KNOWLEDGE-BASED MANAGEMENT OF LEGACY CODES FOR AUTOMATED DESIGN

BY JOHN ERIC KEANE

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Thomas Ellman

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 1996

ABSTRACT OF THE DISSERTATION

Knowledge-Based Management of Legacy Codes for Automated Design

by John Eric Keane

Dissertation Director: Thomas Ellman

Systems for automated design optimization of complex real-world objects can, in principle, be constructed by combining domain-independent numerical routines with existing domain-specific analysis and simulation programs. Such "legacy" analysis codes are frequently unsuitable for use in automated design. They may crash for large classes of input, be locally non-smooth, or be highly sensitive to control parameters. To be useful, analysis programs must first be modified to reduce or eliminate only the undesired behaviors, without altering the desired computation. To do this by direct modification of the programs is labor-intensive, and necessitates costly re-validation.

This dissertation describes research into how legacy analysis codes can be usefully employed in design automation systems. We show that recovery from failure is possible when the failure occurs in the context of a search-based process such as optimization. We discuss the importance of failure context in determining the correct failure recovery action. We then describe an approach to failure recovery that is both context-sensitive and guarantees the integrity of the original computation to which it is applied.

We have implemented a high-level language and run-time environment (together called

LCM) that allow context-sensitive failure-handling strategies to be incorporated into existing Fortran and C analysis programs while preserving their computational integrity. Our approach relies on globally managing the execution of these programs at the level of discretely callable functions so that the computation is only affected when problems are detected. Problem handling procedures are constructed from a knowledge base of generic problem management strategies. We show that our approach is effective in improving analysis program robustness and design optimization performance in several real-world design domains.

Acknowledgements

It would be impossible to acknowledge everyone who has contributed to this work, but there are some without whom it would not have been possible. My deepest thanks to:

My wife, Carol, for her love, support, and endurance through this long process.

My advisor, Tom Ellman, for his patience, persistence, and guidance.

My parents, for their support and concern always.

My wife's parents, for their extra support and assistance during the hardest part.

Reva Kapur, who generously gave me a place to stay when I needed one.

Thanks to all of the members of the Rutgers HPCD project for their many contributions to this work. Special thanks to Saul Amarel and Lou Steinberg for their leadership.

Thanks to my committee – Michael Lowry, Naftaly Minsky, and Don Smith – for their thoughtful critiques and comments. This dissertation owes much to their careful review.

The research presented in this document is supported in part by NASA grants NCC-2-802 and NAG2-817. This research is also part of the Rutgers-based HPCD (Hypercomputing and Design) project supported by the Advanced Research Projects Agency of the Department of Defense through contract ARPA-DABT 63-93-C-0064.

Dedication

To Carol and Susannah.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

One of the "grand challenges" of Artificial Intelligence is to develop systems that are capable of carrying out complex engineering design tasks at a level of performance that meets or exceeds that of good human design engineers. While the state of the art may still be far from complete automation of design, it is possible to develop systems that can usefully augment human design efforts, by automating some of the relatively routine parts of the design process.

One such design task is the optimization of a design object for a particular purpose by adjustment of some continuous-valued numeric parameters that describe the design. An example might be the modification of the flap lengths that make up a jet-engine nozzle so as to minimize the amount of fuel required for engine performance under a specified set of conditions. An automated design system that could carry out this kind of optimization could "tweak" existing designs to improved performance, or could refine a conceptual design to concrete design. These tasks require the evaluation of potentially large numbers of candidate designs, making them good candidates for automation. Existing simulation and analysis programs could provide the basis for the development of such systems.

In this work, we address one significant class of problems encountered when constructing systems for automated optimizing design from "legacy" simulation and analysis programs: failure of the simulation programs during the design process. We describe the system we have implemented (**LCM:** The Legacy Code Manager) for incorporating failure handling strategies into legacy programs used in design, without direct modification of the legacy code. We give details of a language for reasoning about failures in design systems, and a

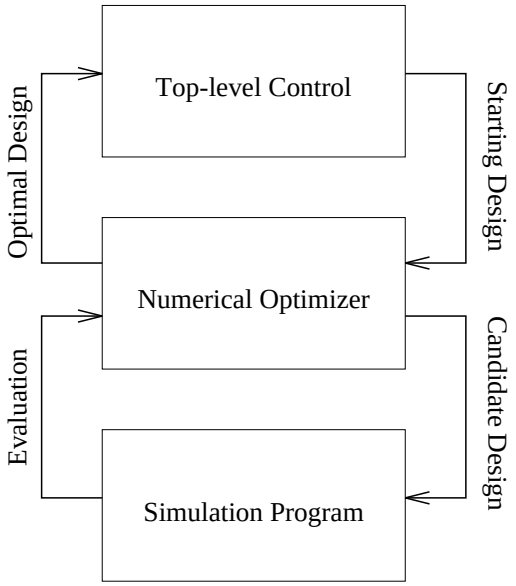


Figure 1.1: Idealized Model Design System Structure

database of generic strategies for failure handling. Finally, we show that the LCM system allows the rapid and safe incorporation of effective failure handling into design systems.

1.1 A Model Structure for Automated Design Strategies

Systems for automated design optimization of complex real-world objects can, in principle, be constructed by combining domain-independent numerical routines with existing domain-specific analysis and simulation programs. Figure 1.1 shows the simplified structure of such a system.

In this model, the top level of the system is responsible for selecting an initial design from which to begin the optimization process. This is passed to the numerical optimizer, along with additional information used to control the optimization process, such as constraint functions and bounds on the design parameters. The optimizer explores the design space by passing candidate designs to the simulation program and getting back the results. The optimizer continues to test candidate designs until it determines that it has found the best possible design subject to the constraints. (A more detailed example will be presented in

the next section.)

This model of the design process, though simple, would be adequate for many design problems if optimizers always reliably returned the true optimum and simulators never failed. The fact that this is not the case is the motivation for our work.

1.2 An Example

In this section we will present an example that illustrates a variety of failures that can occur when design optimization is attempted using legacy simulation codes in the evaluation function. We will then use the example to introduce some key ideas of our thesis.

1.2.1 A Nozzle Design Optimization Strategy

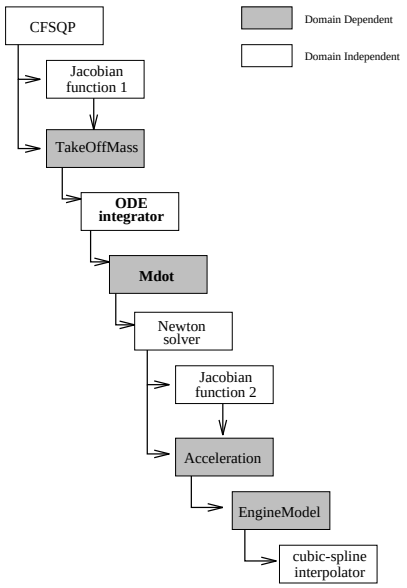


Figure 1.2: Optimization Strategy for Nozzle Problem

Consider the problem of designing the nozzle of a supersonic jet aircraft engine. Informally, the problem is to specify a set of lengths for the movable flaps that make up the nozzle of the aircraft engine so that the amount of fuel required to fly a goal mission using that nozzle is minimized.

Suppose that an *optimization strategy* for this problem has the structure shown in Figure 1.2. It is constructed around the problem domain-specific simulation programs, `Acceleration` and `EngineModel`, the CFSQP optimizer, and domain independent routines from the Numerical Recipes in C library.

The top level of the strategy is the `CFSQP` constrained sequential quadratic programming optimization routine [Lawrence *et al.*, 1994]. It calls `TakeOffMass` as the evaluation function to minimize, and a library routine `Jacobian-function-1` to compute numerical gradients. `TakeOffMass` takes three parameters describing the lengths of the nozzle flaps, a goal mission to simulate, and returns the total fuel mass required to fly the mission. The fuel mass for the mission is obtained by integrating the instantaneous fuel consumption function `Mdot` over the entire mission using the `ODE-integrator` library routine.

The function `Mdot` is obtained by solving for the instantaneous acceleration required at each point in the mission as a function of the throttle setting and angle of attack of the aircraft, using the multi-dimensional root-finder library routine `Newton-solver` (with `Jacobian-function-1` to compute the forward-difference Jacobian). The `Acceleration` function gives the instantaneous acceleration as a function of the instantaneous mass and the engine thrust computed by `EngineModel`. Finally, the engine model computes instantaneous thrust as a function of the current operating conditions of the engine and nozzle, using engine performance data interpolated from empirical results by the `cubic-spline interpolator` library routine.

This optimization strategy requires a starting design from which to begin. It is "correct" in the sense that given a point sufficiently close to the global optimum design (and with careful selection of the parameters that control the behavior of CFSQP) it will converge to the optimum. It is easy to verify empirically that it is virtually unusable as shown. For nearly all starting points in the search space, the optimization strategy will terminate abnormally, usually from errors occurring in the simulation routines. These errors are the result of coding errors (bugs) in the simulation routines, failure of the numerical methods (such as root-finders and interpolators) used in the simulation, and violations of implicit and

explicit modeling assumptions embodied in the simulator.

1.2.2 Failure in the Nozzle Design Strategy

There are many possible ways for a particular failure to manifest in this strategy. How the strategy will behave when a failure occurs will depend both on how and where the error is detected, and how it is handled (if at all). We list here some of the possible ways that one particular error might be detected and handled in our example strategy.

Consider what would happen if the optimizer attempts to evaluate a nozzle design that is adequate for only part of the mission, but causes the engine to produce insufficient thrust to meet the mission requirements at some point. This is a violation of an implicit modeling assumption that the engine-nozzle combination will be adequate for the goal mission. For points on the ODE integration of the mission for which the nozzle is OK, there exists a solution to the acceleration equations, and `Mdot` is defined.

Where the nozzle is inadequate, however, a number of failures might occur. In trying to solve for the required acceleration, the `newton` routine will attempt to increase the thrust computed by the `EngineModel` by trying higher throttle settings. Eventually, it will request a throttle setting that is greater than 100%, a physical impossibility. At this point:

- The `EngineModel` routine could fail to detect the problem, and crash.
- The `EngineModel` routine could detect the problem, and exit.
- The `EngineModel` routine could detect the problem, and return an extrapolated thrust value.
- The `EngineModel` routine could fail to detect the problem, and request a `cubic-spline` interpolation outside of the engine performance table. This could result in a plausible but incorrect thrust, or a wildly implausible thrust being returned.

Any of the above failures that do not lead to immediate termination could result in subsequent failure as the incorrect result propagates through the computation. Most importantly,

even if error handling somehow enabled the simulation to run to completion, the results would be invalid because the nozzle is, in fact, inadequate for the mission.

1.2.3 Failure during search

Failures such as those given in the previous section mean that the simulation cannot proceed, but they need not cause the termination of the optimization process. Because numerical optimization is a process of search, the end result does not directly depend on the evaluability of any single point along the search path. Only the terminal point actually returned as the optimum *must* be evaluable. Consequently, when a failure occurs during an optimization or other search-based process, it may be possible to *handle* it: to take some action that will allow the process to continue.

1.2.4 Failure Context and Recovery

Consider a second example of failure in the evaluation function: suppose the optimizer attempts to evaluate a nozzle design that is *just* adequate for the goal mission.

When the `Newton solver` routine attempts to solve for the acceleration at a point where the engine thrust is just enough, the solution throttle setting is approximately 100%. In the course of searching for the solution, `Newton solver` will almost certainly request some thrust values for throttle settings over 100%. In this case, if the `EngineModel` function can return an extrapolated value, the correct result will ultimately be obtained, as the solution converges to 100%. The failure, in this case, makes no difference to the correctness of the ultimate result.

This example illustrates the importance of understanding the *failure context:* it will determine whether a failure handling action is possible, and what form it should take. We use "context" here to include a number of things: the computational method in which the failure occurs, the purpose for which the value was to be used, the point in the search space where the failure occurs, and the behavior of the failing function at nearby points.

Finally, we note that in order to obtain sufficient failure context information to be able to reason as in the above example, we must have access to information about the internal structure of the evaluation function of the optimization strategy, and semantic knowledge about the functions being computed.

1.2.5 Computational Validity

Once the "throttle out of bounds" problem has been identified, we could make changes in the optimization strategy code to address it. We can add code to `EngineModel` to allow extrapolation from the table when the throttle is out of bounds. We would then have to add additional code to the `Mdot` function to test and ensure that the end result returned by **Newton Solver** was a valid point – that it had converged within the table. This test would require communication between `Mdot` and `EngineModel`, perhaps through a global variable. In the event that the end result was not valid, `Mdot` would have to signal the failure in some way that would allow the `TakeoffMass` function to return a "bad value" indication to `CFSQP`.

The changes described above involve making changes at several levels of the optimization strategy, with information passing across levels in unobvious ways. Given the number of changes required, it would be virtually impossible to guarantee that no unwanted side-effects would be introduced. A reasonable precaution would be to thoroughly re-test the code to ensure that it still behaves as desired.

This issue of ensuring the validity of the end results of the optimization strategy is an important one, particularly in real-world industrial design. If the correctness of the simulation cannot be ensured, the results of optimization are useless. We would like to have a means of performing failure handling *safely:* in a way that preserves the validity of the computation.

1.2.6 Legacy Analysis Codes and Automated Design

The idea of constructing an automated design system around a legacy analysis code is tremendously appealing. The legacy code is a codification of the expert domain knowledge necessary to evaluate the quality of a design. It usually represents an enormous "sunk cost" in initial development and validation. If it has been utilized (by design engineers) for any length of time, its behavior is probably pretty well understood within certain ranges: its strengths, its limitations, its idiosyncrasies. It is a useful design tool not because it is a perfect predictor of reality, but because the relationship between its predictions and reality is well-defined for some class of inputs.

There are disadvantages to using legacy codes. They predefine, to a large extent, the representation of the design object, and will have a large influence on the representation of the entire design problem. They may be sensitive to design factors in ways that are not relevant to the desired design problem, and may be insensitive to factors that are. They are usually "black boxes", written in languages like Fortran or C, and are consequently opaque to automated reasoning about their internal operation.

The single greatest disadvantage of using legacy codes is that they were never intended for use inside an automated design system, and can behave in ways that make them all but unusable. They may crash for large classes of input. They may be numerically unstable, changing output values wildly for small input changes. They may exit to the operating system when they cannot proceed, or write out error messages without other indication of problems. They may be highly sensitive to values of control parameters. They may return completely absurd values without any indication of a problem whatsoever.

The alternative to using an existing simulation program is to develop a new one for the purpose. If time and cost were not factors, this would nearly always be preferable (at least from the standpoint of the person constructing the design automation system), as the program would be developed specifically for the automated design process. Any new simulation code, however, must still be validated and ultimately must be *trusted* by design engineers before the results of a design process utilizing it will be accepted. And

although good software engineering tools and techniques exist to help ensure the correctness and robustness of newly-developed code, it is not possible to guarantee that even purpose-developed code will be lacking in flaws. As we will show in a later section, it is the nature of simulation programs to be unevaluable for some inputs.

1.2.7 Numerical Tools

Many of the same arguments in favor of using existing simulation programs are equally applicable to using existing numerical tools in constructing automated design systems. They are available, (relatively) inexpensive, validated, trusted, and well-understood. Much effort has gone into the development of routines for numerical optimization, and it would be nice to take advantage of it.

As with legacy simulation programs, existing numerical optimization tools encounter difficulties when used in design systems. These routines characteristically perform well when the function they are applied to is smooth, continuous, and unimodal – characteristics rarely true (except over tiny regions) for legacy simulators.

Developing new numerical optimization algorithms specific to each design problem is probably impractical. Even modifying existing optimization algorithms may not be easy, and raises questions about the trustworthiness of the end results.

1.3 Current Approaches

In the past, when these problems have been encountered by developers of automated design systems, there have been two choices: modify the simulation program or modify the numerical optimizer.

Where the simulation program is available and safely modifiable, new code can be added into the program that tests for conditions that would cause a failure to occur. For example, if a simulation is determined to be crashing because it is attempting to take the square root of a negative number, a test can be introduced before the computation to ensure that only

positive values are used. A negative number could be coerced to be positive, or more effort could be expended to find out why it is becoming negative, and correct the problem there.

Direct modification is slow, labor-intensive, and expensive. It requires a detailed understanding of the operation of the portion of the simulation program affected by the modification, and carries a great risk of introducing new bugs.

The problem is not always that analysis programs are "bad", or even that they can be "fixed". The failures may reflect the physical reality of the attempted simulation. The nature of real-world design objects is that they are subject to physical constraints which, if violated, make any evaluation of the object meaningless. For example, if the flap lengths of the nozzle are such that the nozzle cannot be connected into a four-bar linkage, the simulation cannot proceed. These constraints give rise to regions of the design space for which the value of the evaluation function is truly undefined. Optima (or solutions to equations) may lie at the boundary of these regions (*e.g.* where a constraint is active at the optimum). In these cases, the problem is not in the simulator, but in the attempt to apply it for a meaningless input.

Where keeping the simulator from crashing can't readily be accomplished, it may be possible to constrain the optimization process to avoid attempted evaluation of failing points. For example, in the nozzle design problem, a constraint function can be defined that is zero when the nozzle flaps are connectable, otherwise returning a value that indicates the degree to which they miss being connected; this constraint can be used by a constrained optimizer to avoid attempting to evaluate unconnectable nozzles. This approach has been successfully employed in work by [Schwabacher, 1996], who calls it "introducing modeling constraints."

When introducing modeling constraints is prohibitively difficult, the remaining alternative is to attempt to allow the optimization process to recover gracefully from failure.

Any approach that requires the direct modification of a program will be labor-intensive, error-prone, and of limited reusability even when similar types of errors occur in other locations.

1.4 My Approach

Our approach is motivated by three key ideas about failure handling in design optimization:

1. Failures occurring in the context of a search-based process such as design optimization can be *handled*, provided that certain properties of the result are maintained.

2. Successful failure handling requires reasoning about the *context* in which the failure is occurring.

3. If failure handling is to be of any use, it must be done *safely:* in a way that ensures the validity of the end results.

We have found the difficulty of developing failure-tolerant design optimization systems around legacy simulation programs to be a significant barrier to their use. To re-engineer simulation programs into optimizable functions it is necessary to add functionality to avert, handle, and capture information about failures; and to communicate this information to the optimization process in which it is embedded. Yet this must be weighed against the seemingly contradictory objective of not modifying the legacy code in any way that would necessitate its re-validation.

We have developed a methodology for achieving these goals, a run-time system that allows design optimization programs to be executed with safe, context-sensitive failure handling, and a set of tools that automate much of the effort required to build such failure handlers. The implementation of these elements comprises the Legacy Code Manager (LCM) system.

Figure 1.3 shows the example optimization strategy for the Nozzle design problem with the LCM run-time system architecture for failure handling. The key elements shown are:

- **Function Wrappers**

  Each discretely callable routine in the optimization strategy has code associated with it that takes control whenever that routine is called (*prologue* code) and exits (*epilogue* code). This wrapper code is represented by the bold boxes surrounding each routine

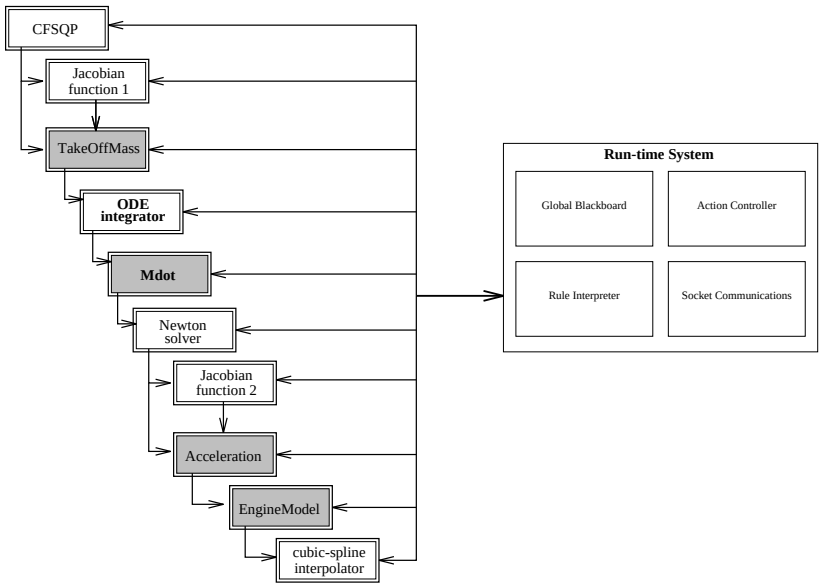


Figure 1.3: Nozzle Strategy With Failure Handling Incorporated

box in Figure 1.3. Associated with each wrapper are rules that determine the behavior of the wrapper. When wrappers get control, they communicate with the LCM run-time system, passing information about the parameters and return values of the routine being wrapped, and invoking the rule interpreter to determine what action is necessary.

Though rules are associated with individual wrappers, rules in many wrappers may participate in a single coherent failure recovery strategy. They can do this because they can transfer control to any other wrapper through the LCM run-time system, and inter-communicate through the global system blackboard. (The issue of ensuring the correct execution context during a non-sequential transfer of control will be addressed later, in section 2.8.1).

Wrappers receive control and handle exception conditions occuring within the routines they are wrapping. They can trap and take control of calls to the system exit routine. They can re-start a function, or terminate a "runaway" routine after a predetermined interval.

- **Rule Interpreter**

The rule interpreter is a Prolog-like inference engine that evaluates horn-clause-like rules using a modified form of resolution theorem proving. The rules are written in a language that includes many special-purpose predicates pertaining to the current computational context and failures that can occur. Rules may reference and alter values on the global system blackboard.

The rules associated with a wrapper are evaluated both before and after the execution of the wrapped routine. They test for failure conditions that require some form of recovery action.

- **Global Blackboard**

  The global blackboard contains state information about the computation (such as the run-time stack), and state variables that are created and referenced by the rules. The blackboard is both globally-accessible and stack-based, current state values and state value history can be explicitly accessed. The blackboard maintains a record of recovery actions taken during execution.

  The blackboard also provides a selective caching mechanism that records the history of function evaluation. It can be used for "memoizing" functions, or for generating approximations to function values on failure.

- **Action Controller**

  Associated with each rule in a wrapper are one or more actions. When a rule succeeds, the actions associated with it are carried out by the action controller. Actions include re-starting a failing routine with new parameters, invoking a routine as a new sub-process, invoking an alternative computation to replace a failing routine, or returning control to a routine other than the one in which the failure occurred. The action controller is also responsible for recording its activities on the global blackboard.

- **Socket Communications**

The wrapped code on the left hand side of Figure 1.3 does not necessarily have to execute in the same process/address space as the LCM run-time system, or even on the same machine. Communication between wrappers and the LCM run-time system can take place through a socket-based interface. The socket-based approach offers additional safety by providing greater run-time isolation, but at a greatly increased overhead cost. It is often necessary, however, to run legacy programs that are intended to be "stand-alone" systems.

1.4.1 Automating Failure Handler Incorporation

We have implemented and tested a system that automates much of the effort required to incorporate effective failure handlers into legacy analysis programs for use with the LCM run-time system. It is based on the key idea that while successful failure handling is context-sensitive, similar contexts may occur at multiple levels within individual systems, and across systems as well. We have developed a library of "generic" failure-handling strategies, each applicable to a class of failures, and a means to apply them to specific failure instances. It takes as input an unwrapped design optimization strategy such as that of Figure 1.2, and produces a ready-to-run wrapped strategy like that of Figure 1.3.

The compile-time part of the LCM system comprises several parts:

- A knowledge base of generic failure-handling schemata. These are sets of declarative rules describing a failure (or class of failures) that can occur in evaluation functions, and what action (or actions) can be taken.

- A knowledge base of function semantics in the form of a multiple-inheritance hierarchy of function types. This information is used to determine the applicability of schema to particular failures in an automated design system.

- A mechanism to instantiate generic failure schemata into specific failure-handling procedures, and install instantiated rules into wrapper code.

- A set of tools and utilities to incorporate intercepting code wrappers into C, Fortran, and Lisp legacy programs.

1.5 Thesis and Claims

It is the principle thesis of this dissertation that the LCM system improves the performance of design optimization systems built around "legacy" analysis and simulation programs. It does so by improving the robustness of the design optimization process through the incorporation of *failure handling* in a manner that facilitates rapid development, ease of re-use, and preserves the integrity of the legacy codes.

We now present a series of claims about our approach, and discuss each.

1.5.1 Claims

Improvement to Optimization Performance

We claim that the incorporation of failure recovery into design optimization strategies significantly improves optimization end results.

We will demonstrate the effectiveness of our approach by contrasting the results of carrying out optimizations with and without failure recovery incorporated. We will show that incorporation of failure recovery allows optimization to proceed that would otherwise fail, and that it improves the quality of design achieved per unit of computational cost.

Effectiveness of Multi-Level Failure Handling

By wrapping components of legacy simulation programs, our methodology allows failure handling to be incorporated at multiple levels within the simulation code.

We claim that this methodology permits failure handling strategies to be defined that are superior to strategies that can only operate outside of the legacy code. We will demonstrate this by contrasting the results of optimizations carried out with multi-level failure handling

with failure handling that is entirely outside. We will show that there is improvement in the quality of design achieved per unit of computational cost.

Context Sensitivity of Failure Handling

We claim that our methodology allows the incorporation of failure handling strategies into legacy programs that are sensitive to failure context: the computational method in which the failure occurs, the purpose for which the value was to be used, the point in the search space where the failure occurs, and the behavior of the failing function at nearby points.

We further claim that such context-sensitive failure handling strategies are superior for some failure instances than failure handling strategies that do not. We will demonstrate this by contrasting the results of optimizations with and without context-sensitive failure handling.

Composability of Failure Handling Strategies

We claim that one of the strengths of our approach is that strategies can easily be combined to increase failure handling power in the legacy program. We will demonstrate this by contrasting the results of optimization with and without composed strategies.

Safety

Our methodology incorporates failure handling strategies into legacy programs through the mechanism of code wrappers: routines that intercept control from the wrapped routines on entry and exit, and that error handling and communication with the LCM run-time system.

We claim that the use of these wrappers allows failure handling to be safely incorporated into legacy programs, eliminating the need to re-validate those programs. We will define three classes of failure recovery and prove that each preserves the correctness of the end results.

Ease of Failure Strategy Development

In the LCM system we have defined and implemented a language, toolkit, and run-time environment specifically for developing and testing failure handling strategies.

We claim that these allow for easier and more rapid development of new failure handling strategies, as compared to use of more conventional programming languages. We will give several examples of failure handling strategies developed using LCM, and will compare and contrast some with examples of traditional failure handling implemented directly in Lisp and C.

Degree of Automation

We claim that LCM automatically generates wrapper code for legacy programs from information contained in a database of function information. We further claim that generic failure handling strategies defined in the system database of strategies can be selected by the user and automatically instantiated into these wrappers.

We will describe these databases in detail, and give examples of generated wrappers and instantiated strategies.

Generality of Approach

We claim that our methodology and its implementation are general across a class of design optimization systems. We also claim that it is effective for a range of failure classes and contexts.

We will demonstrate this generality by showing that we have instantiated failure handling strategies, without modification, into design systems for multiple problem domains. We will also show that individual strategies can be instantiated in many ways within a single design optimization system.

Novelty

Finally, we claim that our approach is novel in a number of ways. We believe that our use of globally-communicating function wrappers is unique, as is our language for defining context sensitive failure handling strategies. We claim also to have advanced the integration of AI techniques and traditional approaches to numerical optimization.

We will support this claim with a discussion of related work at the conclusion of this dissertation.

1.6 Organization of this Dissertation

The following chapter provides a brief background on numerical optimization and automated design, as well as descriptions of the design domains used to test the LCM system. It describes some common classes of failure encountered in design optimization, and possible recovery strategies. Finally, it presents a proof that failure recovery can be carried out in a way that preserves the validity of the computation.

Chapter 3 describes the system that we have implemented, LCM, and chapter 4 gives the results of experimentation with LCM on a number of design optimization problems.

Chapter 5 discusses related work, and the concluding chapter summarizes the work and evaluates the performance of LCM against the claims made.

Chapter 2
Background

In this chapter we will briefly present a number of topics relevant to the discussion of the LCM system that will follow. We will describe the general problem of constrained non-linear optimization, and discuss how it applies to problems of design. We will give a concise description of the structure and behavior of several of the numerical tools that we have employed in constructing design automation programs. We will define an "optimization strategy," a concept we will use throughout the later discussion of our system.

We will then give a brief overview of each of the design problem domains for which we have tested the LCM system. For each, we will formally define the design problem, present an optimization strategy for solving that problem, and describe some of ways in which the optimization strategy can fail.

Finally we will identify characteristics of failures that are common across the problem domains, and discuss some possible strategies for handling them in the context of the design process. We will give a concise characterization of possible failure recovery actions, and will show that failure recovery can be effected in a way that preserves the validity of the final results of the optimization.

2.1 Constrained Non-linear Optimization

The problem of optimizing design of an artifact can be cast in the form of a constrained non-linear optimization problem, or a non-linear programming problem [Vanderplaats, 1984, Moré and Wright, 1993, Gill *et al.*, 1981]. These problems have the form:

Given a representation of a design object in the form of a vector of numeric design parameters $\overline{X} = (x_0, x_1, ..., x_n)$, a non-linear constraint vector function on the design parameters $\overline{\mathbf{g}}(\overline{X})$, and a non-linear objective function $\mathbf{f}(\overline{X})$, then:

$$\min \mathbf{f}(\overline{X})$$

$$\text{subject to } \overline{\mathbf{g}}(\overline{X}) \leq \{\mathbf{0}\}$$

The values of each of the design parameters are drawn from their respective sets of valid values $\{\mathcal{V}_0, \mathcal{V}_1, ..., \mathcal{V}_n\}$, such that $x_0 \in \mathcal{V}_0, ..., x_n \in \mathcal{V}_n$. The *space of possible designs* is therefore the Cartesian product of the sets of valid values $\mathcal{V}_0 \times \mathcal{V}_1 \times ... \times \mathcal{V}_n$.

We will assume, without loss of generality, that there is a single set of values $\overline{X}_{opt}$ that minimizes $\mathbf{f}(\overline{X})$ (subject to the constraint function $\overline{\mathbf{g}}$), which we refer to as the *global optimum*. In practice, it may not be possible to determine the precise global optimum, and there may several $\overline{X}_i$ for which $\mathbf{f}(\overline{X}_i) - \mathbf{f}(\overline{X}_{opt}) \leq \epsilon$, but we will consider these solutions to be equivalent.

We will define a *design process* as the means by which, given $\mathbf{f}$ and $\overline{\mathbf{g}}$, a succession of $\overline{X}$ are chosen and evaluated until $\overline{X}_{opt}$ is determined. This may be performed by a human design engineer, or, algorithmically, by an automated system. Throughout this thesis, we will assume that the design process is principally carried out by a constrained non-linear optimization algorithm.

2.2 Domain-independent Numerical Tools

We make use of a number of domain-independent numerical algorithms in this work. We will now describe the operation of some of the *second-order* numerical methods employed: those that take an evaluation function as an input argument. We will discuss how the information returned by the evaluation function is utilized in each.

2.2.1 Optimization Methods

There is no single "best approach" to non-linear numerical optimization. Many methods are available, and each may be effective for some problems. All are sensitive to factors such as scaling of the parameters, starting point selection, and termination conditions. None are guaranteed to find the global optimum.

If the search space is finite, enumerable, and small, and the cost of evaluation is small, exhaustive search may be possible. Exhaustive search is rarely a practical alternative, particularly for functions of real-valued parameters.

Where the evaluation function to be optimized has a very large number of local optima or is extremely non-smooth or discontinuous, stochastic optimization methods may be of use. By incorporating a degree of randomness into the search, noise and local minima can sometimes be avoided. Stochastic optimization methods tend to be very expensive in the number of evaluations they require, and are less suitable for use in design problems where evaluation is expensive.

Some non-linear numerical optimization methods use gradient information to direct their search. Generally, these methods alternate calculation of a local gradient approximation with a single-dimensional sub-optimization along a promising trajectory. Gradient-based methods are most effective on reasonably smooth unimodal functions. They are extremely sensitive to starting point selection, and the number and magnitude of local minima in the function. When they can be successfully used, these methods can be quite economical in their use of evaluations.

We have chosen to use two different optimization algorithms, one gradient-based that accepts non-linear constraints, and the other unconstrained and not gradient-based.

The CFSQP Optimizer

A *quadratic programming problem* is a non-linear programming problem where the objective function $\mathbf{f}$ is quadratic, and the constraint vector function $\mathbf{g}$ is linear. The gradient-based

CFSQP_optimizer

- Takes as input:
  - an initial point,
  - an objective function,
  - and a set of constraint functions.
- If the initial point is infeasible, search for a feasible point by (recursively) minimizing the maximum of the contraint functions until all are ≤ 0.
- Until either the local gradient is approximately 0, or the step taken is smaller than some ϵ, or the change in the objective function from the prior step is smaller than some ϵ do:
  - Compute the local gradient of the constraint and the objective functions, and update the approximation of the Hessian.
  - Solve for P, the solution to the quadratic programming problem of the quasi-inverse Hessian.
  - Search along a line from the last step to P until a point X is found that satisfies all of the constraints, and has a smaller objective function value.
  - Step to the new point.
- Returns:
  - the best point found.

Figure 2.1: CFSQP Algorithm

CFSQP optimization algorithm [Lawrence *et al.*, 1994] operates by solving a converging sequence of quadratic programming problems that are approximations to the objective function. Figure 2.1 gives the an overview of the operation of the algorithm (For a detailed description of CFSQP, refer to [Lawrence *et al.*, 1994]).

The CFSQP algorithm iterates between two phases: in the first, local gradient information about the objective and constraint functions is used to find a direction in the parameter space in which the function is likely to be both improving and feasible (satisfies the constraint vector function). In the second, the algorithm steps back along a line from the predicted optimal point to the best point found so far until a better feasible point is found. During both phases, the evaluation function is called to evaluate points, but the way in which the

results of the evaluation are used is different, depending on the phase. During the first phase, the evaluation function is used to numerically approximate the gradient. During the second phase,the evaluation function is used only to compare the relative merits of different points in a pairwise manner. The process is repeated until the termination criteria are satisfied.

The CFSQP algorithm is a *search-based method:* there is no specific set of points that *must* be evaluated by this algorithm for any one optimization problem. The points evaluated will depend on the initial point provided, the value selected for the gradient epsilon, and values returned by the evaluation function during the search. Any number of different points can be used for the gradient computation; for example, forward-, backward-, or center-difference computational methods could be used.

The Downhill-Simplex Optimizer

Neither gradient-based nor stochastic, the Downhill-Simplex optimization algorithm is a simple multi-dimensional optimizer that is roughly analogous to a one-dimensional interval search method. It is generally less evaluation-efficient than CFSQP, but may be more robust for some problems. Figure 2.2 gives a high-level description of the operation of the algorithm (for a detailed description, refer to [Press *et al.*, 1986]).

The algorithm takes as input an initial simplex – a set of N+1 points for an N-dimensional problem – and an objective function. It operates principally by repeated reflection and contraction of the simplex (though it can sometimes expand it), until all of the vertices are within some epsilon of the optimal point.

Unlike the quasi-Newton methods, Downhill-Simplex does not compute a local gradient to set its search direction. It utilizes the evaluations of the objective function in only one way: to determine the ordering of the vertices. In this regard, it is using the information in the same manner as the quasi-Newton methods do during the line search phase of each iteration.

As with CFSQP, this is a search-based method: there is no specific set of points that must be evaluated in order to find the optimum. The algorithm is deterministic, but the

DownHillSimplex

- Takes as input:
  - an initial simplex,
  - and an objective function.
- Until the Euclidean distance between the vertices of the simplex is less than ϵ, do:
  - Evaluate the vertices of the simplex using the objective function.
  - Order the vertices according to their evaluations [best-vertex, next-best-vertex, ... worst-vertex].
  - Make a new simplex by reflecting the worst vertex through the hyperplane of the rest of the simplex.
  - If the new vertex is better than the next-worst, try doubling the reflection distance; if worse, try halving the distance.
  - If halving the distance doesn't give a better point than the next-worst vertex, contract the entire simplex by a fixed amount.
- Returns:
  - the last simplex.

Figure 2.2: Downhill Simplex Algorithm

NewtonRaphson

- Takes as input:
  - an initial-point,
  - and an objective function.
- Until either the Euclidean distance between successive steps is less than some ϵ, or the value of the last point evaluated is within some ϵ of 0, do:
  - Compute local gradient (Jacobian) with a numeric gradient approximation function.
  - Solve for the point P, the solution to the inverse Jacobian.
  - Search along line from P to the last step point until a point X is found that is better than the last step point by some threshold amount.
  - Step to the new point.
- Returns:
  - The last step point.

Figure 2.3: Newton-Raphson Algorithm

points evaluated will depend on the selection of the initial simplex, and on the degree to which the simplex is contracted when reflection is not successful.

2.2.2 The Newton-Raphson Equation Solver

The Newton-Raphson algorithm provides a means of solving for the roots (zeros) of a system of non-linear equations. It can be very sensitive to the selection of a starting point: for good choices it will converge to a solution quickly, bad choices can cause the algorithm to fail to converge at all. The algorithm is outlined in Figure 2.3 (for a detailed description, refer to [Press *et al.*, 1986]).

As can be seen in Figure 2.3, the Newton-Raphson algorithm is very similar in form to CFSQP. (CFSQP is, in fact, referred to as a "quasi-Newton method" [Lawrence *et al.*, 1994]). The search for a solution is an iterative process involving two phases: a computation of the local gradient, and a line search to find an improving point. Newton-Raphson is an

unconstrained method: there is no notion of feasibility of a point. Gradient information is not accumulated, the direction for the line search is based on the most recent gradient evaluation.

As with the CFSQP algorithm, the information returned by the evaluation function is used differently depending on where it is being called. During the line search, the exact values of the evaluations are less important so long as the property that points nearer the solution have lower values is true.

The Newton-Raphson algorithm is a search-based method, in that it does not depend on the evaluation of any specific set of points to converge to solution. The points evaluated will be a function of the initial point provided, the gradient computation method used, and the values returned by the evaluation function during the search.

2.2.3 The Runge-Kutta ODE Integrator

The Runge-Kutta algorithm integrates an N-dimensional system of first-order ordinary differential equations. The algorithm given in Figure 2.4 has also been taken from Numerical Recipes in C.

The algorithm shown in Figure 2.4 takes as input a vector of initial values, a time interval over which to integrate, the number of time steps to use in the integration, and the derivative vector function dy/dx(x,t).

Unlike the optimization and root-finding algorithms, Runge-Kutta contains no element of search. For a given time interval and discretization, we can predict for any input derivative function what points will be evaluated. This characteristic is desirable in that it assures that the algorithm will require a predictable amount of computation to return a result. It is undesirable in that we cannot predict in advance what the computational error will be. There exist adaptive Runge-Kutta methods that offer the converse tradeoff: they may take arbitrarily long to return, but they will return a result within a desired degree of accuracy.

RungeKutta

- Takes as input:
  - an initial value for the dependent variable, y(t0),
  - a starting value for the independent variable, t0,
  - an ending value for the independent variable, tn,
  - the number of steps to take from t0 to tn, nsteps,
  - the derivative function to integrate.
- Loop for ti from t0 to tn in nsteps steps, do:
  - Compute the value for each step according to the equations:

$$\begin{aligned} h &= t_{i+1} - t_i \\ k_1 &= hf(t_i, y(t_i)) \\ k_2 &= hf(t_i + \frac{h}{2}, y(t_i) + \frac{k_1}{2}) \\ k_3 &= hf(t_i + \frac{h}{2}, y(t_i) + \frac{k_2}{2}) \\ k_4 &= hf(t_i + h, y(t_i) + k_3) \\ y(t_{i+1}) &= y(t_i) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \end{aligned}$$

- Returns:
  - The value of y(tn).

Figure 2.4: Fixed Fourth-order Runge-Kutta Algorithm

2.3 Desirable Properties of Evaluation Functions

In the prior section, we showed how a number of second-order numerical tools utilize evaluation functions. By looking at the examples of the algorithms, we can draw some conclusions about how evaluation functions should behave to be successfully used with these methods.

In general, search-based methods determine a promising direction in which to proceed from the values returned by the evaluation function, based on the the assumption that the localized behavior of the function will be a useful predictor of the global behavior of that function. The success of this technique will depend on the degree to which this expectation is met by the behavior of the evaluation function.

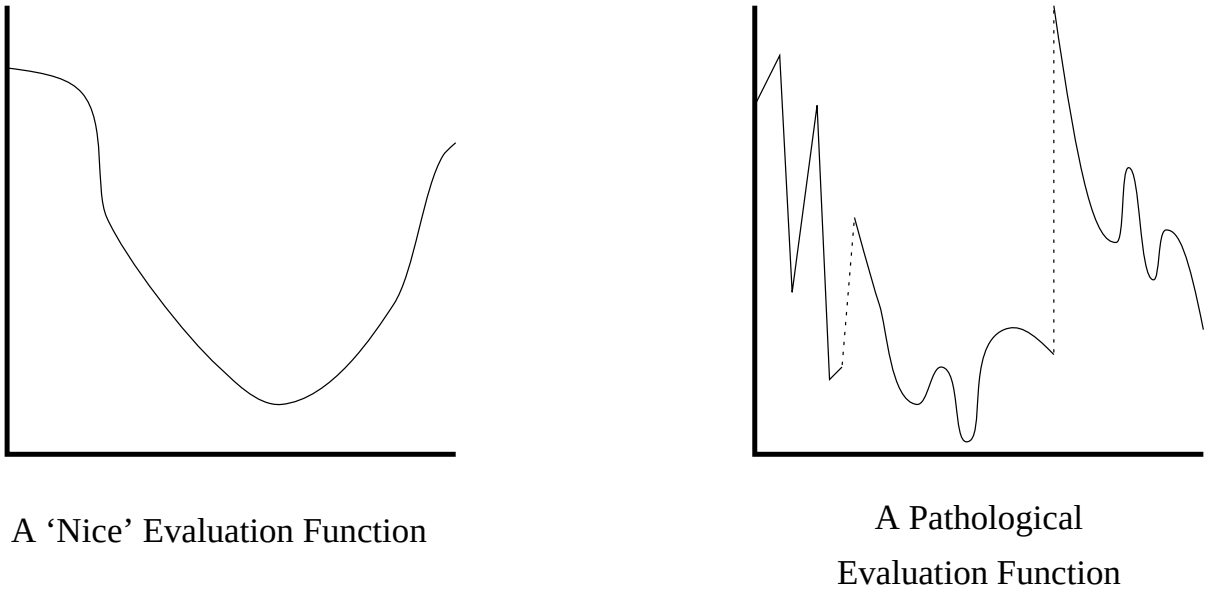


Figure 2.5: Good and Bad Evaluation Functions

Figure 2.5 contrasts the graphs of two one-dimensional evaluation functions. The numerical algorithms previously discussed will be unlikely to experience difficulty with the former, however it is our experience that the latter are more characteristic of evaluation functions for "real-world" design problems, as has been seen from the descriptions of the test design domains.

For the purposes of line searching in CFSQP and Newton-Raphson, and for ordering the vertices in Downhill-Simplex, the evaluation function should have the property for any pair

of points A and B, if point A is feasible and nearer (Euclidean distance in parameter space) the global optimum than point B, the evaluation of A should be less than the evaluation of B. We can see from examination of the algorithms that for these purposes the exact values returned are not important, so long as the order relation holds.

For the purposes of gradient computation in CFSQP and Newton-Raphson, the evaluation function should have the property that the values returned during gradient computation should, when used to compute the gradient (or the Hessian) yield a result that gives a "reasonable" indication of the direction and distance of the global optimum. Evaluation functions that have this property will be continuous, unimodal, locally approximatable by low-order polynomials, and defined everywhere.

The Runge-Kutta algorithm, by contrast, depends on the evaluability of its function at specific points. A change in the value of any single point will directly affect the result of the entire computation. Runge-Kutta is not a search-based method.

When a failure occurs in an evaluation function, a knowledge of how the value returned by a function would have been used is a part of the failure context. We will show later how we can use this knowledge to decide how to handle a failure in a way the preserves the desired properties of the evaluation in that context.

2.4 Optimization Strategies

For any given design problem, there are multiple ways in which the optimization can actually be carried out. A design optimization may be carried out in a single step, or in a sequence of sub-optimizations. Certain parameters may be determined to be entirely determined by others, and be *incorporated* or removed from the search and replaced with a direct calculation. These decisions about how the optimization should be done lie outside the scope of this work. We assume throughout that we have been handed a working construct that incorporates these choices.

We take from Ellman [Ellman and Murata, 1996] the term *optimization strategy* to describe this executable configuration of optimization methods and other domain-independent numerical tools with domain-specific simulation programs. The optimization strategy is a completely specified algorithm, containing all of the elements required to carry out the desired design optimization process. It is correct in the sense that in the absence of any failure during its execution, it will compute the desired optimum design. Several examples of optimization strategies will be given in the next section.

2.5 Test Design Problem Domains

2.5.1 Racing Yacht Hull Design

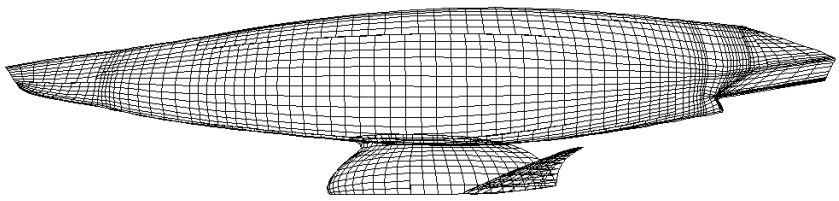


Figure 2.6: The hull and keel of *Stars & Stripes '87*

Statement of the Problem

Given:

- A prototype description of a racing yacht, such as that shown in Figure 2.6.
- A parameterization of the design of the hull into a set of values that specify a degree of deformation from the prototype along each of the major dimensions: beam (width), length, and draft (height).
- A set of upper and lower bounds on the parameter values.
- A set of imposed constraints on the design parameters, called the "12-meter rule."

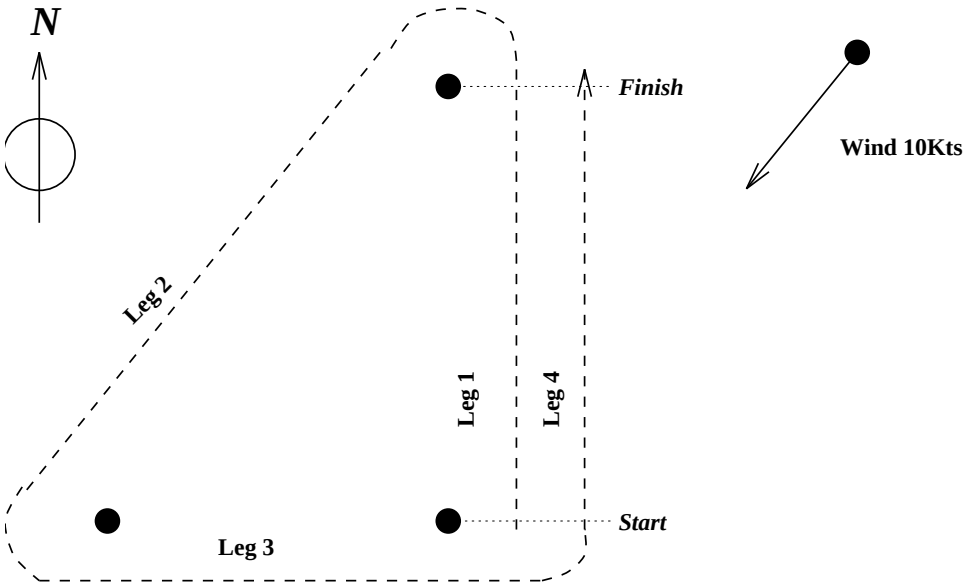


Figure 2.7: A Four-leg Racecourse

- A goal, consisting of a racecourse description such as that shown in Figure 2.7, a windspeed and direction, specifying the conditions under which the yacht is to perform.
- An evaluation function taking the prototype, the goal, and the design parameters, that returns the minimum time required to sail the specified yacht around the goal racecourse under the goal wind conditions.

Find:

- A set of values for the design parameters within the bounds given such that the 12-meter rule is satisfied and the evaluation function (*course-time*) is minimized.

In addition to the 12-meter handicapping rule, the design is subject to various physical constraints, (*e.g.* "the hull must displace a sufficient volume of water to at least equal the mass of the vessel, or the yacht will sink"). The 12-meter rule is intended to constrain certain key elements of the design to within a fairly small range of values, essentially allowing hull size (sailing yachts with larger hulls are generally faster than those with smaller hulls) to be traded off against total sail area (yachts with larger sails are generally faster than those with smaller sails).

Because of the high level of competitive skill involved in yacht racing, very small differences (on the order of one part in a thousand) in yacht performance can make the difference between winning and losing. This design problem is one in which a very small improvement in overall performance can have a disproportionately large payback.

Description of an Optimization Strategy

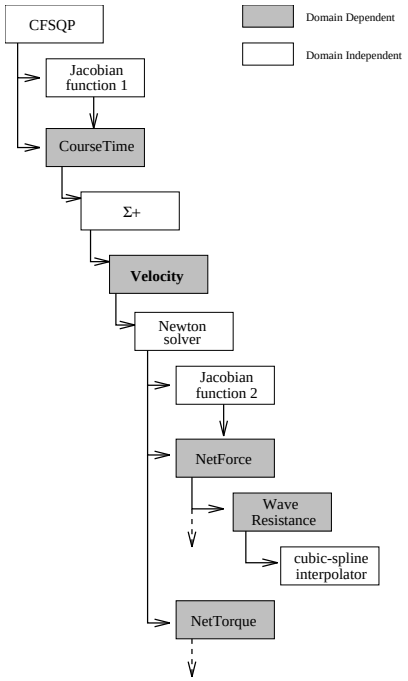


Figure 2.8: Optimization Strategy for Yacht Hull Design

Figure 2.8 shows an optimization strategy for the yacht hull design problem. The strategy takes an initial prototype, goal racecourse and wind conditions, and optimizes the three hull parameters for minimum course time.

The strategy is constructed from domain-independent numerical codes, including the CFSQP optimization algorithm and routines from Numerical Recipes in C and other sources. The domain-dependent functions were developed by Tom Ellman and John Keane, and were originally derived from a commercial racing yacht analysis package, **AHVPP1**, developed and marketed by AeroHydro, Inc. [ME, 1992]. Tables of empirical data were either derived or extracted verbatim from **AHVPP1**.

The strategy operates by using the `CFSQP` optimizer to minimize the value computed by the `CourseTime` function. `CourseTime` takes as input a 13-parameter description of a complete prototype yacht, three parameters representing the current hull design, and the goal racecourse and wind conditions. It returns the minimum time, in seconds, that the prototype yacht, using the design hull, would require to sail the goal racecourse. The `CourseTime` function is also called indirectly to compute the local gradient by the library routine `Jacobian-function-1`.

The course time is computed by solving for the maximum straight-line velocity that the yacht can sail along each leg of the race course separately, using the `Velocity` function. `Velocity` takes as input the current design, the direction in which the yacht is sailing, the direction and speed of the wind, and returns the maximum speed that the yacht can go in that direction. The results are summed over all legs with the function `Summation`.

The maximum velocity is computed by using the `Newton-solver` library routine to solve for the velocity and angle of heel of the yacht such that the linear and rotational forces as computed by the `NetForce` and `NetTorque` functions are balanced. When the yacht is in motion, it is affected by forces that arise from its movement through the air and the water. Some of these forces are linear, and will tend to propel it forward, others will tend to resist its motion. Driving forces tend to diminish with increasing velocity, and resistive forces to increase. Rotational forces behave similarly. When the forces are in balance, the yacht is moving at a constant velocity at a constant angle of heel.

The `NetForce` and `NetTorque` functions utilize subordinate functions (only `WaveResistance` is shown) that compute force components as a function of the yacht geometry, the windspeed and heading, and the velocity and heel.

The `WaveResistance` function shown computes a resistive force arising from the energy lost by the yacht in raising a wake. It utilizes a table of empirical data derived from tank tests on hull models, and the `cubic-spline interpolation` library routine to determine intermediate values. `WaveResistance` is typical of several such functions at this level of the strategy.

Examples of Failure

The yacht hull design optimization strategy, as described, is subject to a variety of failures. A partial list follows:

- Interpolation outside a data table.

  A number of functions to compute component forces used by the `NetForce` and `NetTorque` functions use tables of empirical data with an interpolation function. It is often the case during the course of an optimization (and during the course of root finding) that a trial point will be evaluated that requires data outside the table. An example is the wave resistance data, which gives wave resistance as a function of ship velocity, and is defined only down to 0 knots. The `Newton solver` will often attempt to evaluate a point with negative velocity in the course of solving for the balance of forces.

  When an interpolation is requested outside the table, the interpolation function itself usually generates a detectable system error. In some cases, a value is returned that is absurdly large or small (due to the behavior of spline interpolation functions outside their defined range), causing subsequent failure as the result is used in other calculations.

- Inability to satisfy the 12-meter constraint rule.

  The optimization strategy described satisfies the 12-meter constraint rule by solving for the sail area necessary to satisfy the constraint as a function of the other design variables. For some designs, a negative sail area may be required to satisfy the constraint – a physical impossibility. When this occurs, the function that computes the sail area signals a system error.

- No solution to balance of forces.

  Some yachts are unstable in some wind conditions, they will tip over. If an unstable yacht is evaluated, there is no solution to the balance of forces, and the `Newton solver`

will eventually terminate gracelessly by calling the C exit function. This terminates the entire design session.

- Negative velocity or heel solution.

  The simulation code for this design problem was developed with the assumptions that yacht's velocity would always be non-negative, and the angle of heel would always be between 0 degrees and 90 degrees. Because of these assumptions, there are some anomalous solutions to the balance of forces for negative heel or velocity. The `Newton solver` will occasionally converge to these spurious roots, which can result in negative course times being returned.

- Optimizer returns point not local optimum.

  The 12-meter constraint rule contains some discontinuities (the result of "penalty" conditions) that can cause sudden changes in the sail area of a yacht, and correspondingly discontinuous changes in the evaluation function. These discontinuities give rise to "ridges" in the design space that can be extremely difficult for the CFSQP optimizer to handle. Where the optimum lies in the vicinity of a ridge, CFSQP will often terminate at a point that is neither the global nor a local optimum.

- Modeling constraint violation.

  The simulation code checks to ensure that the design to be evaluated is physically meaningful. Yachts that, for example, fail to displace their mass will cause the simulator to signal a system error.

- Unexplained failures.

  There are points, distributed throughout the design space, for which the optimization strategy simply crashes. These appear to be the consequence of coding problems in the simulation programs.

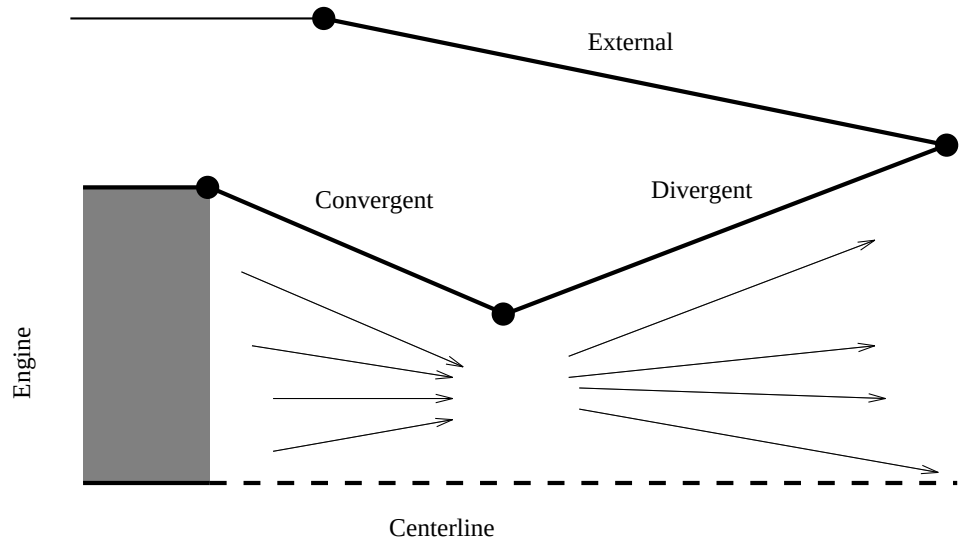


Figure 2.9: Three-flap Convergent-Divergent Nozzle

2.5.2 The Conceptual Design of Jet Engine Nozzles

A jet engine nozzle derives its power from the compression and combustion of fuel and air, and the subsequent acceleration and re-expansion of the hot exhaust gasses. Instrumental in this process is the *nozzle* of the engine. Nozzles on engines intended to operate on supersonic aircraft will typically be constructed from movable elements so the geometry can be changed for different operating conditions.

Figure 2.9 shows a stylized representation of a side view of an axi-symmetric *convergent-divergent* nozzle for a supersonic jet engine, configured as a four-bar linkage with one degree of freedom. The hot, compressed gasses coming from the engine (at the left) are first expanded through the convergent section of the nozzle, and then through the divergent section. Ideally (ignoring certain losses), the velocity of the fluid at the point of maximum constriction in the nozzle should be exactly Mach 1.0 (the speed of sound at ambient temperature and pressure). A nozzle configuration may be fixed, or may be able to change in operation, to adjust for changes in ambient conditions.

Statement of the Problem

Given:

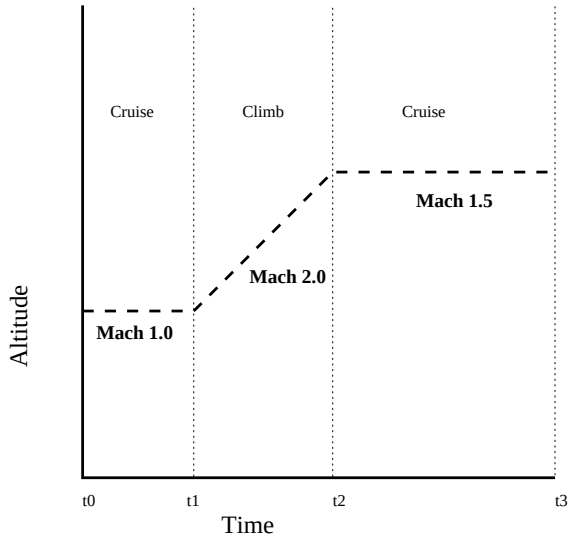


Figure 2.10: Multiphase Mission

- A prototype description of a complete aircraft.
- A parameterization of the design of the nozzle into a set of three flap lengths: convergent, divergent, and external.
- A set of upper and lower bounds on the parameter values.
- A goal mission for the aircraft to fly, consisting of a sequence of *phases;* each phase consists of a starting and ending altitude, a speed (expressed in Mach), and a duration in seconds. An example mission is shown in Figure 2.10.
- An evaluation function taking the airplane prototype, the goal mission, and the design parameters that returns the mass of the aircraft at the beginning of the mission when the plane is loaded with just enough fuel to fly the goal mission.

Find:

- A set of values for the design parameters within the bounds given such that the evaluation function (*takeoff mass*) is minimized. The takeoff mass depends on both the fixed "empty mass" of the plane and the variable mass of the fuel.

The fuel efficiency of a jet engine is highly dependent on nozzle design. Even small changes in the efficiency of the nozzle will have a very large impact on the lifetime cost of operating the aircraft. This design problem is also one in which very small improvements may have a large economic impact.

Description of an Optimization Strategy

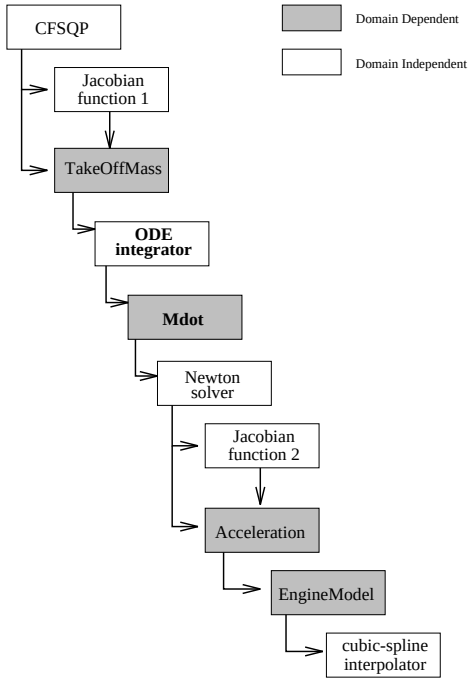


Figure 2.11: Optimization Strategy for Nozzle Problem

The calling structure of an *optimization strategy* for this problem (previously shown in Chapter 1) is shown again in Figure 2.11. The domain-dependent simulation code for this problem was developed at Rutgers by Andrew Gelsey, Don Smith, and Keith Miyake in collaboration with domain experts from Lockheed and GE Aerospace. A Lisp re-implementation was later undertaken by Takahiro Murata.

We will briefly re-cap the operation of this strategy here. The top level of the strategy is the `CFSQP` constrained sequential quadratic programming optimization routine [Lawrence *et al.*, 1994]. It calls `TakeOffMass` as the evaluation function to minimize, and a library routine `Jacobian-function-1` to compute numerical gradients. `TakeOffMass` takes three

parameters describing the lengths of the nozzle flaps, a goal mission to simulate, and returns the total fuel mass required to fly the mission. The fuel mass for the mission is obtained by integrating the instantaneous fuel consumption function `Mdot` over the entire mission using the `ODE-integrator` library routine.

The function `Mdot` is obtained by solving for the instantaneous acceleration required at each point in the mission as a function of the throttle setting and angle of attack of the aircraft, using the multi-dimensional root-finder library routine `Newton-solver` (with `Jacobian-function-1` to compute the forward-difference Jacobian). The `Acceleration` function gives the instantaneous acceleration as a function of the instantaneous mass and the temperature and pressure of the gasses at the entrance to the nozzle, as computed by `EngineModel`. Finally, the engine model computes instantaneous thrust as a function of the current operating conditions of the engine and nozzle, using engine performance data interpolated from empirical results by the `cubic-spline interpolator` library routine.

Examples of Failure

- Interpolation outside a data table.

  The simulation models for this problem depend heavily on the use of tables of empirical data which are interpolated to provide intermediate values. It is frequently the case that the optimizer or root finder will request an evaluation that requires data outside the table. For example, if the nozzle design being evaluated does not enable the engine to produce enough thrust to meet the mission profile, the `Newton solver` will eventually attempt to evaluate throttle settings greater than 100%. The table of engine performance data is only defined up to 100%, so these attempts cause the interpolation method to signal an error.

- No solution to acceleration equations.

  The thrust produced by a jet engine as a function of throttle setting is discontinuous,

due to the action of the afterburner. The thrust varies relatively smoothly from 0-50%, but experiences a sudden increase at the point where the afterburner is engaged. This discontinuity means that there may be points in the mission profile for which there is no exact solution matching the profile. When this occurs, the `Newton solver` terminates abnormally after a specified maximum number of iterations by calling the C exit function, resulting in the termination of the optimization strategy.

- Multiple roots to acceleration equations.

  For some points in the mission, there may be multiple valid solutions to the acceleration equations. This may cause the `Newton solver` to reach one solution for one design, but reach another solution at a nearby point. This gives rise to discontinuities in the evaluation function that (as previously discussed) can cause CFSQP to fail.

- Sanity check failure.

  There are some explicit "sanity" checks built into the nozzle simulation programs. These are there to determine whether a portion of the solution being computed is valid. One such check tests to see that the work required to move the nozzle flaps through a range of motion is positive. A bad choice of discretization by the simulation program can result in negative work, and invalidates the simulation results. If this is detected, an error is signaled.

- Optimizer returns point not local optimum.

  Because the evaluation function for the nozzle design problem is non-smooth, the optimizer can terminate at a point that is not a true local optimum.

- Modeling constraint violation.

  There are a number of physical constraints on the design of the nozzle that are explicitly tested for in the simulation code. These include the ability to actually connect the flaps to form a four-bar linkage, the ability of the nozzle to move through a range of positions, and the ability of the nozzle to create a sufficiently small constriction to

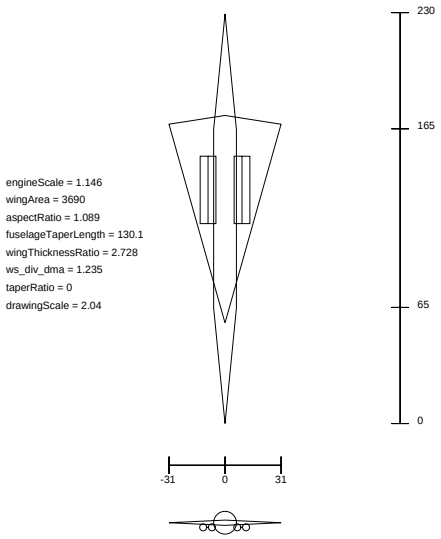


Figure 2.12: Conceptual Design of Supersonic Civil Transport

allow the proper operation of the engine ("choke the flow") at all points in the mission. Violation of any of these modeling constraints is signaled by an error.

- Unexplained failures.

  Though developed to the highest software standards by trained professionals, the nozzle simulation model still occasionally crashes for reasons that defy easy explanation.

2.5.3 Conceptual Design of Airframes for Supersonic Civil Transport

There are many factors in addition to nozzle geometry that will affect the fuel efficiency of an aircraft. The geometry of the fuselage and wing of the aircraft, as well as the number, size, and placement of the engines will all impact performance. This problem extends the nozzle design problem, by including a more detailed description of the aircraft. Figure 2.12 shows a typical design.

Statement of the Problem

Given:

- A prototype description of a complete aircraft.
- A parameterization of the design of the aircraft into twelve values: the three nozzle flap lengths plus one new nozzle parameter, an engine scaling parameter, four wing specification parameters, and three fuselage parameters.
- A set of upper and lower bounds on the parameter values.
- A goal mission for the aircraft to fly, consisting of a sequence of *phases;* each phase consists of a starting and ending altitude, a speed (expressed in Mach), and a duration in seconds. An example mission is shown in Figure 2.10.
- An evaluation function taking the airplane prototype, the goal mission, and the design parameters that returns the mass of the aircraft at the beginning of the mission when the plane is loaded with just enough fuel to fly the goal mission.

Find:

- A set of values for the design parameters within the bounds given such that the evaluation function (*takeoff mass*) is minimized.

Description of an Optimization Strategy

The simulation code for this problem includes that of the previous section, with new domain-dependent code added to evaluate changes in the aircraft structure previously assumed to be constant. The overall goal is the same, however, the minimization of takeoff mass of the aircraft flying a specified mission.

Two separate implementations of the simulation programs used in this optimization strategy were utilized in this work: one written in a mixture of Lisp and C by Arunava Banerjee, and one written entirely in C by Gelsey, *et. al.*. The calling structures of the two differ only in minor ways.

Examples of Failure

The failures in this model include all of those previously encountered in the nozzle model. In addition, new functions and new modeling constraints increase the failure frequency so that a much smaller percentage of the design space for this problem is evaluable.

2.6 Classes of Failure

From the descriptions of failures in the previous section, it is apparent that some forms of failure occur across the problem domains. The existence of domain-independent failures suggests that we may be able to devise techniques for domain-independent failure handling.

In this section we will characterize some "generic" failure categories, drawing on the examples given previously. We will describe how they can be detected, and suggest possible techniques for handling the error in a way that will allow the design process to continue.

2.6.1 "Normal" Termination of a Search-based Method at an Undesired Point

Failure Description and Detection

We have seen that this failure can occur wherever search-based methods are used in the optimization strategies, but most commonly (or at least most visibly) at the top level optimization. The failure occurs when the search-based method returns normally, but the result returned is incorrect. In the case of an optimization, the result is not a local optimum; in the case of a root-finder, the result is a spurious root.

This failure is most often caused by noise or non-smoothness in the evaluation function used by the search-based method. It may also be caused by a poor selection of control parameters for the search-based method: such as the numerical gradient computation step size, the convergence tolerance parameters, or the initial starting point (the seed).

Failures of this kind can be difficult to detect, since no explicit error is returned. In

the absence of other knowledge about the correctness of the result (such as knowing that a valid root of an equation cannot be negative), detection must rely on additional exploration of the evaluation function in the neighborhood of the returned result. Such exploration must necessarily be heuristically-guided, and will not be guaranteed to detect the failure. A probabilistic approach to the detection of this failure can be taken, in which exploration is carried out (under some assumptions about the distribution of values of the function in the design space) until the likelihood that the point is a failure has been reduced below some predefined threshold.

Handling the Failure

This failure arises from the interaction of the search-based method and the evaluation function. Two classes of strategy immediately suggest themselves: change the behavior of the optimization method or change the behavior of the evaluation function.

Numerical optimizers in general (and CFSQP specifically) provide parameters that may be used to modify the behavior of the algorithm. These include the step sizes taken during search, the step sizes and method used to compute gradients (where applicable), and convergence criteria. These parameters often must be selected on a trial-and-error basis.[1] No single optimization method is universally superior; one way to change the behavior of the optimization algorithm is to substitute another.

Another approach is to try and make the evaluation function appear to be more smooth and continuous. Smoother functions can be fit in the vicinity of noise or ridges. Substitute values can be provided for unevaluable and infeasible points. When evaluation functions contain tabular interpolation, discontinuities can arise from the method of interpolation; different interpolation methods can be used to obtain continuity at higher-order derivatives. Non-smoothness can arise from the use of adaptive discretization methods within an evaluation function (such as applying a grid for integration over a surface); replacing adaptive

[1] The CFSQP manual on selecting a value for the `eps` parameter: "If the user does not have a good feeling of what value should be chosen, a very small number could be provided and `iprint=2` selected so that the user could keep track of the process of optimization and terminate CFSQP at an appropriate time."

algorithms with fixed-step algorithms may help.

A less obvious tactic is to change the entire problem formulation. [Ellman *et al.*, 1995] give an example where a reformulation of the parameters of the nozzle design problem leads to the effective removal of a "ridge" in the evaluation function that caused CFSQP to fail.

Because of the difficulty of detecting that this failure has occurred (particularly with respect to optimization), a general strategy of "shoot first and ask questions afterwards" may be appropriate. Where the suspicion exists that a search-based method might be failing, apply several corrective measures speculatively and see if any improvement can be found.

2.6.2 Constraint Violation in the Simulation Model

Failure Description and Detection

The simulation programs of the prior section all embody a set of constraints on the design objects and conditions that they can validly evaluate. Some of these are tested for explicitly in the simulation code, and if violated will cause an error to be signaled. Others are not explicitly tested, but can lead to catastrophic failure of the simulation if violated. Such limitations on the simulation programs will be a part of any design domain.

Constraint violation in the simulation is normally detected when an error is signaled, either by the simulator or by the operating system (as a downstream consequence of violation of an implicitly defined constraint). Detection of implicit constraints can be made explicit in many cases through the introduction of "sanity checks" on the inputs or outputs of a function in the simulation code: functions that test that the values are within acceptable (domain-specific) limits. This is usually a good idea, since some implicit constraint violations may cause the simulation to return plausible but incorrect results without an error being signaled.

Handling the Failure

The best strategy for handling failures arising from constraint violations is probably to avoid violating the constraints to begin with. Where it is possible to specify the constraints in the form of a constraint function, it can be provided to a constrained search-based method to avoid evaluations of infeasible points.

When not all constraints can be put into a constraint function, or when unconstrained search-based methods are being used, evaluations of unevaluable points will probably occur. Handling the failure requires first that the signaled error be caught so that the optimization strategy does not terminate. The correct recovery action will then depend on the failure context. In some cases, it may be sufficient to treat the unevaluable point as "infinitely bad," although this technique creates discontinuous ridges in the value of the evaluation function, and can grossly distort gradient computations. Another possibility is to return a value interpolated or extrapolated from nearby points. It might also be possible to restart the optimization process at a new point in the design space, effectively "jumping over" a region of failure.

2.6.3 Failure of an Iterative Method to Terminate

Failure Description and Detection

Numerical methods that rely on iterative convergence to a solution (many search-based methods fall into this category) may not converge for some inputs. When this occurs, the optimization process is effectively terminated, as the evaluation function never returns a value. We see this problem occurring in all of the design domains where the Newton-Raphson root finder is being used.

Non-convergence can be detected by placing arbitrary limits on the number of iterations or amount of processor time convergent methods can use. There is some unavoidable risk in setting limits that a slowly-convergent, but genuinely evaluable point will be identified as a failure.

Handling the Failure

Experience suggests that failures caused by non-convergence are very difficult to predict, and may not be confined to contiguous regions of the search space (though they have been seen to be more likely in some regions than in others in the test domains).

Once a non-convergent failure has been detected, it can be treated as an unevaluable point and handled as discussed in section 2.6.2.

2.6.4 Interpolation Failures

Failure Description and Detection

We have seen that the use of tables is common in the test design domains. They are used to approximate functions that are too difficult to compute directly, or for which no operational theory currently exists. Unlike most functions, the constraints on tabular interpolation are explicit and easy to test (at least in those cases where the tables are rectangular and completely populated).

Good interpolation tools should detect attempts to interpolate outside table indices and signal an error. Where they do not, explicit sanity checks can be incorporated to detect and signal such attempts.

Handling the Failure

In many cases, some form of extrapolation of the tabular data may be valid, especially for small excursions from the table. Linear extrapolation is relatively easy to perform, but is likely to become increasingly inaccurate with distance. Extrapolation also carries the risk that genuinely infeasible points will be considered valid by the search-based method, and will be returned as results.

If extrapolation is not used, the point can be treated as unevaluable, and handled as in section 2.6.2.

2.7 Classes of Recovery Actions

A *recovery action* is an action taken in the event of failure that is intended to effectively handle the failure in the current context. We showed in the previous section that there are a wide range of possible recovery actions, depending on the failure.

All of the recovery actions described previously can be characterized as belonging to one of only two classes of action. One requires altering the failing function, the other requires altering the data on which the failing function is being computed:

- **Return**

  One possible recovery from a failure is to return some value from a failing function that will allow its caller to continue. When a failure occurs, any function that has been called, but has not yet returned (that is, the failure context is in the execution scope of a function) can be considered to have failed. As an example, if a failure occurs in the **Aero Drag** function of Figure 2.8, a recovery action could return a value from any one of the functions calling it {**Net Force, fdjac, NR-Solver,** $\sum +$**, Coursetime**}.

  A recovery involving returning a value is not limited to returning a constant, but may involve performing any alternative computation. A return action is equivalent to a one-time replacement of all of the computation below some point in the optimization strategy with another computation.

- **Restart**

  The other possible recovery action is to restart some failing function. This is only useful if the value of some input upon which the function depends is altered. As in the case of a return action, any of the functions active when the failure occurs can be considered the target of a restart.

  A restart action does not directly change the function being computed by substitution of one computation for another, but changes the data on which it is computed, which may result in the same effect.

In implementing our system, we must provide the ability to restart or return from any failing function in order to allow for the kinds of recovery actions we have previously discussed.

2.8 Correctness and Validity of Results

We will now address one of the key points of our thesis. We claimed that we can incorporate failure handling into existing programs *safely* – ensuring that the end results of a computation with failure handling are valid. We will define three classes of failure recovery and show that each preserves validity.

Failure recovery functionality is incorporated into optimization strategies through the mechanism of code wrappers. When a function $f_i(\vec{x})$ in the strategy is wrapped, we replace $f_i(\vec{x})$ with a new function $g_i(\vec{x})$ with the property that: $g_i(\vec{x}) = f_i(\vec{x})$ for all $\vec{x}$ such that $f_i(\vec{x})$ does not fail. The behavior of the wrapper function $g_i(\vec{x})$ for $\vec{x}$ such that $f_i(\vec{x})$ fails is defined by the recovery action(s) incorporated in g_i.

Correctness

We would like to ensure the correctness of a result returned by a wrapped legacy code: that the result is identical to the result that would be returned for the unwrapped (and validated) code. We can do that if we can prove that no failure occurred (and therefore no recovery actions were taken) during the execution of the code. From the definition above, we can see that if no failure occurred, the functions computed by any wrapped component of the legacy code will be identical to the unwrapped component. It must be possible to test to see that no failure occurred.

This guarantee places a very stringent condition on the result that no failure have occurred. We would like a more liberal condition that takes into account the unimportance of failure occurring during a search-based method.

Search-Validity

We define the recursive property of *search-validity* as follows: A computation is *search-valid* iff, **either**:

- It is not a search-based method **and:**
  - It does not contain a search-based method within its execution scope, and no failure occurs during its execution, **or**
  - It contains one or more search-based methods within its execution scope, and any failure that occurs, occurs within the scope of a search-based method, and the search-based method computation is search-valid.
- It is a search-based method **and:**
  - The evaluation of the return point is search-valid, **and**
  - The original termination conditions of the method for normal termination are all satisfied, **and**
  - The evaluations of all points used to determine the termination conditions of the method are search-valid.

The property of search-validity means that the result of a computation was not invalidated by any error occurring *during a search process.* If the computation to produce the result contains no search-based methods, it is the same as the correctness condition.

To see that the recursive condition ensures a form of correctness, suppose we have a search-based method with no search-based methods in its evaluation function. Then the results of an evaluation are search-valid iff no failures occur. By the condition above, the result of the search-based method will be search-valid iff the result of the search evaluates with no failure, and the evaluation of any point used in testing the termination conditions (such as computing the norm of a gradient, or testing for a zero) contains no failure. In the absence of failure, the correctness guarantee holds, and the result returned by the search-based method must be valid, regardless of any failure that occurred during the search because:

- the result is correct, **and**
- the termination criteria are satisfied, **and**
- the results used to test the termination criteria are correct.

The implication of this definition is that there may be points that are unevaluable in the original unwrapped code that become evaluable once failure handling is incorporated, yet still represent valid evaluations.

Conservative Constraints

Even the conditions of search-validity are more restrictive than necessary. We observe that there are conditions under which we can change a portion of the function being computed and be assured of a valid result. Specifically, if a failure recovery action taken during the computation of an inequality constraint function (in an otherwise search-valid computation) yields a result that is *known* to be *more conservative* with respect to the constraint, and the constraint is *not* active at the terminal point of the search, then the result of the search can be considered to be valid.

To see that this is so, consider that we know that if the constraint is satisfied under the failure recovery, it must also satisfy the original function, since the recovery is known to yield a result that is more conservative with respect to the constraint. Since the constraint is not active at the terminal point of the search, the terminal point is unaffected by the precise location of the constraint boundary, so long as it is within the feasible region. If the result is otherwise search-valid, it must remain search-valid.

Testability

Because the correctness and search-validity conditions are dependent on whether failure recovery actions have occurred during a computation, they can be checked and reported by an implemented system. We will add the requirement that this condition be testable at the end of any computation to the other elements of failure context supported by the LCM system.

The difficulty with the conservative constraints condition is that, unlike the others, it is not directly testable. It requires the *a priori* knowledge that a failure recovery yields a more conservative result.

2.8.1 Correctness of Execution Context

Throughout the previous discussion, we have assumed that the routines being wrapped are functions: that their output depends exclusively on the values of the input parameters. In practice, we may wish to wrap routines that maintain internal state, that are dependent on the values of global variables, or that produce outputs that are not visible in the parameter or return values of the routine.

These problems can be addressed to a certain degree by judicious *checkpointing* of the execution state of the system prior to the first invocation of a non-functional routine.[2] If the routine must be re-executed in the course of a failure recovery action, the original state can be restored first, thus ensuring that any state changes introduced during the failing execution will be discarded.

State checkpointing can be computationally expensive, and is not guaranteed to capture all relevant process state; for example, data passed between routines by means of disk files will not be correctly checkpointed if only memory and register state are checkpointed.

[2]This can be accomplished through the Unix `fork` system call, see discussion in section 3.3.3.

Chapter 3
System Architecture

In this chapter, we will present the architecture of the system that we have implemented, the Legacy Code Manager (LCM). The LCM system supports the key ideas that we have developed: the incorporation of safe, context-sensitive failure handling, taking place at multiple levels, into legacy programs. It addresses our secondary concerns of facilitating the development and re-use of such failure handling strategies. The LCM system comprises a run-time system for executing optimization strategies with inter-communicating rule-based wrappers; a language for building re-usable failure handling strategies; and a set of databases and a compiler for implementing failure handling into legacy code.

We will first present the LCM run-time system architecture, which is responsible for actually carrying out failure recovery strategies. We will identify the important components of the system, and describe how each operates in the failure recovery process.

We will then present the language that we have defined for specifying failure recovery schemata. We will present the major features of the language, and show how they support the implementation of failure recovery schemata in a modular, re-usable fashion that is sensitive to failure context. We will give detailed examples from our database of implemented failure recovery schemata.

Finally, we will discuss the LCM compile-time system architecture, and show how we incorporate failure recovery into legacy code safely, through the automatic generation of code wrappers. We will discuss how we use a socket-based approach to ensure the controlled execution of legacy systems designed to be run as stand-alone programs.

3.1 The LCM Run-time System Architecture

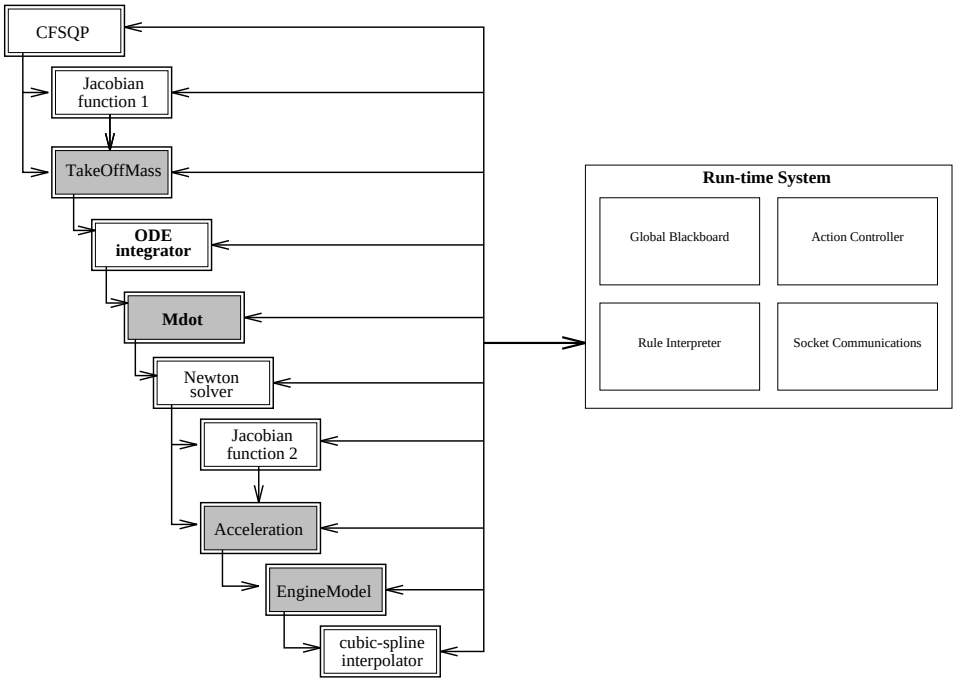


Figure 3.1: Optimization Strategy with Failure Handling

Figure 3.1 shows the nozzle design optimization strategy presented earlier, with failure handling incorporated. (We will defer a discussion of fuction wrapper generation and incorporation to the end of this chapter.) In the figure, the wrappers around each element of the strategy can be seen to be communicating with the LCM *run-time* system.

The LCM run-time system provides the execution framework for failure handling. It comprises several key components:

- The *function wrappers,* which allow control of callable components of the strategy.
- The *system blackboard* which contains information about global state and failure context, and history of function evaluation.
- The *rule interpreter,* which executes the rules that determine if and when failure handling is to take place.
- The *action controller* which carries out the recovery action.

We will describe each of these components, and discuss its role in failure recovery.

3.1.1 Function Wrappers

Wrapper Structure and Operation

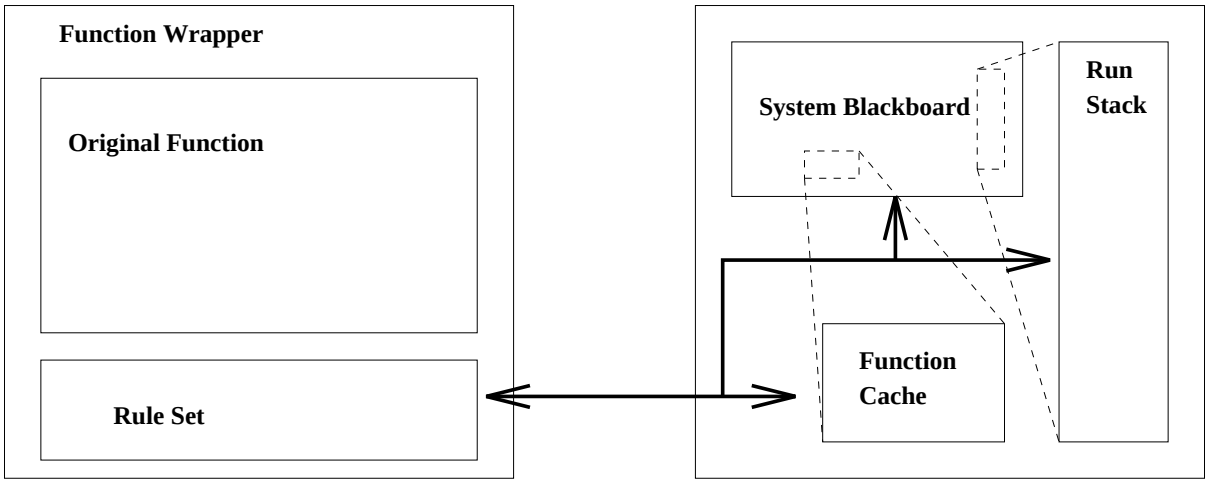


Figure 3.2: Function Wrappers

When an optimization strategy is compiled for use with the LCM system, new code is linked in. This code is in the form of routines, called *function wrappers* that have the same name and formal parameters as existing routines. When the original routines are compiled, their entry names are transformed by appending a postfix string (such as "_WRAPPED"). The result of this is that the wrapper routines will receive control whenever the original routine is referenced by a calling routine. Wrapper routines also contain code that enables them to receive control whenever an error condition is generated within their dynamic scope of control.

Figure 3.2 shows the structure of a wrapper. Associated with each wrapper are the rules that control the behavior of the wrapper, and the original wrapped function. The wrapper rules reference and alter state information on the global blackboard. State information includes including context automatically maintained by the LCM run-time system, such as caches of function evaluations and the run-time stack, and information placed on the blackboard by rules in other wrappers.

When they are called, wrappers pass control, and the calling parameter information, back to the LCM run-time system. The rule interpreter is invoked against the rules associated with the wrapper before the wrapped function executes. If no action is taken, control is returned to the wrapper, which then calls the wrapped function with the parameters originally passed.

The wrapper will receive control again when either the wrapped function returns normally, or it catches an error condition raised during the execution of the wrapped function. Control is passed again to the LCM run-time system along with any return value or error condition for a post-execution invocation of the rule interpreter. If no action is taken, control is returned to the wrapper, which returns any return value to the original caller function.

The normal execution flow of control is altered only when a rule succeeds that requires a non-null action.

Error Detection and Classification

Wrappers are responsible for actually taking control when errors occur in wrapped functions. We will now describe the kinds of errors that can be detected and handled by the function wrappers.

- **Asynchronous Errors**

  Errors detected and reported by the hardware and operating system are *asynchronous* errors, and may occur anywhere in the execution of a function. Asynchronous errors include arithmetic errors (function domain errors) and system error signals like SIGBUS and SIGSEGV. These errors are trapped by the LCM system using provided operating system condition-handling mechanisms (such as the `signal` function). Error conditions may be tested in rules by means of the `Condition` predicate. Error class assignment follows Common Lisp conventions (see [Steele, 1990]).

  Condition handler scope is dynamic: control is transferred from the location of the error to the most recently executed handler for the error. The dynamic scope of condition handling leads to the possibility of conflict if condition handlers are already part of the

optimization strategy.

The LCM system allows for the detection of an additional asynchronous error: a *timeout* error. The timeout is specified as a maximum duration (in wall-clock seconds) for a function to execute. If the function fails to terminate in that time, the timeout error is raised, and the most recently executed timeout handler receives control. Timeouts are intended to limit runaway function execution, and must be defined with caution as wall-clock time for execution will depend on the hardware on which a function is executing, and the load on the system.

A condition that is not normally considered an error is a *system exit*. In the context of an optimization strategy, however, a system exit is equivalent to an asynchronous error: if left unhandled, it will result in the premature termination of the optimization. Exits are handled by interception, using the Unix function `atexit`. It should be noted that `atexit` is also dynamically scoped, and the possibility of conflict exists if exit trapping is already part of the optimization strategy.

- **Synchronous Errors**

  Synchronous errors are those that do not cause an immediate non-local transfer of control. These errors must be explicitly tested for by the LCM system. They include bad error return codes, input parameter values outside of valid ranges, and performing regular expression matches on output files for error messages. While most asynchronous errors are system-defined, most synchronous errors are function-specific, and must be defined by the user. Input parameter values and function return values may be tested in rules using provided predicates. No built-in mechanism has been provided yet for the testing of error messages in standard output.

  Also included in the category of synchronous errors is the class of *semantic failures*. These occur when a function executes and returns with no error indication, but the value computed is not semantically correct. The most common example is an optimization that terminates at other than the global optimum for the function. Other

examples include equation solvers returning values that are not zeros of the input equations, or finding a correct but physically impossible root for systems with multiple roots. These failures may be difficult to detect, and require explicit "sanity" testing provided by the user through the `Sanity` predicate.

3.1.2 The Blackboard

The system blackboard is the mechanism by which state information is recorded and referenced across all the components of the LCM system. State information on the blackboard determines the truth value of predicate evaluations in schema rules.

Blackboard Operation

The blackboard is a collection of state variables, each with a unique identifier. By convention, an identifier is specified in two parts: a root and a symbol. This allows state variables to be conveniently grouped together by the rules in which they participate. State variables may be added to or removed from the blackboard as the result of evaluating schema rules. Some state variables are provided by the LCM run-time system for use by built-in predicates, such as the run-stack and error condition information (system state variables have a root of "SYSTEM").

One potential problem in using global state variables is the conflict that is created when a rule is used in a re-entrant fashion. This will often occur in this system, where functions of the optimization strategy are called recursively. In order to allow rules to be written so that they do not need to be explicitly aware of recursive use, each state variable is managed by the LCM system as a stack-tree of values. Each state variable is initialized at the time of its addition to the blackboard. If a rule is re-entered, the variable's state is pushed, *as is the state of all variables added to the blackboard subsequent to that variable.* As no new value is set for the subsequent variables, after the push operation they appear to be uninitialized. Every time this operation occurs, the *state frame* is said to have been pushed. Values for earlier frames continue to exist on each variable's value stack, but are said to be *shadowed* by

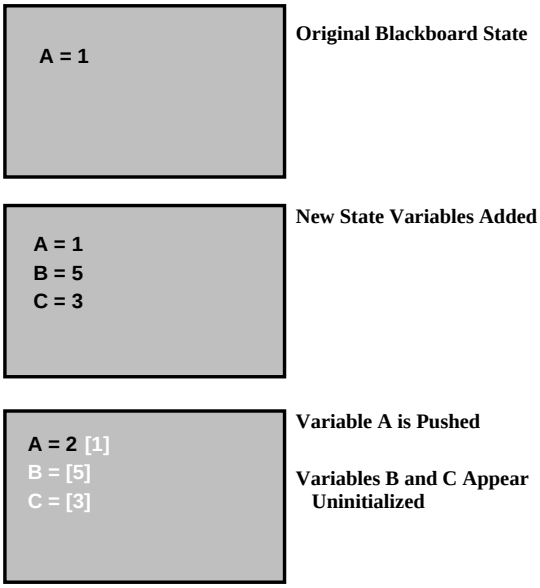


Figure 3.3: Blackboard Frame Push

the current state value. The effect of a frame push is to make the visible (unshadowed) state of the blackboard as it was when the state variable being pushed was first entered onto the blackboard, while not destroying any existing state information. This is shown in Figure 3.3.

```
                6   5
             8  4 --
Stack Top -->  7  3
             2--
             1

Sequence: Push 1, Push 2, Push 3, Push 4, Push 5,
          Pop, Push 6, Pop, Pop, Pop,
          Push 7, Push 8, Pop
```

Figure 3.4: Value Stack Tree

When a frame is popped, the values are not removed from the value stack of the state variables in that frame. Instead, the top of stack pointer is changed to reflect the current top of the value stack. The next time the frame is pushed for those variables, a new branch of the stack-tree is created. A trace of a single state variable's value stack tree is illustrated in Figure 3.4. Like the realized calling tree, any path from a node to the root reflects one recorded state of the stack. The value stack-tree is retained so that rules may be written to

test for state values occurring *at any time* during the execution of an optimization strategy.

The Caches

When caching is enabled for a function, the return value of the function is stored in a cache, maintained on the system blackboard, and hash-indexed by the parameter values, whenever the function is successfully evaluated. A unique cache exists for each cached function in the optimization strategy. A built-in predicate uses cached values to approximate the value of the function for failing evaluations.

The cache index is constructed by recursively "unrolling" the scalar and aggregate parameters (lists, arrays, structures) of the parameter list into a single "flat" list of values. Because only the formal parameters of functions make up the index, functions that depend on the values of global variables may not be reliably cached. It is the responsibility of the user to identify non-cachable functions in the functional semantics knowledge base, though the LCM cache manager will warn the user if caching is attempted for two different values on the same index.

Caches may also be stored to disk files and reloaded to provide databases of function evaluation information.

3.1.3 The Rule Interpreter

Whenever the LCM system receives control during the execution of an optimization strategy (when an asynchronous error occurs, or at the transfer of control between two functions), the rule interpreter is invoked against the currently selected rule set. Rules are sorted by their specified precedence, and the topmost rule is selected for evaluation.

The rule interpreter uses Prolog-like resolution theorem proving to evaluate the clauses of the conditions of each rule. The arguments of `And` and `Or` clauses are evaluated left-to-right, with evaluation halting as soon as the result is known (i.e. on evaluation of a false predicate for `And` or a true predicate for `Or`). Clause recursion is permitted.

As predicates with unbound variables may be true for many different bindings, the rule interpreter backtracks among alternatives until the rule is satisfied, or all bindings have been attempted. No provision currently exists for speedup in the backtracking process (as with the Prolog "cut" operator).

If the rule conditions are satisfied, the variable bindings are retained while any conditional global state updates are performed, and the action specified for the rule is carried out. If the rule conditions are not satisfied, or a null action has been specified for a succeeding rule (the rule suceeds only to update global state), the rule is removed from the sorted list, and the interpreter is re-invoked on the next rule in sequence, until the rule list has been exhausted.

3.1.4 Actions and Control

The normal execution flow of control in an optimization strategy is altered when a wrapper rule succeeds that requires a non-null action. The flow of control of an action depends on the type of the action:

- Local Return

  A local return is a return from the currently active frame. Control is returned to the most recently active wrapper, along with a return value. The wrapper returns control and the value to the original caller of the wrapped function.

- Non-local Return

  A non-local return is a return from a stack frame other than the most recent frame. For a non-local return, control is transferred via a non-local goto (a Lisp *throw* or a C *longjmp*) to an earlier frame. Control is then given to the wrapper for the function for that frame, along with the value to be returned. The wrapper returns the value and control to the original caller of the wrapped function for that frame.

  A non-local return carries some risk. Because control is being transferred out of the normal sequence of execution, there is the possibility that the optimization strategy

may be left in an internally inconsistent state. Some storage allocated may never be recovered, causing memory leakage.

- Local Restart

  A local restart occurs when control is passed back to the wrapper for the currently active frame, along with a modified parameter list. The wrapped function for the frame is re-called by the wrapper, and passed the new parameter list. Flow of control follows as during the original call of the wrapped function.

- Non-local Restart

  A non-local restart occurs when control is given to a frame other than the most recent frame, along with a modified parameter list. Control is transferred via a non-local goto, as in the non-local return. The wrapper for the target frame receives control, and re-invokes the wrapped function for that frame.

  A non-local restart carries risks similar to those for a non-local return. In addition, because it is, in effect, attempting to invoke the restarted function in a reentrant manner, unexpected results may occur if the restarted function produces side-effects, or depends on global variables.

- Signal

  A signal is a non-local transfer of control where the target of the transfer is not known in advance. It is used to decline responsibility for handling a condition and allow another portion of the LCM system to receive control.

3.2 Recovery Schemata

3.2.1 The Realized Calling Tree

The *realized calling tree* is a graphical representation of some or all of the functions (discretely callable routines) that make up the *optimization strategy* that is input to the LCM system,

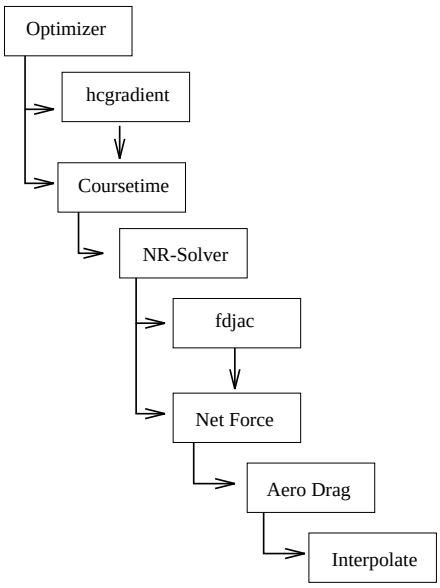


Figure 3.5: A Simple Realized Calling Tree

and their flow-of-control (call and return) relationships. Figure 3.5 shows a simple example of a realized calling tree.

The calling tree is characterized as *realized* because:

1. It is incomplete. It contains only those functions that have been identified as being of relevance to the LCM system.

2. It is derived from analysis of the blackboard run-time stack during one or more executions of the optimization strategy. It therefore reflects the *actual* calling relationships, rather than the *potential* calling relationships that would be produced by static analysis.

The realized calling tree is a compact representation of all recorded states that the blackboard run-time stack can take on while executing a particular optimization strategy. The list of functions along any possible path from a node in the tree to the root of the tree is a single achievable stack state. This knowledge will be required by the LCM system to select among failure recovery strategies.

3.2.2 Failure Handling Schemata

A failure handling schema is a description of a class of problems, occurring in a (possibly) broadly-defined context. Each schema consists of three parts:

- A Structural Template

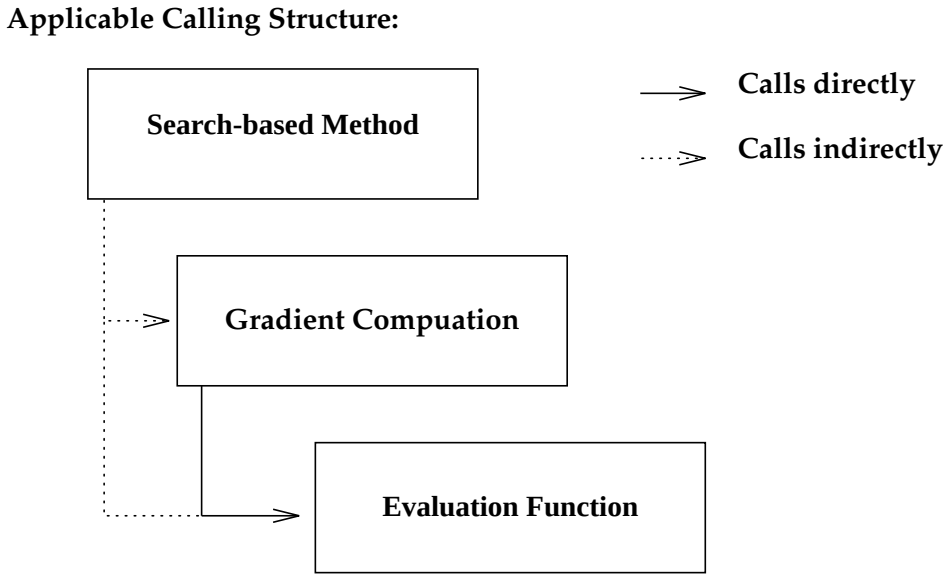


Figure 3.6: Schema Structural Template

The structural template defines the *class* and *calling context* of a set of functions in the target system for which the schema is applicable. The form of the template is the list of calling context relationships between the modules of the template, e.g.: $\{F_1 \Rightarrow F_2, F_1 \Rightarrow F_3, F_2 \rightarrow F_3\}$ where $\rightarrow$ denotes a direct calling relationship (F_1 is the *calling parent* of F_2) and $\Rightarrow$ denotes an indirect calling relationship (F_1 is the *calling ancestor* of F_2). This template is shown graphically in Figure 3.6.

Each function in the template is identified by the most generic class (in the functional semantics database) for which the schema is applicable to that function. A match occurs when a set of nodes can be found in the realized calling tree of the target system for which the calling context relationships between all of the nodes is the same as specified in the template, and the class of each of the functions is the same or a

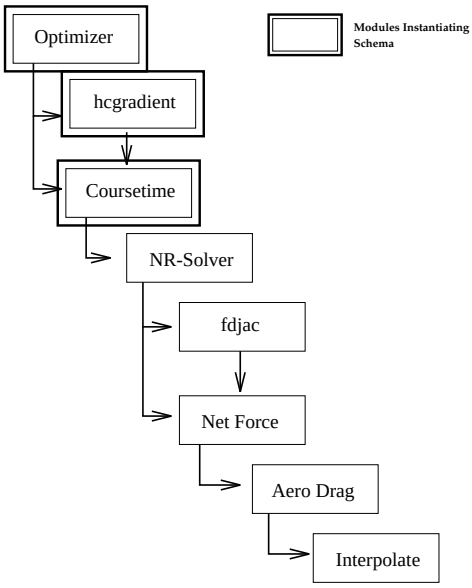


Figure 3.7: Possible Instantiation of Structural Template

subclass of the function classes in the template. An example template match is shown in Figure 3.7.

- A Rule Set

  Each schema is defined by one or more rules of the form: ($Condition^{+}$ *Action State-update**).

  Each rule specifies one or more conditions that define when and where the rule is applicable; a consequent action that may alter the flow of control of the executing system; and (optionally) a set of updates to the global state information.

  The *condition* portion of a rule consists of one or more logical predicates joined by boolean operators. The predicates reference the parameters of routines, state variables created and modified by other rules, or special global state information maintained by the blackboard system. A detailed description of defined predicates is given in a later section.

  The *action* portion of a rule defines the action to be taken when the condition is true. The action can specify a return from a currently-executing function, a restart of a failing function with new parameters, or a signal to allow another schema to handle

the condition. An ordered list of alternative actions may be specified in a rule.

The *state-update* portion of a rule allows one or more elements of global state information to be changed before the application of the action portion of the intercessor. This allows for additional levels of control over intercessor rule application.

- The Instantiation Symbol Set

  The schema is defined with some of its symbols left unbound, to be determined when the schema is instantiated into a particular context. Some of these symbols (like the names of functions and their parameters) can be automatically instantiated with information from the functional semantics database, others (such as appropriate default return values) must be provided by the user. The union of the all of the instantiation symbols from all the rules in a schema is the instantiation symbol set for the schema.

3.2.3 Example Schema and Instantiation

The rules shown in Figure 3.8, along with the structural template of Figure 3.6, define a schema that can be described as follows:

> IF *any detectable error condition occurs during the execution of a function being called directly or indirectly by a numerical gradient-search-based method (such as an optimizer or an equation solver),* AND *the run-time context of the error is that the function utilized by the search-based-method for computation of the gradient is not executing,* THEN *return from the routine in which the error occured with a return value of* `?bad-value` *(to be provided at instantiation time).*

This schema can be instantiated into the calling tree for the optimization strategy diagrammed in Figure 3.7. The function labeled **Optimizer** is a subclass of `Search-based-Method`, **hcgradient** is a subclass of `Gradient-Computation`, and **Coursetime** is a subclass of `Evaluation-Function`, therefore the schema can be instantiated as shown by the bold boxes by providing the following bindings for the instantiation symbols:
((?bad-value . 9.99E99) (?gradfn-name . hcgradient)).

In this calling tree, the function labeled **NR-Solver** is also a subclass of `Search-based-Method`, **fdjac** is a subclass of `Gradient-Computation`, and **Net Force** is a

```
(Rule 1
        :condition      '(and (Condition T)
                              (CurrentFrame ?x)
                              (Parent ?parent ?x)
                              (ModuleName ?parent ?name)
                              (NotEqual ?name ?gradfn-name))

        :action         (make-instance 'action
                              :action-code      'Return
                              :return-form      '?bad-value)

        :state-update   NIL

        :symbols        '(?gradfn-name ?bad-value))

(Rule 2
        :condition      '(and (Condition T)
                              (CurrentFrame ?x)
                              (Not (Parent ?parent ?x)))

        :action         (make-instance 'action
                              :action-code      'Return
                              :return-form      '?bad-value)

        :state-update   NIL

        :symbols        '(?bad-value))
```

Figure 3.8: Simple Bad-value Schema

subclass of `Evaluation-Function`, so the schema can also be instantiated on those functions by providing the following bindings for the instantiation symbols: ((?bad-value . 9.99E99) (?gradfn-name . fdjac)).

3.2.4 Predicate Descriptions

Run-time Context Predicates

`CurrentFrame(x)`	The current frame is `x`
`ModuleName(f,n)`	The module name of frame `f` is `n`
`FrameNumber(x,n)`	The frame number of frame `x` is `n`
`Parent(x,y)`	The frame `x` is the immediate calling parent of frame y
`Ancestor(x,y)`	The frame `y` is somewhere in the dynamic calling scope of frame `x`
`Inbetween(x,y,z)`	The frame `z` occurs somewhere between frame `x` and frame `y`

Table 3.1: Run-time Context Predicates

The predicates in Table 3.1 allow rules to be conditioned on the state of the run stack maintained on the system blackboard. The `CurrentFrame` and `FrameNumber` predicates are intended to be used functionally to obtain information about the frame at the top of the run-time stack. The remaining predicates can be used either as truth-valued predicate tests or as functions to obtain any parameters left unbound.

Function Reference Predicates

`Formals(f,l)`	The frame `f` has the formal parameter list `l`
`ReturnValue(f,v)`	The current return value of frame `f` is `v`
`ParameterValue(f,l,x,v)`	The frame `f` with the formal parameter list `l` has a parameter value of `v` at parameter index `x`
`CacheValue(f,l,v)`	The frame `f` with formal parameter list `l` has a cached value of `v`
`Distance(f,l1,l2,d)`	The sum-squared distance in parameter space between the formal parameter lists `l1` and `l2` for frame `f` is `d`

Table 3.2: Function Reference Predicates

The predicates in Table 3.2 allow rules to be written that refer to parameters of functions and their values. All of these predicates are intended to be used as functions of the first (n-1) arguments to produce the last argument, e.g. $f(p_1, p_2, ..., p_{n-1}) \rightarrow p_n$.

The formal parameters of functions are represented as recursively-defined lists of scalar values. The *index* provided to the `ParameterValue` predicate is a special accessor object automatically bound from information in the functional semantics database to a pre-defined symbolic name for the parameter of interest (for example, ":SEED" for a function taking a seed as an argument). The `CacheValue` predicate will return a false value if there is no value cached for the formal parameter list given. The `Distance` predicate computes the sum-squared difference of an element-by-element comparison of the two parameter lists given.

Error Detection Predicates

`Condition(x)`	An error condition of class `x` has occurred

Table 3.3: Error Detection Predicates

The `Condition` predicate shown in Table 3.3 provides access to information about errors occurring during execution. If `x` is bound, the predicate will have a true value if an error has occurred with the class or a subclass of the class specified. If `x` is unbound, the predicate will have a true value if any error occurs, and the error will be bound to the argument. Error classification is discussed in detail in a later section.

Blackboard Reference Predicates

The predicates in Table 3.4 allow rules to reference (and alter) global state information maintained on the system blackboard. The predicates `InsertSymbol, RemoveSymbol,` and `SetValue` are intended to be used in the **state-update** portion of the rule (though they may, in fact, appear in the condition portion). The `CurrentValue` predicate is used to obtain the current value of a global state variable. The `ShadowedValue` predicate will be true for each

`InsertSymbol(s)`	Symbol `s` has been inserted into the blackboard
`RemoveSymbol(s)`	Symbol `s` has been removed from the blackboard
`SetValue(s,v)`	The current value of symbol `s` is `v`
`CurrentValue(s,v)`	The current value of symbol `s` is `v`
`ShadowedValue(s,v)`	A shadowed value of symbol `s` is `v`

Table 3.4: Blackboard Reference Predicates

shadowed value appearing in the symbol's value stack. See the discussion of the blackboard for a detailed discussion of value shadowing.

Comparison Predicates

`LessThan(x,y)`	The value of `x` is less than the value of `y`
`GreaterThan(x,y)`	The value of `x` is greater than the value of `y`
`Equal(x,y)`	The value of `x` is equal to the value of `y`
`LessEqual(x,y)`	The value of `x` is less than or equal to the value of `y`
`NotEqual(x,y)`	The value of `x` is not equal to the value of `y`
`GreaterEqual(x,y)`	The value of `x` is greater than or equal to the value of `y`

Table 3.5: Comparison Predicates

The comparison predicates in Table 3.5 may only be used as truth valued tests, and will generate errors if any arguments are left unbound.

Conjunction, Disjunction, Negation

`And(x,y,...)`	Predicate `x` is true, and predicate `y` is true...
`Or(x,y,...)`	Predicate `x` is true, or predicate `y` is true...
`Not(x)`	Predicate `x` is false

Table 3.6: Conjunction, Disjunction, Negation

The logical symbol predicates listed in Table 3.6 allow predicates to be combined. In the `And` predicate, predicates are evaluated sequentially from left to right until one has a value of false, or until all have been evaluated. The `Or` predicate evaluates its arguments from left to right until one is found to be true. In both cases, variable bindings established as predicate

arguments are evaluated are carried forward to subsequent argument evaluations. The `Not` predicate uses negation-as-failure: if its predicate argument fails, it succeeds. None of these predicates can take unbound arguments.

User-defined Predicates

`Sanity(x,y,z,...)`	`x` is a Lisp function taking one argument, the list of `y,z,...` that returns NIL for OK, T for sanity failure.

Table 3.7: User-defined Predicates

The `Sanity` predicate in Table 3.7 provides a universal hook for any user-defined predicate to be included in a rule. The Lisp function of the argument is evaluated in the run-time context of the rule, and may refer to local parameter or global variable values. This predicate is intended to provide a general method for inclusion of domain-dependent sanity checks on the arguments and return values of functions.

Qualifier Predicates

`For-All(s,p)`	For all recorded values of symbol `s` on the blackboard, predicate `p` is true
`Exists(s,v,p)`	For some recorded value `v` of symbol `s` on the blackboard, predicate `p` is true
`Exists-N (s,n,p)`	For exactly `n` recorded values of symbol `s`, predicate `p` is true

Table 3.8: Qualifier Predicates

The predicates in Table 3.8 provide a means of writing rules that can be evaluated against the recorded history of values for blackboard symbols. Because the blackboard retains a history of all values taken on by global state variables during the entire time that those variables exist on the blackboard, it is possible to have rules with scope greater than the current visible state. The universe of possible values for the variable symbol is circumscribed by those values recorded on the blackboard for that variable. Needless to say, the predicate argument `p` should reference the variable symbol argument `v`.

3.2.5 Action Descriptions

Action	Parameters
`Return`	Frame
	Value
`Restart`	Frame
	Parameter Index
	Override Value(s)
	Cycle Option
	Value Ordering
`Signal`	Condition
`Null`	

Table 3.9: Actions

Once a particular failure has occured, been caught, and identified, it must be handled: some action must be taken. Table 3.9 lists the possible actions that can be taken. These be classified into three broad categories: return from an executing function with some value, restart a failing function with altered parameters, or do not handle the failure and either signal a new failure for some other recovery schema to handle, or do nothing.

Returning a Value

When a function fails, its result is undefined, and any subsequent computation that is dependent on the value of that result cannot continue. Under certain circumstances, it is possible to provide a meaningful substitute value and allow the computation to continue.

A return action may be specified for any function active on the blackboard run-time stack at the time that the failure occurs; it is not limited to the lowest-level failing function. Several different actions can be specified that return values:

- Return a fixed value.

  This is the simplest action, and is appropriate in the context of search-based methods, where the value returned will be used for comparison. A value can be returned that is of sufficient magnitude and sign that it will be guaranteed to be "bad" in comparison with any legitimate value.

Fixed-value return is a poor technique to use in the context of gradient-based methods, as it will distort the computed gradient in the neighborhood of the failing point.

- Return the result of an alternative computation.

  When there is more than one way to compute a function, an alternative computation may be substituted for the failing original. Many numerical methods (for example, ODE solvers) have weaknesses that can cause them to fail for some inputs; it may be difficult to predict in advance what inputs will cause failure. Alternative solution methods (perhaps more computationally expensive or less accurate) may be able to succeed where the original method fails.

  This strategy requires accurate knowledge of the semantics of the failing function.

- Return an interpolated value.

  A strategy that represents something of a compromise between returning a fixed value and performing a semantically valid alternative computation is to assume that the failing function is reasonably approximated by a smooth interpolation of successful values in the near neighborhood of the failing point. Since the LCM system is capable of caching successful function evaluations, this option is easy to implement.

  This strategy allows search-based methods to continue without the gradient distortions introduced by the fixed-value method. Since the returned value is an approximation, however, care must be taken to ensure that the search process does not terminate on a point with an interpolated value.

Restarting a Computation

It may be the case that the success of a computation is dependent on the values of control parameters, appropriate values of which may be difficult to determine *a priori*, or which may need to change during the progression of an external process in which the computation is embedded. Examples of such parameters are: numerical gradient computation epsilon,

termination criteria for iterative methods, integration step sizes. In these cases, restarting the failing computation with some parameter value changed may result in successful computation.

A restart action may be specified for any function active on the blackboard run-time stack when the failure occurs. A parameter index and one or more override values must be specified with the restart. When a restart is specified for a function not at the top of the run-time stack, the stack top is reset to the restarted function. Multiple override values will result in multiple restarts, each with one of the values. The results are collected and sorted by a value-ordering predicate (*less-than* by default), and the first element is returned as the value of the restart. Additional options are available to allow a set of parameters to be cycled through multiple times, checking for convergence (see the next chapter for greater detail).

Declining to Handle

Finally, once the conditions of a failure have been examined, there may be no reasonable action that can be taken, or a recovery attempt may itself lead to failure. In this case, the schema can decline to handle the failure.

A signal action may be specified that raises a new error condition, defined by the schema. The new error condition may be handled by another schema, or may result in termination of the computation (there is always default handling of all signals and errors which is provided by the LCM system and results in the termination of all active computation).

A null action may also be specified. This causes the original error to be re-posted as though no error handling had been attempted.

3.2.6 A Class Hierarchy for Recovery Schemata

The recovery schemata in the database are organized in a class hierarchy that parallels the function class hierarchy of the function semantics knowledge base. The hierarchy is graded

by *specificity of context:* schemata near the root of the hierarchy are applicable to broader classes of failure than those near the leaves. The specificity of a context is increased by changing the rule set or structural context for a schema such that it becomes applicable for a strict subset of failures that its immediate parent(s) are applicable to.

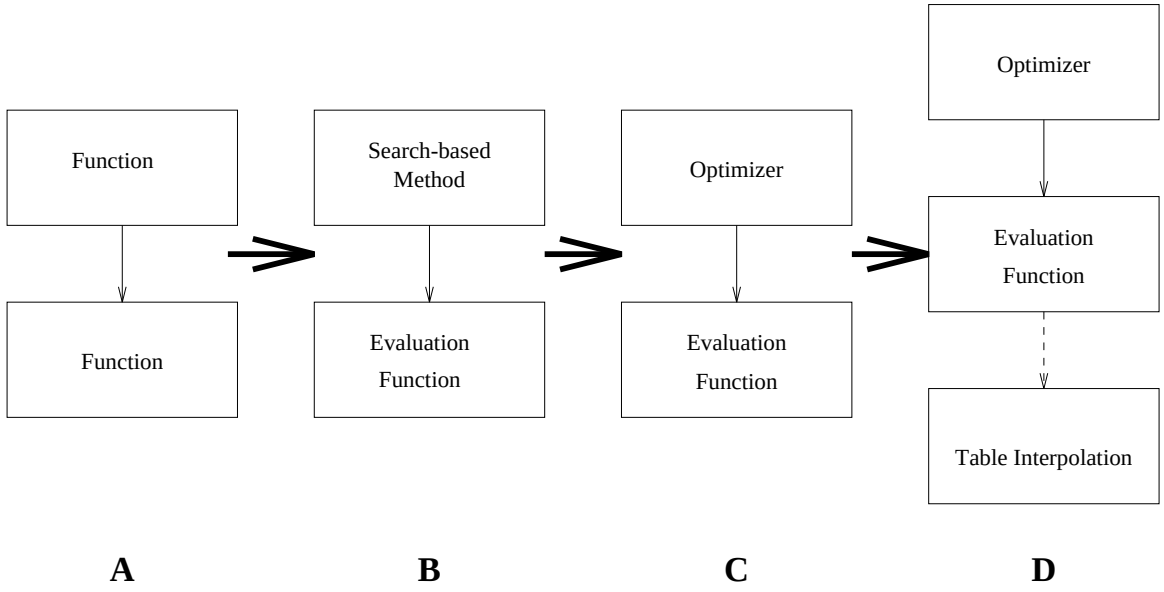


Figure 3.9: Fragment of the Schemata Class Hierarchy

Figure 3.9 shows a fragment of the database showing a progression of increasingly specialized structural contexts for schemata. At the left, schema **A** is the most general, applying to any pair of functions in a direct calling relationship. Schema **B** is more specific, requiring that the caller be a search-based method and the function being called be an evaluation function; schema **C** further restricts this to require that the caller be an optimization method. Finally, schema **D** requires that the failure context include a table interpolation method within the evaluation function.

The hierarchical organization of the schemata database serves two purposes. First, it allows economy of definition of new schemata: when a schema is a specialization of an existing schema, only the differences must be specified. Second, it allows a partial ordering of applicable schemata to a particular failure context. We might usefully assume that, in general, the more specific a recovery schema is, the more effective it is likely to be.

3.2.7 Schemata Descriptions

In this section we give brief descriptions of several of the schemata that have been implemented and tested. Detailed descriptions of the schemata, including the structural context and rules, can be found in Appendix A.

- **SBV** – Simple Bad Value

  The simplest of all of the schema, the **SBV** schema can be instantiated for any detectable error occuring during the execution of an evaluation function for a search-based method.

  When a failure occurs, a user-provided value is returned as the value of the evaluation function. It is presumed that this value will be sufficiently "bad" to ensure that it is worse than any valid evaluation.

- **SI** – Simple Interpolate

  The **SI** schema is a variant of **SBV** that uses execution history context for recovery. When a failure occurs, instead of returning a fixed error value, a value is returned that is the linear interpolation of nearby good points from the cache maintained for that function on the global blackboard.

- **GMC** – Gradient Method Change

  The **GMC** schema attempts to handle a failure occuring during gradient computation by changing the method used to compute the gradient.

  When failure occurs, the run-time stack is checked to determine if the gradient function for which this schema has been instantiated is currently on the stack. If it is, a gradient computation is assumed to be in progress, and the gradient function is re-started with its parameters changed so it will not evaluate the failing point, for example by using a backward-difference computation in place of forward-difference.

- **GA** – Gradient Approximation

The **GA** schema is a computational context-sensitive version of the **SI** schema. It is actually derived from **GMC**. When a failure occurs during gradient computation, a point is approximated for the gradient using interpolation of good points from the cache maintained on the global blackboard. When failure occurs outside of the gradient context, this schema behaves like **SBV** to avoid the risk of search termination on an unevaluable point.

- **TE** – Table-Extrapolation

  The **TE** schema attempts to handle failures that occur when an interpolation is required outside the bounds of a table by providing an n-dimensional linear extrapolation from the table. The schema assumes that tables are (hyper-)rectangular and completely filled.

  When a failure occurs, either because the interpolation method signaled an error, or because a parameter test detected that the interpolation point was outside the indices passed to the interpolation method, the interpolation function is returned from with a value computed by the `LinearExtrapolation` built-in function. The use of this schema depends on the indices and the tabular data being explicitly represented in the arguments to the interpolation method – normally the case for standard library interpolation routines.

- **BPR** – Backtracking Parameter Restart

  The **BPR** schema is the most general of a large family of schemata. The schema allows a function to be re-started multiple times with the value of some control parameter of the function substituted from the elements of a provided list. The return values of the multiple executions are collected, and the best value (minimum or maximum may be specified) is returned as the value of the wrapped function.

- **RMS** – Random-Multi-Start

The **RMS** schema is commonly used in optimization. It is applicable whenever the success of a search-based method will depend on the selection of a starting point (the *seed value*), and a good seed value cannot be determined in advance. The schema re-starts the search each time it completes or fails with a new seed generated within a bounding volume specified by the user, for some specified number of tries, and returns the best result found over all tries.

- **MMS** – Multi-Method-Search

  The **MMS** schema is intended to override the original optimization method in an optimization strategy, and replace it with a set of alternative optimization methods. These are applied sequentially with each method beginning from the result of the prior method, and the best result is retained. If the results improve by more than *epsilon*, the process is repeated.

- **BSM** – Bound Search-based Method

  The **BSM** schema addresses the problem that occurs when an unconstrained search-based method is used for a function where there is a known bounding box that constrains the space of evaluable points. When an evaluation is attempted that falls outside the bounded region (defined by a user-supplied predicate) the search-based method is restarted with a new initial point that lies on the line connecting the failing point to the last point evaluated that was inside the box.

3.2.8 Summary of Recovery Schemata

The schemata presented in the previous section are shown in Figure 3.10. Each link in the graph indicates that the child schema is a specialization of its parent(s). All thirteen schemata detailed in Appendix A are shown, but some (for example, **BPR**) represent a large family of schemata that differ very slightly in detail.

These schemata are by no means intended to represent a covering set for the space of general schemata. The focus of our work has been less on the development of a complete

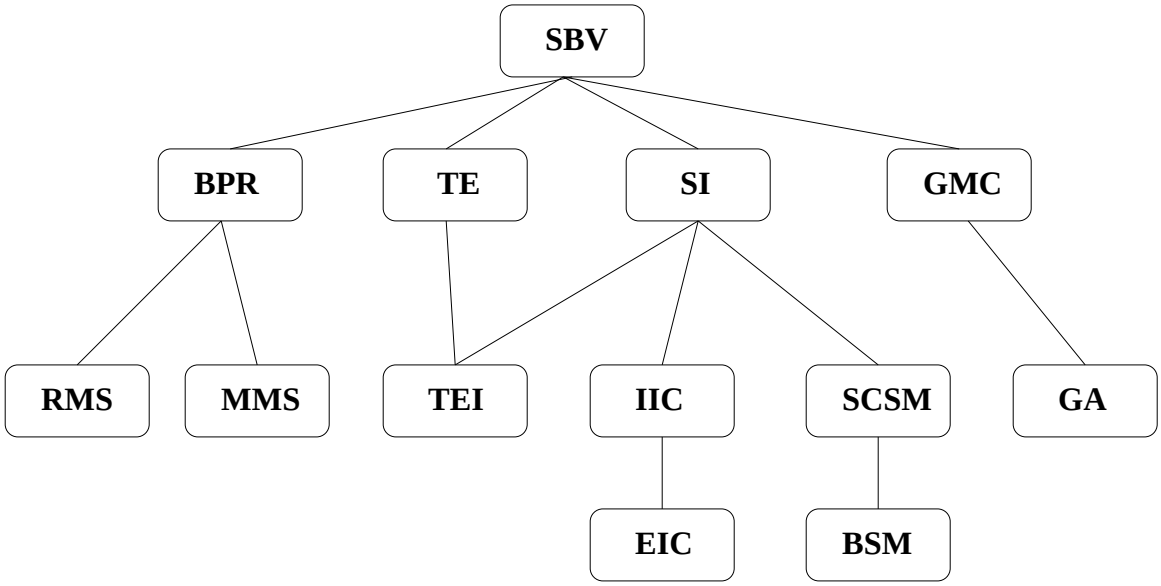


Figure 3.10: Database Ordering of Described Schemata

set of schemata than on the development of the tools that would make building such a set possible. We have implemented only enough different schema to demonstrate the power and flexibility of the approach.

Problems Addressed and Not Addressed

The schemata developed address a range of common failures we have encountered during optimization in our research into automated systems for design. The failures are principally those that are signaled by the presence of a system error condition. We have addressed some semantic failures, particularly those occuring as a consequence of choice of modeling parameters. For these cases, the failure has been largely detected as a consequence of the recovery schema: the action is undertaken with the assumption that it will reveal its necessity as alternatives are explored.

We have not addressed a broad range of other problems that may occur, such as resolving representational differences between system elements, or improving the execution efficiency of optimization strategies. We require that optimization strategies be correct (in the absence of failure).

Because we have chosen to approach optimization strategies at the level of the functional interfaces, the applicability and effectiveness of these schema will depend on the granularity of the strategy to which they are applied. Our approach offers little leverage for highly monolithic programs. We are also dependent on being able to identify the semantics of portions of the target system. The greater the degree to which this is possible, the more likely it will be that appropriate schemata will be found.

A more detailed discussion of the LCM system's strengths and weaknesses will be found at the conclusion of this thesis.

The Number of Possible Recoveries

When a failure occurs, the number of possible recovery actions depends on the type of failure and the context in which it occurs.

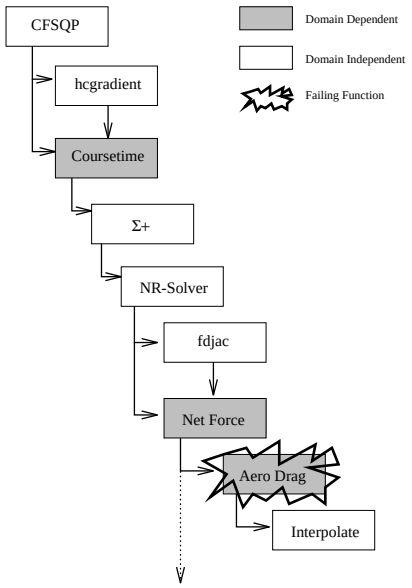


Figure 3.11: Failure Example

Consider an example which has actually occurred in one of our problem domains: an error occuring in the fragment of a realized calling tree shown in Figure 3.11. Suppose a floating-point error (signalled by a SIGFPE) occurs as a consequence of a division by zero in the **Aero Drag** function. At the time the failure occurs all eight of the functions shown

may be active on the run-time stack. This failure context satisfies the structural context requirements for all of the schemata, with the exception of **TE, TEI** and **ECC**.

Looking only at the structural context, there are 10 possible ways to instantiate the **SBV** schema for this failure (7 failing functions are in the execution scope of the search-based method **CFSQP**, and 3 in the execution scope of the search-based method **NR-Solver**); 10 ways for **SI, BPR, RMS, MMS**; 113 ways for **ICC**; and 8 ways for **GMC, GA, BSM, SCSM**; for a total of 195 ways to instantiate a single schema for this failure. This ignores the multiplicative factor of the instantiation values provided by the user for free variables in the schemata, and the possibility of composing multiple schemata for more alternative recovery actions.

If it is the case that the failure in **Aero Drag** is determined to have been caused by an earlier undetected failure in the **Interpolate** function – caused by "interpolation" outside of the range of tabular data using a cubic spline method – then the introduction of a sanity check on the parameters of the **Interpolate** function introduces an additional 198 possible schema instantiations.

Clearly, not all 485 instantiations will be equally effective. The example here was selected to give some idea of the size of the space of possible instantiations. The specificity of a schema may give some clue to its potential usefulness. It is currently up to the run-time user of the LCM system to select the instantiation(s) to use.

3.3 The LCM Compile-time System Architecture

3.3.1 LCM System Inputs and Outputs

The purpose of the compile-time portion of the LCM system is to transform a "skeleton" optimization strategy – one without adequate failure handling – into a strategy that computes the identical function as the original, but with the ability to incorporate robust failure handling. This is accomplished principally by the generation of code wrappers for the original system. These wrappers contain the mechanisms that will allow errors to be detected and

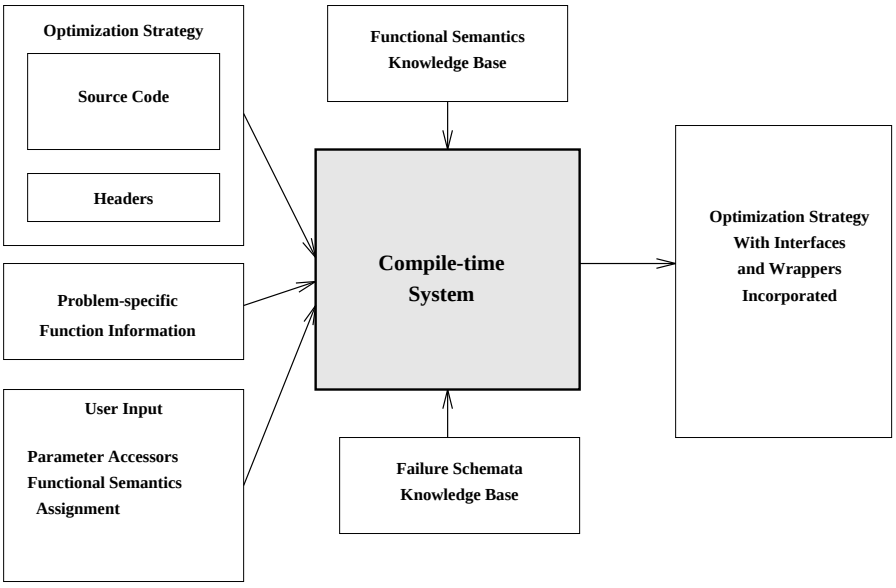


Figure 3.12: Conceptual Inputs and Outputs of the System

handled, and provide the interfaces to the LCM run-time system.

At the end of the compile-time step, the hooks for failure handling are in place, but have not been activated. If nothing else is done, the optimization strategy will run as it did before the new code was incorporated. Specific failure-handling strategies are incorporated as needed at run-time. This process involves selection of appropriate strategies from the database (driven by the type of failure along with the structure and function semantics information gathered during the compile-time phase) and instantiation of these generic schemata into specific contexts within the wrapped code.

Figure 3.12 illustrates the behavior of this part of the LCM system. The original source code and header descriptions of the functions to be wrapped are input. Additional semantic information is provided by the user so that functions may be associated with information in the LCM system database. All of the required interface code is automatically generated, and function symbols are dynamically translated in the original code during compilation. The resultant system is then linked together, resulting in a program that is ready to be run with the LCM run-time system.

3.3.2 The Functional Semantics Knowledge Base

Function Class Hierarchy

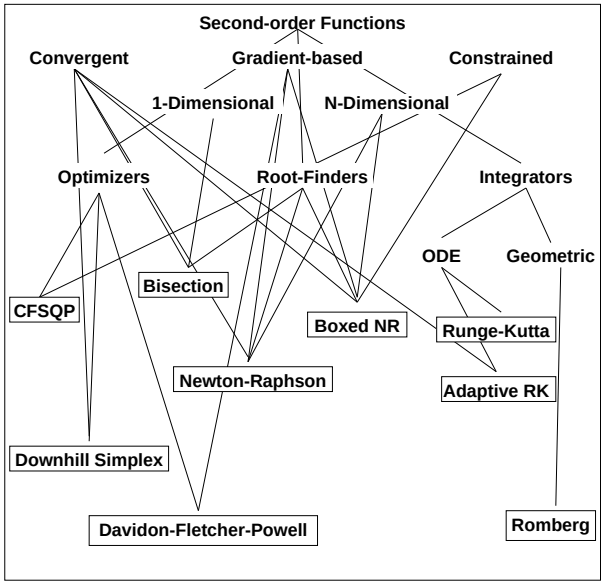


Figure 3.13: Fragment of the Function Class Hierarchy

Figure 3.13 shows a fragment of the function class hierarchy of the knowledge base. The figure shows the type relationships among several second-order numerical methods. As can be seen in the diagram, the hierarchy allows multiple-inheritance of parent classes.

The assignment of a function to a class within this hierarchy is the mechanism by which functional semantics are associated with elements of the optimization strategy. By default, all elements are classified at the root of the hierarchy, the class `function`. When the optimization strategy is initially input to the LCM system, the user must provide additional classification, as needed.

Associated with each class in the hierarchy is information about the set of parameters that the function accepts. Parameters are inherited by subclasses, therefore the set of parameters accepted by a function is the union of the parameters of all its parent classes plus any parameters unique to itself. Parameters are assigned unique, semantically meaningful labels (e.g. :SEED, :CONVERGENCE-TOLERANCE). When a function is first assigned to a class, accessor functions must be created (by the user) for each of the parameters defined for

that class. The function may accept additional parameters not represented in the hierarchy.

Functional Semantics Database

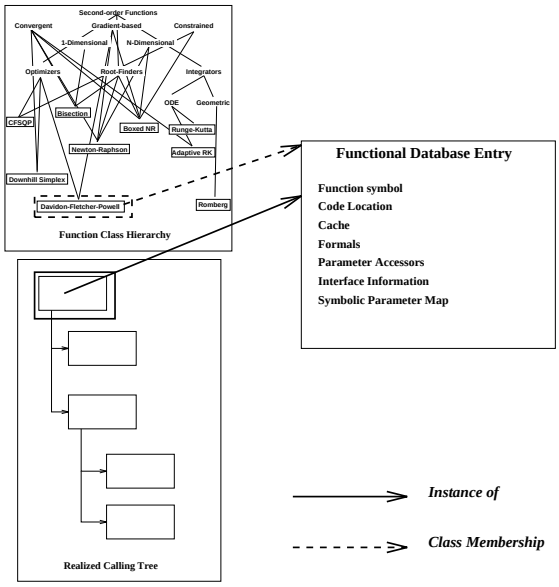


Figure 3.14: Functional Semantics Database

Each function in an optimization strategy that is to be recognized by the LCM system must have an entry in the functional semantics database. As diagrammed in Figure 3.14, each entry relates a node of the realized calling tree with a class in the function class hierarchy. There is one database entry instance per function.

In addition to giving the function a semantic interpretation, the database entry provides all of the information that the LCM system requires to interface with the actual code that implements the function. The name and location of the code are recorded (code may be loaded internal or external to the LCM system). The cache for the function (if required) is defined. Certain other information about the behavior of the function is also stored, such as whether the function is serially or multiply reentrant, whether it is functional (the output is a function only of the formal parameters without side-effects).

Of particular importance are the accessor functions and symbolic parameter maps: these provide the information required to instantiate the generic recovery schemata into operational

Abstract Function:

```
Optimize (:SEED :GRADIENT-STEP-SIZE :MAX-ITERATIONS
          :TERMINATION-TOLERANCE :OPTIMIZE-DIRECTION
          :OBJECTIVE-FUNCTION :DIMENSIONS)
```

Concrete Functions:

```
void cfsqp(int,int,int,int,int,int,int,int,int,int *,int,int,
           int,int *,double,double,double,double,double *,
           double *,double *,double *,double *,double *,
           void (*)(int,int,double *,double *),
           void (*)(int,int,double *,double *),
           void (*)(int,int,double *,double *,
                    void (*)(int,int,double *,double *)),
           void (*)(int,int,double *,double *,
                    void (*)(int,int,double *,double *)));

void powell(double p[], double **xi, int n, double ftol,
            int *iter, double *fret, double (*func)(double []));
```

Figure 3.15: Abstract and Concrete Functions

code. Rules for recovery schemata are written in terms of idealized abstractions of functions, with meaningful symbolic labels for parameters. Only a subset of the parameters that are common to all functions of a particular class are assigned symbolic labels in the abstract function. An example of the difference in representation is given in Figure 3.15, showing two different instantiations as C-language routines of the abstract function `Optimize`.

3.3.3 The Socket Control Interface

Incorporated Code vs. Stand-alone Code

The description of the interaction between the optimzation strategy and the LCM run-time system, as shown in Figure 3.2, has left unspecified whether the strategy will execute within the same process and address space as the LCM system. The simplest and most straightforward way to integrate the two is to link them into a single executable image. In this model, the LCM run-time system is directly incorporated into the optimization strategy.

The advantages are simplicity of interface, and minimal execution overhead.

Direct incorporation is not always possible with a legacy system. Much legacy code has been written as into stand-alone systems, with the implicit assumption that they will execute in their own address space. Stand-alone systems typically are not even serially re-entrant: they are intended to be executed only once in a "clean" address space, and never "clean up" (close files, free unneeded memory, etc.) after themselves.

The socket control interface was developed to accomodate the need to execute stand-alone legacy code under the control of the LCM run-time system. The stand-alone code remains external to the LCM run-time system, executing in its own process and address space, which may even be located on another physical system. The interface is capable of supporting the same level of control as when the optimization strategy is directly incorporated, but at a significantly higher overhead cost.

Socket Control Interface Architecture

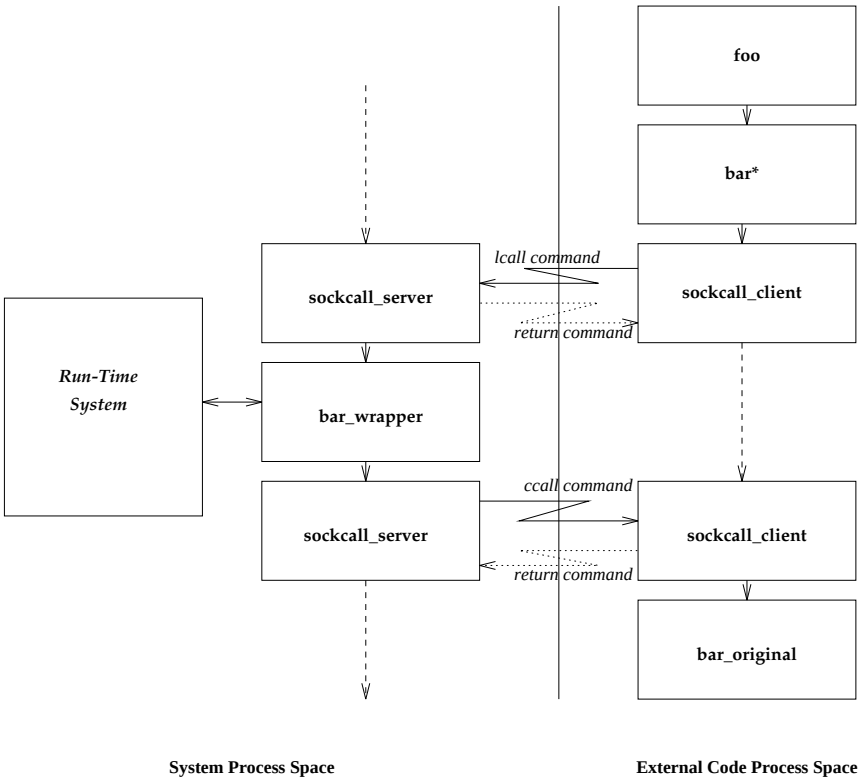


Figure 3.16: The Socket Control Interface

Figure 3.16 illustrates the operation of the socket control interface during a normal call-return sequence. As in Figure 3.2, the function `foo` originally called the function `bar`. In the figure, `foo` now calls `bar*`, a generated stand-in routine for `bar`, which calls `sockcall_client`, the client side of the socket control interface. `sockcall_client` is connected via a socket (a TCP/IP network socket if the connection is across machines, and a Unix socket if not) to `sockcall_server`, the server side of the interface. An `LCALL` command is sent across the socket, along with parameters necessary for the call, and the execution of `sockcall_client` suspends, waiting for a command to be received from `sockcall_server`.

On the server side, the wrapper function for `bar` is called, which first passes control to the LCM run-time system (as previously described) and then re-entrantly calls `sockcall-server`. A `CCALL` command and parameters are sent across the socket, where they are interpreted by `sockcall_client` which makes the call to the renamed original `bar` routine.

When the routine `bar` returns, a `RETURN` command is passed back across the interface to `sockcall_server`, and back through the various routines via the wrapper and LCM run-time system, to `foo`.

Commands

Command	Function
`HELLO, USESOCK`	Initiate session with external code
`SETJMP, ENVBLOCK`	Enable non-local GOTO in external code
`LONGJMP`	Execute non-local GOTO in external code
`LCALL, CCALL, RETURN`	Function call and return
`FORKCALL`	Function call with subprocess fork
`ERROR`	Error notification from external code
`GETMEM, SETMEM, MEMBLOCK`	Examine and change external address
`GETADDR, RTNADDR`	space and symbols
`SETCALL, SETNOTIFY`	Change interaction mode for wrappers
`SETDISABLE, NOTIFY`	in external code
`EXIT`	Terminate external execution

Table 3.10: Socket Control Interface Commands

Information and control are exchanged across the socket control interface by means of

the commands given in Table 3.10.

Session establishment and termination are controlled by the `HELLO`, `USESOCK`, and `EXIT` commands. At the first call to `sockcall_client`, the socket link is established, and a unique socket is assigned to the session. The LCM system may request the client terminate at any point by sending an `EXIT`.

The server may enable non-local transfer of control for a function in external code by issuing the `SETJMP` command. When received, the client calls the system `setjmp` function, and returns the *environment* block to the server with the `ENVBLOCK` command. The server requests a non-local GOTO by sending the `LONGJMP` command along with the appropriate environment block.

A client-to-server call is requested by the `LCALL` command, followed by a block containing the parameters for the call. Function interface routines on both sides of the socket control interface are built at compile-time with the information necessary to package and unpackage parameter blocks. A server-to-client call is similarly requested by the `CCALL` command. A normal return from a call is performed in both directions by means of the `RETURN` command, followed by the a block containing the parameters and return values.

The `FORKCALL` command initiates a special form of the server-to-client call. When the client receives a `FORKCALL` command, it issues the system call `fork` before calling the target routine. The client parent process calls `wait`, which suspends its execution until the child completes. Because the client process has forked, any state changes introduced during the execution of the target routine can be discarded simply by terminating the child process, and resuming execution in the parent. This feature allows the LCM system to perform recovery strategies that require *restart* actions even in cases where the routine to be restarted is not re-entrant. If the child terminates normally, the parent process is sent an `EXIT` command, and terminates. There is a potentially large cost to issuing a `fork`, and `FORKCALL` should not be used indiscriminately.

Abnormal conditions are signalled across the interface with the `ERROR` command, followed by information about the error condition.

The server can examine or change storage in the client process by means of the `GETMEM` and `SETMEM` commands. Addresses corresponding global external symbols can be obtained by the `GETADDR` command. The `MEMBLOCK` and `RTNADDR` commands return the requested values from the client.

Finally, system execution efficiency can be substantially improved by use of the `SETNOTIFY` and `SETDISABLE` commands. These permit the client to forgo passing control to the server system at a particular call or return, instead only sending a brief notification that a call or return has occurred, or not communicating across the interface at all. Normal behavior can be restored by issuing the `SETCALL` command.

3.3.4 Incorporation of Failure Recovery into Legacy Code

The process by which failure recovery schemata are incorporated into an optimization strategy built around a legacy simulation program comprises several steps:

Acquisition and Compilation of the Legacy Code

The first part of the incorporation process is the *acquisition* of the code in the optimization strategy. The LCM system currently requires that source code be available for all functions of the optimization strategy that are to be placed under its control. In addition, C header information describing the form of the functions and their arguments and return values must also be available.

The LCM system is capable of interfacing with code written in Common Lisp, ANSI C, and Fortran-77. If the code is written in C, headers can be easily extracted; if written in Fortran, the public-domain compiler `f2c` [Laboratories, 1990] may be used to generate the equivalent C headers; if written in Lisp, the headers must be hand-written.

The C header information is used by the LCM system to automatically generate the interface routines and code wrappers for the functions being acquired. The legacy code is then compiled, using a special pre-processor that re-maps function symbols for functions

being wrapped. All remaining code is then compiled, and a new executable version of the optimization strategy is linked, containing the original code (unchanged, except for the wrapped routine names) and the interface and wrapper code. The resultant executable requires the LCM run-time system to execute.

Generation of the Realized Calling Tree

Once the code has been acquired, the optimization strategy is run several times, and the run-time stack contents are recorded. The data from these runs are used to generate an initial realized calling tree. The form of the tree need not be complete at this point, it can be extended at any time if required.

Updating of the Functional Semantics Knowledge Base

The user must create an entry in the function database for each discrete function in the realized calling tree, associating that function with a class in the function class hierarchy. In some cases, it may be necessary or desirable to extend the class hierarchy with new subclasses. By default, functions are initially assigned to the root of the hierarchy.

In addition, the user must provide accessor functions for the symbolic parameters of the function class to which each function has been assigned. In most cases, these are very simple Lisp functions that set and retrieve the nth item of the formal parameter list, but they may require more complex transformation, such as remapping from an array represenation to a list.

The burden of this step is reduced to the degree that the code being incorporated uses functions (such as numerical tools) which are already identified in the database. Many functions from the well-known *Numerical Recipes in C* collection [Press *et al.*, 1986], for example, are already defined.

Interactive Addition of Failure Recovery

Before failure recovery can be added, there must first be a failure. The legacy code is run, either in a systematic exploration of the design space, or by running the complete optimization strategy. At some point, it is assumed that an error condition will occur.

During interactive addition of failure recovery, the error is caught by the default handling in the LCM run-time system. The run-time stack is analyzed against the database of failure recovery schemata, and the information in the functional semantics knowledge base. The user is presented with the selection of schemata whose structural templates match the current failure. The user selects one, and is prompted for any additional instantiation data required. The schema is then instantiated into the active rule set for the optimization strategy, and execution is re-started. This process may be iterated until the user is satisfied that all required failure handling has been incorporated.

Finally, the instantiated failure recovery schemata are saved, along with the updated functional semantics knowledge base.

3.4 Knowledge about Context Derived From Legacy Codes

In this chapter we have described the system that we implemented, LCM, and how we use it to incorporate failure handling into design optimization strategies and legacy codes. We summarize here the knowledge acquired about the optimization strategy in this process, where it comes from, and how it is employed.

3.4.1 Knowledge about Structure

We can determine the calling relationships between functions in the strategy because, as a side effect of their execution, the wrapper functions in the strategy update a directed acyclic graph (the *realized calling tree*) that gives the transitive closure of the *calling* $\rightarrow$ *called* relation.

We use the structural knowledge in conjunction with knowledge about function semantics to determine where and how failure schemata can be instantiated into an optimization strategy.

3.4.2 Knowledge about Purpose

Structural information alone is insufficient to determine purpose. In order to infer how the value of a function will be used, it is necessary to have information about the semantics of the functions that are active at the time of the failure. This knowledge is provided by the user of the LCM system when an optimization strategy is first wrapped, and is stored in the functional semantics knowledge base.

We use this information in conjunction with structural knowledge to determine where and how to instantiate failure schemata into an optimization strategy.

3.4.3 Knowledge about Behavior

Finally, because evaluation functions are executed repeatedly during the course of optimization, we can collect a record of their behavior across a span of evaluations.

This knowledge allows both inference about the behavior of functions at points near the point of failure, points, and the use of the values in failure recovery.

3.5 Summary of Implementation

We have defined a model for failure handling and recovery that preserves the correctness of the optimization, and allows the optimization to continue in the presence of failure. In implementing this model, we created a system that:

- Can implement wrapper functions without altering original functions.
- Can be used with standard numerical tools and existing legacy simulation models.

- Can extract specific knowledge from legacy codes.
- Can allow returns and restarts of any functions following failure.
- Keeps track of context required to determine recovery actions.

Chapter 4
Empirical Results

There are two important dimensions along which we have evaluated the LCM system experimentally: the effectiveness of recovery strategies developed under the system in improving the results of optimization, and the utility of the LCM system as a development environment for failure handling strategies. In evaluating the first we wish to determine whether incorporating failure handling does, in fact, improve optimization results, and to what degree. In evaluating the second we wish to assess the effort of using the LCM system to implement failure handling strategies relative to the alternatives.

4.1 Results of Incorporation of Failure Handling on Optimization

4.1.1 Evaluation Methodology

In order to evaluate the effect of individual recovery schema on optimization, we ran a series of optimizations from multiple valid starting points within the design space for an optimization strategy from each test design domain. We repeated each set of optimizations from the identical starting points with and without failure handling schemata incorporated.

The CFSQP optimization algorithm was used as the primary optimization method in all strategies (except when explicitly overridden by the schema). For each trial optimization, we recorded the final result returned, the amount of CPU time used, the number of times the evaluation function was called, and the number of times the schemata were applied.

We have summarized the results for each set of experiments, by domain, in two tables and a graph. The first table of each set gives a measure of the result quality for a single

optimization run, by schema. The first two columns show the number of optimizations attempted and the number that completed (the number for which the optimization method terminated normally). The best result, average result, and standard deviation over all runs are given in the next three columns. By itself, this table gives a measure of the absolute improvement in optimization quality per schema, but no indication of the relative costs.

The second table weights the improvement in optimization quality by taking into account the cost in evaluations. We assume that for most real-world design problems (particularly those involving CFD codes in the evaluation function) the computational cost of the optimization will be dominated by the computational cost of the evaluation function. The first column shows the average number of evaluations per attempted optimization. The next column gives the average number of actions taken by the recovery schema being tested. The CPU utilization average is computed in the same manner as the evaluation average, and has been normalized for differences across machines on which the experiments were run.

The last two columns are intended to give a measure of the relative strength of the various schema. The column labeled "$(.99/\tau)$" predicts the number of function evaluations that would be required to have a 99% likelihood of reaching a result within τ percent of the best-known result for the problem. The single-optimization probability of reaching a result within τ is computed by dividing the number of optimizations that reach results within τ by the total number of optimizations performed for a particular problem. This probability is used to compute the expected number of independent optimizations (starting from randomly selected valid points) that would be required to have a 99% probability of acheiving a result within τ. Finally, the expected number of evaluations is computed by multiplying the expected number of independent optimizations by the average number of evaluations per optimization.

Because the relative strengths of the schemata may depend on the value chosen for the τ threshold, the $.99/\tau$ values for the schema are shown graphically for a range of τ values. The number of evaluations required is on the Y axis, and the τ value on the X axis. Points closer to the X axis at each threshold are superior.

The failure handling schemata were selected for these experiments based on our *a priori* knowledge that they would be likely to be of benefit. We have not done rigorous comparisons of all combinations of schemata under a wide variety of conditions. We wish only to show that there are conditions under which each schema can be shown to provide some benefit, and to demonstrate that the ability to perform failure recovery inside evaluation functions and in a context-sensitive manner can improve optimization results.

4.1.2 Incorporation of Single Failure Recovery Schema

Domain: Synthetic Failure Function

The synthetic failure function is a quasi-quadratic[1] 3-parameter evaluation function with non-linearly bounded failure regions. The function has a known single global optimum that lies on the intersection of two failure surfaces, ensuring that optimization will encounter unevaluable points as it converges on a solution. The function is partially computed using tabular interpolation for the function x^2 for values from 0.0 to 5.0.

Figure 4.1 shows the effects of incorporating several schemata individually into the synthetic failure function. Failure recovery schemata tested were Simple Bad Value, Simple Interpolation, Random Multi Start, Multi Method Search, Backtracking Parameter Restart, Table Extrapolation, Gradient Method Change, and Gradient Approximation.

The results show the difficulty of optimizing this function, as most of the schemata are individually insufficient to allow optimization complete. The **SI** schema depends on having at least a minimal number of points cached in order to extrapolate a result; in these tests the first failure occured before enough points were successfully evaluated. The **MMS** and **BPR** schemata depend on the completion of at least one successful optimization by one of the methods given; with no other failure handling incorporated, no optimization was able to complete. The **SBV** optimizations were able to complete in all cases, as were the **GA** and

[1] A truly quadratic function would be too easy to optimize with the constrained quadratic programming optimizer we used.

Schema	Optimizations Attempted	% Opt. Completed	Best Result	Average Result	Standard Deviation
None	97.	0.0	NA	NA	NA
SBV	97.	100.00	-15.88	-2.00	5.40
SI	97.	0.0	NA	NA	NA
RMS	97.	100.00	-18.52	-9.56	4.60
MMS	97.	0.0	NA	NA	NA
BPR	97.	.00	0.00	NA	NA
TE	97.	0.0	NA	NA	NA
GMC	97.	0.0	NA	NA	NA
GA	97.	100.00	-15.88	-2.00	5.40

Schema	Avg. Evals/ Opt.	Avg. Recovery Actions/Opt.	Avg. CPU/ Opt.	$(.99/\tau)$ Evals
None	9.0	0.0	0.02	NA
SBV	144.29	61.96	0.31	NA
SI	9.00	1.00	0.03	NA
RMS	1745.74	755.00	3.64	NA
MMS	9.0	0.0	0.02	NA
BPR	5.09	0.00	0.01	NA
TE	10.2	1.2	0.01	NA
GMC	9.0	0.0	0.02	NA
GA	143.91	0.04	0.46	NA

Figure 4.1: Single-Schema Results with Synthetic Failure Function

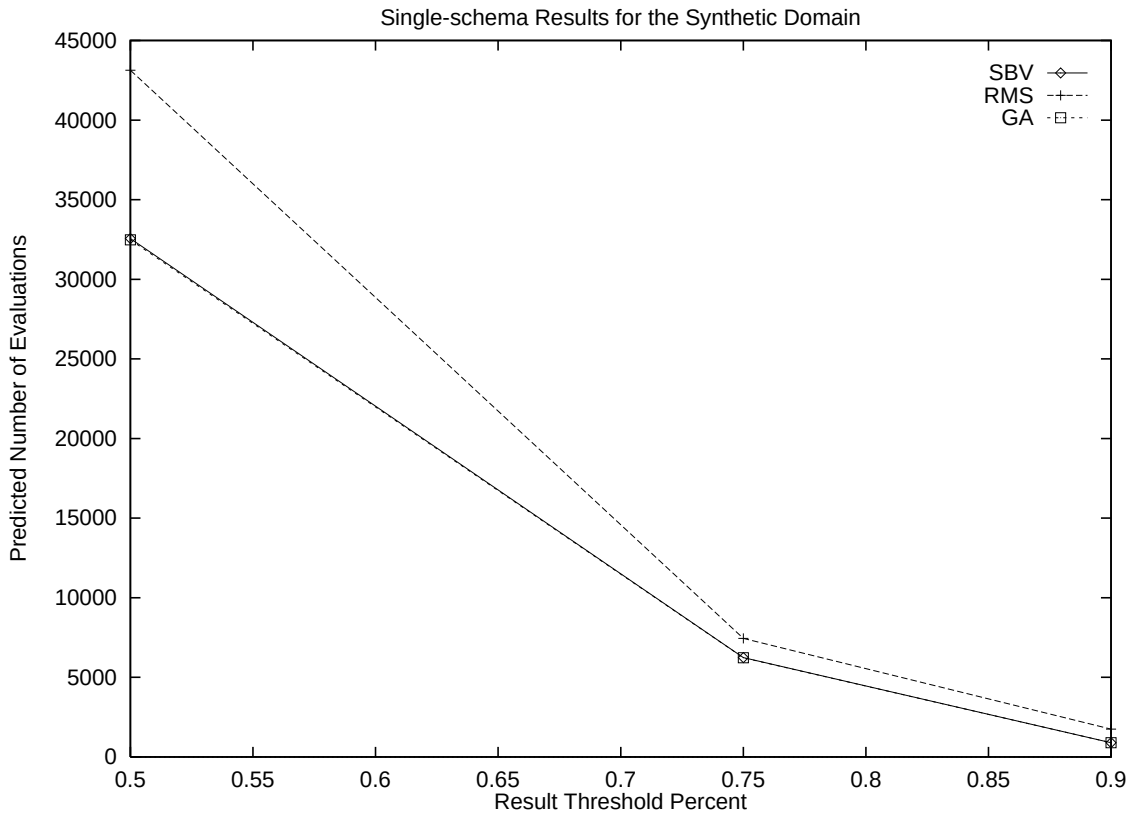


Figure 4.2: Single-Schema Results with Synthetic Failure Function

RMS optimizations. This is not terribly surprising, as both the **GA** and **RMS** schemata also translate evaluation function failure into bad values. Since the **GA** results are identical to the **SBV** results, we can conclude that the portion of the **GA** recovery that takes place during gradient computation was never exercised.

In absolute terms, the **RMS** schema provided the best results. However, none of the runs that completed came close to the best known value for this problem. Considering that the **RMS** results reflect 970 separate optimizations, we can conclude that this function presents a serious challenge to the CFSQP optimizer.

The cost results table shows that although the **TE** schema did not allow optimization to complete, it did, on average, allow it to continue slightly further before the non-linear bounded failure regions were encountered. Since none of the optimizations that actually completed came within a reasonable τ of the best known optimum, the last column of the table is not filled in.

Schema	Optimizations Attempted	% Opt. Completed	Best Result	Average Result	Standard Deviation
None	61.	.00	0.00	NA	NA
SBV	61.	98.36	354.87	360.83	12.22
SI	61.	33.00	354.56	414.24	61.94
RMS	61.	100.00	354.82	355.47	0.57
MMS	61.	.00	0.00	NA	NA
BPR	61.	.00	0.00	NA	NA
TE	61.	100.00	354.87	396.96	43.59
GMC	61.	.00	0.00	NA	NA
GA	61.	98.36	354.87	360.83	12.22
BSM	61.	100.00	371.00	429.63	11.65

Schema	Avg. Evals/ Opt.	Avg. Recovery Actions/Opt.	Avg. CPU/ Opt.	$(.99/\tau)$ Evals
None	9.00	0.00	1.49	NA
SBV	107.03	7.16	17.65	843.27
SI	15.89	7.89	2.24	2194.42
RMS	1068.41	0.00	178.21	1633.39
MMS	9.00	0.00	1.37	NA
BPR	8.13	0.00	1.22	NA
TE	76.03	41.46	12.83	1760.84
GMC	9.00	0.00	1.41	NA
GA	106.93	0.02	17.14	842.50
BSM	63.92	44.79	19.20	NA

Figure 4.3: Single-Schema Results for Yacht Hull Optimization

Looking at the graph in Figure 4.2 shows that of the three schemata that succeeded, **GA** and **SBV** were superior to **RMS**. Note that the scale of the X-axis shows that none of the schemata come much closer than 50 percent of the best known value.

Domain: Racing Yacht Hull Design

Figure 4.3 shows the effects of incorporating several schemata individually into a yacht hull design optimization strategy. Failure recovery schemata tested were Simple Bad Value, Simple Interpolation, Random Multi Start, Multi Method Search, Backtracking Parameter Restart, Table Extrapolation, Interpolate and Incorporate Constraint, Gradient Method Change, Gradient Approximation, and Bound Search Method.

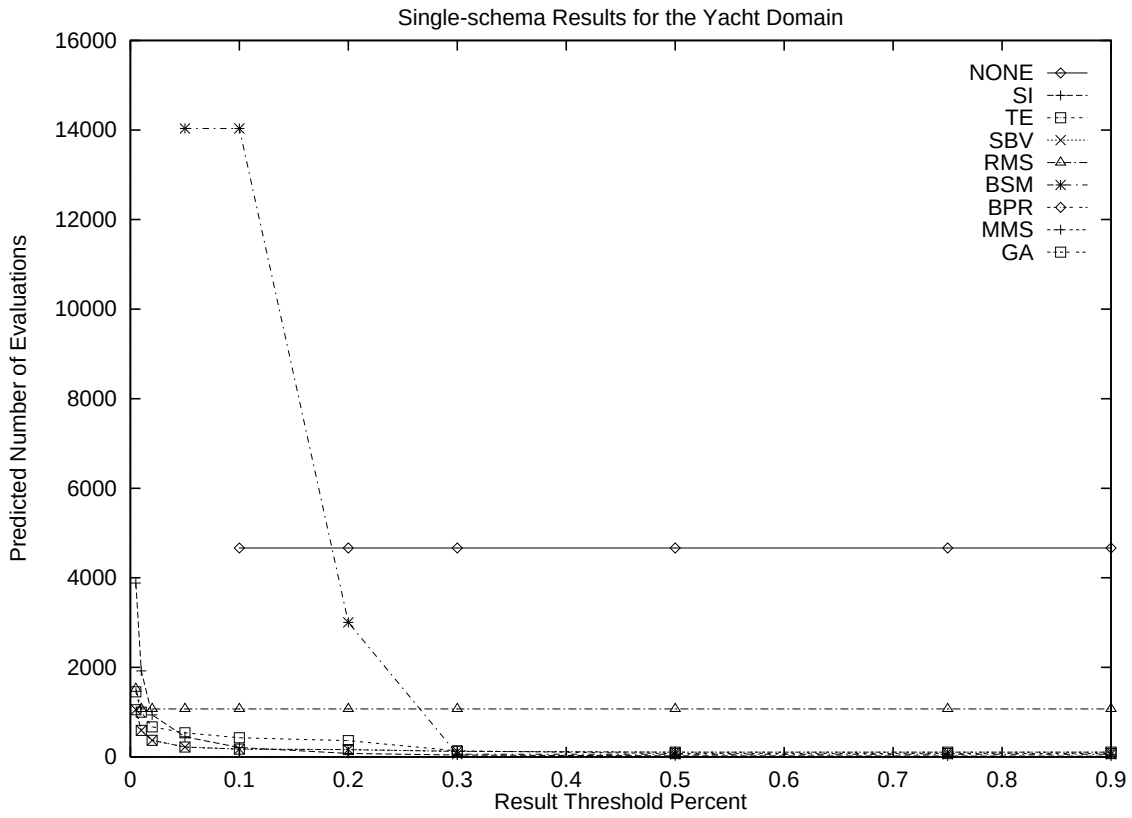


Figure 4.4: Single-Schema Results for Yacht Hull Optimization

These results show that most of the schemata incorporated had at least some effect on improving the optimization. Only the **MMS**, **BPR**, and **GMC** schemata failed to allow the optimization to complete. Several of the single schemata were able to find the best known value for this problem.

The cost-weighted results show that for a τ of 1%, **SBV** and **GA** are approximately equal, indicating that failure during gradient computation is not a factor when individual failure schemata are used. The **TE** schema is about twice as expensive, slightly worse than **RMS**. The **SI** schema is in last place by a larger margin.

The graph in Figure 4.4 shows that **SBV**, **GA**, and **TE** are roughly equally good over the entire range. This would suggest that table extrapolation problems are the principal source of failure in this problem, but that treating them as bad values is about as good as extrapolating them – if you can keep the optimization going, it will get to the global pretty quickly. This may not be particularly surprising considering that the function is nearly

Schema	Optimizations Attempted	% Opt. Completed	Best Result	Average Result	Standard Deviation
None	62.	.00	0.00	NA	NA
SBV	62.	3.23	123071.62	136185.47	13113.86
RMS	29.	100.00	122966.35	123240.12	191.05
MMS	62.	.00	0.00	NA	NA
BPR	62.	.00	0.00	NA	NA
TE	62.	.00	0.00	NA	NA
GMC	62.	.00	0.00	NA	NA
BSM	62.	.00	0.00	NA	NA

Schema	Avg. Evals/ Opt.	Avg. Recovery Actions/Opt.	Avg. CPU/ Opt.	$(.99/\tau)$ Evals
None				
SBV	11.52	9.94	19.81	NA
RMS	2030.55	0.00	5040.96	85632.77
MMS	1.26	0.00	2.75	NA
BPR	1.03	0.00	1.32	NA
TE	7.84	141259.48	49.26	NA
GMC	2.29	0.00	10.00	NA
BSM	1.00	0.00	1.07	NA

Figure 4.5: Single-Schema Results for Nozzle Optimization

quadratic, and except for the failure regions, quite smooth.

The **RMS** schema is seen to be somewhat more expensive over most of the range, and the **BPR** to be considerably more expensive. The **BSM** technique is good up to a point, but degrades at high quality levels, and fails to acheive the known best result.

Domain: Conceptual Jet Engine Nozzle Design

Figure 4.5 shows the effects of incorporating several schemata individually into a nozzle design optimization strategy. Failure recovery schemata tested were Simple Bad Value, Random Multi Start, Table Extrapolation, Gradient Method Change, Gradient Approximation, and Bound Search Method.

The results show that only the **SBV** and **RMS** schemata were able to allow optimization to complete, and the **SBV** schema only worked in 3.2% of the tests. The number of runs for **RMS** is lower than for the other schemata because of the very small standard deviation

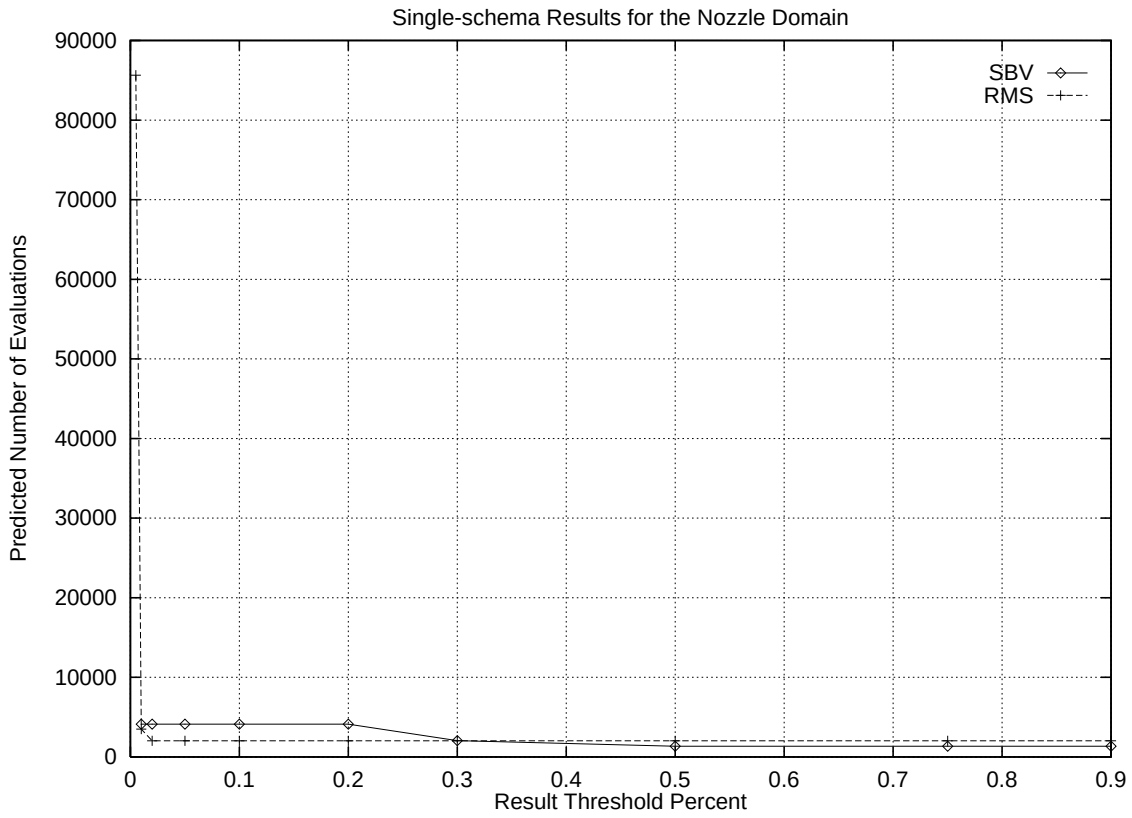


Figure 4.6: Single-Schema Results for Nozzle Optimization

of the results.

The results in the costs table show that, although ineffective overall, the **TE** schema was exercised heavily, suggesting that it may have considerable value in combination with other schemata. The **RMS** schema was able to achieve results within $\tau = 0.5\%$, but at an enormous predicted cost in evaluations.

The graph in Figure 4.6 is interesting in that it shows that neither of the two methods applied individually is superior to the other over the entire range of τ values. From τ of 30% down, it appears that **SBV** has a slight edge, which jumps considerably at the 0.5% threshold.

Domain: Conceptual Aircraft Design

Table 4.7 shows the effects of incorporating several schemata individually into a nozzle design optimization strategy. Failure recovery schemata tested were Simple Bad Value, Random

Schema	Optimizations Attempted	% Opt. Completed	Best Result	Average Result	Standard Deviation
None	10.	.00	0.00	NA	NA
SBV	10.	100.00	115006.13	116540.97	1181.23
RMS	10.	100.00	114403.49	114939.92	313.83
TE	10.	.00	0.00	NA	NA
GMC	10.	.00	0.00	NA	NA

Schema	Avg. Evals/ Opt.	Avg. Recovery Actions/Opt.	Avg. CPU/ Opt.	$(.99/\tau)$ Evals
None	10.00	0.00	871.65	NA
SBV	94.30	32.30	7469.39	NA
RMS	994.20	702.00	87836.23	NA
TE	22.20	5077.10	1921.86	NA
GMC	11.00	0.00	791.32	NA

Figure 4.7: Single-Schema Results for Aircraft Optimization

Multi Start, Table Extrapolation, Gradient Method Change, Gradient Approximation, and Bound Search Method.

As in the nozzle design domain, only the **SBV** and **RMS** schemata were found to be effective independently. As would be expected, the **RMS** schemata yields the best single-run results, though it fails to come within the τ threshold of 1%.

The costs table shows that the **TE** schema was exercised in this problem, despite failing to allow optimization to complete. Other schemata make little headway.

Because of the small standard deviation in the results of the completed runs in this domain, the graph in Figure 4.8 is not very interesting. If a τ threshold of only 20% is desired, the **SBV** schema would be superior to **RMS** by about an order of magnitude.

4.1.3 Incorporation of Multiple Failure Recovery Schema

We next evaluated the results of composing recovery schemata together using the methodology described previously. The results are summarized by domain.

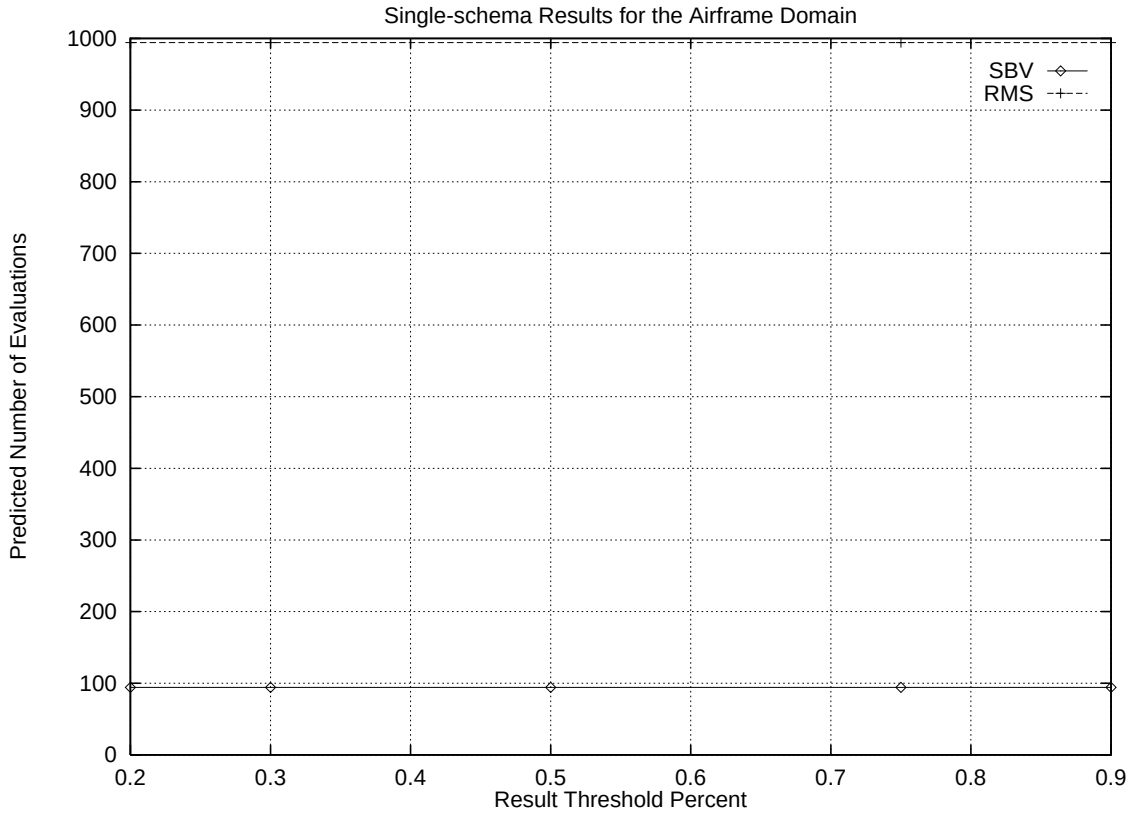


Figure 4.8: Single-Schema Results for Aircraft Optimization

Domain: Synthetic Failure Function

Table 4.9 shows the effects of combining schemata into the synthetic failure function optimization strategy.

The results show that all of the combinations of schemata were successful in allowing optimization to complete. The best and average results differ significantly among combinations, although the standard deviations are all quite close. From this table alone it would be quite difficult to determine the best combination, though **SBV+TE+MMS** has the best overall value.

The graph in Figure 4.10 gives a clearer picture of the combination results. The **SBV+TE+MMS** combination is a clear winner for τ values of less than 50%. Below that, the methods are quite close, with **SBV+RMS** possibly having an edge, but deteriorating rapidly.

It is interesting to note that the best value was only found during testing when the **TE** schema was employed.

Domain: Racing Yacht Hull Design

Table 4.11 shows the effects of combining schemata into a yacht design optimization strategy.

All of the combinations tried resulted in 100% completion of optimizations, and the best overall result was found in every case. The average results are also quite close in in value, and the standard deviations are quote low. The results would seem to indicate that the methods are all about equally effective for a single optimization run.

Because the per-optimization costs are quite different for the various combinations, the cost results favor the cheaper schemata. **SBV+TE** and **SBV+TE+GMC** are the clear winners in this case. The identical results for these two, once again suggests that failure during gradient computation is not a factor in this problem, at least once table extrapolation has been handled. Notice that **GMC** *does* show a significant improvement when added to **SBV+RMS**.

Schema	Optimizations Attempted	% Opt. Completed	Best Result	Average Result	Standard Deviation
SBV+TE	97.	100.00	-14.87	-2.68	4.53
SBV+RMS	97.	100.00	-19.73	-10.40	4.60
SBV+TE+MMS	97.	100.00	-28.00	-2.50	4.80
SBV+TE+GMC	97.	100.00	-14.87	-2.66	4.60
SBV+GMC+RMS+GMC	97.	100.00	-17.63	-9.81	4.65

Schema	Avg. Evals/ Opt.	Avg. Recovery Actions/Opt.	Avg. CPU/ Opt.	$(.99/\tau)$ Evals
SBV+TE	138.68	81.77	0.29	NA
SBV+RMS	1367.77	918.61	2.11	NA
SBV+TE+MMS	781.54	708.77	1.37	347310.66
SBV+TE+GMC	138.68	81.77	0.28	NA
SBV+GMC+RMS+GMC	1358.43	905.15	2.04	NA

Figure 4.9: Multi-Schema Results with Synthetic Failure Function

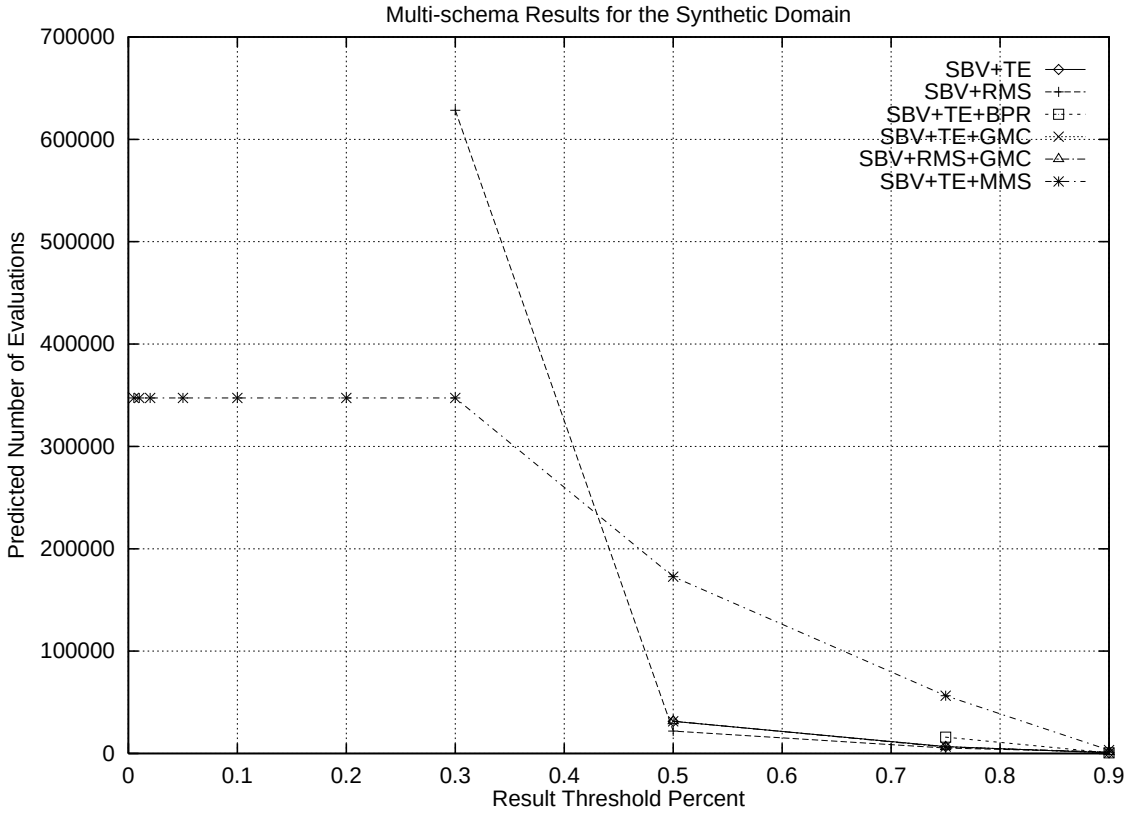


Figure 4.10: Multi-Schema Results with Synthetic Failure Function

Schema	Optimizations Attempted	% Opt. Completed	Best Result	Average Result	Standard Deviation
SBV+TE	61.	100.00	354.87	396.96	43.59
SBV+TE+BPR	61.	100.00	354.87	369.49	21.23
SBV+RMS	61.	100.00	354.84	356.02	1.10
SBV+TE+MMS	61.	100.00	354.87	397.54	48.58
SBV+TE+GMC	61.	100.00	354.87	396.96	43.59
SBV+RMS+GMC	61.	100.00	354.87	355.94	1.25

Schema	Avg. Evals/ Opt.	Avg. Recovery Actions/Opt.	Avg. CPU/ Opt.	$(.99/\tau)$ Evals
SBV+TE	76.03	41.46	12.85	1760.84
SBV+TE+BPR	212.62	224.80	35.96	3469.36
SBV+RMS	819.41	728.11	143.11	2819.67
SBV+TE+MMS	1399.07	3118.05	171.81	32400.84
SBV+TE+GMC	76.03	41.46	12.64	1760.84
SBV+RMS+GMC	839.03	717.64	145.86	2625.25

Figure 4.11: Multi-Schema Results for Yacht Hull Optimization

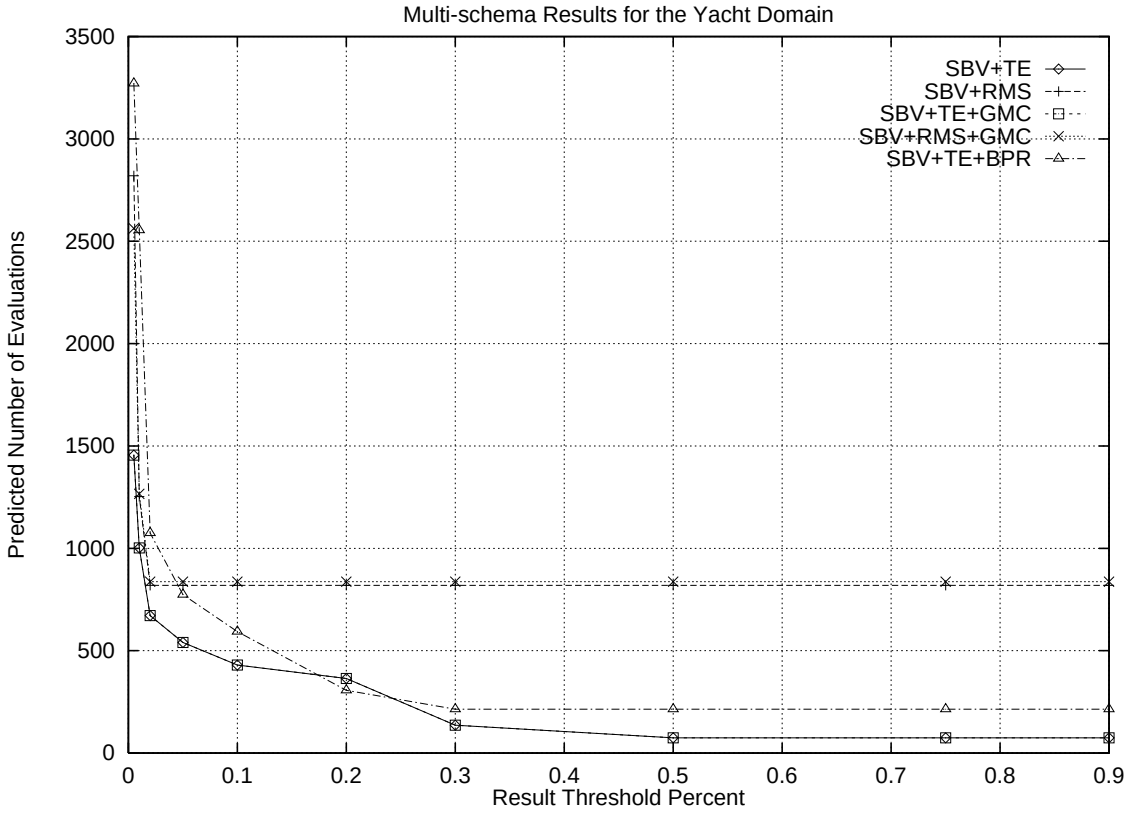


Figure 4.12: Multi-Schema Results for Yacht Hull Optimization

The graph in Figure 4.12 shows that results are pretty consistant across across the range of τ threshold values, except that the combinations that include **RMS** are substantially worse at lower τ values due to the high per-run cost.

The **MMS** results are significantly worse over the entire range, and are not shown on this graph, to avoid compressing the vertical scale.

Domain: Conceptual Jet Engine Nozzle Design

Table 4.13 shows the effects of combining schemata into a nozzle design optimization strategy.

Schema	Optimizations Attempted	% Opt. Completed	Best Result	Average Result	Standard Deviation
SBV+TE	62.	85.48	122433.42	194848.20	274691.66
SBV+TE+BPR	62.	93.55	122447.68	141990.23	14080.70
SBV+RMS	62.	98.39	122433.42	156736.89	198797.66
SBV+RMS+GMC	31.	100.00	122579.23	123816.30	1111.23

Schema	Avg. Evals/ Opt.	Avg. Recovery Actions/Opt.	Avg. CPU/ Opt.	$(.99/\tau)$ Evals
SBV+TE	77.06	851671.25	341.81	7155.58
SBV+TE+BPR	158.10	2049936.90	787.82	14679.58
SBV+RMS	329.23	2768372.20	1244.59	18031.46
SBV+RMS+GMC	1001.29	10309330.00	4531.29	26215.80

Figure 4.13: Multi-Schema Results for Nozzle Optimization

Unlike the previous two domains, the three combinations tested are very unequal in their ability to bring optimization to completion. In this case, adding the **GMC** schema to the combination can be seen to allow more runs to finish, though the best result found is not as good.

Surprisingly, the cost table shows that the combination with the lowest completion rate is the winner when ranked by cost of optimization. For a τ of 0.5%, the **SBV+TE** combination is better by more than a factor of 2 than the next-ranked combination, and by a factor of more than 6 better than the worst.

Looking at the graph over the range of τ values shows that, again surprisingly, the

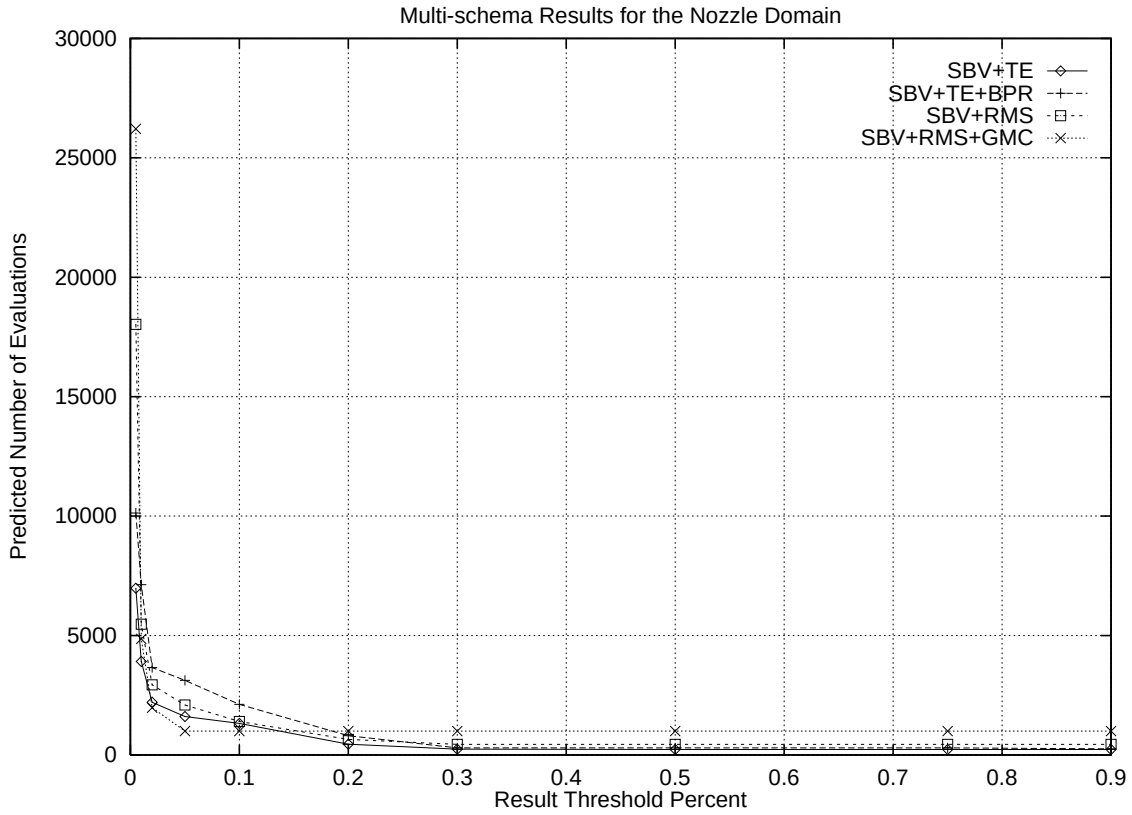


Figure 4.14: Multi-Schema Results for Nozzle Optimization

performance of all of the combinations are roughly equal over most of the range. The **SBV+TE** combination is at least as good as any other combination throughout the range, giving it overall superiority.

Domain: Conceptual Aircraft Design

Table 4.15 shows the effects of combining schemata into an aircraft design optimization strategy.

For this domain, only three combinations were tested because of the high cost of evaluation. The single-run best, average, and standard deviations for all three are close. The cost-weighted results show that **SBV+TE** is significantly better than **SBV+RMS**. Adding the **GMC** schema, however, raises costs without a corresponding improvement in performance.

The graph in Figure 4.16 is consistent with the tabular data, showing **SBV+TE** to be superior by a large margin to **SBV+RMS** and **SBV+RMS+GMC** over the entire range of τ threshold values.

4.1.4 Summary of Schemata Incorporation Results

We will now summarize the conclusions we have drawn from these experiments:

1. Some form of failure handling is required.

   In only one test domain (yacht hull design) was even a single optimization able to run to completion in the absence of failure handling. In all of the experiments, failure seems to be inevitably encountered on the path to the optimum, usually before even a single iteration step has completed from the initial valid starting point.

   In the absence of failure handling, the optimization process appears to be reduced to a process of evaluation of random points in the design space.

2. Handling failures in more than one way is better than only handling them in a single way.

Schema	Optimizations Attempted	% Opt. Completed	Best Result	Average Result	Standard Deviation
SBV+TE	10.	100.00	102285.92	104413.71	709.26
SBV+RMS	10.	100.00	102279.32	103050.87	1052.46
SBV+RMS+GMC	10.	100.00	102278.08	103318.76	1093.93

Schema	Avg. Evals/ Opt.	Avg. Recovery Actions/Opt.	Avg. CPU/ Opt.	$(.99/\tau)$ Evals
SBV+TE	73.00	26238.80	5276.48	3190.73
SBV+RMS	733.90	2032842.70	39708.71	2807.15
SBV+RMS+GMC	739.70	1569623.10	41286.69	3717.65

Figure 4.15: Multi-Schema Results for Aircraft Optimization

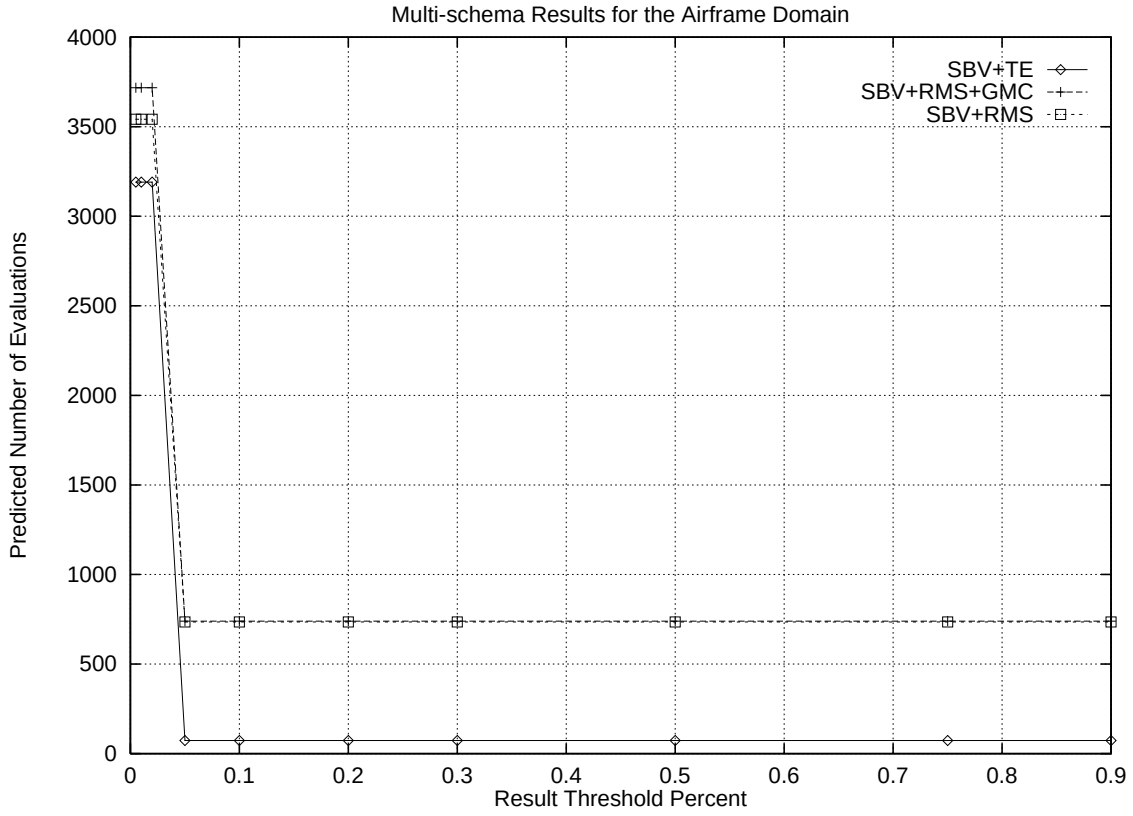


Figure 4.16: Multi-Schema Results for Aircraft Optimization

Across all of the domains, optimization quality per evaluation improved when multiple failure handling was incorporated. In the synthetic and aircraft domains, the only solutions found within 20% of the best known solution were with multiple schemata.

3. General failure handling can be successfully combined with more specific failure handling.

   The combinations of schemata tested include combinations of the **SBV** schema, which will provide handling for *any* error, with schema such as **TE** which only handle very narrow classes of error. The results are uniformly superior to either being used alone.

4. A weak approach can be successful, if cost is not a problem.

   In three of the four test domains (excepting the synthetic failure domain), the **SBV+RMS** combination was successful at a cost of not more than a factor of 3 greater than the best combination. In the synthetic domain, it was the second-best combination method, despite not reaching the best known value.

   This result is somewhat disappointing, as it suggests that computational cost probably can substitute to a large degree for sophistication in failure handling, particularly where the evaluation cost is low.

5. On the other hand, there is a definite, significant improvement to be seen in selecting the right combination of recovery schemata.

   The differential between the best result and the second best ranges from a minimum of 10% lower cost to a maximum of 50% lower cost. The range from best combination to worst combination goes to a maximum of 90% improvement, counting only combinations that succeed in reaching the best known value.

6. Improvement is not monotonic in the number of schemata incorporated.

   There are clear examples in each test domain where adding a schema (for example **GMC** to a combination) results in a larger increase in the computational cost than in the quality of the result. We conclude that it is possible to over-handle failure:

sometimes it is better to allow some failures to terminate the simulation than to try to patch them up and continue, as the end result will not turn out to have been worth the effort.

4.2 Productivity Gains

We have not undertaken a controlled experiment with the intent of determining the degree to which the LCM system increases programmer productivity when implementing failure handling in optimization strategies. Though we believe such an experiment would be worthwhile and revealing, the logistical difficulties in identifying and training a sufficient group of programmers places it outside the scope of our resources. Any discussion of productivity improvement, therefore, must be based on anecdotal evidence, principally derived from the experience of the author.

We do believe that the LCM system provides considerable leverage in incorporating failure-handling into legacy programs over the use of traditional programming languages. If presented with an opportunity to develop a new design optimization system based on an existing legacy simulation model, we would ourselves choose to use the LCM system, particularly if time were a consideration. We are, of course, eminently familiar with LCM, and do not have to undertake the large initial cost of learning a new language faced by any other programmer.

4.2.1 Lines of Code Generated

One (perhaps rather questionable) metric is the number of lines of wrapper code actually generated to carry out a recovery schema. The majority of the generated code handles the interface between a legacy simulation code and the LCM run-time system. It can be argued that to acheive the desired wrapper functionality obtained from the generated code, roughly the same code would have to be developed by hand. While this may be true, it is unlikely that one would consider it worth the effort to hand-code such an interface.

For the C aircraft vehicle model previously described, there are 42 discretely callable functions for which wrappers are generated. There are 2093 lines of generated C code and 2299 lines of generated lisp code. This works out to an average of approximately 105 lines of generated code per function to be interfaced. This completely ignores the portion of the recovery schema for which no code is generated.

If one believes another highly questionable metric, the "30 lines of code per programmer per day" estimate of programmer productivity, the 4392 lines of generated code would represent 133 programmer-days of effort.

4.2.2 Comparison to Legacy Code for Equivalent Functionality

In developing our schema, we have "rationally reconstructed" a number of recovery schemes previously implemented. One such, **BSM**, was originally called *Newton-in-a-box*, and was implemented by Gelsey, Smith, and Miyake at Rutgers for their nozzle design optimization system. Newton-in-a-box differs from **BSM** in being a hard-coded algorithm for bounding a Newton-Raphson root finding algorithm within a specified box. The code was partially adapted from the Numerical Recipes in C routine `newt`, and uses the LINPACK Gaussian elimination routines.

The schema **BSM** is implemented in 8 rules, the two listed in the previous chapter, and 6 other trivial "housekeeping" rules. It can be applied to any identified search-based method by instantiation of a single variable, the bounds of the constraining box. In contrast, the Newton-in-a-box routine is 147 lines of C code, and is single-purpose. (This is not to imply any criticism of the C code, but simply to compare relative sizes.)

Regardless of what metric one uses to estimate the cost of developing the C routine, it would seem fairly obvious that the effort required to initially develop the **BSM** schema should be less. The effort to transfer the schema from one context to another (by instantiation vs. re-coding) should be less by several orders of magnitude.

4.2.3 Comparison to "Compiled Schema"

For performance reasons, we found it advantageous to code "compiled schema": routines that would be inserted into code wrappers in the same fashion as the rule-based schema, that would have access to the same LCM run-time system, predicates, and action functions (like `CacheInterpolate`), but would be hard-coded in Lisp and compiled for speed. We can compare the original rule set with the corresponding Lisp function to get some measure of the relative effort.

As an example, we created a compiled version of **RMS**. The rule-based schema consists of a single rule. The Lisp equivalent consists of a 56-line function. Again, the rule-based version was substantially easier to develop, even though both were developed in the context of the LCM run-time system, with access to the same functionality.

4.2.4 Actual Time to Implement

Finally, we note an occasion within the past year in which the author was requested to add a new modeling parameter to a Numerical Recipes routine that would limit the maximum number of iterations. The iteration maximum was previously defined in the code as a compile-time constant. Modification of the code and tracking down and changing all of the routines in the system that used that routine to add the new parameter was a process taking about 8 hours of the author's time. As an exercise, the author later coded a single-rule wrapper to accomplish the same result (for any iterative method) in approximately 15 minutes.

4.3 LCM Run-time System Performance

We have also attempted to evaluate the additional computational overhead incurred in incorporating failure handling through the LCM system. The cost is dependent on the number of schema, the number of wrapped functions in the optimization strategy, and the method of communication between the LCM run-time system and the optimization strategy used.

4.3.1 Cost of the Wrapper Interface

In order to incorporate failure handling, functions are replaced by wrapper code that transfers control to the LCM run-time system before calling the wrapped function. There is some overhead associated with this transfer of control, as it involves moving parameters into and out of a Lisp environment.

Number of Parameters	CPU without Wrappers	CPU with Wrappers
none	0.01	0.12
1	0.01	0.14
5	0.01	0.21
10	0.02	0.35

Table 4.1: Wrapper Overhead Costs

Table 4.1 shows the results of calling a null C function with different sized parameter lists with and without wrappers. The increase in computation cost shown can be entirely attributed to the wrapper overhead.

Viewed in isolation, the wrapper increases the call overhead by a very large factor that scales linearly with the number of parameters. The performance implications of this will depend on the way in which wrappers are used: if very small, very frequently called routines (where the cost of the call is an insignificant fraction of the execution cost of the function) are wrapped, the overhead cost will be along the lines shown above; if large, infrequently called routines are wrapped, the overhead cost will be proportionally smaller.

4.3.2 Cost of the Rule Interpreter

The cost of rule interpretation will depend on the number of predicates in the rule, and the degree of backtracking that the rule interpreter must undertake to satisfy the predicates. The data in Table 4.2 show the CPU costs (msec per 100 interpretations on a Sun SPARC5) for a number of different schema applied to run-time stacks of varying depth, both succeeding and failing.

Schema	Stack Depth	Success	CPU
SBV	5	N	40
SBV	10	N	50
SBV	15	N	50
SBV	5	Y	60
SBV	10	Y	60
SBV	15	Y	50
GA	5	N	200
GA	10	N	200
GA	15	N	190
GA	5	Y	320
GA	10	Y	290
GA	15	Y	280
BSM	5	N	140
BSM	10	N	120
BSM	15	N	150
BSM	5	Y	370
BSM	10	Y	350
BSM	15	Y	340

Table 4.2: Rule Interpretation Costs

As in the case of the wrapper overhead, the overhead cost of rule interpretation will depend on how frequently rules are evaluated. Rules that are evaluated for functions that are executed thousands of times during a single execution of the evaluation function will create a correspondingly higher overhead than those that are evaluated only a few times.

4.3.3 Cost of the Socket Interface

The execution cost of a call through the socket interface is dominated by the cost of the system routines that support socket communications, and is relatively independent of the number of parameters. This cost is most appropriately measured in total wall-clock time per call, which includes user, system, and overhead considerations.

Each call through the socket (using Unix family sockets, which are implemented through named FIFOs) takes approximately 6.71 msec. This is slower than the call/return interface by a factor of approximately 1000.

As a practical example, the use of the socket interface around all of the functions in the airframe model increased the execution time from approximately .5 seconds per evaluation to 18 seconds per evaluation, a slowdown factor of 36.

Chapter 5
Related Work

5.1 AI Systems for Optimizing Design

5.1.1 ENGINEOUS and Inter-GEN

The ENGINEOUS system [Tong, 1988] is a system for assisting engineers in real-world design optimization problems, developed for General Electric. A significant aspect of its functionality is the ability to utilize external code as part of an evaluation model. The Inter-GEN component of ENGINEOUS [Powell, 1990] is specifically concerned with the problems of managing optimization on evaluation functions that have undesirable characteristics, such as multi-modality and non-smoothness. The approach taken is limited by an acknowledged inability to modify the legacy simulation codes:

> "Since the Inter-GEN approach is implemented in a GE-developed software system called Engineous, which is designed to run simulation codes without requiring that they be modified in any way, Inter-GEN could be used on turbine codes without requiring that they be re-certified. This Engineous feature is a major departure from software systems such as ADS, which usually require that simulation codes be modified and recompiled for integration into an optimization program."

Because of this self-imposed restriction, ENGINEOUS is constrained to handling failures at the level of the optimizer. The Inter-GEN component adds a rule-based framework for managing numerical optimization that can run multiple sub-optimizations within a larger design process.

We have demonstrated that better optimization results can be achieved if handling can be incorporated inside the simulation programs. The LCM system can support the rule-based

approach of Inter-GEN, but extends the failure handling capability to include recovery at the level of routines inside the legacy programs.

5.1.2 DOMINIC II

The DOMINIC II system [Orelup *et al.*, 1988] is a design system that applies a strategy of using multiple optimization methods to obtain robust optimization. There is no specific discussion of DOMINIC II's failure-handling capabilities: but one infers that, if present, they are limited to external detection of failure in the evaluation codes. DOMINIC II is similar to the LCM system in that it has a number of pre-coded strategies for handling apparent optimization failures. These strategies are limited to problems occuring between the optimizer and the evaluation function only, and appear to be "hard-coded", as there is no discussion of a general methodology for development of new strategies.

We believe that the strategies described in DOMINIC II could be relatively easily implemented under the LCM system. With our system we could also apply the same strategies to sub-optimizations occuring within simulation programs.

5.1.3 DA/MSA and NDA

Our work is quite closely related to other research in automated design taking place at Rutgers University, particularly that of Tom Ellman, Andrew Gelsey, Don Smith, and Mark Schwabacher. In developing our schema, we have drawn on their experiences in developing the Nozzle Design Associate (NDA) [Gelsey and Smith, 1995], and the Rutgers Design Associate/ Modeling and Simulation Associate (DA/MSA) [Ellman *et al.*, 1992].

In the course of the development of the DA/MSA and the NDA, failures of various kinds were encountered in the simulation models. It was in these systems that the idea of using a wrapper-based approach to failure handling was first introduced. Throughout their development, hand-coded failure handlers were introduced at failure points in the simulation code.

In our work, we have generalized the structure of several of the specific failure handling strategies that were coded for NDA and MSA, and have turned them into schemata that can be automatically instantiated into a broad class of failure contexts.

The LCM system is an integral part of the Rutgers Design Associate. Other work on this system is described in [Ellman *et al.*, 1993, Ellman *et al.*, 1996].

5.2 Objects, Meta-Objects, and Reflection

By incorporating legacy code into a system developed under the CLOS object-oriented environment, some of the advantages of that environment can be applied to the legacy code. Our work can be viewed (to a degree) as making elements of a legacy system "more object-oriented" through the mechanisms of encapsulation and method subclassing. A good description of the CLOS approach to object-oriented programming can be found in [Keene, 1989].

The wrappers of the LCM system act to specialize the computation of the functions they surround, in the manner of CLOS' "`around`" methods. They follow the O-O paradigm of providing one computation for some class of inputs (in this case non-failing inputs) and providing a new computation for a specialized class of inputs (failing inputs). Wrappers also provide a degree of encapsulation by masking the effects of failure in a called routine from the caller. If legacy systems were routinely written in CLOS, the addition of the wrapper functionality in the form of an actual `around` method on the original method would be straightforward.

The hierarchical organization of the knowledge bases in the LCM system can be viewed as conforming to an object-oriented structure. Routines, failures, and handling schemata are defined in terms of the classes in which they participate, and selection of failure handling methods is driven by the classes of the failing routines and the classes of the error conditions.

The LCM system also provides a meta-level representation of the design optimization strategy and the legacy code that is accessible by schemata rules, allowing reasoning by the

wrapped strategy on its own structure, and control of its own execution. This capability of the system to reason about itself has been termed *reflection*, and has been discussed in the AI community for a long time [Maes, 1988] [Aiello and Levi, 1988]. Common Lisp and CLOS offer considerable leverage for implementation of reflective systems, through the intrinsic self-referential features of the language [des Rivières, 1988], and through the Meta-Object Protocol (MOP) [Kiczales *et al.*, 1992]. We will discuss some AI systems that take a reflective approach in the following section.

5.3 Model-based Diagnosis

The area of model-based diagnosis, though more often confined to the identification of fault causes within *physical* systems, is clearly related to our work. The idea of model-based diagnosis, first formalized by [Reiter, 1987] is that the behavior of a system under consideration can be compared with the behavior of a model of the system. When the behaviors are found to differ, the model is used to reason about the possible causes of the failure. This approach has been applied in many domains; for example, in [Dague *et al.*, 1992] a model based on order-of-magnitude reasoning is used to diagnose faults in electronic circuits. Model-based diagnosis research has historically been closely tied with the qualitative reasoning community, as in the recent work of [Subramanian, 1995], which uses QSIM-based qualitative models [Kuipers, 1990] to diagnose multiple faults in dynamic physical systems.

Model-based approaches have been taken to software fault analysis. The AUTOGNOSTIC system of [Stroulia, 1994] is a good example of a model-based reflective system. AUTOGNOSTIC models the functional tasks of a problem-solving system under its control. When a failure occurs within the problem-solving system, AUTOGNOSTIC uses the model and a trace of the behavior of the problem-solving system's actions to infer a cause, and re-configure the problem-solver. The language used for representation and reasoning about the function of elements of problem-solving systems, FPS, is discussed in [Stroulia and Goel, 1995].

A similar approach is seen in Livingstone [Williams and Nayak, 1996], a kernel for model-based reactive self-configuring autonomous systems. Livingstone has been selected by NASA as part of the flight software for the first New Millennium mission, Deep Space One.

In the UNPROG system [Hartman and Chandrasekaran, 1995], Functional Representation (FR), a theory and language for modeling the behavior of program elements, is used to construct a model of program behavior for the purpose of automated understanding of legacy code. The model thus derived from a legacy program could be used to identify failure causes and guide corrective actions. The ACTI model of [Edwards, 1995] is a related approach for representing the semantics of software systems. ACTI is particularly interesting from our perspective because of its focus on modeling a system at a routine level, rather than on a line-by-line basis.

Finally, an approach to software "work-arounds" [Liver, 1995] is philosophically similar to the LCM system idea of patching up failures and allowing computation to continue in some fashion. The work-around approach does not depend on pre-defined templates for failure recovery, as in LCM, but derives work-arounds by using functional reasoning to identify implicit redundancy in systems.

5.4 Management of Numerical Software

5.4.1 Expert Systems for Numerical Software

Several systems have been developed to assist users in selecting the appropriate numerical software from large libraries of subroutines. Systems like SAIVS [Lucks and Gladwell, 1992] ODEXPERT [Kamel, 1993] and GAMS [Boisvert, 1985] rely primarily on eliciting information from the user about the problem to be solved, and use that information with expert-system rules to select the correct routines from the knowledge base. ODEXPERT is significant in that it does not rely exclusively on user-provided information. It has additional capabilities to automatically test (by numerical experimentation) the input problem to determine some important characteristics, and this requires utilizing some limited exception

handling.

These approaches are all aimed at aiding a user in selecting an algorithm, and in some cases, some of the parameters required. They do not manage the execution of the numerical software, nor do they provide any capability (beyond that which might already exist in a numerical routine) for handling failure during the execution.

The KAM system developed by Yip [Yip, 1990] and the POINCARE system of Sacks [Sacks, 1990] both used numerical experimentation under the control of an expert system to derive information about the global behavior of systems described by differential equations. Neither system addresses issues of failure during simulation.

5.5 Numerical Methods for Optimization

Numerical optimization is an area that has received a great deal of attention. A good introduction to numerical optimization can be found in [Beightler *et al.*, 1979], and in a more application-oriented approach in [Gill *et al.*, 1981]. A practical discussion of applying numerical optimization to problems of engineering design is given by [Vanderplaats, 1984], and [Moré and Wright, 1993] surveys commercially available optimization and root-finding software.

Throughout this literature is the assumption that the functions to be optimized do not fail. There is nowhere a treatment of failure handling, though Moré does discuss ways in which simulation models being used for optimization can be modified to make successful optimization more likely. His analysis of the causes of non-smoothness in simulation models parallels ours at several points. The approach suggested is one that has serious drawbacks in real-world design problems: to sacrifice the accuracy of the simulation in favor of smoothness in the evaluation function. As we have discussed earlier in this work, an incautious application of this technique will yield successful optimizations with meaningless results, though the use of smoother approximations during some stages of the search may be a good strategy.

5.6 Software Systems Architectures

5.6.1 The I $\hat{3}$ Architecture

The ARPA Intelligent Integration of Information (I $\hat{3}$) Reference Architecture [Hull and King, 1995] addresses issues in the use of legacy information sources in new contexts. They propose a broad 5-level architecture, with five primary families of services: Coordination, Management, Semantic Integration and Transformation, Functional Extensions, and Wrapping.

We believe that our work can be cast within the framework of this architecture as an example of Wrapping Services, involving mediation:

> "**7) WRAPPING SERVICES**
> These services are used to make information sources comply with an internal or external standard. This standard may involve the interface to the information source or the behavior of the information source. Some wrappers simply transform the output or interface of an information source or software component. Other wrappers modify the meaning or behavior of information sources or software components; this may involve creating new internal interfaces, or even exposing internal interfaces to services using the wrapped information source/component."

The I $\hat{3}$ architecture is principally useful as a framework for discussion of issues in software interoperability. It is not (at present) an implemented system. While we did not implement the LCM system as a part of the architecture, we are pleased to find that we can explain it after the fact in those terms.

5.6.2 Law-Governed Systems

Naftaly Minsky has originated the concept of "Law-governed Systems" [Minsky, 1991, Minsky, 1995, Minsky, 1996], where global rules can constrain the interactions between components of a system. There are many philosophical similarities in in our ideas, particularly the availability and use of non-local information in controlling module interactions, and imposing controls by means of an external execution framework. Like a law-governed system, the LCM system has declarative rules that are applied at different levels of the system.

We share a "system-level" view of software interaction that differs from a more traditional "pairwise-interaction" view.

One dimension in which our work differs is in the degree of global applicability of our rules. Our "laws" are somewhat more like local ordinances: we have focused on defining strategies that are specifically *not* universally applicable – they derive their strength from reasoning about context. Our work is more narrow in scope as well, being principally focused on issues of failure occuring in, and utilization of knowledge about the automated design process.

5.7 Software Engineering and Fault Tolerant Software

5.7.1 Software Engineering for Robustness

Identifying ways in which more robust software systems can be constructed has been a goal of the software engineering community since its inception. The object-oriented paradigm, with its focus on modularity and abstraction has given rise to programming languages that make it possible to write code that is arguably less failure-prone. The ideal is certainly that well-designed programs should clearly specify their pre- and post-conditions, be free of side-effects, and never return an unexpected value.[1] A classic treatment is given in the description of the CLU language in [Liskov and Guttag, 1986].

Leaving aside the question of whether programs developed using object-oriented languages are, in fact, less susceptible to failure than others, we find two problems. The first, and more trivial, is that industrial legacy simulation programs about which we are concerned tend not to be written in these languages, nor by software engineers. The second issue is that for simulation programs that incorporate search-based methods, there is no useful way to completely specify the set of inputs for which the simulation will successfully run.

Our approach violates somewhat the spirit of system modularity, insofar as we try to

[1] "What, never?" "No, never!" "What, *never?*" "Hardly ever..."

expose (and exploit) some of the internal structure of the legacy codes. The use of wrappers does minimize the potential for undesired consequences, and actually depends on the *internal* modularity of the systems in which it is employed.

5.7.2 Exception Handling

The issue of exception handling, or what to do when programs behave in unexpected ways has been the subject of research in the software engineering and programming language research communities. Flaviu Cristan [Cristian, 1989] gives rigorous definitions of the basic concepts, such as robustness, exception, failure, and error. He proposes mechanisms for exception handling in hierarchical modular systems. [Yemini and Berry, 1985] give a survey of a number of proposals, in addition to their own "replacement" model. Alex Borgida proposed treating exception handling as in integral mechanism for effectively dealing with problems of database inconsistency [Borgida, 1985]. A potentially useful proposed set of extensions to scientific programming languages to handle computational exceptions is given in [Hull, 1988].

We see our work as complementary to this research effort. We agree on the principle of allowing subclassing of failure handlers to specific contexts, but disagree on the utility of providing default failure handling. The simulation systems with which we have worked challenge the definition of what constitutes an "exceptional" event: for the overwhelming majority of inputs within a bounded space of designs (99% or more) failure is returned. In this regard, we take Borgida's view that exception handling must be a normal part of our system's operation.

Our approach is unique in its "meta-system" viewpoint: information and control can pass freely across multiple levels of the system in which the failure occurs. This feature enables the context sensitivity of our recovery strategies, which we have shown to be of benefit.

Our approach is also novel in allowing "generic" failure handling to be specialized by automatic instantiation into a system. The SESAME programming language developed by Cristian [Cristian, 1982] allows for the specification of "default handlers" based on the

principle of automatically recovering to an earlier system state when a failure occurs, but this is a "one size fits all" solution. Because we have considered failure occuring in the context of a search-based process, we have gained additional flexibility in dealing with failure.

Finally, our focus on the safe introduction of failure handling into legacy systems written in languages without these mechanisms is new.

If the use of exception handling mechanisms were to become common practice in numerical software, it would reduce, to a degree, the necessity for our system. Nonetheless, extant exception handling mechanisms lack the support for reasoning about global context of our approach. An examination of any library of standard numerical tools, or examples given in journals such as ACM Transactions on Mathematical Software will reveal that exception handling is still, at best, a gratuitous element of most numerical software.

5.7.3 Fault-tolerant Software

Research has been done on developing systems that are capable of tolerating completely unpredictable faults, usually arising from hardware failure. Systems that must be used in particularly critical applications, such as automatic vehicle control, air defense systems, or life-support monitoring must be capable of either returning a correct answer, or at least recognizing and reporting unanticipated failures. Approaches to this problem all contain an elements of disparity and redundancy: multiple independent components of the software systems cross-check each other. The June, 1993 special issue of IEEE Transactions on Reliability contains a number of articles surveying and comparing techniques for developing fault-tolerant systems. The article by Hudak, *et. al.* [Hudak *et al.*, 1993] reports on the success of five different techniques in a controlled study at CMU.

While we are addressing issues of failure and robustness, we have not given any consideration to the problem of hardware failure during simulation execution. The design process is far from having the level of criticality that would cause us to be concerned. We believe that our concept of context in failure recovery might well be applicable to issues of high-reliability

systems, particularly where finding some reasonable recovery action is of paramount importance (such as when a failure is detected during the last moments of an automatically-guided aircraft landing).

5.7.4 Software Engineering for Safety

Another approach to the issue of safety in in highly critical applications is *fault-tree analysis*. This methodology, adapted from an engineering technique used for the safety analysis of electro-mechanical devices involves the identification of the immediate pre-conditions for catastrophic failure in a software (or software/hardware system), and regression of those conditions back to the system inputs that will cause them. An overview of the technique is given in [Leveson and Harvey, 1983] and [Leveson, 1991]. The software fault tree approach requires *a priori* knowledge about ways in which the software system can fail catastrophically. It is an analysis tool for improving the robustness of a system by identifying safety-related weaknesses in advance. The technique is powerful because of its goal-regression approach: only a very limited subset of the possible inputs to the system need to be considered (versus the combinatorial problems of ensuring adequate test case coverage).

Software fault-tree analysis can be difficult to accomplish for some software systems because of the difficulty of regressing failure conditions back through complex functions. One approach to address this is discussed in [Yau *et al.*, 1995], where numerical techniques are used dynamically in lieu of static analysis of code. The fault-tree approach is also incorporated into an approach to software re-use described in [Engelhardt, 1996]. These two systems have similarities to LCM in their focus on addressing the robustness of legacy software. The LCM system differs in both the level of granularity at which it treats legacy code (routines vs. lines of code), and in its "fail-first-then-fix" approach to system failure.

A chilling example of software failure in a critical system is found in a report on the failure of the Patriot missile system during the 1991 Gulf war [Office, 1992]. Fault-tree analysis could (in principle) have identified the conditions under which the failure occurred.

5.8 Automated Re-engineering and Re-use of Legacy Systems

5.8.1 AMPHION

The AMPHION system [Lowry, 1995] is a successfully implemented Automated Programming system that derives correct programs by composition of legacy library routines. It uses an elegant formal methods approach, and has a powerful graphical user interface. AMPHION utilizes a knowledge base that provides semantic information about library function syntax and semantics.

AMPHION's dependence on correct formal descriptions of the semantics of the library components it uses to construct programs may limit it somewhat. It is unclear how AMPHION would handle failure-prone library components components like numerical optimization routines and multi-dimensional root-finders. We believe that our work is complementary to that of AMPHION in providing a framework for managing the uncertainty of certain classes of numerical routines.

5.8.2 Commercial Software Renovation

Automating the renovation and re-engineering of legacy software systems is of tremendous commercial interest at present. At the 1995 IJCAI workshop on AI and Software Engineering, several papers on work in progress were presented, including a description of AI approaches used by Lockheed Martin InVision consulting services [Filman, 1995]. The paper is interesting principally because it describes a very large scale deployment of knowledge-based techniques, primarily in the service of acquisition of information about legacy systems. Except for the most routine tasks, the re-engineering process is still largely manual.

5.9 Wrapping Mathematical Software

5.9.1 VEHICLES

The VEHICLES knowledge-based environment for the conceptual design of spacecraft is a tool for assisting design engineers [Bellman, 1991]. It allows simulations and optimizations to be carried out as part of the design process, using numerical tools. The VEHICLES system utilized wrappers around legacy mathematical programs [Landauer, 1993] [Miller and Quilici, 1992]. The primary concern addressed by code wrappers in this system was interfacing with the legacy routines, and making it easier for the users to employ existing modeling programs for new purposes.

A further, more general discussion of the role of code wrapping in systems for design can be found in [Bellman and Landauer, 1995]. She defines a *wrapping*:

> "A wrapping is an expert interface that describes the uses of a resource in complex software system [sic]. It contains an explicit, machine-processable description of the resource used for management of the system architecture, both short term (run time) and long term" (configuration).

We believe that the emphasis of our approach to wrappers is different in two significant ways: First, that we are concerned with wrapping at a *system* level, rather than just a component level. Our wrappers communicate and transfer control across levels, and alter the behavior of the system as a whole. Second, that wrappers as defined and used in the cited works do not not address the issue of failure in the wrapped code and how it should be handled.

Chapter 6
Conclusions

6.1 Evaluation of the LCM System

6.1.1 Discussion of Claims and Results

We will now review the claims made in the first chapter, and discuss each in light of the results present.

Improvement to Optimization Performance

We claimed that the incorporation of failure recovery into design optimization strategies significantly improves optimization end results.

The results show that in the absence of any failure handling, optimization effectively degenerates to a process of random probing in the search space, as failure prevents the optimizer from making any progress.

Stochastic optimization techniques such as simulated annealing or genetic algorithms may provide some ability to eventually reach an optimum, as they are known to be less affected by discontinuities in the evaluation function. Some limited form of failure handling would still have to be incorporated, to prevent failure points from causing abnormal termination of the search process. At a minimum, this would include something like the Simple Bad Value schema, as well as trapping system exits and providing a runaway task timeout. The principal drawback of stochastic techniques is that they are well-known to be expensive in number of evaluations required.

The results also show that in all the problem domains tested, incorporation of some

or all of the recovery schemata tested yielded improved results, either in absolute terms (optimization reaching a valid point not otherwise reached), or as a function of the cost of the optimization. Conversely, it was seen that not all failure handling schemata were effective on the test problems, either individually or in combination.

Effectiveness of Multi-Level Failure Handling

We claimed that our multi-level methodology permits failure handling strategies to be defined that are superior to strategies that can only operate outside of the legacy code.

In all of the problem domains tested, inclusion of the Table Extrapolation schema in combination with one or more other schemata gave the best results. The **TE** schema must be incorporated inside simulation code, at the point(s) where table interpolation is being performed. The success of this schema appears to come from the nature of the interaction between the search based methods and the functions in which the tables are being used: allowing the table to be extrapolated in a reasonable way (as opposed to the behavior of a spline interpolator outside a table) does, in most cases, permit the search based method to reach a legitimate solution back inside the table. The alternative of terminating the simulation on this error results in the loss of information that could have furthered the search.

We also showed that, as the sole failure handler, a schema that operates inside a strategy may not be sufficient. We believe this is because of the *specificity* of the schema to a particular problem. As it only handles a small class of problems (though problems of that class may occur frequently), the occurrence of other failures will cause the optimization to terminate. These results suggest that the best overall strategy will be to combine very specific schemata such as **TE** with more general schemata such as **SBV** in order to ensure the most effective failure handling when possible, but some recovery for all errors encountered.

Context Sensitivity of Failure Handling

We claimed that our methodology allows the incorporation of failure handling strategies into legacy programs that are sensitive to failure context. We further claimed that such context-sensitive failure handling strategies are superior for some failure instances than failure handling strategies that do not.

We demonstrated by examples (Gradient Method Change, Gradient Approximation, Simple Interpolation, Bound Search Method) how several of our schemata are sensitive to failure context.

Context sensitivity was definitely not seen to be universally beneficial. None of the context-sensitive schemata were shown to be superior to all other schemata when tested individually. In combination, inclusion of the Gradient Method Change schema was seen to yield significant improvement to the **SBV+RMS** combination, but have insignificant effect on the **SBV+TE** combination in the Yacht design domain; in the Nozzle and Aircraft design domains, it improved single-run performance, but actually degraded cost-weighted optimization performance.

We conclude that there are appropriate applications for this technique, but that it must be used with caution. In particular, it appears that inclusion of the context-sensitive schema allowed *unprofitable* optimizations to continue past the point where they would otherwise have terminated, resulting in a greater relative cost than the additional improvement warranted.

Composability of Failure Handling Strategies

We claimed that one of the strengths of our approach is that strategies can easily be combined to increase failure handling power in the legacy program.

We demonstrated through experiments in each of our four domains that optimization quality uniformly improved with multiple schemata over any individual schema. We also showed that improvement is not monotonic in the number of schemata incorporated: some

combinations can improve the single-run performance (as measured by percentage completed, and the average and standard deviation of the results), but actually do worse when the cost per run is considered.

Safety

We claimed that the use of these wrappers allows failure handling to be safely incorporated into legacy programs, eliminating the need to re-validate those programs.

We showed that our wrapper-based approach only affects the computation of a simulation program when a failure occurs. We proved that we can guarantee that a wrapped code will therefore return the same result as the unwrapped code if no failure occurs during the computation.

We defined a new condition called "search-validity," and showed that the results of a computation incorporating a search-based method are search-valid if any failure that occurs during the computation occurs only during the non-terminal portion of the search. A result that is search-valid can be trusted to be consistent with the semantics of the unwrapped code, even though failure occurred during the computation.

These guarantees allow the results of optimizations carried out with validated legacy codes that have been wrapped to be trusted without re-validation.

We also presented an additional condition for which failure recovery in the computation of an inequality constraint for a constrained optimization method could be shown to preserve search-validity, even when the failure occurs on the terminal point of the search. We did not specify an operational test for determining the necessary condition, however.

Ease of Failure Strategy Development

We claimed that the methodology and tools we have developed allow for easier and more rapid development of new failure handling strategies, as compared to use of more conventional programming languages.

We did not support this claim with a carefully controlled study, but rather summarized

our own experience in these domains, and gave some selected comparisons between schemata and failure handling coded directly in other languages. Such a study would require taking groups of experienced programmers and providing them with sufficient training and practice that they could be considered proficient in our methodology. We could then take a legacy system that we have not previously seen (preferably provided by an industry collaborator) and code an optimization strategy without additional failure handling around it. We would divide the programmers into groups and allow each to incorporate whatever failure handling they deem necessary to produce a result that is globally optimal and valid. We could then assess the results of their efforts according to the time required to incorporate the failure handling, the quality of the end results of the optimizations, and the computational effort expended to achieve those results.

Degree of Automation

We claimed that the LCM system automatically generates wrapper code for legacy programs from information contained in a database of function information. We further claimed that generic failure handling strategies defined in the system database of strategies, once selected by the user, are automatically instantiated into these wrappers.

We described our databases, and gave examples of the information stored in them. We gave examples of the process by which generic schemata are instantiated into legacy systems. We described (although not in great detail) the operation of these code wrappers, and how they are generated from database entries.

We have automated much of the mechanical effort required to incorporate failure handling into optimization strategies. Key decisions that are being made by the LCM system include:

- What interface code is necessary for wrapping legacy functions.
- How generic schemata get instantiated into specific wrappers.
- What the failure context is when failure occurs.
- What recovery action to take for the failure context.

- Whether the end result of optimization meets validity criteria.

We have not automated some key aspects of the system. We are dependent on the user to provide information about the semantics of functions that are not already in the functional semantics knowledge base. The user must also provide syntactic information necessary to access the parameters and return values of those functions.

The decision of which of several applicable failure handling schemata to incorporate into an optimization strategy is not automated, but is supported by the LCM system. While the user must identify the schema to instantiate and the level at which to instantiate it (where multiple choices exist), the LCM system provides information about the behavior of the functions involved in the failure, the failure context, and the candidate schemata. It is evident from our experiments that this decision can be a crucial one to the overall performance of the optimization strategy. We believe that it might be possible for the LCM system to provide additional guidance to the user in, perhaps based on information gathered over a history of using the schemata in different contexts.

Generality of Approach

We claimed that our methodology and its implementation are general across a class of design optimization systems. We also claimed that it is effective for a range of failure classes and contexts.

We defined a structure for automated design strategies constructed from domain-independent numerical tools and domain-specific simulation programs. We believe that this structure is sufficiently generic to include many problems of vehicle design where the evalauation criteria for the design depend on the integration over time of forces arising from the vehicle's motion. We showed that this to be the case for two substantially different problem domains: aircraft and yachts.

We note here that our approach places few constraints on the programs to which it is applied. The principal restriction is that the failure recovery take place in the context of

a search-based process. While a design optimization strategy meets this criterion, other search-based algorithms exist.

We have demonstrated that our generic schemata can be instantiated in multiple ways even within a single optimization strategy. We have shown that the schemata we have developed are capable of handling a variety of failures in all of the test design domains.

We have tested our schemata in different problem domains, and have seen that no one schema or combination is superior in all of them. It remains to be seen whether new schemata would continue to need to be developed as the LCM system is applied to new problem domains, or if the schemata already developed form a sufficient basis to adequately handle any failure. We speculate that it will probably be the case that the existing schemata will be useful in new problem domains, but that better results will be attained if at least some more problem-specific schemata are developed.

Novelty

Finally, we claimed that our approach is novel in a number of ways. We believe that our use of globally-communicating function wrappers is unique, as is our language for defining context sensitive failure handling strategies. We claim also to have advanced the integration of AI techniques and traditional approaches to numerical optimization.

We gave examples of research in a variety of related areas, and differentiated our work from theirs. We showed how our work could, in some cases, incorporate or complement other work in the areas of code re-use and design optimization.

6.1.2 Limitations

The applicability of our wrapper-based approach is limited by the structure of the original analysis programs to which it is to be applied. Highly structured systems with a large number of small discrete subroutines provide more opportunity for useful scheme incorporation than systems with a few highly monolithic modules. The greatest leverage comes when analysis

programs use well-known standardized numerical routines for which schema already exist.

The LCM system is also limited by having no way of detecting data passed between subroutines through global variables, such as Fortran commons or C `externs`. Routines that create or depend on side-effects can lead to unexpected failure when an scheme alters the normal call-return flow of control. It is incumbent on the model developer to understand what portions of the analysis model are not re-entrant.

Although the availability of source code for analysis programs is not an absolute necessity for scheme inclusion, semantic information about function parameters and return values *is* required. In practice, it is probably desirable to have access to the source, as semantic information cannot be obtained solely from function headers.

We have tested the LCM system exclusively on design optimization strategies developed locally. A more rigorous test of our approach would require taking a real industry legacy simulation code, and incorporating failure handling into an optimization strategy built around it.

6.1.3 Contributions

We believe our research makes several contributions:

- **Ideas**

  We have developed the idea of managing failure at a system level. In particular, we have introduced the concept of failure *context* in a way that takes into account more than the local circumstances that precipitated the failure. We have defined context to include all of the current computational state, the recent history of the computation, and state information collected over the course of repeated computation.

  We have extended the idea of code wrappers from code that works strictly between pairs of software modules to code that operates amongst a collection of inter-operating software modules. We have also developed the idea that failure handling can take place exclusively in the code wrappers.

We have shown when failure occurs in the context of a search-based process, there are unique opportunities for failure recovery that do not compromise the validity of the end results of the process.

- Methodologies

  We have developed a language and an architecture for reasoning about failure and failure context during design optimization.

  We have developed a methodology for incorporating failure handling safely into legacy programs that avoids the need for re-validation of the codes.

- Tools

  We have implemented a tool for the more rapid construction of useful design optimization systems around existing legacy code in a way that preserves the integrity of the legacy code.

  We have populated a knowledge base about function syntax and semantics with information about a number of standard numerical tools, including the CFSQP optimizer, and routines from the Numerical Recipes in C library. This knowledge base can be used in implementing failure handling into future design optimization strategies that use these tools.

  We have compiled a database of generic failure handling schemata that can be used as a basis for incorporating failure handling into new optimization strategies.

6.1.4 Directions for Future Research

The degree of automation in the LCM system could be extended from its present state. In particular, the selection of a good failure handling strategy and where to instantiate it in a design optimization strategy is a task that our system might at least provide some guidance towards appropriate choices. To do this would require developing a better understanding of

the relationship between the structure of failure handling schemata and their effect on the optimization process.

We feel we have only scratched the surface of information that can be extracted about the internal behavior of legacy systems. With the run-time environment that we have already developed, it should be possible to carry out extensive numerical experiments on legacy codes, with the intent of developing a "qualitative" model of their behavior that could be used to guide optimization and improve failure handling.

We have also exploited only a small amount of information about the purpose for which computations are being carried out. With a richer knowledge base about function behavior (particularly in the area of the behavior of numerical tools), more sophisticated failure handling might be achievable. Ultimately, with an adequate understanding of computational intent, whole functions within legacy systems could be supersede by new functions that provide better information to the numerical tools.

Development of a more user-friendly interface to the LCM system, while not strictly a research issue, would make it more accessible, and increase the likelihood that it might ultimately be useful in a real industrial design context.

A side effect of our wrapper approach to legacy code is the ability to monitor the interactions between elements of the optimization strategy, and to collect information about the behavior of routines. We are planning to explore this further by developing ways of visualizing the behaviors (and particularly the failures) of legacy programs. We can use the wrapper control to selective explore portions of the legacy system without concern that their behavior is being interfered with. Ultimately, we would like to be able to find a way of visually relating the behavior of the legacy system to the parts of the system responsible for the behavior, as part of a tool to aid the development of new design optimization systems.

References

[Aiello and Levi, 1988] L. Aiello and G. Levi. The uses of metaknowledge in AI systems. In P. Maes and D. Nardi, editors, *Meta-level Architectures and Reflection*. Elsevier Science Publishers B. V., 1988.

[Beightler *et al.*, 1979] Charles S. Beightler, Don T. Phillips, and Douglass J. Wilde. *Foundations of Optimization – second edition*. Prentice Hall, 1979.

[Bellman and Landauer, 1995] Kirstie L. Bellman and Christopher Landauer. Designing testable, hetergeneous software environments. *Journal of Systems Software*, 29:199 – 217, 1995.

[Bellman, 1991] Kirstie L. Bellman. An approach to integrating and creating flexible software environments supporting the design of complex systems. In *Proceedings of WSC '91: The 1991 Winter Simulation Conference*, pages 1101–1105. SCS, 1991.

[Boisvert, 1985] et al. Boisvert. GAMS: A framework for the management of scientific software. *ACM Transactions on Mathematical Software*, pages 313 – 355, December 1985.

[Borgida, 1985] Alexander Borgida. Language features for flexible handling of exceptions in information systems. *ACM Transactions on Database Systems*, 10(4), December 1985.

[Cristian, 1982] Flaviu Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, C-31(6):531 – 540, June 1982.

[Cristian, 1989] Flaviu Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*, pages 68 – 97. Blackwell Scientific Publications, 1989.

[Dague *et al.*, 1992] P. Dague, O. Raiman, and P. Devès. Troubleshooting: When modeling is the trouble. In W. Hamscher, L. Console, and J. de Kleer, editors, *Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[des Rivières, 1988] J. des Rivières. Control-related meta-level facilities in LISP. In P. Maes and D. Nardi, editors, *Meta-level Architectures and Reflection*. Elsevier Science Publishers B. V., 1988.

[Edwards, 1995] S. Edwards. A formal model of software subsystems. Technical report, The Ohio State University Department of Computer and Information Science, 1995. Ph.D. Thesis.

[Ellman and Murata, 1996] Thomas P. Ellman and Takahiro Murata. Deductive synthesis of numerical simulation programs from networks of algebraic and ordinary differential equations. In *Proceedings of the Eleventh Knowledge-Based Software Engineering Conference.* IEEE, 1996.

[Ellman *et al.*, 1992] Thomas Ellman, John Keane, and Mark Schwabacher. The Rutgers CAP project design associate. Technical Report CAP-TR-7, Rutgers University, Department of Computer Science, New Brunswick, NJ, 1992.

[Ellman *et al.*, 1993] Thomas Ellman, John Keane, and Mark Schwabacher. Intelligent model selection for hillclimbing search in computer-aided design. In *Proceedings of the Eleventh National Conference on Artificial Intelligence.* AAAI, 1993.

[Ellman *et al.*, 1995] Thomas Ellman, John E. Keane, Takahiro Murata, and Mark Schwabacher. A transformation system for interactive reformulation of design optimization strategies. Technical Report HPCD-TR-42, Rutgers University, Department of Computer Science, New Brunswick, NJ, 1995.

[Ellman *et al.*, 1996] Thomas Ellman, John Keane, Mark Schwabacher, and Ke-Thia Yao. Multi-level modeling for engineering design optimization. Technical Report HPCD-TR-44, Rutgers University, Department of Computer Science, New Brunswick, NJ, 1996.

[Engelhardt, 1996] B. Engelhardt. Tools and methods for development, maintenance, and reuse of quality software. Technical Report UPN 323-08: Maintaining Software Safety, NASA, September 1996.

[Filman, 1995] R. Filman. Applying AI to software renovation. In *Working Notes, Third Workshop on AI and Software Engineering: Breaking the Toy Mold.* IJCAI, 1995.

[Gelsey and Smith, 1995] Andrew Gelsey and Don Smith. A computational environment for exhaust nozzle design. In *Proceedings, Computing in Aerospace 10*, San Antonio, TX, March 1995. AIAA. AIAA-95-1016.

[Gill *et al.*, 1981] Philip E. Gill, Walter Murray, and Margaret Wright. *Practical Optimization.* Academic Press, 1981.

[Hartman and Chandrasekaran, 1995] J. Hartman and B. Chandrasekaran. Functional representation and understanding of software: Technology and application. In *Proceedings 5th Annual Dual-Use Technologies and Applications Conference.* IEEE, May 1995.

[Hudak *et al.*, 1993] John Hudak, Byung-Hoon Suh, Dan Siewiorek, and Zary Segall. Evaluation & comparison of fault-tolerant software techniques. *IEEE Transactions on Reliability*, 42(2):190 – 204, June 1993.

[Hull and King, 1995]
R. Hull and R. King. REFERENCE ARCHITECTURE for the intelligent integration of information, May 1995. <URL http://isse.gmu.edu/I3_Arch/X0001_0.TitleTOC.html >.

[Hull, 1988] et al. Hull. Exception handling in scientific computing. *ACM Transactions on Mathematical Software*, pages 201 – 217, September 1988.

[Kamel, 1993] et al. Kamel. ODEXPERT an expert system to select numerical solvers for initial value ODE sytems. *ACM Transactions on Mathematical Software*, pages 44 –62, March 1993.

[Keene, 1989] S. Keene. *Object-oriented programming in Lisp*. Addison-Wesley, 1989.

[Kiczales *et al.*, 1992] G. Kiczales, J. des Rivières, and D. Bobrow. *The Art of the Meta-Object Protocol*. The MIT Press, 1992.

[Kuipers, 1990] B. Kuipers. Qualitative simulation. In Daniel S. Weld and Johan deKleer, editors, *Qualitative Reasoning about Physical Systems*, pages 470 – 475. Morgan Kaufmann, Palo Alto, CA, 1990.

[Laboratories, 1990] AT&T Bell Laboratories. f2c - A Fortran to C Compiler, 1990. Program.

[Landauer, 1993] Christopher Landauer. Wrapping mathematical tools. In *Proceedings of the 1990 SCS Eastern MultiConference*, pages 261 – 266. SCS, 1993.

[Lawrence *et al.*, 1994] C.T. Lawrence, Zhou J.L., and Tits A.L. CFSQP Version 2.1 (Released November 1994); Copyright (c) 1993 — 1994, all rights reserved, 1994. Program.

[Leveson and Harvey, 1983] N. Leveson and P. R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, SE-9(5), September 1983.

[Leveson, 1991] Nancy Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2):35 – 46, February 1991.

[Liskov and Guttag, 1986] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[Liver, 1995] B. Liver. Repair of communication systems by working around failures. In *Proceedings of Applied Artificial Intelligence 9*, pages 81–99, 1995.

[Lowry, 1995] M. Lowry. Automating software reuse. In *Working Notes, Third Workshop on AI and Software Engineering: Breaking the Toy Mold*. IJCAI, 1995.

[Lucks and Gladwell, 1992] M. Lucks and I. Gladwell. Automated selection of mathematical software. *ACM Transactions on Mathematical Software*, pages 11 – 34, March 1992.

[Maes, 1988] P. Maes. Issues in computational reflection. In P. Maes and D. Nardi, editors, *Meta-level Architectures and Reflection*. Elsevier Science Publishers B. V., 1988.

[ME, 1992] AeroHydro Inc. Southwest Harbor ME. AHVPP1 - SAILING YACHT PERFORMANCE PREDICTION PROGRAM, 1992. Program.

[Miller and Quilici, 1992] L. Miller and A. Quilici. A knowledge-based approach to encouraging reuse of simulation and modeling programs. In *Proceedings, 4th International Conference on Software Engineering and Knowledge Engineering*. IEEE, 1992.

[Minsky, 1991] N. Minsky. Law-governed systems. *Software Engineering*, pages 285 – 302, March 1991.

[Minsky, 1995] Naftaly Minsky. Law-governed regularities in object systems; part 1: Principles. In *Theory and Practice of Object Systems*. 1995. To be published.

[Minsky, 1996] Naftaly Minsky. Independent on-line monitoring of evolving systems. In *Proceedings of The 18th International Conference on Software Engineering (IWWW.CSE)*, 1996.

[Moré and Wright, 1993] Jorge J. Moré and Stephen J. Wright. *Optimization Software Guide*. SIAM, 1993.

[Office, 1992] U. S. General Accouting Office. Patriot missile defense: Software problem led to system failure at Dharan, Saudi Arabia. Technical Report GAO/IMTEC-92-26, U. S. General Accounting Office, P.O. Box 6015, Gaithersburg, MD 20887, February 1992.

[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.

[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.

[Press *et al.*, 1986] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes in C, 2d ed.* Cambridge University Press, New York, NY, 1986.

[Reiter, 1987] R. Reiter. A theory of diagnosis from first principles. In W. Hamscher, L. Console, and J. de Kleer, editors, *Model-Based Diagnosis*. Morgan Kaufmann, 1987.

[Sacks, 1990] Elisha Sacks. A dynamic systems perspective on qualitative simulation. *Artificial Intelligence*, 42(2-3):159–188, March 1990.

[Schwabacher, 1996] Mark Schwabacher. The use of artificial intelligence to improve the numerical optimization of complex engineering designs. Technical report, Rutgers University, Department of Computer Science, New Brunswick, NJ, 1996. PhD Thesis.

[Steele, 1990] Guy L. Steele. *Common Lisp, the Language – Second Edition*. Digital Press, 1990.

[Stroulia and Goel, 1995] E. Stroulia and A. Goel. Functional representation and reasoning for reflective systems. In *Proceedings of Applied Artificial Intelligence 9*, pages 101–125, 1995.

[Stroulia, 1994] E. Stroulia. Failure-driven learning as model-based self-redesign. Technical report, Georgia Institute of Technology Department of Computer Science, December 1994. Ph.D. Thesis.

[Subramanian, 1995] S. Subramanian. Qualitative multiple-fault diagnosis of continuous dynamic systems using behavioral modes. Technical report, The University of Texas at Austin Department of Computer Science, December 1995. Ph.D. Thesis.

[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.

[Vanderplaats, 1984] Garret N. Vanderplaats. *Numerical Optimization Techniques for Engineering Design: With Applications*. McGraw-Hill, 1984.

[Williams and Nayak, 1996] B. Williams and P. Nayak. A model-based approach to self-configuring systems. In *Proceedings of AAAI-96*. AAAI, August 1996.

[Yau *et al.*, 1995] M. Yau, S. Guarro, and G. Apostolakis. Demonstration of the dynamic flowgraph methodology using the Titan II space launch vehicle digitial flight control system. In *Reliability Engineering and System Safety*. Elsevier Science Ltd., 1995.

[Yemini and Berry, 1985] Shaula Yemini and Daniel Berry. A modular verifiable exception-handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2), April 1985.

[Yip, 1990] Kenneth Yip. Generating global behaviors using deep knowledge of local dynamics. In Daniel S. Weld and Johan deKleer, editors, *Qualitative Reasoning about Physical Systems*, pages 470 – 475. Morgan Kaufmann, Palo Alto, CA, 1990.

Appendix A
Schemata Descriptions

In this appendix we will give detailed descriptions of several of the schemata that have been implemented and tested with our system. We present the failure context for each schema, a description of the recovery action, and a discussion of the rationale behind it.

The rule syntax shown in these examples is essentially as implemented. Some minor syntactic changes have been made to improve readability, and remove some "bookkeeping" rules (generally rules for which the condition is `(And T)` and the action is `NULL`, which are executed only for their side-effects). The reader should not be unduly mislead by the apparent simplicity of many of the rules: the power of the built-in predicates and actions, and the composability of the rules allows for complex failure recovery strategies to be described in very terse form.

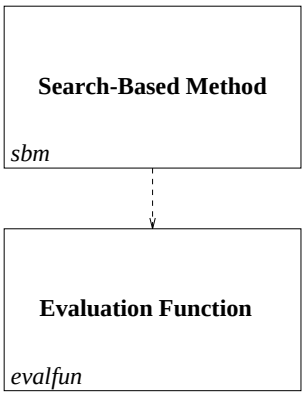
A.1 Simple Schemata

The simplest of all of the schema, the Simple Bad Value **SBV** schema can be instantiated for any detectable error occuring during the execution of an evaluation function for a search-based method.

When a failure occurs, the user-provided value *?errorval* is returned as the value of the evaluation function. It is presumed that this value will be sufficiently "bad" to ensure that it is worse than any valid evaluation.

The Simple Interpolate (**SI**) schema is a specialization of the **SBV** schema. It is applicable under the same failure context as **SBV**.

Structural Template:



Rules:

```
(Rule 1
        :condition (Condition ANY)
        :action    (Return :From ?evalfun
                           :Value ?errorval)
        :symbols (?errorval))
```

Figure A.1: Schema: Simple Bad Value (**SBV**)

Derived From:

Simple Bad Value (SBV)

Changes:

```
(Rule 1
        :condition (And (Condition ANY)
                        (Formals ?formals))
        :action (Return :From ?evalfun
                        :Value (CacheInterpolation ?evalfun ?formals))
        :symbols () )
```

Figure A.2: Schema: Simple Interpolate (**SI**)

When a failure occurs, instead of returning a fixed error value, a value is returned that is the linear interpolation of nearby good points from the cache. The built-in function `CacheInterpolation` sorts the cache for the evaluation function by distance from the failing point, and uses a singular-value decomposition (SVD) technique to provide its best linear approximation of the interpolation of a simplex formed by the nearest (dimension+1) points. SVD is used because it will return a reasonable approximation even when the points are collinear, as is often the case during optimization.

We would expect **SI** to be useful where there is no obvious fixed "bad" value for an evaluation function, or where a large "bad" value might cause the search-based method to move away from the global optimum. A risk of **SI** is that it might result in a value that would cause the search-based method to terminate at the failing point.

A.2 Gradient Context Schemata

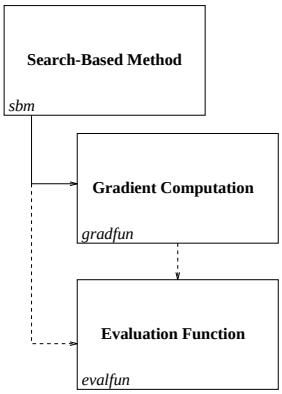
These related schemata are all intended to handle failures occurring in the context of a search-based method that computes a numerical gradient. They depend on knowing when a gradient computation is in progress by the presence of a recognized gradient function on the run-time stack.

The Gradient Method Change (**GMC**) attempts to handle a failure occuring during gradient computation by changing the method used to compute the gradient.

When failure occurs, the run-time stack is checked to determine if the gradient function for which this schema has been instantiated is currently on the stack (the variable *?gradfun* will be automatically bound to the *gradfun* tag in the structural template at schema instantiation). If it is, a gradient computation is assumed to be in progress, and the gradient function (which is assumed by default to use a 3-point method) is re-started with the :N-POINTS symbolic parameter set to 2.

We would expect that this will be most effective where optimization is taking place near the border of an unevaluable region, or where failure points are distributed throughout a

Structural Template:



Rules:

```
(Rule 1
        :condition (And (Condition ANY)
                        (CurrentFrame ?x)
                        (Ancestor ?ancestor ?x)
                        (ModuleName ?ancestor ?name)
                        (Equal ?name ?gradfun))
        :action    (Restart
                        :At ?gradfun
                        :Override ((:N-POINTS . 2))))))
```

Figure A.3: Schema: Gradient Method Change (**GMC**)

region. We expect that some accuracy in approximating the gradient will be lost, which might affect the behavior of the search-based method.

Minor variations of **GMC** re-starts the gradient function with the symbolic parameter :METHOD set to either FORWARD-DIFFERENCE or BACKWARD-DIFFERENCE. These schemata can be combined, as needed.

Derived From:
Gradient Method Change (GMC)

Changes:

```
(Rule 1
        :action (Return :From ?gradfun
                        :Value (CacheInterpolation ?gradfun ?formals)))
```

Figure A.4: Schema: Gradient Approximation (**GA**)

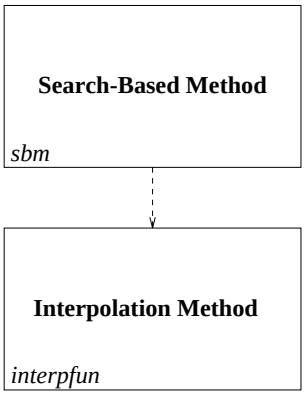
The Gradient Approximation (**GA**) schema is derived from **GMC**, and is applicable in the same failure context. It is also a specialization of the **SI** schema, for a more restrictive structural context. This schema addresses the problem of using either SBV alone, which will tend to distort gradient computation, or SI alone, which carries a risk of allowing a search-based method to terminate for a failing point. This schema could reasonably be used in conjunction with SBV: because it is more specific than SI, which is derived from SBV, it will be applied first in contexts in which either could apply.

A.3 Interpolation Context Schemata

The Table Extrapolation (**TE**) schema attempts to handle failures that occur when an interpolation is required outside the bounds of a table by providing an n-dimensional linear extrapolation from the table.

When a failure occurs, either because the interpolation method signaled an error, or because the `Bounds-Check` sanity predicate detected that the interpolation point was outside

Structural Template:



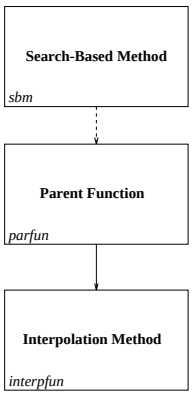
Rules:

```
(Rule 1
        :condition (And (CurrentFrame ?x)
                        (Formals ?x ?formals)
                        (ParameterValue ?x ?formals :INDICES ?ind)
                        (ParameterValue ?x ?formals :POINT ?pt)
                        (Or (Condition ANY)
                            (Sanity #'Bounds-Check ?ind ?pt)))
        :action (Return :From ?interpfun
                        :Value (LinearExtrapolate ?interpfun ?formals)))
```

Figure A.5: Schema: Table Extrapolation (**TE**)

the indices passed to the interpolation method, the interpolation function is returned from with a value computed by the `LinearExtrapolation` built-in function. The use of this schema depends on the indices and the tabular data being explicitly represented in the arguments to the interpolation method – normally the case for standard library interpolation routines.

Structural Template:



Rules:

```
(Rule 1
        :condition (And (CurrentFrame ?x)
                        (Parent ?x ?parent)
                        (Equal ?parent ?parfun)
                        (ParameterValue ?x ?formals :POINT ?pt)
                        (Or (Condition ANY)
                            (Sanity #'Bounds-Check ?ind ?pt)))
        :action (Return :From ?interpfun
                        :Value (CacheInterpolation ?interpfun ?formals))
        :symbols (?ind) )
```

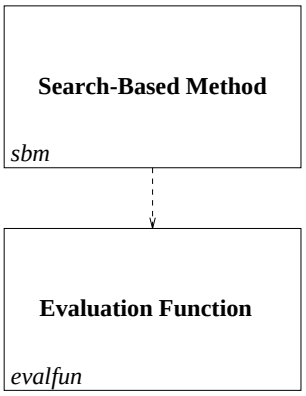
Figure A.6: Schema: Table Extrapolation (Implicit) (**TEI**)

Sometimes functions contain embedded interpolation tables. If the table and indices are not explicitly represented, **TE** cannot be applied, however, by specializing the structural context, we can use the built-in function `CacheInterpolation` to perform approximately the same recovery. In this case, the indices to be used in the sanity check must be provided at schema instantiation time. It is also necessary that sufficient successful evaluations have been cached for this function that the data stored in the cache gives a good approximation

of the contents of the edges of the table.

A.4 Alternative Modeling Parameter Schemata

Structural Template:



Rules:

```
(Rule 1
        :condition (Or (Condition ANY)
                       (Condition NONE))
        :action    (Restart
                            :At ?sbm
                            :Override ((?symparm . ?vallist))
                            :Cycle ?ntimes
                            :Best ?ordpred)
        :symbols (?symparm ?vallist ?ordpred ?ntimes))
```

Figure A.7: Schema: Backtracking Parameter Restart (**BPR**)

The Backtracking Parameter Restart (**BPR**) schema is the most general of a large family of schemata. It is applicable whenever the value computed by a search-based method method is dependent on so-called *modeling parameters:* parameters of more-or-less arbitrary value that control the manner in which the method will perform its search. The **BPR** schema allows the method to be re-started multiple times with the value of some symbolic parameter of the function specified by *?symparm* to be taken from the elements of a list of values *?vallist*. The `Or` of `(Condition ANY)` and `(Condition NONE)` ensures that the rule will be applied *after* each evaluation of the search-based method, regardless of whether or not an error condition occurs.

The value returned is the first element of the list of collected return values, as ordered by the predicate provided by *?ordpred*, after failing evaluations have been removed. The *?ntimes* argument provides an additional level of control over the iteration: if specified as `NIL`, the search-based method will be restarted once for each element in *?vallist*; if specified as a number, the entire set will be applied in sequence that many times; if specified as `T`, the set will be applied in sequence, and at the end of the 2, 3, ... cycle, the best result of the current cycle will be compared with the best result of the prior cycle (as determined by *?ordpred*), and if improvement has occurred, the cycle will repeat.

Derived From:
Backtracking Parameter Restart (BPR)

Changes:

```
(Rule 1
        :condition (Or (Condition ANY)
                       (Condition NONE))
        :action    (Restart
                             :At ?sbm
                             :Override ((:SEED . (RandomSeed ?bounds)))
                             :Cycle ?ntimes
                             :Best ?ordpred)
        :symbols (?bounds ?ordpred ?ntimes))
```

Figure A.8: Schema: Random Multi-Start (**RMS**)

The Random Multi-Start (**RMS**) schema is a specialization of **BPR** that is commonly used in optimization. It is applicable whenever the success of a search-based method will depend on the selection of a starting point (the *seed value*), and a good seed value cannot be determined in advance. The schema re-starts the search each time it completes or fails with a new seed generated within a bounding volume specified by the user, for some specified number of tries, and returns the best result found over all tries. The value of *?ntimes* is typically in the range of 10-100, and *?ordpred* is `#'<` for minimization.

We would expect this method to be of use whenever the search-based method is known

to fail for some portions of the space, but where there is no good principled method for determining those portions of the space in advance. It trades off a more efficiency for completeness by enforcing a more extensive search. In the extreme, we can transform any deterministic search-based into a stochastic search algorithm by applying a sufficiently large value of *?ntimes*.

A.5 Method Substitution Schema

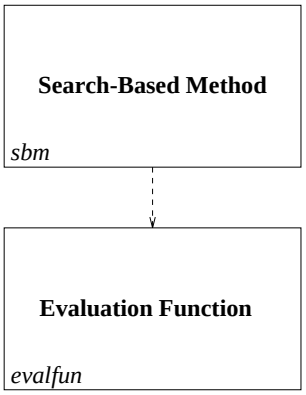
The schema in Figure A.9 begins to demonstrate some of the complexity that is possible with interacting rules. This schema is intended to override the original optimization method in an optimization strategy, and replace it with a set of alternative optimization methods. These are each applied, and the best result is retained. If the results improve by more than *?epsilon*, the process is repeated. Notice that the global state information is retained across multiple restarts of the search-based method.

These rules differ in some regards from the previous examples. Notice that `Rule 1` has no action, and is evaluated only for its side-effects on the global state. Its condition of `T` ensures that it will succeed immediately upon attempted execution of *?sbm*. `Rule 2` will always be checked immediately after, and `Rule 3` if `Rule 2` fails. Since either a restart or a return is executed, the original *?sbm* is never executed. When the restart is executed, the effect will be to re-enter the process at `Rule 1`.

We expect this strategy to do well when the selection of optimization methods compliments each other. Where one method might get "stuck" at a local optimum, another might be able to progress. It may well be expensive in evaluations, however.

The version of **MMS** shown here is somewhat simpler than the version currently implemented. That version has one significant difference: it applies each optimization method sequentially to the result of the last optimization, so as not to discard any progress made. Two additional rules (and a more complex Lisp loop) are required in the more complex version.

Structural Template:



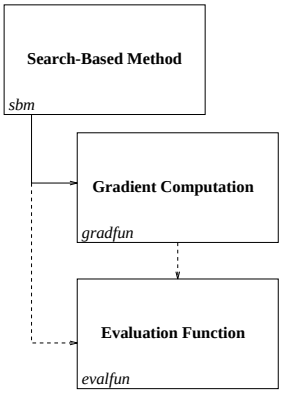
Rules:

```
(Rule 1
        :condition T
        :state-update (And (CurrentFrame ?x)
                           (Formals ?x ?formals)
                           (CurrentValue BestValue ?bv)
                           (SetValue LastValue ?bv)
                           (SetValue BestPoint
                                     (loop for m in ?methods minimize
                                           (apply ?evalfun (apply m ?formals))))
                           (CurrentValue BestValue ?nv)
                           (SetValue DiffValue (- ?bv ?nv)))
        :action NIL        :symbols (?methods))
(Rule 2
        :condition (And (CurrentValue DiffValue ?dv)
                        (GreaterThan ?dv ?epsilon)
                        (CurrentValue BestPoint ?bp))
        :action (Restart :At ?sbm
                         :Override ((:SEED . ?bp)))
        :symbols (?epsilon))
(Rule 3
        :condition (And (CurrentValue DiffValue ?dv)
                        (LessEqual ?dv ?epsilon)
                        (CurrentValue BestPoint ?bp))
        :action (Return :From ?sbm
                        :Value ?bp)
        :symbols (?epsilon))
```

Figure A.9: Schema: Multi-Method Search (**MMS**)

A.6 Bound Search Method Schemata

Structural Template:



Rules:

```
(Rule 1
        :condition (And (CurrentFrame ?x)
                        (Formals ?x ?formals)
                        (Sanity ?outsidebox ?formals)
                        (CurrentValue LastGood ?lg))
        :action (Restart :At ?sbm
                         :Override ((:SEED . (Back-InBox-Seed ?outsidebox
                                                              ?formals
                                                              ?lastgood))))
        :symbols (?outsidebox))
(Rule 2
        :condition (And (CurrentFrame ?x)
                        (Formals ?x ?formals)
                        (Not (Sanity ?outsidebox ?formals))
                        (Ancestor ?ancestor ?x)
                        (ModuleName ?ancestor ?name)
                        (NotEqual ?name ?gradfun))
        :state-update (SetValue LastGood ?formals)
        :action NIL
        :symbols (?outsidebox))
```

Figure A.10: Schema: Bound Search-based Method (**BSM**)

The schema shown in Figure A.10 attempts to address the problem that occurs when an unconstrained search-based method is used for a function where there is a known bounding box that constrains the solution space.

The schema operates by keeping track of successful evaluations that are inside the bounding box, and not part of the gradient computation for a gradient-based method. The most recent successful evaluation point is always found in the state variable `LastGood`. When an evaluation is attempted that falls outside the region defined by the user-supplied predicate *?outsidebox*, the search is restarted with a seed computed by the built-in function `Back-InBox-Seed`, which returns a point along the line connecting the failing point to the point saved in `LastGood` that is back inside the box.

We would expect this schema to be successful when the bounded permissible region for the evaluation function is large enough that there is a reasonable chance of landing in it when the search is restarted inside the box.

In the current implementation of this schema (once called *Newton-in-a-box*), some additional rules provide bookkeeping that detects when the schema is being unsuccessful and is repeatedly moving outside the bounding box.

A more generally applicable schema, seen in Figure A.11, can be derived from **BSM** by requiring a general constraint function, instead of the bounding box predicate. In this case, when a failure occurs, a line search is attempted by the built-in function `Good-Point-OnLine` to find a new starting point on the line connecting the point saved in LastGood and the failing point. This search is less computationally efficient than the direct intercept calculation possible when the bounding box is known.

This schema can be used to, in effect, transform an unconstrained search-based method into a constrained method. It uses a somewhat simplistic strategy to bring satisfying the constraint, and requires that at least one evaluable point be available.

A.7 Introduce Constraints Schemata

This schema is applicable whenever a failure occurs within the context of a constrained search-based method. It is a refinement of the **SI** schema with a more constrained context,

Derived From:
Bound Search-based Method (BSM)

Changes:

```
(Rule 1
        :condition (And (CurrentFrame ?x)
                        (Formals ?x ?formals)
                        (Sanity ?constrfun ?formals)
                        (CurrentValue LastGood ?lg))
        :action (Restart :At ?sbm
                         :Override ((:SEED . (Good-Point-OnLine ?constrfun
                                                                ?formals
                                                                ?lastgood))))
        :symbols (?constrfun))
(Rule 2
        :condition (And (CurrentFrame ?x)
                        (Formals ?x ?formals)
                        (Not (Sanity ?constrfun ?formals))
                        (Ancestor ?ancestor ?x)
                        (ModuleName ?ancestor ?name)
                        (NotEqual ?name ?gradfun))
        :state-update (SetValue LastGood ?formals)
        :action NIL
        :symbols (?constrfun))
```

Figure A.11: Schema: Simple Constrain Search-based Method (**SCSM**)

and a more sophisticated strategy.

When a failure occurs in the function *fun*, the **ICC** schema restarts the constrained search-based method *?sbm* with a newly created constraint function, *?newcf*, added to the list of existing inequality constraints for *?sbm*. This new function is actually the evaluation function *?evalfun*, but evaluated in the context of a new state variable introduced into the system blackboard.

The effect of the state variable is to trigger a new rule inserted into the rule set for *?fun* by **ICC** that causes *?evalfun* to return with a computed value that represents the degree of violation of the constraint. This value is computed by the by the using the built-in predicate `Distance` to compute the 2-norm distance between the failing point and the nearest successfully evaluated point in the cache for *?fun*.

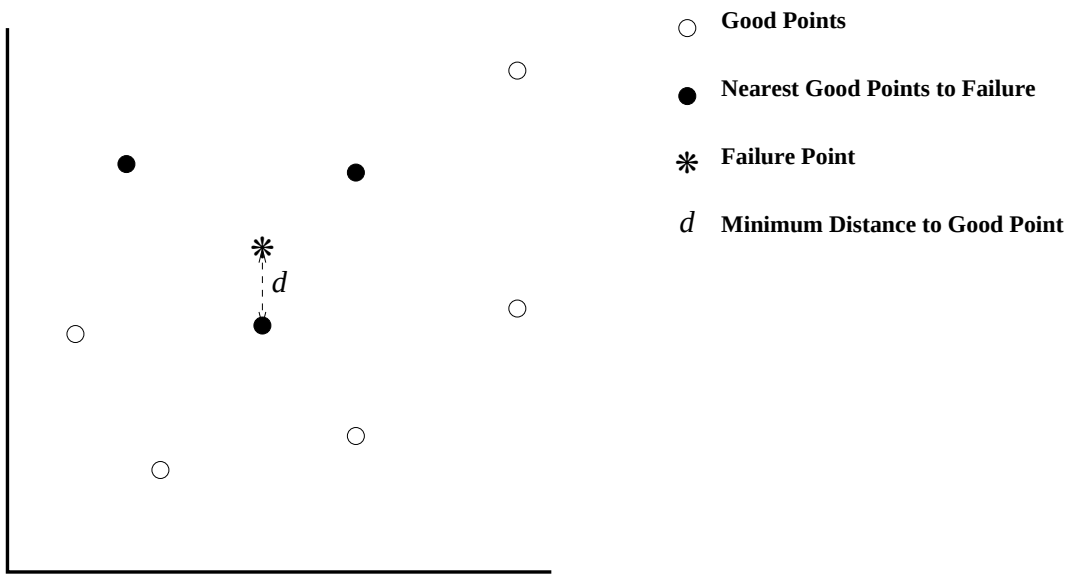


Figure A.12: Schema: Interpolate and Introduce Constraints (**ICC**)

When the evaluation function is re-evaluated *outside* of the scope of the new state variable (that is in its normal role as the objective function for the optimization), an approximate value is interpolated from the cache for *?fun*, and execution is allowed to continue. Figure A.12 illustrates the operation of **ICC**.

The use of the minimum distance to a good point as the measure of the constraint

violation is intended to have the effect of causing the constrained method to move in the best known direction in which the function is evaluable. It is possible for strange behaviors to be created, however, as the path of the optimizer attempting to satisfy the constraints moves nearer to some cached good points and away from others. In general, we would expect that it would be able to find its way back to some evaluable region, even if not by the shortest path.

A specialization of **ICC** is the Extrapolate and Create Constraint schema (**ECC**). This schema differs in being only applicable to tabular interpolation methods, but because the evaluable region of the function is explicit in the indices of the interpolation table, it is possible to create an exact constraint function that gives the degree of constraint violation (distance outside the table). We would consequently expect that **ECC** would be able to provide more effective behavior in the constrained search-based method.

Vita

John E. Keane

1984 B.A., Rutgers College

1984-90 City Federal Savings Bank, Somerset, NJ. Last position held: Associate Vice President/ Director Information Services Planning.

1990-91 Teaching Assistant, Department of Computer Science, Rutgers University.

1991-96 Graduate Assistant, Department of Computer Science, Rutgers University.

1992 M.S. in Computer Science, Rutgers University.

1993 T. Ellman, J. Keane, and M. Schwabacher. Intelligent Model Selection for Hill-climbing Search in Computer-Aided Design. *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, D.C.

1995 T. Ellman, J. Keane, T. Murata, and M. Schwabacher. A Transformation System for Interactive Reformulation of Design Optimization Strategies. *Proceedings of the Tenth Knowledge-Based Software Engineering Conference*, Boston, Massachusetts.

1996 J. Keane and T. Ellman. Knowledge-based Re-engineering of Legacy Codes for Robustness in Automated Design. *Proceedings of the Eleventh Conference on Knowledge-based Software Engineering*, Syracuse, NY.

1996 Ph.D. in Computer Science, Rutgers University.