply by moving them from the `***criteria` to the `***constraint` datafield and adding an initial value of the penalty parameter after the criterion token. Parameters of the evolutionary search are found under the `***convergence` heading. If a feasible point is found during the search, it will, by definition of the constraints, improve on the safe design. Such procedure is recommended as approximated Pareto front can miss optimal designs. The evolutionary optimizer finds, after 10000 analyses, the following (feasible) optimum design that mixes high resistance and high module fibers : $((0_{12})^{\text{CEHR}}/(\pm 45)_9^{\text{CEHM}}/(0_{20})^{\text{CEHR}})_s$, which has a total thickness of $H = 1.25\,cm$, and $\lambda_{\text{buckl}} = 2.29$, $\lambda_{\text{maxeto1}} = 1.02$, $\lambda_{\text{maxeto1}} = 1.07$, $E_x = 81.69\,GPa$, $E_y = 25.75\,GPa$. One can notice how high module fibers are rotated with respect to loading axes, so that they contribute to transverse stiffness without enduring too large principal strains. With respect to the multiple objective solution, this design is thinner and has a larger longitudinal stiffness.

5 Conclusions

When manufacturing composites laminates, a large numbers of parameters can be controlled that have an influence on the process (e.g., injection pressure, injection gate location, compaction rate), or on the final structure, or even on both process and structure (fibers arrangements, number of plies, choice of materials). In addition, numerous manufacturing and design criteria (e.g., injection maximum pressure or time, stiffness, failure) need to be considered when choosing these parameters.

The inherent complexity of composite design make it an important example where the architecture of the simulation-optimization conditions the final design. Two examples have been given to illustrate

the point : a process-structure and a material selection-structure coupled optimization.

The simulation-optimization interface presented in the text is object-oriented, internal, and is based on four natural classes, optimization variables, criteria, optimizers and simulation environment. The classes are assembled through systematic use of object composition and the abstract factory pattern.

It is likely that well-funded interface patterns will become increasingly necessary as more practical optimizers, using all information available about the simulation (approximations, sensitivities, numerical cost of sub-simulations) come into use.

References

[1] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[2] S. Meyers. *Effective C++ : 50 ways to improve your programs and designs*. Addison-Wesley professional computing series. Addison-Wesley, 2 edition, 1997.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addisson-Wesley professional computing series. Addisson-Wesley, 1994.

[4] J. O. Coplien. *Multi-paradigm design for C++*. Addison-Wesley, 1998.

[5] J. Besson and R. Foerch. Large scale object-oriented finite element code design. *Computer methods in applied mechanics and engineering*, 142:165–187, 1997.

[6] J. Besson and R. Foerch. Object-oriented programming applied to the finite element method – part 1: General concepts. *Revue Européenne des Eléments Finis*, 7(5):535–566, 1998.

[7] Y. Dubois-Pélerin, T. Zimmermann, and P. Bomme. Object-oriented finite element programming: II. a prototype program in smalltalk. *Computer Methods in Applied Mechanics and Engineering*, 98:361–397, 1992.

[8] B.W.R. Forde, R.O. Foschi, and S.F. Stiemer. Object-oriented finite element analysis. *Computer & Structures*, 34(3):355–374, 1990.

[9] M.S. Eldred, D.E. Outka, W.J. Bohnhoff, W.R. Witkowski, V.J. Romero, E.R. Ponslet, and K.S. Chen. Optimization of complex mechanics simulations with object-oriented software design. *Computer Modeling and Simulation in Engineering*, 1(3), August 1996.

[10] M.S. Eldred, A.A. Giunta, B.G. van Bloemen Waanders, S.F. Wojtkiewicz, W.E. Hart, and M. P. Alleva. *DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis. Version 3.0 Users Manual*. Sandia Natl. Laboratories, Albuquerque, NM, USA, December 2001. Sandia Technical Report SAND2001-3796P.

[11] The Mathworks Inc., Natick, MA, USA. *MATLAB Documentation, release 12.1*, 2001.

[12] North Western Numerics, Ecole des Mines de Paris, Onera, INSA de Rouen, Seattle, WA, USA or Centre des Matériaux P.-M. Fourt, Evry, France. *Z-set – User Manual, release 8.0*, 2001. available at http://www.nwnumerics.com.

[13] Parametric Technologies Corp., Waltham, MA, USA. *ProEngineer – User Manual, release 22*, 2000.

[14] Swanson Analysis Systems, Inc., Houston, PA, USA. *ANSYS – User Manual*, 2001.

[15] B. Esping and O. Romell. Structural optimization using OASIS-ALADDIN. Technical report, NATO/DFG Advanced Study Institute, Berchtesgaden, Germany, September-October 1991.

[16] G.J. Moore. *MSC/NASTRAN Design Sensitivity and Optimization User's Guide*. The MacNeal-Schwendler Corporation, Los Angeles, CA, USA, 1992.

[17] Samtech, S.A., Liège, Belgium. *SAMCEF-OPTI, version 4.1*, 1991. SAMTECH Publications.

[18] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford Univ. Press, New York, USA, 1996.

[19] J. Horn and N. Nafpliotis. Multiobjective optimization using the niched pareto genetic algorithm. Technical Report IllIGAL 93005, Department of General Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 1993.

[20] M. Luersen and R. Le Riche. Globalisation de l'algorithme de Nelder-Mead: Application aux composites. Technical report, INSA de Rouen, LMR, av. de l'Université, 76801 Saint Etienne du Rouvray cedex, France, december 2001. in French.

[21] Z. Gürdal, R.T. Haftka, and P. Hajela. *Design and Optimization of Laminated Composite Materials*. John Wiley and Sons, Inc., New York, USA, 1999.

[22] A. Saouab, J. Bréard, and G. Bouquet. Contrôle optimal des procédés LCM. In *Compte Rendu JNC 12*, volume 1, pages 1117–1126, ENS Cachan, 2000. in French.

[23] J.-M. Berthelot. *Composite Materials : Mechanical Behavior and Structural Analysis*. Mechanical Engineering Series. Springer, 1999.

[24] J. Gaudin and R. Le Riche. *LAMKIT – Online Presentation*. EADS Centre Commun de Recherches, 2001. available at http://www.eads.net/lamkit.

[25] R. Le Riche and J. Gaudin. Design of dimensionally stable composites by evolutionary optimization. *Composite Structures*, 41:97–111, 1998.

A The abstract factory

The abstract factory pattern, originally described in [3] and improved in [6], is a way to create families of objects deriving from the same class without specifying their concrete type. The concrete type will be decided at run time. The abstract factory is based on dynamic inheritance ([1]). In the example of Figure 3, `all_criteria` of the `SIMULATION_ENV` class are instances of classes derived from `CRITERION`, say `EX_CRITERION` and `BUCKLING_CRITERION`. The `CRITERION::calculate(...)` method is called in `SIMULATION_ENV` through instructions like,

```
double c = all_criteria[i]->calculate(...);
```

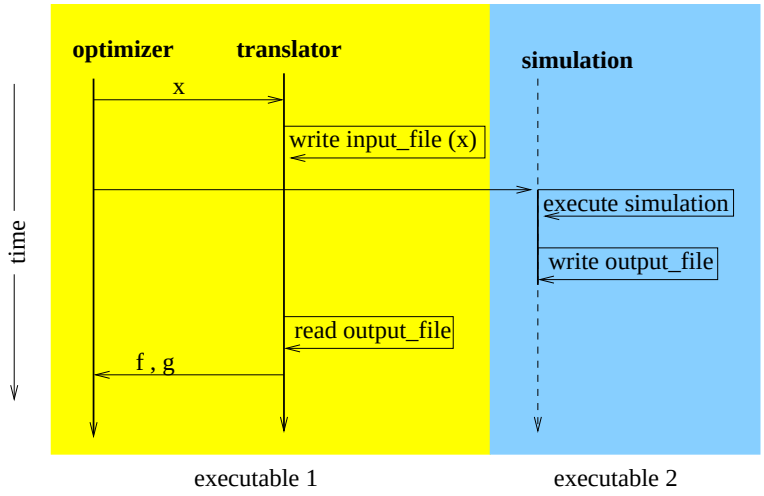

Figure 1: External simulation-optimizer interface, one evaluation.

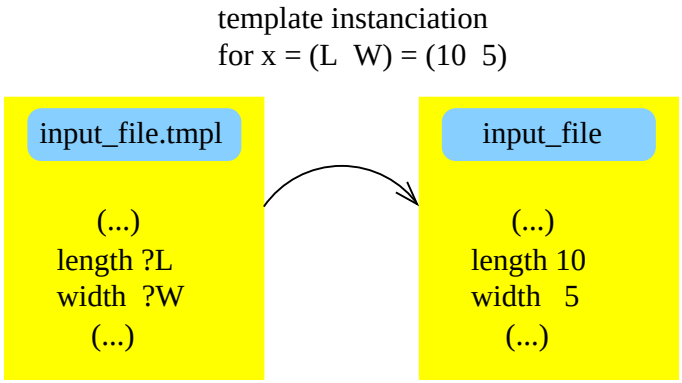

Figure 2: Template instantiation as generic interfacing mechanism (from [12]).

The pointer is dereferenciated at run time, so that specific `EX_CRITERION::calculate(...)` or `BUCKLING_CRITERION::calculate(...)` methods are called, according to the true type of `all_criteria[i]`. When a new class, e.g., `MAXETO1_CRITERION`, is added to the project, the abstract factory makes it possible to automatically instantiate it throughout the code. Another advantage of the abstract factory, with respect to other factory patterns ([3]), is that the new instantiation takes place without modifying or compiling previous files (it remains necessary to re-link the code with the new compiled files). The abstract factory can be seen as a “plugin” mechanism. Dynamic inheritance then makes the new methods `MAXETO1_CRITERION::calculate(...)` available without any change to old files.

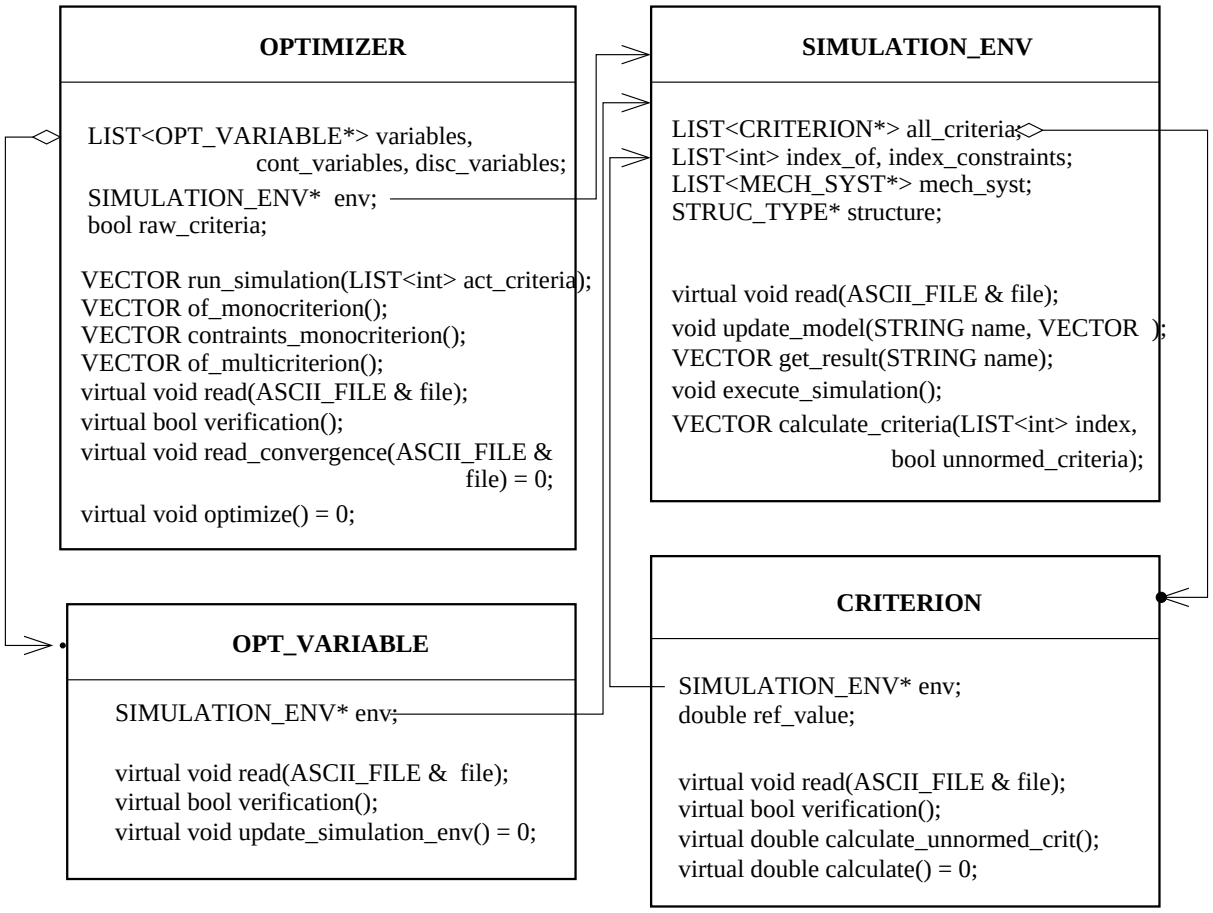

Figure 3: Overview of the simulation-optimizer interface.

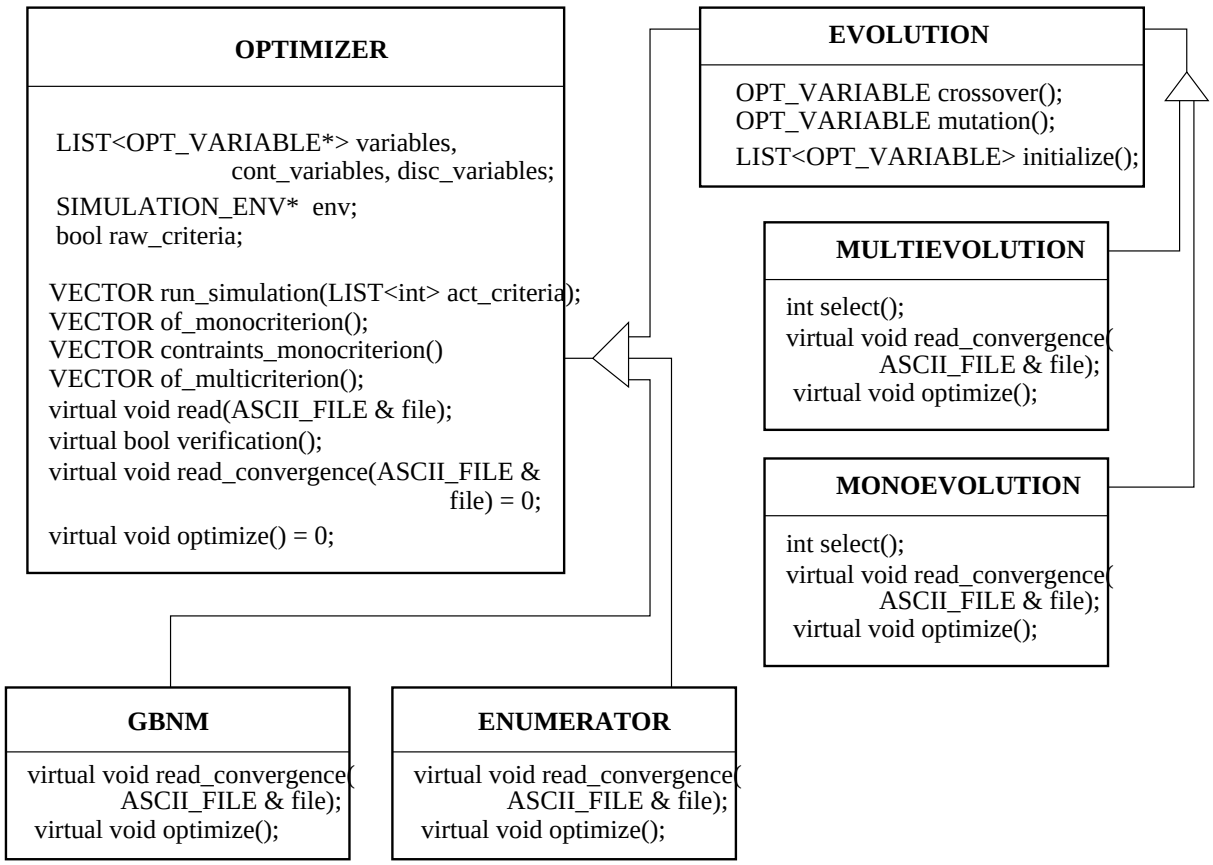

Figure 4: Optimizer inheritance hierarchy.

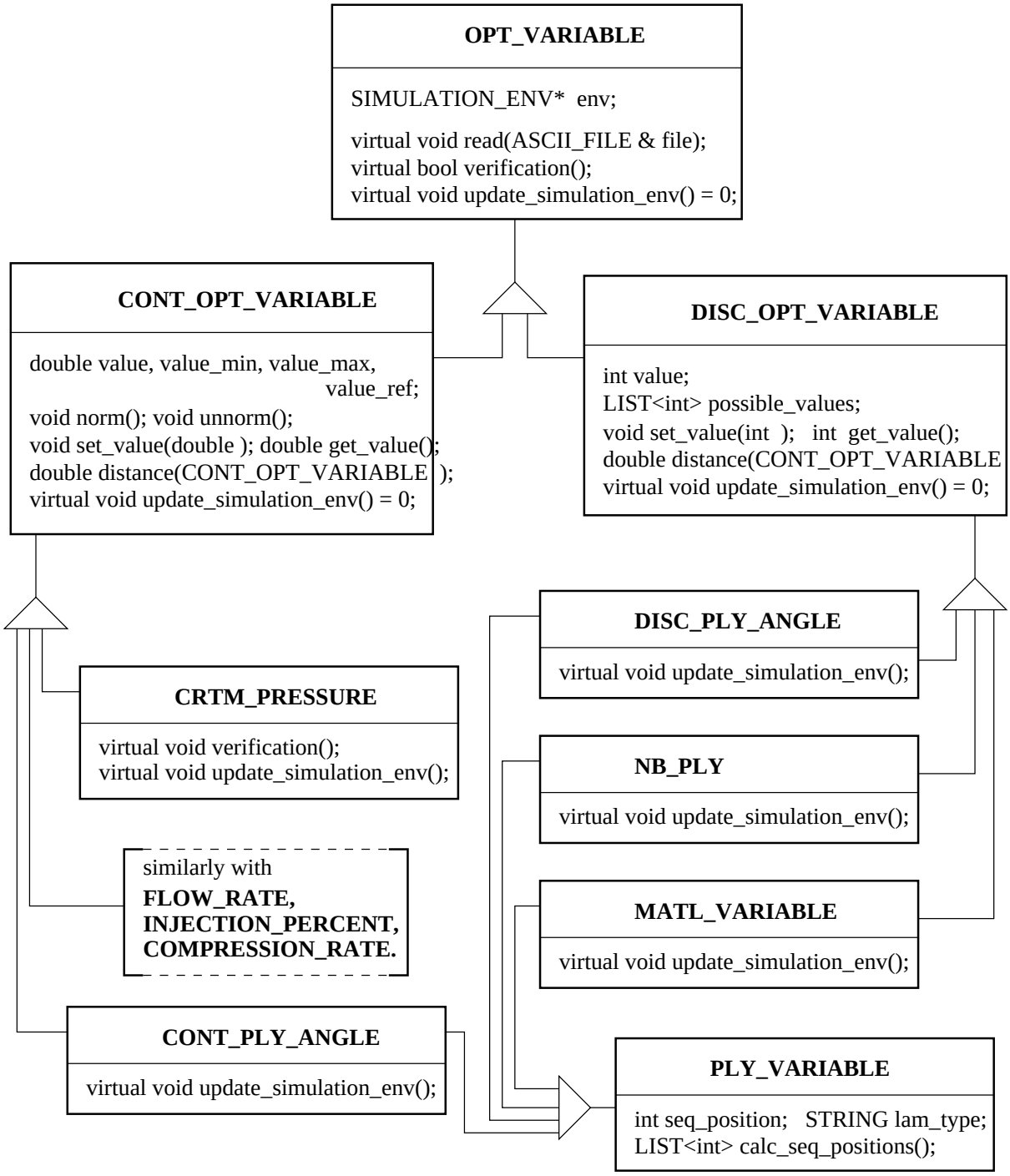

Figure 5: Design variables inheritance hierarchy.

```
VECTOR OPTIMIZER::of_monocriterion()
{  return run_simulation(index_criterion); }

VECTOR OPTIMIZER::constraints_monocriterion()
{  return run_simulation(index_constraints); }

VECTOR OPTIMIZER::run_simulation(LIST<int> active_crit)
{  if (variables_different_from_last_call()) {
        for (int i=0; i<length(variables);i++) variables[i]->update_simulation_env();
        env->execute_simulation();
   }
   return env->calculate_criteria(active_crit,unnormed_criteria); }

VECTOR SIMULATION_ENV::calculate_criteria(LIST<int> active_crit,
                                                                    bool unnormed_crit)
{  VECTOR cv;
   for (int i=0;i<length(active_crit);i++) {
      int j = active_crit[i];
      if (unnormed_crit) cv[i]=all_criteria[j]->calculate_unnormed_crit();
      else cv[i]=all_criteria[j]->calculate();
  }
  return cv; }
```

Figure 6: Sample code of the simulation-optimization interface.

process-structure.opt

```
****optimize enunerator
  ***variables
    **cont_crtm_pressure
      *continuous
        name pressure
        ref 1.e+05
        init 1.e+05
        min 5.e+04
        max 9.e+05
    **cont_ply_angle
      *continuous
        name t1
        ref 1.
        init 15.
        min 0.
        max 90.
      *specific
        seq 1
        lam_type gal
```

process-structure.opt (cont.)

(Similarly with 2nd., 3rd. and 4th. ply angle variables. The numbers after `seq` are 2, 3 and 4, respectively.)

```
  ***criteria
    crtm_time 50.
  ***constraint
    crtm_pressure_max 0.  2.e+05
    Ey 0.  40.e+09
    dev_Ey 0.  8.e+08
  ***convergence
    no_increments 19
    (optionally  exhaustive_search )
****return
```

Figure 7: Optimization input file for the coupled process structure problem and `enumerator` optimizer.

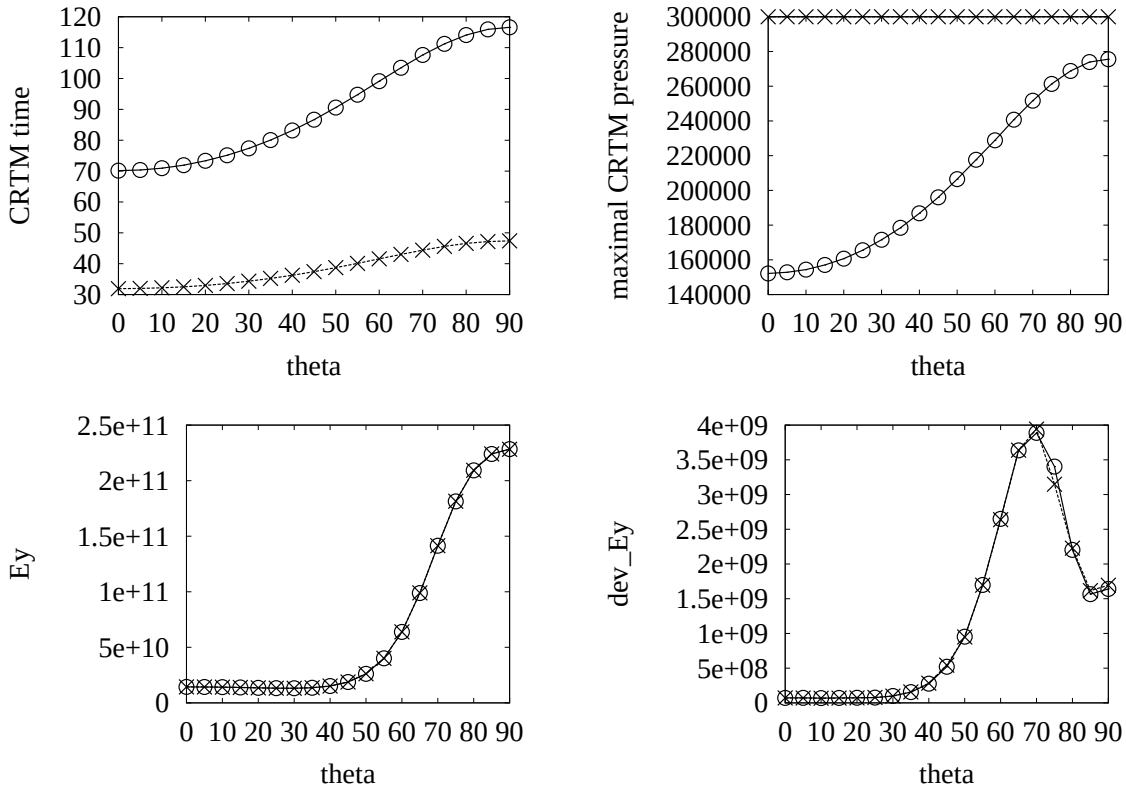

Figure 8: Criteria variations of a $(\pm\theta)_{4s}$ laminate as a function of θ and injection pressure. Pressure is $1.e+05\ Pa$ for circled lines and $3.e+05\ Pa$ for crossed lines.

thickmat.opt

```
****optimize multievolution
 ***variables
  **ranked_matl
    *discrete
      name m1
      values GE CEHR CEHM
      init CEHM

    *specific
      seq 1
      lam_type balanced_sym
  **nb_ply
    *discrete
      name n1
      values 1 2 3 4 5 6 7 8 9 10
      init 2
    *specific
      seq 1
      lam_type balanced_sym
  **disc_ply_angle
    *discrete
      name t1
      values 0 45 90
      init 45
    *specific
      seq 1
      lam_type balanced_sym
```

thickmat.opt (cont.)

(Similarly with 2nd. and 3rd. layers, the numbers after `seq` are 2 and 3)

```
 ***criteria
   thickness 0.005
   buckling 30.0
   maxeto1 1.0
   maxeto2 1.0
   Ex 85.e+9
   Ey 45.e+9

 ***convergence
   *evolution
     population_size 1000
     prob_muta 0.3
     prob_Xover 0.4
     max_nb_analyse 20000
   *specific
     sig_share 100.
     size_comparison_set 95

****return
```

Figure 9: Multiple objective optimization file of the structural and material problem.

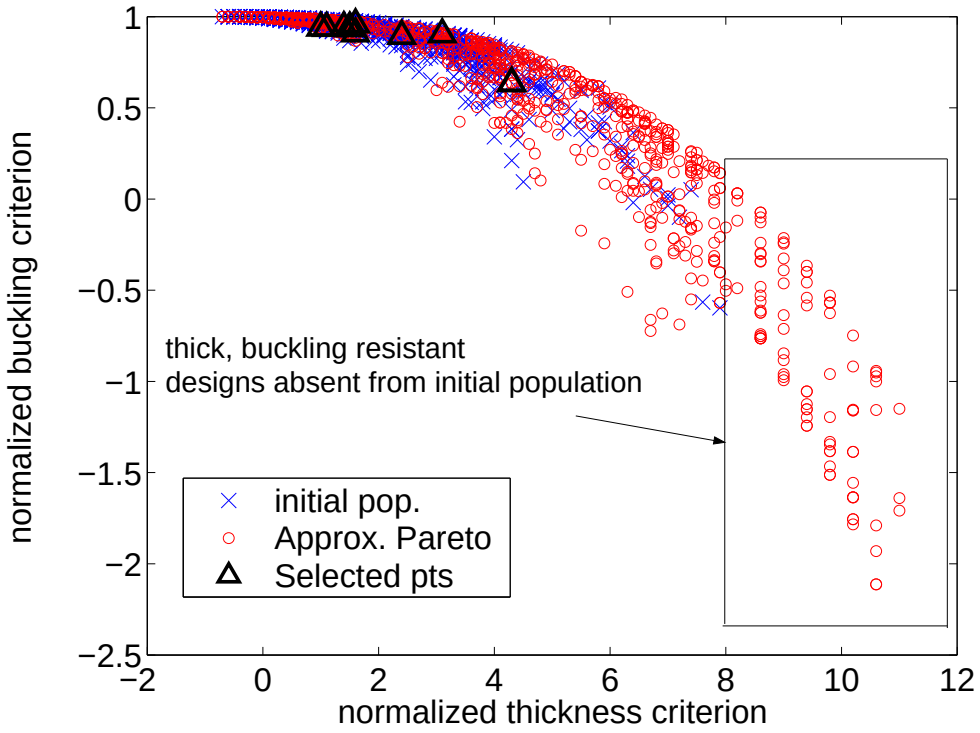

Figure 10: Material selection and structural optimization. Initial population, approximated Pareto front, and selected points of the Pareto front projected on the $(thickness, buckling)$ normalized criteria plane.

thickmat.opt	thickmat.opt (cont.)
****optimize evolution ***variables …cf. Figure 9 ***criteria thickness 0.005 ***constraint buckling 0. 1.0 maxeto1 0. 1.0 maxeto2 0. 1.0 Ex 0. 50.e+9 Ey 0. 25.e+9	***convergence *evolution population_size 500 prob_muta 0.3 prob_Xover 0.4 max_nb_analyse 10000 *specific lag_mult_step 1.0 tourn_size 2 ****return

Figure 11: Input file for constrained single objective evolutionary optimization.

criterion	normed expression	unnormed expression
`alphax` : longitudinal coefficient of thermal expansion.	$\lvert\alpha_x\rvert/\alpha_x^{\text{ref}} - 1$	α_x
…similarly with `alphay`, the coefficients of hygral expansion `betax` and `betay`, and the total strains `EPSx` and `EPSy`.		
`Ex` : longitudinal Young's modulus.	$1 - E_x/E_x^{\text{ref}}$	E_x
…similarly with `Ey`, shear's modulus, `Gxy` and Poisson's ratio, `vxy`.		
`maxeto1` : 1st ply failure using strain in fiber direction, written as a load factor, $\lambda_{\text{fail}} = min_{\text{plies}}(\varepsilon_1^{\text{ref}}/\lvert\varepsilon_1\rvert) \geq \lambda_{\text{ref}}$.	$1 - \min_{\text{plies}}(\varepsilon_1^{\text{ref}}/\lvert\varepsilon_1\rvert)/\lambda_{\text{ref}}$	$min_{\text{plies}}(\varepsilon_1^{\text{ref}}/\lvert\varepsilon_1\rvert)$
…similarly with `maxeto2`, `maxeto12`, principal stresses failure criteria, `maxsig1`, `maxsig2` and `maxsig12`, Tsai-Wu, Hoffman, and Tsai-Hill failure criteria, `max_wu`, `max_hoffman` and `max_hill`.		
`buckling` : critical buckling load.	$1 - \lambda_{\text{buckl}}/\lambda_{\text{ref}}$	λ_{buckl}
`crtm_time` : injection (by CRTM) time, T_{crtm}.	$T_{\text{crtm}}/T_{\text{crtm}}^{\text{ref}} - 1$	T_{crtm}
`crtm_pressure_max` : maximum pressure during injection (by CRTM), P_{crtm}.	$P_{\text{crtm}}/P_{\text{crtm}}^{\text{ref}} - 1$	P_{crtm}
`thickness` : total laminate thickness, H.	$H/H_{\text{ref}} - 1$	H
`dev_alphax` : standard deviation of the longitudinal coefficient of thermal expansion.	$\lvert S(\alpha_x)\rvert/S^{\text{ref}}(\alpha_x) - 1$	$S(\alpha_x)$
…similarly with `dev_alphay`, `dev_betax`, `dev_betay`, `dev_EPSx`, `dev_EPSy`, `dev_Ex`, `dev_Ey`, `dev_Gxy`, `dev_vxy`, `dev_maxeto1`, `dev_maxeto2`, `dev_maxeto12`, `dev_maxsig1`, `dev_maxsig2`, `dev_maxsig12`, `dev_max_wu`, `dev_max_hoffman`, `dev_max_hill`, `dev_buckling`, `dev_crtm_time`, `dev_crtm_pressure_max`.		

Table 1: Summary of optimization criteria implemented by derivation of `CRITERION`. S denotes the standard deviation, $^{\text{ref}}$ and $_{\text{ref}}$ reference values.

	thickness (mm)	E_1 (GPa)	E_2 (GPa)	v_{12}	G_{12} (GPa)	ε_1^{ref}	ε_2^{ref}
Graphite Epoxy (GE)	0.5	46.	10.	0.31	4.6	0.005	0.004
Carbon Epoxy HR (CEHR)	0.125	115.	10.	0.33	5.	0.005	0.004
Carbon Epoxy HM (CEHM)	0.125	230.	14.4	0.32	5.	0.0015	0.001

Table 2: Materials of the structural and material optimization problem.

designs	unnormed criteria					
	H (cm)	λ_{buckl}	λ_{maxeto1}	λ_{maxeto2}	E_x (GPa)	E_y (GPa)
$(\pm 45_9^{\text{CEHM}}/0_{18}^{\text{CEHR}}/\pm 45_2^{\text{CEHR}})_s$	1	1.84	0.63	0.63	62.93	28.44
$(0_2^{\text{CEHR}}/\pm 45_{10}^{\text{CEHM}}/0_{18}^{\text{CEHR}})_s$	1.05	1.87	0.74	0.74	70.32	28.60
$(0_{16}^{\text{CEHR}}/\pm 45_{16}^{\text{CEHM}})_s$	1.2	1.75	0.62	0.60	51.56	30.02
$(0_{18}^{\text{CEHR}}/\pm 45_{16}^{\text{CEHM}})_s$	1.25	1.84	0.67	0.66	54.20	30.10
$(\pm 45_7^{\text{CEHR}}/0_{20}^{\text{CEHR}}/\pm 45_8^{\text{CEHM}})_s$	1.25	1.93	0.72	0.74	57.75	26.82
$(90_{20}^{\text{CEHR}}/0_{32}^{\text{CEHR}})_s$	1.3	1.29	0.98	0.78	75.11	50.72
$(\pm 45_{10}^{\text{CEHR}}/\pm 45_9^{\text{CEHM}}/0_{14}^{\text{CEHR}})_s$	1.3	2.77	0.58	0.59	44.78	26.46
$(0_{28}^{\text{CEHM}}/90_{10}^{\text{GE}})_s$	1.7	3.16	0.51	1.37	100.90	33.46
$(90_{12}^{\text{GE}}/0_{34}^{\text{CEHR}})_s$	2.05	2.92	1.10	0.88	53.80	31.49
$(0_{18}^{\text{CEHR}}/90_{16}^{\text{CEHR}}/0_{18}^{\text{GE}})_s$	2.65	10.94	1.40	1.12	52.76	25.98

Table 3: Designs selected from Pareto front of Figure 10. Criteria are defined in Table 1.