

A Transformation System for Interactive Reformulation of Design Optimization Strategies

Thomas Ellman and John Keane and Takahiro Murata and Mark Schwabacher
Department of Computer Science, Hill Center for Mathematical Sciences
Rutgers University, Piscataway, New Jersey 08855
{ellman,keane,murata,schwabac}@cs.rutgers.edu

Abstract

Numerical design optimization algorithms are highly sensitive to the particular formulation of the optimization problems they are given. The formulation of the search space, the objective function and the constraints will generally have a large impact on the duration of the optimization process as well as the quality of the resulting design. Furthermore, the best formulation will vary from one application domain to another, and from one problem to another within a given application domain. Unfortunately, a design engineer may not know the best formulation in advance of attempting to set up and run a design optimization process. In order to attack this problem, we have developed a software environment that supports interactive formulation, testing and reformulation of design optimization strategies. Our system represents optimization strategies in terms of second-order dataflow graphs. Reformulations of strategies are implemented as transformations between dataflow graphs. The system permits the user to interactively generate and search a space of design optimization strategies, and experimentally evaluate their performance on test problems, in order to find a strategy that is suitable for his application domain. The system has been implemented in a domain independent fashion, and is being tested in the domain of racing yacht design.

1 Introduction

Numerical design optimization is a notoriously unreliable process. Optimization programs often take excessive time to reach termination. Furthermore, upon termination, optimization programs often fail even to reach locally optimal designs. These difficulties typically arise

if the constraints or objective functions are expensive to evaluate, the problem contains many design parameters, or the search space contains pathologies, such as ridges, plateaus or un-evaluable points. Design engineers can, in principle, overcome these difficulties, by carefully formulating the inputs to numerical optimization codes. In particular, by carefully choosing the search space, objective function and constraints, a design engineer can dramatically reduce the duration of the optimization process, and improve the quality of the resulting design. Unfortunately, a design engineer may not know the best formulation in advance of attempting to set up and run a design optimization process. The best formulation will vary from one application domain to another, and from one problem to another within a given application domain.

In order to attack this problem, we have developed a software environment that supports interactive formulation, testing and reformulation of design optimization strategies. Our system represents optimization strategies in terms of second-order dataflow graphs. Reformulations of strategies are implemented as transformations between dataflow graphs. The system permits the user to interactively generate and search a space of design optimization strategies, and experimentally evaluate their performance on test problems, in order to find a strategy that is suitable for his application domain.

This paper presents an overview of our approach to reformulation of design optimization strategies. In Section 2 we define the types of optimization problems we are considering, and present an application domain - sailing yacht design - that we are using as a testbed. In Section 3, we describe our language for representing design optimization strategies. In Section 4, we describe the manner in which a user typically interacts with our system in order to generate and test optimization strategies. In Section 5, we present a catalog of transformations for converting one optimization strategy into another. In Section 6 we illustrate how the transformations can be used to construct optimization strategies for the yacht

```

Problem-Class( $e_1, \dots, e_N, k_1, \dots, k_M$ ):

Given:
  Environment Parameter Values: ( $e_1, \dots, e_N$ )
  Threshold Parameter Values:   ( $k_1, \dots, k_M$ )
  Design Parameters Names:      ( $d_1, \dots, d_L$ )

Optimize:
  Objective Function:
     $F(d_1, \dots, d_L, e_1, \dots, e_N)$ 
  Equality Constraints:
     $Eq-1(d_1, \dots, d_L, e_1, \dots, e_N) = 0$ 
    ...
     $Eq-T(d_1, \dots, d_L, e_1, \dots, e_N) = 0$ 
  Inequality Constraints:
     $In-1(d_1, \dots, d_L, e_1, \dots, e_N) \leq k_1$ 
    ...
     $In-M(d_1, \dots, d_L, e_1, \dots, e_N) \leq k_M$ 

```

Figure 1: Generic Optimization Schema

domain. We also describe preliminary tests aimed at evaluating the performance of various derived optimization strategies in the yacht domain. Finally, in Section 7 we describe a related body of research on using Machine Learning techniques to construct strategy selection rules.

2 Design Optimization Problems

Design optimization problems can be characterized in terms of the schema shown in Figure 1. One is given a list (d_1, \dots, d_L) of parameters that describe an artifact. One hopes to find a list of values of these parameters that will minimize some objective function F , subject to a collection of equality constraints $\{Eq_i\}$ and inequality constraints $\{In_i\}$. The objective function, equality constraints and inequality constraints all depend on the design parameters (d_1, \dots, d_L) as well as a collection of “environment parameters” (e_1, \dots, e_N) that describe the environment in which the artifact must operate. Finally, the inequality constraints depend on various objective “threshold parameters” (k_1, \dots, k_M) , the values of which specify how “tight” or “loose” each constraint will be. The threshold parameters and the environment parameters define a parameterized class of design problems, i.e., each combination of values of the environment and threshold parameters defines a particular problem instance in the domain under study.

The class of design problems constituting the racing yacht domain is described in Figure 2. The objective is to design a yacht with a steady-state *Velocity* that is as high as possible when sailing in winds with a given *Windspeed* and sailing at a given angular *Heading* with

```

Yacht-Problem-Class( $Heading, Windspeed, Rating$ )

Given:
  Design Parameters:
    Length, Beam, Draft, KeelHeight,
    BallastMass, SailArea, Velocity, Heel.
  Environment Parameters: Windspeed, Heading.
  Threshold Parameters:   Rating.

Optimize:
  Objective Function:      Velocity(Design)
  Equality Constraints:
    NetForce(Design,Heading,Windspeed)=0
    NetTorque(Design,Heading,Windspeed)=0
    NetBuoyancy(Design,Heading,Windspeed)=0
  Inequality Constraints:
    NetRating(Design,Heading,Windspeed)≤Rating

```

Figure 2: The Yacht Design Problem Class

respect to the wind¹ Thus *Windspeed* and *Heading* are the environment parameters for this problem class. The design parameters include geometric quantities (*Length*, *Beam*, *Draft*, *KeelHeight*) describing the size and shape of the hull of the yacht. They also include the amount of lead needed to make the boat float properly (*BallastMass*) and the area of the sail (*SailArea*). Finally the design parameters also include *Velocity* and *Heel* angle at which the yacht will sail. Although these two parameters are formally included among the design parameters, they cannot be manipulated independently of the others. They are constrained by equations that describe the physics of sailing yachts. These equations assert that the net (longitudinal) forces, net (angular) torques and net (vertical) buoyancy are each equal to zero, when the yacht is in steady state motion. Finally, the yacht must satisfy an inequality constraint, called the “rating constraint”, imposed by the racing authorities. The rating constraint specifies that an arithmetic formula involving geometric parameters of the yacht design must have a value no greater than a given *Rating*.

3 Representation of Strategies

A wide variety of choices must be made in the course of setting up and running a design optimization process. Some of these choices are outlined in Figure 3. These choices concern issues that are not normally addressed

¹Strictly speaking, racing yachts are designed for a particular race course consisting of a sequence of legs at several different headings. We are using a simplified version of the problem for the purposes of this presentation.

- **Abstraction and Decomposition:** Will the overall optimization problem be split into subproblems that can be solved sequentially (or in parallel)? If so, what will be the subproblems? What will be the strategy for solving each subproblem?
- **Setting up the Search Space:** Over what search space will each subproblem optimization be carried out? How will each space be parameterized, i.e., what coordinate system will be used?
- **Handling of Constraints:** Which equality constraints will be handled by the numeric optimization algorithm? Which will be solved externally to the optimizer, resulting in a reduction of the dimension of the search space? Which inequality constraints will be forced to be active, i.e., made into equality constraints? Which will be allowed to become inactive?
- **Selecting Approximations:** What approximations will be used in computing the objective function, and evaluating the constraints? How will any fitting coefficients in those approximations be fit to the problem at hand? Will approximations be recalibrated during the optimization process? If so, how often, and when?
- **The Optimization Algorithm:** What optimization algorithm will be used for each subproblem? How will each tolerance parameter be set? How will gradients be computed? How will line searches be carried out? How many seed points will be used? How will seed points be selected?

Figure 3: Choices in Setting up an Optimization Process

by numerical analysts who develop numerical optimization codes. Numerical analysts will typically take the inputs to the constrained optimization process as a given, and focus instead on the most efficient and/or robust numerical algorithm for handling them. In contrast to this, we assume the existence of various algorithms for solving constrained optimization problems. We focus instead on questions of how such codes will be used to solve problems: How will their inputs be formulated? How will they be exercised repeatedly in the course of solving an overall design problem?

We have developed a high level language for representing design optimization strategies. The language permits one to express the range of strategies outlined informally in Figure 3. Our system represents design optimization strategies in two different ways. Strategies

First Order Operations:

- Arithmetic, Trigonometry, Interpolation, Conditionals, Relational, Boolean, Set and List Operations.

Second Order Operations:

- Solving systems of algebraic equations.
- Integrating systems differential equations.
- Optimizing functions subject to constraints.
- Iterating functions to converge on fixed points.

Figure 4: Primitives in the Strategy Language

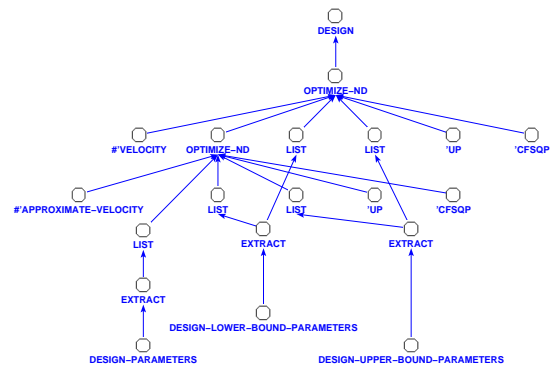


Figure 5: Dataflow Graph

are displayed *externally* to human users as second-order dataflow graphs. An example of such a dataflow graph is shown in Figure 5. The dataflow graphs are represented *internally* in LISP syntax. This dual representation allows us to express transformations between strategies as simple rewrite rules operating on LISP syntax, while using the visual dataflow representation to communicate with human engineers who are typically not familiar with LISP syntax.

The primitive operations supported in our strategy language are described in Figure 4. The primitives include first order operations like arithmetic and interpolation that make up the objective and constraint functions, as well as higher-level, second order operations like root finding, integration, optimization and convergence, that determine the overall organization of the optimization strategy.

4 Interactive Strategy Reformulation

The user begins by preparing an initial design optimization strategy to serve as the starting point of the strategy development process. The initial strategy will typically be formulated in a manner analogous to the yacht design problem class specification, shown in Figure 2. All potentially relevant quantities are included as parameters. The governing physical equations are included as equality constraints. Various measures of design quality appear either in the objective function or in the constraints. After preparing the initial strategy, the user selects a transformation from a menu and applies it to the initial strategy, to create a new, derived strategy. In many cases, a given transformation can be applied in more than one way. In such cases, the system asks the user which instantiation of the transformation he wants to apply. The system then applies the selected transformation and displays the revised strategy to the user.

As the user interacts with the strategy transformation system, he generates a tree in the space of design optimization strategies. The system keeps track of this search tree during the strategy development process. (See Figure 6.) Each node in the tree holds a description of an optimization strategy. Each arc in the tree is labeled with the transformation used in converting the parent strategy into the child strategy. The tree thus implicitly describes all the differences between ancestors and descendants in the tree. The user can move up and down the tree at will to backtrack to previous strategies to try new alternatives. He can run the strategy in any node on a set of test problems and store the results in a database associated with that node. He can also generate plots of various kinds of data that describe the behavior of strategies on test problems, e.g., a plot of the path through a search space, or a plot of the evolution of a measure of merit, or other quantities appearing in the constraints and the objective function. The user can also annotate the current strategy node with his own observations and conclusions about the behavior of the strategy. The annotated tree of strategies and databases thus serves as a record of the whole design strategy development process.

5 Transformations between Strategies

Transformations that reformulate the search space are described in Figure 7. The transformation called “Drop Parameter” is useful whenever the user believes that a design parameter may not have a great impact on the objective function or constraints appearing in the problem. By removing the parameter from the space, the user may improve the efficiency of the strategy without loss of design quality. For example, in the yacht domain, the user might believe that the *Beam* parameter

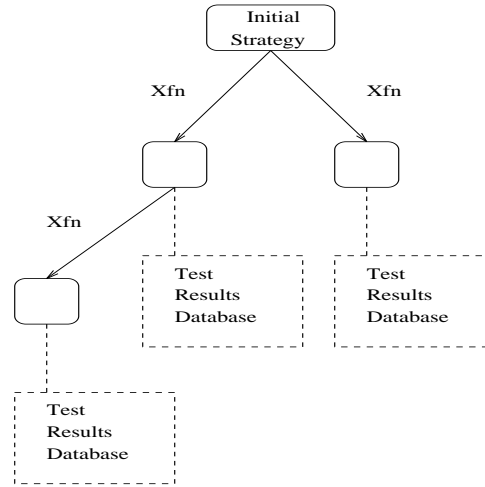


Figure 6: Design Strategy Development Record

is not important and should be dropped from the search space. The transformation called “Parameterize Intermediate Quantity” allows the user to introduce a new parameter in to the search space. The user may select any intermediate quantity appearing in a constraint or objective function and allow that quantity to be directly manipulated by the design optimization algorithm. For example, in the yacht domain, the user might decide that the displacement of the yacht (i.e., the amount of water displaced by the hull) should be made into an explicit design parameter.

The transformation called “Constrain Intermediate Quantity” allows the user to impose a new equality or inequality constraint on the problem. This transformation can be used in several ways. One use is simply to convert an inequality constraint into an equality constraint. For example, in the yacht domain the user might reason that the optimal design will have a sail that is as large as possible while satisfying the *NetRating* inequality constraint. This inequality constraint will therefore be exactly satisfied in any optimal solution. By making this into an equality constraint, the user can restrict the search to smaller subspace with a potential improvement in efficiency. The transformation called “Constrain Intermediate Quantity” can also be used for another purpose. For example, in the yacht domain, the *NetRating* function contains conditional expressions that apply rating penalties to yachts whose geometries fall outside certain critical dimensions. These penalties result in non-smoothness in the value of *NetRating*. Optimal designs very often lie in subspaces of the search space where the penalties come into effect, i.e. were the *NetRating* is non-smooth. The user can apply the transformation called “Constrain Intermediate Quantity” to introduce a new equality constraint that will force the solution to

Drop Parameter: Given an optimization over parameters $x_1, \dots, x_i, \dots, x_n$: (1) Drop x_i from the argument list of each constraint or objective function; (2) Replace each reference to x_i in any constraint or objective function with a reference to the seed value of x_i .

Parameterize Intermediate Quantity: Given an optimization over parameters x_1, \dots, x_n , and any expression $f(x_1, \dots, x_n)$ appearing as an intermediate quantity computed by some constraint or objective function: (1) Introduce a new design parameter y , add y as an additional argument to each constraint or objective function; (2) Replace each appearance of $f(x_1, \dots, x_n)$ in a constraint or objective function with a reference to y ; (3) Impose an equality constraint requiring that $y = f(x_1, \dots, x_n)$.

Constrain Intermediate Quantity: Given an optimization over parameters x_1, \dots, x_n , and any expression $f(x_1, \dots, x_n)$ appearing as an intermediate quantity computed by some constraint or objective function: Introduce a new constraint asserting that $f(x_1, \dots, x_n) = 0$, $f(x_1, \dots, x_n) \leq 0$, or $f(x_1, \dots, x_n) < 0$, etc.

Solve Equality Constraint: Given an optimization over parameters $x_1, \dots, x_i, \dots, x_n$, and some constraint of the form $f(x_1, \dots, x_i, \dots, x_n) = 0$: (1) Remove x_i from the set of design parameters; (2) Drop x_i from the argument list of each constraint or objective function; (3) Arrange for each constraint or objective function to symbolically or numerically solve $f(x_1, \dots, x_i, \dots, x_n) = 0$ for x_i in terms of $x_1, \dots, x_{i-1}, \dots, x_{i+1}, \dots, x_n$.

Figure 7: Transforms to Reformulate Search Space

lie in a such a non-smooth subspace, with a potential improvement in efficiency.

The transformation called ‘‘Solve Equality Constraint’’ allows the user to reduce the dimension of the search space by solving for one (or more) parameters in terms of others. The transformation allows one to solve symbolically (at the time the strategy is being developed) using MAPLE [Char *et al.*, 1992] as an equation solver. The transformation also allows the equation to be solved numerically at run time using a numerical root finder like the Newton-Raphson algorithm. In either case the effect is to reduce the dimension of the search space, with a potential for improvement in efficiency. For example, in the yacht domain, the *NetBuoyancy* constraint can be solved symbolically for *BallastMass*,

thus removing this parameter from the search space. The *NetForce* and *NetTorque* constraints can be solved numerically (simultaneously) for *Velocity* and *Heel*, thus removing these parameters from the search space. Furthermore, once the *NetRating* inequality constraint has been converted into an equality constraint, it can be solved symbolically for the maximum legal *SailArea*, removing this parameter from the search space.

Transformations that introduce approximations into objective and constraint functions are described in Figure 8. The transformation called ‘‘Freeze Intermediate Quantity’’ applies to any intermediate quantity appearing in a constraint or objective function. It allows the user to treat that quantity as if its value were independent of the design parameters. For example, in the yacht domain, one might assume that the quantity *ApparentWindAngle* (the angle at which the wind appears to be blowing, in the frame of reference of the yacht) is not likely to change very much over the course of the optimization. The transformation will freeze such a quantity at its value at the beginning of the optimization process. This type of transformation is useful for two reasons. To begin with, it avoids repeated computation of quantities that do not change very much, thus saving time if re-computation would have been expensive. In addition, this transformation may enable further improvements in efficiency. For example, in the yacht domain, the quantity *EffectiveSailArea* actually depends on the *Heel* angle at which the yacht is sailing. If this quantity is frozen at its initial value, the *NetTorque* equation can be solved symbolically, with a dramatic improvement in efficiency.

The transforms called ‘‘Drop Equation’’ and ‘‘Decompose Equations’’ both serve to reduce the number of equations that need to be solved simultaneously. For example, the ‘‘Decompose Equations’’ transformation can be used in the yacht domain to replace the simultaneous solution of the *NetForce* and *NetTorque* equations with a sequence in which *NetTorque* is solved for *Heel* first, assuming a fixed *Velocity*, and then *NetForce* is solved for *Velocity*, assuming the value just computed for *Heel*. The transformation called ‘‘Drop Equation’’, could be used to introduce an even more radical approximation, by fixing *Heel* at its seed value and eliminating the *NetTorque* equation entirely. These transformations are useful for two reasons. To begin with, numerical solution of two systems of equations, each with one equation in one unknown, is usually faster than simultaneously solving two equations in two unknowns. In addition, single equations can be solved using a more reliable and robust method, such as bisection, rather than Newton-Raphson, which is brittle and unreliable.

Transformations that introduce iterative optimization are described in Figure 8. These transformations

Freeze Intermediate Quantity: Given an optimization over parameters x_1, \dots, x_n , and any expression $f(x_1, \dots, x_n)$ appearing as an intermediate quantity computed by some constraint or objective function: Replace $f(x_1, \dots, x_n)$ with the result of applying f to the seed values of parameters x_1, \dots, x_n .

Drop Equation: Given an expression e appearing in some constraint or objective function, such that e solves the equation $f(y) = 0$, using a numerical method, starting with a seed value s : Replace e with s .

Decompose Equations: Given an expression e appearing in some constraint or objective function, such that e solves the simultaneous equations $f(y_1, y_2) = 0$, $g(y_1, y_2) = 0$ using a numerical method, starting with seed value s_i for each y_i : Replace e with a new expression operates in two steps: (1) Solve $f(y_1, s_2) = 0$ for y_1 ; (2) Solve $g(y_1, y_2) = 0$ for y_2 . (Generalizations of this transformation split $N + M$ simultaneous equations into two steps, solving N equations in the first step and M equations in the second step.)

Figure 8: Transforms to Introduce Approximations

can be used in combination with transformations for search space reformulation and approximation, described above, to construct a variety of strategies, including abstraction, decomposition and multi-level modeling. For example, consider the transformation called “Introduce Sequential Optimization”. This transformation replaces a single optimization, with a sequence of two optimizations. If the user subsequently uses the “Drop Parameter” transformation to remove different sets of parameters from each stage of the optimization, he constructs a strategy that decomposes the optimization problem into two subproblems, that are solved sequentially. For example, in the yacht domain, these transformations could be used to construct a decomposition under which the parameters describing geometry near the waterline (*Length* and *Beam*) are optimized in the first stage, and the parameters describing underwater geometry (*Draft* and *KeelHeight*) are optimized in the second stage. Decomposition is useful in part because of a direct improvement in efficiency, since solving the two subproblems is usually faster than solving the one original problem. Furthermore, decomposition allows different optimization strategies (e.g., different approximations) to be used on different subproblems.

The transformation called “Introduce Sequential Opti-

Introduce Sequential Optimization: Replace a single optimization strategy $s \rightarrow r$ starting with seed s and returning result r with a sequence of two optimization strategies $s \rightarrow p \rightarrow r$, where the first optimization converts s to p and the second optimization converts p to r .

Introduce Convergence Loop: Replace an optimization strategy $s \rightarrow r$ starting with seed s and returning result r , with a convergence loop that iterates $s \rightarrow r$ until some termination condition is met.

Introduce Regional Sampling: Replace an optimization strategy $s \rightarrow r$ starting with seed s and returning result r , with a new strategy that repeatedly generates seeds in some neighborhood of s , and applies the original strategy to each seed.

Figure 9: Transforms to Iterate Optimization

mization” is useful whenever the user plans to use transformations that potentially degrade the quality of the resulting design. For example, transformations that drop parameters, decompose problems and introduce approximations all have the potential to result in suboptimal designs. The user may be willing to accept a suboptimal design in return for an improvement in the speed of optimization. If the user is not willing to make such a tradeoff, he may introduce a sequential optimization, and apply the quality degrading transformations to the first stage only. In the resulting strategy, the first stage of optimization serves only to find a seed point for the second stage. The second stage takes the seed and optimizes it using a strategy that does not sacrifice quality.

The transformation called “Introduce Convergence Loop” serves an analogous purpose. This transformation replaces a single optimization with a convergence loop that applies the original optimization strategy until a convergence test is met. This transformation is particularly useful when used to iterate an optimization strategy that involves a decomposition of the design parameters, or a local approximation to the constraints or objective functions. For example, in the yacht domain, suppose the user applies transform called “Freeze Intermediate Quantity” to construct a strategy in which the *EffectiveSailArea* is frozen at its seed value. The user might then apply the transform called “Introduce Convergence Loop”, to iterate the approximated strategy until it converges. The overall effect is a strategy that periodically re-calibrates the frozen value of *EffectiveSailArea*. We conjecture that there exist conditions under which this sort of re-calibrating strategy

can be proven to converge at exactly the same points as the original strategy which has no approximations.

One final transformation is called “Introduce Regional Sampling”. This transformation constructs iterative optimization strategies that operate by repeatedly generating seed points in some neighborhood, and initializing a base optimization strategy from each seed point - returning the best overall resulting design. This transformation is useful when the objective function contains local optima that are not global optima. Multiple local optimal may simply result from the physics of the design problem. In addition, multiple “apparent” local optima often arise due to “noise” generated by unreliable models for computing quantities appearing in the constraints and objective functions. In both cases, regional sampling is needed to keep optimization algorithms from getting stuck at points that are local, but not global optima. Sampling is also useful when the equality and inequality constraints are satisfied in regions that are not connected to each other.

6 Experiments

We have implemented both the strategy representation language and the strategy transformation system described above. We are currently testing the system in the domain of racing yacht design. For this purpose, we implemented the initial yacht strategy described in Figure 2. We then applied various sequences of transformations to generate four other strategies. The entire group of strategies is described in Figure 10. We do not consider these strategies to be the final word in yacht design. We do intend that they illustrate the range of strategies that can be generated by our system.

We are testing the yacht design strategies using CFSQP [Craig *et al.*, 1994] as the underlying numeric optimization method. CFSQP is a state-of-the-art implementation of the Sequential Quadratic Programming method. Sequential Quadratic Programming is a quasi-Newton method that solves a nonlinear constrained optimization problem by fitting a sequence of quadratic programs to it, and then solving each of these problems using a quadratic programming method. We tested two design strategies (*S3* and *S4* from Figure 10) on each of six different test problems. We recorded average CPU time and the average quality (i.e., velocity) of the resulting yacht design. Results of these tests are shown in Figure 11. These data show that *S4* ran significantly (47.5%) faster than *S3*, while suffering only a small (0.465%) decline in relative quality. Additional tests are in progress. Nevertheless, these results indicate that our interactive reformulation system can have a significant impact on the performance of a design optimization strategy.

- (*S*₁) Eight design parameters (*Length*, *Beam*, *Draft*, *KeelHeight*, *BallastMass*, *SailArea*, *Velocity*, *Heel*) three equality constraints (*NetForce*, *NetTorque* and *NetBuoyancy*) and one inequality constraint (*NetRating*).
- (*S*₂) Arranged to solve *NetForce* and *NetTorque* equality constraints numerically for *Velocity* and *Heel*; Solved *NetBuoyancy* equality constraint symbolically for *BallastMass*. (Remaining design parameters: *Length*, *Beam*, *Draft*, *KeelHeight*, *SailArea*.)
- (*S*₃) Converted *NetRating* into an equality constraint and solved it symbolically for *SailArea*. (Remaining design parameters: *Length*, *Beam*, *Draft*, *KeelHeight*.)
- (*S*₄) Constrained two non-smooth intermediate quantities in *NetRating* to lie at a points of non-smoothness. Solved these equality constraints symbolically for *Beam* and *Draft*. (Remaining design parameters: *Length* and *KeelHeight*)
- (*S*₅) Introduced approximations into *NetForce* and *NetTorque* functions in order to solve balance of force and torque equations symbolically.

Figure 10: Derivations of Yacht Design Strategies

Strategy	Δ Quality	Δ CPU Time
S3	-0.012%	0%
S4	-0.477%	-47.5%

Figure 11: Performance of Optimization Strategies

7 Related Work

In a related body of work, we have developed methods of using Machine Learning to develop rules for selecting design optimization strategies. This work is based on the belief that the best optimization strategy will vary from domain to domain, (e.g., in yacht design as compared to say, aircraft design), as well as from problem to problem within a domain. For example, in [Schwabacher *et al.*, 1995] we reported experimental results showing that the decision of whether to constrain intermediate quantities appearing in the *NetRating* constraint will depend on the particular yacht design problem at hand. We used inductive learning methods to develop rules whose conditions refer to the environment and threshold parameters of the yacht design problem class (e.g., *Heading* and *Windspeed*) and whose actions recommend particular design optimization strategies. We plan to extend this

learning approach to develop rules for making a wider range of decisions that go into formulation of a design optimization strategy. In the long run, we expect three types of reformulation selection rules to result from these investigations: (1) Rules that apply to all engineering design domains (e.g., rules applicable to yacht design and aircraft design, and other domains, etc.); (2) Rules that apply to all design problems within a single domain (e.g., rules applicable to all yacht design problems, but not necessarily to all aircraft design problems); and (3) Rules that apply to some problems within a domain but not to all problems within a domain (e.g., rules applicable to designing yachts for traveling downwind in high winds, but not necessarily applicable to other yacht design problems).

Several other investigators have attempted to use knowledge-based methods to improve the performance of numerical optimization, including [Orelup *et al.*, 1988; Tong, 1988; Powell, 1990; Hoeltzel and Chieng, 1987]. This work has focused primarily on automating the choice of underlying numerical algorithm, and tolerance parameters to that algorithm. In contrast to this work, we have focused on the automating the formulation of other inputs to the optimization algorithm (e.g., the choice of design parameters, the representation and approximation of constraints and objective functions), and on the overall optimization strategy (e.g., how to use optimization as a subroutine in an iterative process). Nevertheless, we consider the choice of method and tolerance parameters to be significant, and expect our work to be complementary to this previous work. One particular type of search space reformulation, called “model reduction” has been described in [Papalambros and Wilde, 1988] and implemented in [Choy and Agogino, 1986]. This work has focused on developing methods that are provably correct, i.e., methods that are guaranteed improve performance without any loss of design quality. In contrast, we have focused on developing an experimental system for testing reformulations when no such guarantees are available.

8 Acknowledgments

Our research is supported by the National Aeronautics and Space Administration through NASA Grant NCC-2-802 and by the Advanced Research Projects Agency of the Department of Defense through contract ARPA-DABT 63-93-C-0064. It has benefited from discussions with Saul Amarel, Andrew Gelsey, Don Smith, Haym Hirsh, Richard Keller, Lou Steinberg, Gerry Richter.

References

[Char *et al.*, 1992] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt.

First Leaves: A Tutorial Introduction to Maple V. Springer-Verlag and Waterloo Maple Publishing, 1992.

[Choy and Agogino, 1986] J. Choy and A. Agogino. Symon: Automated symbolic monotonicity analysis system for qualitative design optimization. In *Proceedings ASME International Computers in Engineering Conference*, 1986.

[Craig *et al.*, 1994] L. Craig, J. Zhou, and A. Tits. User’s guide for cfsqp version 2.1: A c code for solving (large scale) constrained nonlinear (minimax) optimization problems, generating iterates satisfying all inequality constraints. Technical Report TR-94-16r1, Institute for Systems Research, University of Maryland, November 1994.

[Hoeltzel and Chieng, 1987] D. Hoeltzel and W. Chieng. Statistical machine learning for the cognitive selection of nonlinear programming algorithms in engineering design optimization. In *Advances in Design Automation*, Boston, MA, 1987.

[Orelup *et al.*, 1988] M. F. Orelup, J. R. Dixon, P. R. Cohen, and M. K. Simmons. Dominic ii: Meta-level control in iterative redesign. In *Proceedings of the National Conference on Artificial Intelligence*, pages 25–30, St. Paul, MN, 1988.

[Papalambros and Wilde, 1988] P. Papalambros and J. Wilde. *Principles of Optimal Design*. Cambridge University Press, New York, NY, 1988.

[Powell, 1990] D. Powell. Inter-gen: A hybrid approach to engineering design optimization. Technical report, Rensselaer Polytechnic Institute Department of Computer Science, December 1990. Ph.D. Thesis.

[Schwabacher *et al.*, 1995] M. Schwabacher, T. Ellman, and H. Hirsh. Learning when reformulation is appropriate for iterative design. IJCAI-95 Workshop of Machine Learning in Engineering, 1995.

[Tong, 1988] S. S. Tong. Coupling symbolic manipulation and numerical simulation for complex engineering designs. In *International Association of Mathematics and Computers in Simulation Conference on Expert Systems for Numerical Computing*, Purdue University, 1988.