Intelligent Intercessors in Analysis Models for Automated Design

John Keane
Thomas Ellman
Department of Computer Science
Rutgers University
New Brunswick, NJ
{keane, ellman}@cs.rutgers.edu

January, 1996

Abstract

Systems for automated design optimization of complex real-world objects can, in principle, be constructed by combining domain-independent numerical codes with existing domain-specific analysis and simulation models. Unfortunately, existing “legacy” analysis models are frequently unsuitable for use in automated design. They may crash for large classes of input, be numerically unstable or locally non-smooth, or be highly sensitive to control parameters. Direct modification of legacy codes to correct these problems is often rendered infeasible by the high cost of re-validating the modified code.

This paper describes an approach to incorporating knowledge-based handling of failures into design optimization systems that does not require code modification, yet allows for fine-grained control of model execution. We have constructed a toolkit for the development of robust design optimization systems that builds “intelligent intercessors” into existing analysis models. These intercessors are compiled from high-level rules to code that is inserted between discretely callable components of the design system. Intercessors serve to detect failures; take corrective action when possible; and transfer control to an appropriate destination when corrective actions fail. We show that this approach is effective in improving analysis model robustness and design optimization performance in the domain of conceptual design of jet engine nozzles.

Acknowledgments

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.
1 Evaluation Models for Automated Design

Before one can build a system for the automated design of a “real-world” object (such as a racing yacht or a jet engine nozzle), it is necessary to have one or more suitable evaluation models. These models take a representation of the design object as input, analyze it, and return a value or values that will be used by the design system as a basis for comparison of designs. It may be the case for a particular design problem that analysis programs already exist, developed and intended for use by human design experts. These “legacy” codes can represent an enormous investment in development and validation effort. It would be advantageous to be able to use them as the basis for the evaluation models of the design system, in conjunction with domain-independent numerical tools such as optimizers, root finders, integrators, and interpolaters.

Though construction of a complete design system from these components may appear to be straightforward, it is extremely likely that any naive implementation will work very poorly, if at all. Existing analysis codes, though extensively validated and used by human designers, are often unsuitable for direct use as evaluation models. They are frequently brittle: for inputs that violate their internal modeling assumptions or constraints, they may terminate abnormally, loop indefinitely, return absurd values; or worse, return plausible but incorrect values. They may have regions of numerical instability or local non-smoothness, and may be very sensitive to values of control parameters. Most standard numerical tools lack the ability gracefully to detect or handle failures occurring in the functions to which they are applied, and will crash or return meaningless results when problems are encountered. Finally, existing analysis programs may be unsuitable for use as evaluation models in an automated design system because they rely on human setup, intervention during execution, or “sanity checking” evaluation of their results.

2 Addressing Deficiencies in Analysis Programs

The problem is not necessarily that the analysis codes are “bad”, or even that they can be “fixed”. The nature of real-world design objects is that they are subject to physical constraints which may not be violated. These constraints give rise to discontinuous “ridges” in the value of the evaluation function in the design space that border the unevaulable regions. Optimal designs often lie at the edge of these regions (e.g. where a constraint is active at the optimum).\(^1\) When these regions are non-linearly bounded (and therefore the evaluability of a particular point cannot easily be predicted \textit{a priori}) finding the optimum in their neighborhood is a process of search that will necessarily involve the attempted evaluation of unevaulable points.

Of course, it is also true that for an evaluation model that utilizes numerical methods such as Newton-Raphson root-finders, it is not even possible to predict in advance for which inputs the method will successfully converge. Finally, it is also extremely likely that a

\(^1\)As a simple example, consider an evaluation model that predicts the speed at which a racing yacht will sail for a given windspeed and direction. For a yacht with a hull of fixed length, heading directly downwind, the speed will increase as the hull displacement is decreased. There is a physical constraint that the hull must always displace the mass of the boat, or the boat will sink. The optimum (fastest) hull will therefore be the smallest that still floats.
real-world analysis program of any complexity will occasionally crash for reasons completely unrelated to the physics it is modeling.

It is important to understand that a failure of an evaluation model in the course of a search-based process such as optimization or numerical root-finding need not be fatal. Strictly speaking, only the terminal point of the search need be evaluable. When failure occurs, it is necessary that the search-based process recognize that there has been a failure, and handle it in some manner that allows the search to continue. Good evaluation models are those that either handle failure in some meaningful way that prevents the failure from being seen by the search-based process, or that communicate the failure to the search-based process in a manner that allows it to proceed.

We believe that the difficulty of adapting existing analysis codes into suitable evaluation models is a significant barrier to the development of significant new design automation systems. To convert analysis programs, it is necessary to add code to avert failures, handle failures, capture information about failure causes, and communicate with the optimization process in which it is embedded. Our experience with adapting several real-world analysis programs has shown us that the manual effort required to modify these programs can be substantial. The problem is compounded when the analysis programs are legacy codes: the source, when available, is likely to be obscure and inadequately commented, making modification a process with a high likelihood of introducing unwanted side-effects. Legacy codes that have been directly modified need to be re-validated before designers will trust their results. Moreover, though similar problems may occur in different analysis programs, modifications to one program are rarely directly transferrable to another.

3 Context Sensitivity in Failure Handling

Correct handling of failure in an evaluation model can depend on the context in which it is being used when the failure occurs. As a simple example of this kind of context sensitivity, consider a failure occurring in an evaluation model that has been called by a gradient-based optimizer. If the optimizer has no explicit mechanism for failure handling, the evaluation model must return some value that will allow the optimization to continue, or terminate abnormally.

Suppose that the evaluation model returns a very large “bad” value to the optimizer on failure. If this evaluation is used by the optimizer to determine whether to “step” to a new design, the optimizer will then be able to continue, but the large “bad” evaluation ensures that this point cannot be an optimum. Suppose, however, that the failure occurred for an evaluation that was to be used to compute an approximation to the evaluation function gradient. In this case returning a large “bad” value would be a poor choice, as it could grossly distort the computed gradient. A better alternative might be to evaluate another nearby point, and approximate the failing value.

An even better alternative might be to change the gradient computation method for the current point, say from a 3-point central-difference method to a 2-point forward-difference method. This last alternative would require a means of interceding between the optimizer and the gradient calculation routine (assuming that the gradient routine is a discrete routine), and causing a different calculation to be carried out than was originally specified. Notice that this would require that the evaluation function not only be aware of its calling context...
(called by the gradient computation routine instead of directly by the optimizer), but also that execution control be transferred out of the normal call-return sequence.

4 Intercessors

To address some of the issues raised above, we have developed a toolkit and runtime environment that supports the incorporation of intercessors into evaluation models. An intercessor is a set of rules that describes a failure (or class of failures) that can occur in the evaluation model, and what action (or actions) can be taken. Intercessors have the following important characteristics:

- They are reusable. They can be defined in generic terms and instantiated into specific code, avoiding the need to “re-invent the wheel” for each new analysis model.

- They do not require source code modification of the model. Intercessors compile into code that exists outside and monitors the interactions between discretely callable components of the evaluation model. This avoids both the need for a detailed understanding of the internals of the analysis program, and the need to re-validate it.

- They can catch error conditions and abnormal terminations, allowing search-based processes to continue after failure.

- They can alter the execution sequence of components of an evaluation model, and can examine and modify the calling parameters and return values of model components. As will be described later, this enables a number of strategies for coping with failures.

- They have access to global execution state information, such as the run stack, and can communicate across levels of a system through a global blackboard. This allows context-sensitive intercessors to be defined easily.

- They have access to the execution history of components of the evaluation model through a caching mechanism.

5 Example

The HPCD project at Rutgers University is concerned with issues in the automation of real-world design problems where the analysis requires the utilization of high-performance computational codes. Project work is currently being done in a number of real-world design domains. The example given here draws from our experience in the domain of conceptual design of jet-engine nozzles.

5.1 The Conceptual Design of Jet Engine Nozzles

The goal of this design problem is to specify the length of the three flaps that comprise a jet engine nozzle so that the amount of fuel consumed by the aircraft while flying a specified mission is minimized. The analysis code for this problem was developed locally in collaboration with domain experts from the aerospace industry.
Figure 1 gives a simplified picture of the structure of the base design system. The system is constructed from a nested sequence of applications of several standard numerical tools which take functions as arguments (the optimizer, ODE integrator, and root-finder) with a set of domain-specific analysis programs that model the physics of the engine and nozzle (the engine model), the lift and drag of the aircraft (the aircraft model), and the resulting motion of the aircraft (the acceleration model).

A candidate nozzle design is evaluated by solving for the control parameters (throttle setting and angle of attack of the aircraft) of the acceleration function such that the acceleration of the aircraft is exactly that required by the path defined by the mission profile at a time $t$. Once the throttle setting is known, the fuel mass flow at that time can be computed. The fuel mass flow is integrated backwards over time from the end of the mission to the beginning, to give the total of the fuel consumed during the mission. This quantity, plus the fixed portion of the aircraft mass, gives the takeoff mass of the aircraft. This is the quantity that the optimizer attempts to minimize by altering the nozzle geometry.

5.2 Intercessors in the Nozzle Evaluation Model

The following are some of the intercessors that were implemented in the process of developing the nozzle evaluation model.

- **Simple Bad Value**

  There are several different ways in which the evaluation of a nozzle can fail. A candidate design may be ill-formed in some way that is detected during analysis (for example, the nozzle geometry specified may not be able to be physically realized), or an imposed constraint may be violated during simulation (for example, the throttle setting may not exceed 100% at any point during the mission). It is also possible for the simulator to crash (segmentation faults, floating point exceptions) for unknown programmatic reasons for some designs.
This simplest of the intercessors causes the ODE integrator to return an extremely large “bad” value whenever a failure occurs anywhere within its execution scope.

- Context-Sensitive Bad Value
  This intercessor also returns a bad value when a failure occurs during optimizer line search. However, if the failure occurs during gradient computation (for a gradient-based optimizer), the gradient is computed using an alternative method that does not require the failing point.

- Table Extrapolation
  There are many tables of empirical data in the analysis models, which are interpolated to appear as continuous functions. In the course of optimization and root-finding points are often evaluated which require data a short distance outside the table. This intercessor extrapolates the table, allowing the search process to continue, on the assumption that as the search converges, it will move back inside the table.

- Random Multistart
  This intercessor implements a technique common in numerical optimization: the optimization is re-started multiple times with randomly-generated valid starting seeds. The best value returned across all optimizations is returned.

- Multimethod Search
  This intercessor causes a search-based method to be replaced by a sequence of methods. Each is tried in turn, starting from the results of the prior method, until no further improvement is found. This intercessor is normally applied to the optimizer.

- Non-convergence Detector
  For some designs and missions the root finder will fail to converge to a solution for the acceleration. This results in a failure of the evaluation function to terminate. Detection of this is accomplished by means of a separate process that is started at the same time that the root-finder is called, and causes an error to be generated when a defined time limit has been exceeded.

- Bounded Root-finding
  For some points in the design space, multiple solutions are possible, corresponding to sub- and super-sonic solutions. This intercessor restarts the root-finding process at a point inside the bounding box when a bounds constraint on the evaluation function is detected. Persistent attempts to step outside the defined bounds result in an error being generated for handling by another intercessor.

6 Intercessor Toolkit Architecture

6.1 The Structure of Target Design Systems

We are interested in incorporating intercessors into systems having the general form of that described in the example: compositions of domain-independent second-order numerical codes
such as integrators, root finders, and optimizers with domain-dependent analytic codes. We believe this model covers a large class of real-world design optimization problems.

### 6.2 Function Wrappers and Their Automated Construction

Implementation of intercessors requires the addition of new code to the system to be interceded. In order to avoid the drawbacks of making modification within existing routines we have chosen to implement intercessors exclusively through the mechanism of *wrappers*: new code that intercepts control on entry and exit of discretely callable wrapped routines in the system. The idea of wrapping code is familiar in the software engineering community, however wrappers are usually seen as controlling interactions between routines in a pairwise manner, not as communicating across different levels of a system. A single intercession may be implemented across multiple wrappers, and a single wrapper may participate in multiple intercessions.

The basic wrapper structure upon which intercessors are constructed is diagrammed in Figure 2, as applied to the example of the nozzle design system. Each wrapper is a small routine that possesses the external symbolic name of the routine it is wrapping; the wrapped routine has had its symbolic name changed by the addition of a unique prefix. When the wrapped routine is called by its original external symbolic name, the wrapper receives control and the calling arguments, instead of the wrapped routine. It may perform whatever condition checking or actions are required by the intercessions in which it is participating, before transferring control to the wrapped routine. (Alternatively, as part of an intercession it may call another routine instead, or return immediately.) When the wrapped routine returns, control is once again given to the wrapper, which may perform any post-processing required.

The base wrappers (the interception code without intercessions) may be constructed automatically by modification of the external symbol information in compiled code, or by a compiler preprocessor for source code (as we have currently implemented). Function head-
ers are required in either case to provide the wrapper with information about the formal arguments of the function.

Wrappers communicate with each other and access global state information through the system blackboard. Wrappers may also transfer control to each other directly, through a mechanism analogous to the Common Lisp “throw” and “catch” functions, allowing a called wrapper to return control to a calling ancestor wrapper several levels above it.

An additional side-effect of the wrapper mechanism is the ability to keep a record of the calling arguments and return values for wrapped functions. As will be seen later, this allows intercessors to be defined not only with respect to the current problem state, but also the state history.

The blackboard maintains information about global problem state, and allows wrappers to communicate through special state variables. It also maintains a trace of all intercessor action for debugging and analysis purposes.

6.3 Definition of Intercessors

An intercessor is defined by one or more rules of the form:

If Condition
Then
  Action
Global-state-update
  State

Each rule specifies a set of conditions that define when and where the rule is applicable; a set of consequent actions that alter the flow of control of the executing system; and a set of updates to the global state information.

The condition portion of an intercessor rule consists of one or more logical tests joined by boolean operators. The tests may reference the parameters of routines, state variables created and modified by other intercessors, or special global state information maintained by the blackboard system. This includes the context state variables such as called-in-the-context-of, which may be used to define an intercessor that occurs only when some function is called within the dynamic scope of some other function; and number-of-times-called-in-the-context-of, which may be used in intercessors to detect convergence problems. Conditions may also be defined for exceptions such as bus errors or floating point exceptions, timeouts, calls to the system exit routine, or special intercessor-defined signals. Exception intercession is dynamically scoped, with the most recent, most specific intercessor receiving control for a given exception.

The action portion of an intercessor defines one or more actions to be taken when the condition is true.

Actions include: Restart, which allows a routine at the current or higher level to be restarted with different parameters; Alternate, which specifies an alternate computation to be performed in place of the specified system routine; ReturnFrom, which specifies a function to return from, and a list of override values; Signal, which allows an intercessor-defined exception to be raised, resulting in the transfer of control to another intercessor.

When multiple actions are defined the intercessor becomes a backtracking intercessor. The first time the rule is applied in a particular runtime context, the first action is taken. If
the rule conditions evaluate true again within the same runtime context, the next action on the list is tried. This may be repeated until all actions have been exhausted. This mechanism allows for multiple different intercessions to be attempted where it is not possible to know in advance which will work.

The state portion of an intercessor 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.

6.3.1 Example of an Intercessor Definition

If net-force-function(V) is called in the context of
newton-raphson-solver(X0, *Function0fX) Global-state-update
LastIterationValue := V ;

If (the number of times net-force-function(V) is called in the context of newton-raphson-solver is greater than 100)
Then
 Return from newton-raphson-solver with a value of LastIterationValue ;

Figure 3: Example Intercessor Rules

Figure 3 is an example of the rules defining a simple intercessor to allow a higher-level computation to continue when a Newton-Raphson root-finder has failed to converge after 100 iterations by returning the trial point of the last iteration as the root. (The rule syntax shown is exemplary rather than literal.)

The first rule of the intercessor takes no action, but sets a global state variable LastIterationValue to the value of the calling argument V of net-force-function whenever it is called in the dynamic scope of newton-raphson-solver. The second rule in the intercessor uses a built-in feature of the blackboard system that can keep count of the number of times a routine is called within the dynamic scope of another routine. When net-force-function has been called more than 100 times the intercessor forces a return from newton-raphson-solver with the value of V that was most recently saved in LastIterationValue.

6.4 Intercessor Schema

An intercessor schema is an intercessor definition in which the names of functions and variables have been replaced with class variables that constrain the values that may be used to instantiate those variables. Functions in the intercessor toolkit are represented by objects organized in a class hierarchy. Multiple inheritance is permitted. Formal arguments common
to a class of functions are represented by slots in the function object, thus formal arguments may be uniformly symbolically referenced within a class.

### 6.4.1 Example of an Intercessor Schema

If 1D-FUNCTION

*is called in the context of*

ROOT-FINDER

Global-state-update

\[
\text{LastIterationValue} := \text{1D-FUNCTION:ARG1};
\]

If *(the number of times 1D-FUNCTION is called in the context of ROOT-FINDER is greater than \{Limit\})*

Then

*Return from ROOT-FINDER with a value of LastIterationValue;*

Figure 4: Intercessor Schema Example

Figure 4 shows the intercessor schema corresponding to the intercessor shown in Figure 3. The class 1D-FUNCTION is defined to have a single input argument, ARG1. The class ROOT-FINDER is defined to have a single return value, RETURN-VALUE. The variable \{Limit\} must be provided at instantiation time. This intercessor schema may be instantiated for any pair of functions where one is a 1D-FUNCTION and the other is a ROOT-FINDER.

### 7 Results and Discussion

#### 7.1 Evaluation Methodology

There are two dimensions along which we expect intercessors to have an impact in the development of evaluation models for automated design: ease of implementation, and improved quality of final result as a consequence of greater robustness in the evaluation model.

The first dimension is hard to assess rigorously, as the toolkit and runtime environment are still being developed. Our own experience suggests that, at least for intercessions requiring action at multiple levels of the evaluation model (such as the bounded root-finding intercessor) implementation is greatly facilitated by the toolkit architecture. On at least a few occasions, we have found that we can easily re-use intercessors developed earlier, at a small fraction of the effort. We plan to do a more controlled study when the system is more mature.

To evaluate the effect of including intercessors in evaluation models, we selectively included intercessors from our intercessor library into evaluation models from three different
problem domains. We then ran a series of optimizations with various goals for each configuration, and compared the results to the known best results for each goal. We averaged the performance over all optimizations to obtain an overall rating for each.

As a measure of the execution cost of each run, we counted the number of times the evaluation function was called. In real-world design systems, the cost of evaluation far exceeds the computational cost of any other part system.

7.2 Domain: Synthetic Failure Function

Figure 5 shows the effect of intercessor incorporation on a synthetic quasi-quadratic\(^2\) 3-parameter evaluation function with non-linearly bounded failure regions. The function has a single global optimum that lies on the intersection of two failure surfaces, ensuring that optimization will encounter unvaluable points as it converges on a solution.

The Simple Bad Value and Context-Sensitive Bad Value intercessors were tested independently. 100 optimizations were run for each method from valid random starting points. The quality was normalized relative to the true global optimum, which for this function is known exactly. Evaluations shown are the average per successful optimization (except for the first row, which is the total number of evaluations). A successful optimization is one in which the optimizer terminates normally, returning the best value found.

<table>
<thead>
<tr>
<th>Intercessor</th>
<th>% Completed</th>
<th>% Quality</th>
<th>Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0%</td>
<td>NA</td>
<td>876.0</td>
</tr>
<tr>
<td>Simple</td>
<td>100%</td>
<td>77.1%</td>
<td>164.7</td>
</tr>
<tr>
<td>Context Sensitive</td>
<td>100%</td>
<td>80.7%</td>
<td>191.4</td>
</tr>
</tbody>
</table>

Figure 5: Results with Synthetic Failure Function

The results for this contrived problem show the value of intercessor incorporation in achieving any successful optimization at all. Optimization quality was significantly improved, at a greater computational cost, when the context-sensitive intercessor was used. The difficulty of optimizing even this relatively simple function in the vicinity of the failure regions is seen by the relatively low overall quality score.

7.3 Domain: Racing Yacht Hull Design

The objective of this problem is to design a hull for a racing yacht, subject to various imposed constraints and environmental conditions, such that the time required to sail the yacht around a goal race course is minimized. The evaluation model for this domain is based on analysis code derived from a commercial racing yacht analysis package, AHVPP1, developed and marketed by AeroHydro, Inc. [AeroHydro, 1992]. Of primary importance in constraining the design is the so-called “12-meter rule” which forces the weighted sum of various dimensions of the yacht to total 12 meters. This constraint forces a number of ridges into the evaluation function.

\(^2\)A truly quadratic function would be too easy to optimize with the constrained quadratic programming optimizer we used.
Figure 6 shows the effect of intercessor incorporation on the yacht hull evaluation model. The Simple Bad Value and Context-Sensitive Bad Value intercessors were tested independently. Twenty-five optimizations were run for each method from a fixed starting point\(^3\) for single-leg racecourses under different windspeed and direction goals. The quality indicates the percentage of results that were within \(\epsilon\) of the best known optimum for each goal. Evaluations were averaged over all successful optimizations.

<table>
<thead>
<tr>
<th>Intercessor</th>
<th>% Completed</th>
<th>% Quality</th>
<th>Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>8%</td>
<td>8%</td>
<td>219.0</td>
</tr>
<tr>
<td>Simple</td>
<td>100%</td>
<td>32%</td>
<td>131.9</td>
</tr>
<tr>
<td>Context Sensitive</td>
<td>100%</td>
<td>48%</td>
<td>180.5</td>
</tr>
</tbody>
</table>

Figure 6: Results for Yacht Hull Design System

The results show that incorporation of intercessors improves the quality and reduces the amortized cost of optimization. The context-sensitive intercessor improves the optimization quality, at a higher average evaluation cost.

### 7.4 Domain: Conceptual Jet Engine Nozzle Design

Figure 6 shows the effect of intercessor incorporation on the nozzle evaluation model described earlier. In addition to the Simple Bad Value and Context-Sensitive intercessors tested in the other domains, the Table Extrapolation, Random Multistart, and Multimethod intercessors were tested, the last two independently applied at the optimizer. The Multimethod optimization was instantiated for a gradient-based, constrained quadratic programming optimizer (CFSQP\cite{1994LAWRENCE}; and a non-gradient optimizer, downhill-simplex\cite{1986PRESS}). Eighty-one optimizations were run for each method from known valid starting points for eight different mission goals. The quality indicates the average normalized closeness to the best known optimum for each goal. Evaluations were averaged over all successful optimizations.

<table>
<thead>
<tr>
<th>Intercessor</th>
<th>% Completed</th>
<th>% Quality</th>
<th>Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0%</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>Simple (S)</td>
<td>0%</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>Table Extrapolation (TE) + S</td>
<td>100%</td>
<td>20.3%</td>
<td>115.0*</td>
</tr>
<tr>
<td>Context Sensitive (CS) + TE</td>
<td>100%</td>
<td>52.9%</td>
<td>141.0*</td>
</tr>
<tr>
<td>Random Multistart + CS + TE</td>
<td>100%</td>
<td>94.0%</td>
<td>402.8</td>
</tr>
<tr>
<td>Multimethod + CS + TE</td>
<td>100%</td>
<td>91.9%</td>
<td>619.4</td>
</tr>
</tbody>
</table>

\(^*\) Evaluation data from partial run

Figure 7: Results for Conceptual Nozzle Design System

\(^3\)In this design system, design modifications are performed as deformations from a starting prototype hull.
These results also indicate that no optimization is possible without some degree of intercession. The Context-Sensitive and Table Extrapolation intercessors were found to be required at a minimum. Optimization quality was improved by both of the optimization intercessors; surprisingly, Random Multistart is pareto superior to the more “sophisticated” Multimethod.

7.5 Discussion

Our results indicate that in order to use analysis models for design problems in which failure is likely to be encountered during the optimization, some form of intercession must be incorporated into the models before they can successfully be used as evaluation models. They also show the value of context sensitivity, communication between wrappers, and intercession at different levels in improving evaluation model behavior.

We have shown that a wrapper-based approach to intercessor implementation can be effective without requiring direct modification of analysis program code. We have also demonstrated some degree of generality to our approach by incorporating the same intercessors across multiple problem domains.

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 intercessor incorporation than systems with a few highly monolithic modules. The greatest leverage comes when analysis programs use well-known standardized numerical routines for which intercessors already exist.

The 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 intercessor 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 intercessor 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.

8 Future Work

At this time, much of the process of intercessor incorporation is manual. The user must determine that a failure is occurring, select an intercessor from a library, and instantiate it into the analysis code. We are planning to automate much of this process into an interactive evaluation model development system. The system will dynamically detect failure, present a heuristically-ordered list of applicable intercessors to the developer for selection, instantiate it into the model, and restart model execution.

Many failures can be avoided during search-based processes by constraining the search-based method. The intercessor architecture provides a means to introduce constraints into search-based methods occurring in the evaluation model. We are planning extensions to the system’s ability to acquire and manipulate constraints. In particular, we are planning to
take advantage of the system’s ability to cache function evaluations to allow for the definition of *qualitative sanity constraints*, such as monotonicity or modality, on functions within an evaluation model. The evaluation caches could also be used as source of training data to learn failure-prediction functions for analysis program subroutines.

### 9 Related Work

The ENGINEOUS system [Tong, 1988] is a system for assisting design engineers in real-world scale problems that was developed for General Electric. It includes the ability to bring in external code as part of an analysis model, and a framework for execution that provides some exception-handling. Our work provides a more robust and general approach to the problem of recognizing and handling exceptions in analysis codes.

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. Intercessors can be similarly applied at the level of the optimizer function in a design system, but are general enough to be applied to other classes of function.

The ARPA Intelligent Integration of Information (I3) Reference Architecture [Hull and King, 1995] is concerned with similar issues in the use of legacy information sources in new contexts. They define a broad 5-level architecture, with five primary families of services: Coordination, Management, Semantic Integration and Transformation, Functional Extensions, and Wrapping. Our work appears to fit into this architecture as a kind of a Wrapping Service, involving mediation.

Naftaly Minsky has written on the concept of “Law-governed Systems” [Minsky, 1991], where global rules can constrain the interactions between components of a system. There are many similarities in our ideas, including using non-local information in controlling module interactions, and imposing controls by means of an external execution framework. Our work is more specific to the problems encountered in design automation systems.

Our work is closely related to research taking place at Rutgers University, particularly that of Andrew Gelsey and Don Smith [Gelsey and Smith, 1995]. In developing our intercessor schema, we have drawn on their experiences in developing the Nozzle Design Associate (NDA), and the Modeling and Simulation Associate (MSA), and used them as a basis for generalization.

### References


