Computational Frameworks for Collaborative, Multidisciplinary Design

David E. Acton

AE 8500 Project Report
June 4, 1998

School of Aerospace Engineering
Georgia Institute of Technology

Advisor: Dr. John R. Olds

TABLE OF CONTENTS

1. Introduction

Significant research has been conducted into the development and deployment of analysis integration techniques. This paper discusses these research activities in the context of collaborative frameworks for conceptual space systems design. It begins by discussing the current state of aerospace systems design processes, then recommends a path toward higher efficiency in design. It continues by describing various computational technologies that have been or are being developed to achieve this goal, and concludes by presenting several demonstration problems that make use of these technologies.

2. Background

2.1 Research Context

The Space Systems Design Laboratory (SSDL) at Georgia Tech is one of three branches of the Aerospace Systems Design Laboratory (ASDL). SSDL is focused primarily on developing conceptual design strategies and tools for advanced launch vehicles, satellites, and interplanetary spacecraft. In addition to developing these strategies and tools, the group demonstrates them on a series of sample design problems, ranging from a reusable launch vehicle powered by a rocket-based combined-cycle engine, to a solar-electric-powered spacecraft mission to Mars' moon Phobos.

The material discussed in this report is based on computational framework research performed within the SSDL, and therefore tends to have an emphasis on space systems design. The demonstration problems involve both launch vehicles and interplanetary spacecraft. Obviously then, the analysis tools used in these demonstrations deal with space-related disciplines.

One good reason why space systems design is well suited to an integrated design framework is its inherent complexity. Figure 1 shows a typical Design Structure Matrix (DSM) for a launch vehicle design process performed in the SSDL. The first thing to notice is the relatively large number of disciplines involved when developing such a vehicle. This DSM does not even show some other disciplines that may be involved in some more detailed designs, such as "Stability & Control," "Avionics," and "Operations & Facilities."

The second thing to notice about the DSM is the high level of coupling between the various disciplines. It is impossible to design a complex system such as a launch vehicle in a single pass through the disciplines. Decisions that are made early in the process (for example, in "Layout & Packaging") will undoubtedly need to be modified when the results of later analyses are generated. Typically, the process is iterated until convergence of the design variables is reached. That is, when the outputs of all disciplines do not change beyond a specified tolerance during successive iterations. This all means that the entire design process may take many iterations, each involving numerous complex analyses. An integrated design framework can help organize the transfer of data between different disciplines.

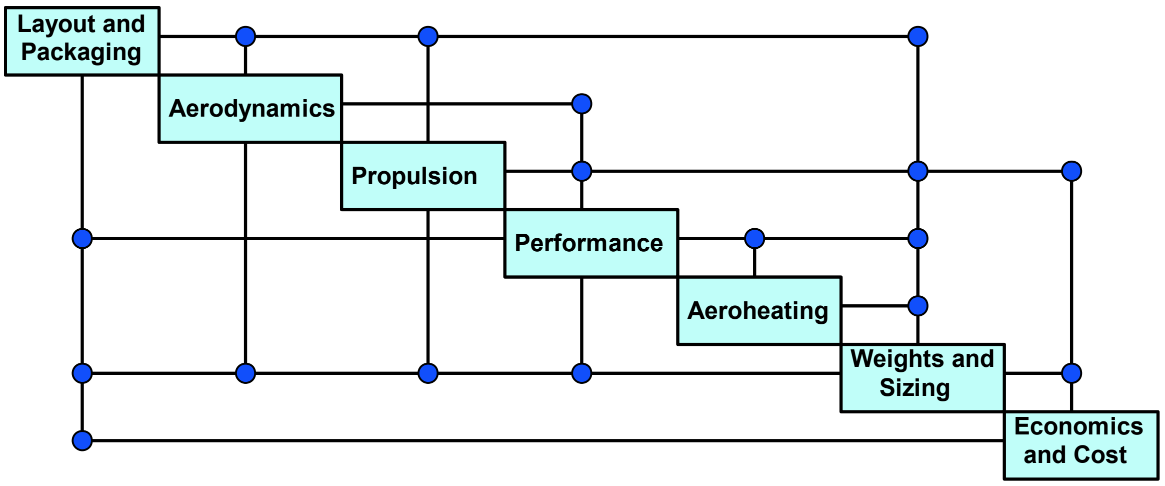


Figure 1. Typical Design Structure Matrix for an advance launch vehicle.

Another good reason for applying design frameworks to aerospace-related problems stems from the large amount of resources spent performing the various analyses. The space design community makes wide use of industry-standard *legacy codes*, referring to computer programs that were developed years ago for specific engineering disciplines. These legacy codes, whether they are command-line based FORTRAN programs, graphical CAD programs, or PC-based spreadsheets based on industry standard formulas, typically require a great deal of human interaction to execute. The user may need to hand-edit text input files, then read through and extract information from text output files. He may have to sit in front of a CAD terminal for hours drawing the model and analyzing it, or he may edit and retrieve cell values of a spreadsheet using a mouse at a PC. Unfortunately, much of the tedium that evolves from using these legacy codes arises from the pure amount of repetition involved. If a computer-based integrated design framework can reduce the tedium by performing repetitive tasks automatically, it has instantly enhanced the design process.

Of course, the framework research discussed here can be applied to almost any design process, in any field of engineering, science, architecture, etc. Any process that makes use of distinct computational analyses, and requires interaction between these analyses, should benefit from an integrated framework. Obviously, the processes that reap the most benefit from the framework are those that are most difficult to perform in the first place. If your design process consists only of several algebraic equations, it would be more efficient to write a single program to do the calculations, rather that deal with the overhead of an integrated framework. It should also be noted that the analyses need not all be computer-based. After all, not all engineering knowledge can be reduced to computer routines. Perhaps one element of a design process involves querying the local expert in a particular field. Even though the automation capability of the framework may not be tapped here, the efficient data exchange made possible with the environment will certainly aid the process.

2.2 Conceptual Design Strategies

In today's industry, the method of designing complex engineering systems usually takes one of two forms, depending often on the size of the design team, the tradition and experience of its members, the culture of the organization, and the specific point within the design process. Each design method seeks to integrate knowledge from distinct disciplines to synthesize a complete system. The major differences between design methodologies are in how information from the disciplines is integrated, and how knowledge is actually produced within each discipline.

Engineering disciplines usually provide more than one *piece* of knowledge. For example, a structural analyst on the design team for a launch vehicle may run a particular computer code that calculates the mass properties of the design. That analyst may run a separate code that predicts the flexible modes of the structure. Each discipline is therefore made up of several *contributing analyses* (CA's), each of which contributes to the total knowledge base.

2.2.1 Loosely-Integrated Analyses

The most prevalent design methodology for several decades has made use of *Loosely-Integrated Analyses*. This implies heavy use of standard legacy codes, but provides for very little electronic integration of CA's. Experts in each discipline manually perform each CA, and data is exchanged between disciplines along written or verbal lines, or possibly through *ftp* of data files (See Figure 2). This method is usually seen in large organizations, working on large, complex systems, often at the preliminary design phase.

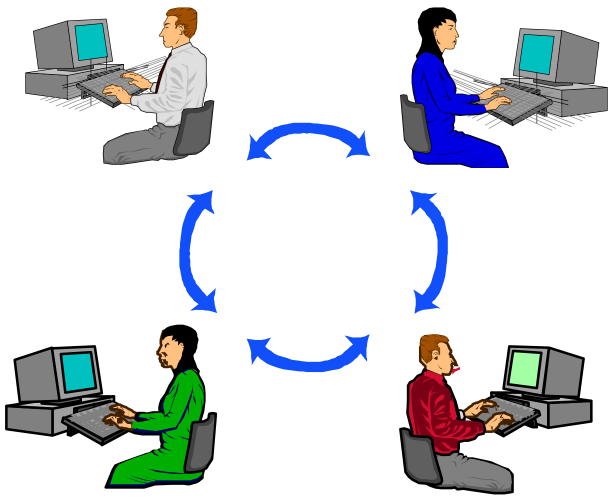


Figure 2. Loosely-integrated analyses.

Benefits of this methodology include the ability to work over geographic and computer platform boundaries, retention of disciplinary expertise in a collaborative environment, and high-fidelity results. Disadvantages include very slow cycle time, difficulty of data-exchange, large human and computer resource requirements, and often the inability to truly optimize the entire system.

2.2.2 Monolithic Design Code

The second design methodology, use of a *Monolithic Design Code*, takes the opposite extreme from the loosely-integrated approach. The monolithic code integrates CA's as subroutines of a large synthesis “executive.” Intended for single user execution, usually in the conceptual design phase, a monolithic code uses response surfaces, low-fidelity codes, and other methods to approximate the higher fidelity results that would be obtained using legacy codes (See Figure 3). Examples of monolithic codes include FLOPS and ACSYNT in the aircraft industry, and AVID and ODIN in the spacecraft field.

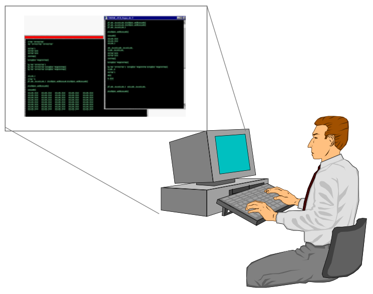


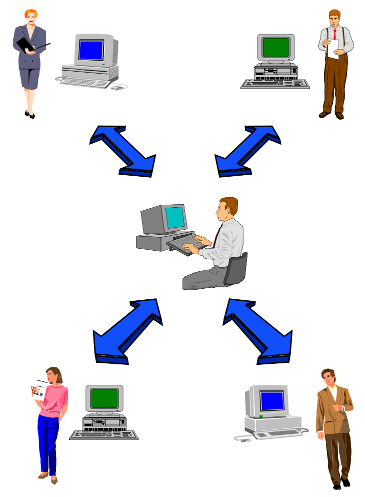
Figure 3. Design framework using a monolithic code.

Advantages of this methodology include fast execution, easy data flow, and limited resource requirements. Disadvantages include lower fidelity results, and possible alienation of disciplinary experts, due to lack or reliance on legacy codes. Another problem that stems from the use of low-fidelity tools at the conceptual level is that potential problems which will arise under more detailed design may be difficult to catch. In general the cost of design changes rises almost exponentially as a system progresses down its design life cycle. With a low-fidelity conceptual design model, some of the “show-stopper” problems will not be detected early on. Optimally, we would like to bring more of the engineering knowledge from the Preliminary and Detail design levels back into the Conceptual stage, and this means using higher-fidelity codes earlier.

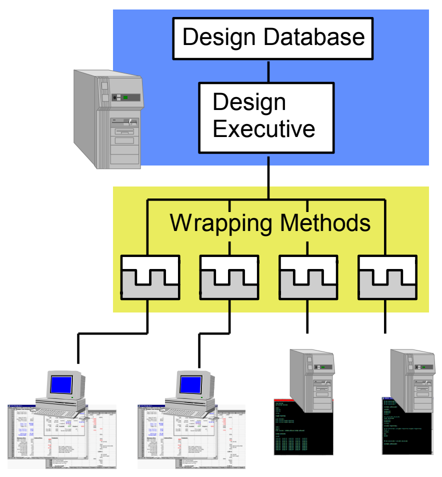
3. A Future Design Strategy — Tightly-Integrated Analyses

With current methodologies, it is difficult to bring high-fidelity analysis into the conceptual design phase. Correspondingly, when the engineering system is transitioned into preliminary and detail design phases, the convenience of easy data exchange and fast cycle time is lost. Current research is directed towards developing collaborative design frameworks that bridge the gap between the existing design strategies, and thereby share the benefits of each. An optimum design scheme is one that reduces cycle time, yet keeps the fidelity and accuracy of legacy codes and disciplinary experts.

The key to accomplishing the above goal is to transform the complex disciplinary analyses into "design-oriented" analyses, capable of being "plugged into" a larger system executive. These *Tightly-Integrated Analyses* will not be approximations, but rather true legacy tools set up by experts. The codes are "wrapped" using shell-scripts, remote computer control, and other methods. The system executive automatically executes each contributing analysis when needed, returning a high-fidelity result that can be easily passed to the next analysis (See Figure 4).



(a) Layout of the design framework



(b) Integration schematic

Figure 4. Tightly-integrated analysis framework.

The system executive serves two purposes. First, it provides for data exchange between separate analyses, and design data storage. Second, it controls the execution of each CA towards a system-level goal. That goal could be the optimization of one or several system-level metrics (such as vehicle dry weight or development cost). The flexibility of a system executive also allows for the use of Multidisciplinary Design Optimization (MDO) techniques to reduce design iterations or improve final results.

4. Thrust Areas

While the goals and benefits of a tightly-integrated design framework are obvious, the mechanisms by which to achieve it may not be. The primary problem is how to integrate and automate legacy codes and resources that were designed to be executed by humans. After all, if it were easy to do, why would anyone attempt a design in a loosely-integrated environment? Issues that will be encountered include

- Designing and developing a computational architecture that will support integration and design process deployment
- Difficulty in automating sometimes complex interfaces to legacy analyses
- Dealing with dissimilar computer platforms for analyses
- Extending the framework to access geographically distributed computer systems

The research discussed in this paper addresses these issues and examines methods to resolve them in a tightly-integrated collaborative design framework. This is a good time to differentiate between a *design framework* and a *computational architecture*. We define a framework as a methodology for connecting disciplinary analyses. The framework refers not to a particular piece of software, but rather to the idea of how codes should be tied together (if at all) in a design environment. We define an architecture as the enabling software environment that supports a particular framework. The architecture may be best represented by the system-level executive described above when discussing the tightly-integrated framework.

It is important to note that this research is not intended to develop a particular computer architecture for engineering design, but rather to examine the techniques necessary for integrating analysis tools into such an architecture. Much work has been done in both academia and industry to construct particular computational infrastructures that support integrated design (one architecture developed at Georgia Tech, IMAGE, was actually used in this research. See Section 4.1.1). This is a bold undertaking, and requires detailed knowledge of design schema evolution, database design, process management, and message passing, along with a keen understanding of how engineers actually perform a design. However, what many design architectures leave out is a clear description of how to integrate a wide variety of codes. Admittedly, the architecture designer cannot predict the complete array of analyses that a user may wish to integrate, and therefore cannot define every technique for wrapping such analyses.

Rather, he supplies the “ports” into the architecture, into which the engineer “plugs” his wrapped analysis codes into. In effect, most of the computational architectures being developed are intended to support the idea of a tightly-integrated design framework.

When discussing a future design framework, one must first look at the way design is done in industry now. We cannot just wipe decades of engineering experience and tradition away, and require a completely new way of designing spacecraft, aircraft, or any other engineering system. That would mean telling engineers to change the way they make decisions and perform design. There are too many legacy codes and legacy knowledge that must be tapped. Rather, we should look at the way design should be done in the future, what efficiencies we can obtain, and develop ways to bridge the gap between now and then.

4.1 System Executive Design

The structure and implementation of the system-level executive is a critical issue in designing the environment. The executive defines the user interface to the design process while it is running, as well a possibly during problem setup. The executive also determines the method of data storage, data retrieval, and data exchange. The effectiveness of a particular executive may be measured by its ease of use during both setup and execution, robustness to both user error and design instability, expandability to include additional analyses, and level of automation.

Two system executive implementations were investigated, including a Unix-based single-user architecture that provides sophisticated integration and database tools, and a Web-based architecture that allows distributed, multi-platform information access and analysis execution.

4.1.1 Unix Interfaces

The architecture chosen in this category was the Intelligent Multi-disciplinary Aircraft Generation Environment (IMAGE)[1]. Developed by Dr. Mark Hale as part of his Ph.D. dissertation, IMAGE is a modular computing architecture based on a Form, Model, Process

[1] The name IMAGE is a bit of a misnomer since the architecture really contains nothing that makes it inherently appropriate for aircraft design versus any other sort of system design.

design theory. In its current form, IMAGE consists of a graphical user interface, based on the Tk/tcl scripting language, above an object-oriented modeling system (See Figure 5). It contains a structured scheme for integrating analyses, supports communication via PVM, and can be compiled for a variety of Unix platforms.

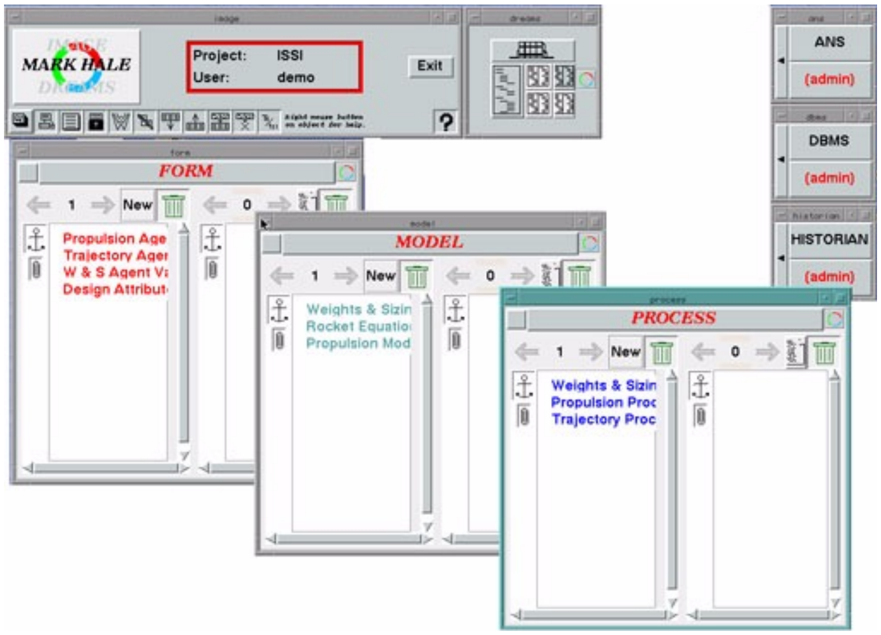
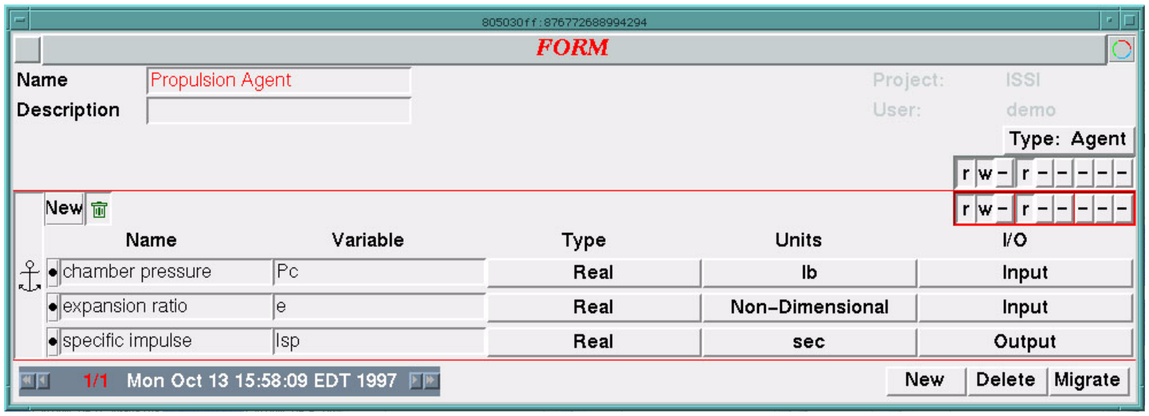


Figure 5. The IMAGE integrated design architecture.

It is useful to describe the primary elements of IMAGE in order to develop a context within which one may later discuss its effectiveness for certain problems. IMAGE is based on the notion of design schemas. A schema is a collection of various objects that all have a common basis. A Form schema will likely contain a collection of variables related to a particular system subsystem. An example from the aerospace field would be the collection of variables related to a propulsion system. These may include performance variables such as engine thrust and I_{sp}, as well as geometric variables such as inlet area and expansion ratio. A Model schema will contain objects which define a model of a particular phenomenon. This phenomenon may be the aerodynamic behavior of a vehicle in atmospheric flight. In order to model this behavior, we run a computer code to simulate it. The Model schema will describe the computer code and how it is executed. A Process schema will contain objects which link particular models to particular design variables. It is no good if we have an aerodynamic analysis, and we try to apply it to propulsion design variables. The Process schema provides the coordination mechanism.

During problem description and setup, the designer defines the object within each of the Form, Model, and Process schemas. Typically there are multiple of each type of schema, perhaps representing multiple disciplines within a design process. For example, in the RBCC IMAGE demonstration to be described in Section 6.1, four separate Form schemas are used, each representing one of Propulsion, Trajectory, Weights & Sizing, and Economics & Cost. Even for one discipline, one may generate several Form schemas, each containing the variables that are used for a particular code. This may arise if one has access to say three different trajectory codes. Each code has its own special set of input variables, whose names may not coincide with the other two codes' input variables. An example Form schema is shown in Figure 6.


805030ff:876772688994294

FORM

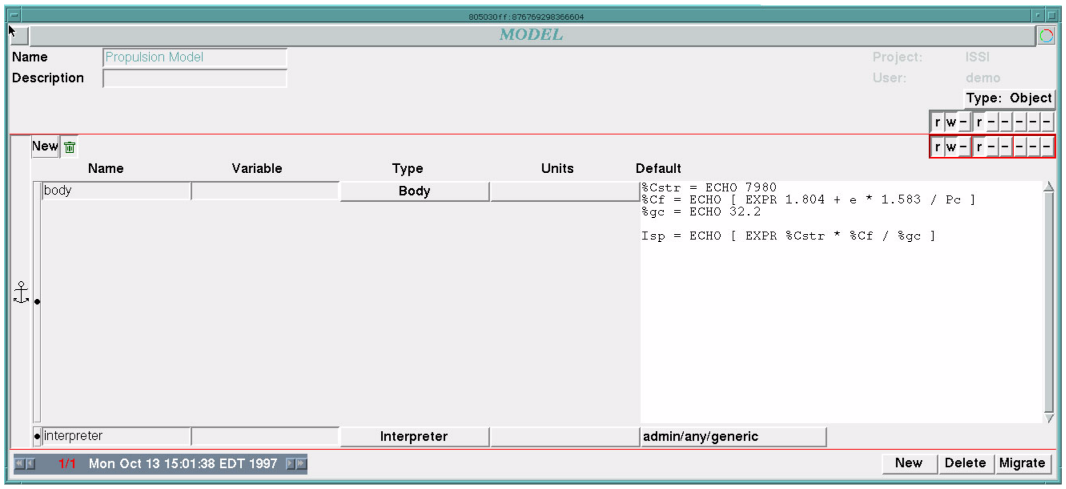
Name: Propulsion Agent
Description:
Project: ISSI
User: demo
Type: Agent

Name	Variable	Type	Units	I/O
chamber pressure	Pc	Real	lb	Input
expansion ratio	e	Real	Non-Dimensional	Input
specific impulse	Isp	Real	sec	Output

1/1 Mon Oct 13 15:58:09 EDT 1997

Figure 6. Example Form schema containing design variables.

A Model schema contains both a Body and Interpreter object. The Body object provides a location to specify the exact syntax to execute an agent. (IMAGE uses the term *agent* to refer to contributing analyses after they have been wrapped and integrated into the architecture) The Interpreter object defines the language in which the Body content is to be understood. In most problems, the Body syntax is written in a modified form of the Tcl scripting language, so the Interpreter is simply a modified Tcl interpreter. An example Model schema is shown in Figure 7.



805030ff:876769298366604

MODEL

Name: Propulsion Model
Description:

Project: ISSI
User: demo
Type: Object

Name	Variable	Type	Units	Default
body		Body		
interpreter		Interpreter		admin/any/generic

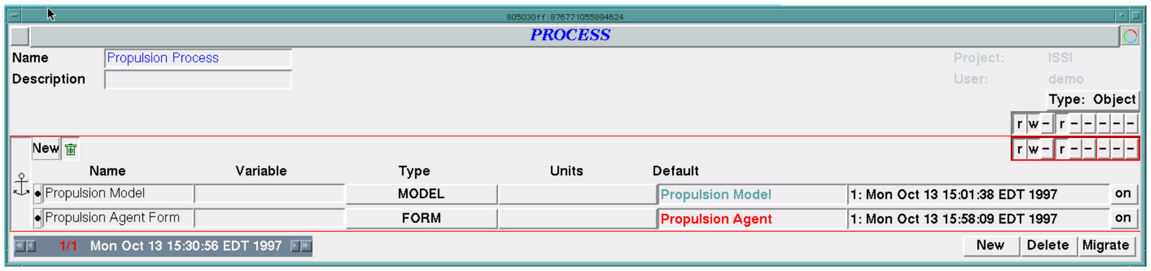
```
%Cstr = ECHO 7980
%Cf = ECHO [ EXPR 1.804 + e * 1.583 / Pc ]
%gc = ECHO 32.2

Isp = ECHO [ EXPR %Cstr * %Cf / %gc ]
```

1/1 Mon Oct 13 15:01:38 EDT 1997

Figure 7. Example Model schema defining an analysis agent.

Process schemas simply link an instance of a Model schema to an instance of a Form schema. This allows variables described in the Form schema to be directly associated with command line arguments, output variables, etc. contained within the Model schema's Body content. An example Process schema is shown in Figure 8.



805030ff:876771055894624

PROCESS

Name: Propulsion Process
Description:

Project: ISSI
User: demo
Type: Object

Name	Variable	Type	Units	Default		
Propulsion Model		MODEL		Propulsion Model	1: Mon Oct 13 15:01:38 EDT 1997	on
Propulsion Agent Form		FORM		Propulsion Agent	1: Mon Oct 13 15:58:09 EDT 1997	on

1/1 Mon Oct 13 15:30:56 EDT 1997

Figure 8. Example Process schema linking a particular Form schema to an agent model.

After defining the Form, Model, and Process schemas, the designer specifies the design process. Suppose that we have defined Propulsion, Trajectory, Weights & Sizing, and Economics & Cost analyses, but which order do we perform them in? Do we have iteration between the analyses, or is each only executed once? Is there an optimizer wrapped around one or more of the design variables? These choices are realized through a natural-language processor meant to aid in translating the structure of a typical engineering design process. If each analysis is to run sequentially, one might tell IMAGE:

```
First find Propulsion then find Trajectory then find
Weights then find Cost
```

The natural-language processor will convert this series of commands into a graphical process tree, with each node representing one of the *system support problems* (or contributing analysis). The exact execution parameters of each system support problem may be defined, including such items as manual vs. automatic start and single vs. infinite execution. Figure 9 shows the design process after being interpreted by the natural-language processor.

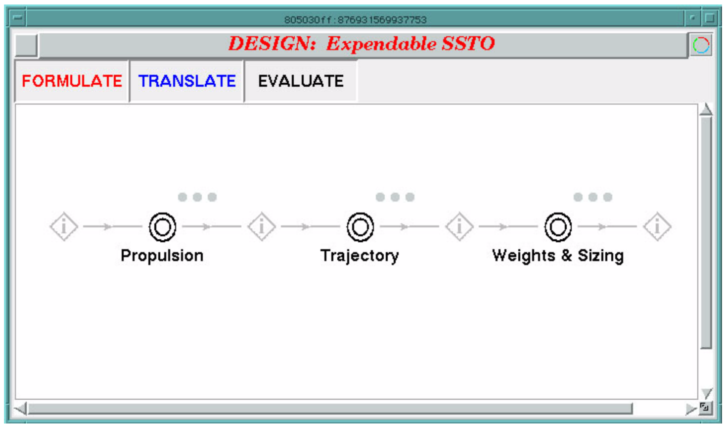


Figure 9. Sample design process for an SSTO all-rocket vehicle design.

Once all schemas and processes are defined, the design problem can be executed. During execution, interpreters are started as agents are called by IMAGE. Small red, yellow, and green lights next to each system support problem illuminate signifying its current state. When the problem is completed (or even during execution), variable values within the Form schemas may be examined. A very useful feature of IMAGE is its ability to retain past values of each design variable. Instead of keeping only the most recent number, output file, or picture, IMAGE stores past versions, and records each with the date, user, and project with which it is associated. For numerical values, the progression of design variables may be plotted to help in visualizing the convergence of a design.

The IMAGE environment was used primarily in the Hyperion RBCC demonstration (See Section 6.1) and the NASA Langley IDS demonstration (See Section 6.3).

4.1.2 Web Interfaces

The World Wide Web (WWW or Web) provides a natural architecture on which to build distributed computational systems. The combination of its Internet heritage, simple and consistent data request formats, and customizable levels of security makes the Web appealing to someone planning to run widely distributed codes.

The Web uses the HyperText Transport Protocol (HTTP) and Uniform Resource Locators (URLs) to pass information between various locations. The way URLs are formatted, any page of information, or any Web-accessible computer program may be accessed via a simple one-line text string. The page or program may return content ranging from simple text data to graphical images, plots, and sound files. This rich content is passed directly back to the user via the HTTP protocol, and displayed in his or her Web browser. Another benefit of the distributed Web environment is that no real distinction is made between a Web server located upstairs and one located across the country. Unlike remote shell commands in Unix, where the remote machine must recognize and authorize the user to perform commands in its operating system, the Web provides essentially open access to anyone that requests a particular URL. Of course, this may present a security risk, and there are techniques for protecting Web data and programs, and the levels of protection are highly customizable by the administrator running the server.

Web servers may be run from a multitude of platforms, including Unix machines, Macintoshes, and Windows-based PCs. This makes the problem of handling different computing environments easier to deal with, as long as each environment support execution of Common Gateway Interface (CGI) programs from its server.

This research has investigated the use of the Web architecture to support remote execution and automation of several legacy codes. The codes are initiated by a simple URL call from a preceding Web page. The workings behind the URL will be discussed in Section 5.3. For the current time, all codes and Web pages are served off a single Unix-based server. In the future, however, any code may be moved to a separate server to demonstrate the possibility of distributed execution.

The Web architecture provides a familiar and attractive interface between the designer and the knowledge-base. As the world's computer users become increasingly interested in using Web-based services, it seems natural to extend engineering design into this environment. However, the field of Web-programming is a relatively new one, and therefore is still developing. Even with its obvious benefits, the Web architecture falls short in some areas. The Web was originally designed simply as a way to display information, and therefore provided limited means for the user to actually interact with that information. Only recently, with the advent of the Java and JavaScript programming languages, have users been able to manipulate the information they see in their browsers. Even these languages, however, have a long way to go before they can compete with the highly developed graphical interfaces provided in the Unix, Macintosh, and Windows development environments.

For most Web servers, the only database existing behind the interface is the Unix or PC filesystem containing the Web pages. In order to use the Web as an engineering environment, information must kept and tracked in a robust database. It does us no good to execute an analysis on the Web if the data it generates disappears as soon as we move to another Web page. It is certainly possible to connect say an object-oriented database behind a Web server, with Java and C++ help to provide the required hooks. However, an exercise such as this would require as much expertise as connecting the same type of database behind a graphical Unix/X-Windows interface.

Another major problem for a Web-based design environment is the inability to define an automatable process model. The Web was designed for human interaction, so it usually involves manually clicking buttons and selecting hyperlinks to move forward in a process. Unlike the Unix architecture, which can spawn processes when their preceding processes are finished, a Web server will typically just show you the results page, and wait. Again, methods have been and are being developed such as *Server Push* that address this issue, but they have some time before they fully mature.

4.2 Integrating Legacy Codes

Legacy resources, while providing high-fidelity results, are often not design-oriented. Design-oriented resources must be suitable for automated and repetitive use, provide consistent results, accept wide variance of input parameters, and often allow for visualization of output. A CAD system such as I-DEAS or ProEngineer is an excellent example of a legacy tool that would be hard to automate. Even if one could simulate mouse clicks on the screen (which can be done with a variety of software), how is one to know exactly where to click for every possible session at a CAD terminal. It is like saying that you could sit down in front of I-DEAS, close your eyes, and create or modify a part on the screen solely by knowing where your cursor is. Problems such as unexpected error dialogs, or slight displacement of on-screen objects from session to session, will make automating such a legacy tool quite cumbersome.

Now, many companies are designing into their software capabilities that allow for automating some processes, and this is a step in the right direction. What about those programs that are not receiving that sort of attention? Should we restrict ourselves to those codes that are already automatable at the expense of loosing fidelity or learning a new program? Probably not. So we must develop techniques ourselves to automate certain tools.

Another problem arises in interpreting the output from certain legacy tools. For example, a finite element code may churn through a rigorous analysis and output a large data file with element stresses and strains. In order to interpret the results, one usually takes the file into a piece of post-processing software for visualization. The operation of this post-processor acts much like a CAD system. Moreover, each post-processing session often follows a unique sequence since the results data set changes in each iteration, thereby making automation that much harder.

Even for a relatively simple command-line program, automation takes some skill. For example, the space design community often uses the trajectory analysis code POST (Program to Optimize Simulated Trajectories). The output of this code includes a single large text file containing the numerical trajectory results for the run. The numbers that the designer is interested in are located within the file, or derivable from numbers in the file, but extraction of the numbers usually requires loading the file in a text editor and manually picking out the required values. When POST is automated, this tedious task is performed completely by the computer.

4.3 Dissimilar Computer Platforms

Legacy resources may exist on a range of different computer platforms, thereby complicating the issue of integration and data exchange. POST is typically executed on Unix platforms, including SGI, Sun, and HP workstations. Propulsion and aerodynamics codes are also typically run in the Unix environment. In the SSDL, we also have spreadsheet based tools, including Weights & Sizing, and Economics & Cost. These run as Microsoft Excel files on a Macintosh personal computer.

It is important to respect the freedom that engineers have to develop analysis tools for their platform of choice. Whatever platform works best for their particular applications should be the one used, and we as integrators must take this fact and work with it. Of course, an Excel spreadsheet could theoretically be transformed into a C or FORTRAN program to be run in Unix, but what happens when the spreadsheet designer makes a modification to the tool? Is the spreadsheet then transformed each time an updated version appears? The most efficient route (albeit after the integration hurdles have been surmounted) is to retain the analysis tools in their native environment. It has therefore been a major thrust of this research to develop techniques for remotely executing tools that reside on PC platforms, from within a Unix environment. See Section 5.2 for more details.

4.4 Widely Distributed Resources

Often times, legacy resources may not all be accessible locally. A particular code may reside on a system in one part of the country, while the design team is situated in another. This is often the case with proprietary software. The software developer wants others to have access to his code from an execution standpoint, yet not actually keep the code on their own systems. This also exposes the possibility of a *distributed design team*. Suppose the disciplinary experts for a project are each located in a different location, while the systems engineers who specify requirements and monitor the design process reside in yet another location. If the design framework does not allow for distributed access to resource setup and analysis results, then the users are back to the problems of a loosely-integrated environment.

As discussed in the section regarding Web-based architectures, the Web provides a natural facility for accessing remotely located data and programs. There must also be a mechanisms in the Unix environment that allow for such remote access as well. In the RBCC IMAGE demo discussed in Section 6.1, one analysis tool (an Excel spreadsheet) was transported to NASA Marshall Space Flight Center (where the demonstration was being shown) and installed on a computer there. The Unix-based system executive, IMAGE, was still running off of a Sun workstation back at Georgia Tech, over a hundred miles away. However, after simple changes to the wrapping script, IMAGE was able to make an agent call to the spreadsheet located at Marshall.

5.

Specific Techniques

5.1 Wrapping Unix-based codes

As an example of a Unix-based command-line executed analysis tool, we look at POST (Program to Optimize Simulated Trajectories). This is a common tools used to analyze the ascent and reentry performance of conceptual launch vehicles. POST remains today as it was originally written decades ago, a FORTRAN program run from a Unix command-line, taking a single namelist formatted text input file (commonly referred to as the *input deck*), and generating several text output files. An expert carefully sets up the input deck for a nominal vehicle design.

When the trajectory portion of the vehicle design is to be converged with other analyses, or the converged vehicle is to be optimized, a few key design parameters in the input deck are changed. These may include **initial vehicle weight**, **aerodynamics reference area**, **engine specific impulse**, or **wing loading constraint**. Some design variables may be more than just a single number. For example, the engine specific impulse or aerodynamic coefficients may actually be tables of values, generated for different altitudes and/or mach numbers. In this case, the entire table is substituted into the input deck via a C-style #include statement.

After execution, certain key values are extracted from the output files. As the output file gives a complete history of the vehicle trajectory, a designer might be interested in extracting the **final burnout weight**, **maximum heat rate** or **wing loading** encountered during the trajectory. He may also extract certain numbers that help him calculate other performance values offline, such as **mass ratio** or **mixture ratio**. Data output from POST that is in tabular form may be plotted to gain visual insight into the vehicle's performance.

Defines the type of script to be interpreted (ksh, csh, perl, etc.).

Header block gives version and usage info, along with a command line example.

Directory and machine-name variables are assigned early to reduce the amount of editing if one should change.

The “C Pre-Processor” (cpp) is used to substitute variable values into an input deck template.

A remote shell command initiates POST on the SGI machine “atlas”.

Unix commands “grep” “tail” and “awk” are used to parse the output file for the desired output variables.

Small Perl scripts are used to perform calculations that ksh cannot handle (due to exponential notation). Notice the reuse of the $SCRIPTDIR variable.

Final results are printed to the terminal screen. If desired, the values could be piped to an external file, with or without the leading text.

```
#!/usr/bin/ksh

# Unix shell script to wrap POST for Hyperion
# Written by David Acton
# January, 1998
#
# Usage: run_hyp [input filename] [wing normal force limit]
#                [initial weight] [Sref]
# Outputs: final weight
#          mass ratio
#          mixture ratio
#          max wing normal force
#
# Example: run_hyp hyperion_LEO_20 -1.442915e6 824522.5 6302.54

RSH_MACHINE=atlas
SCRIPTDIR=/home/asdl2/issi/post/scripts/bin
POSTDIR=/home/asdl2/issi/post
DATADIR=/home/asdl2/issi/post

# substitute values into template
cpp -P \
       -D_DATADIR_=\'$DATADIR \
       -D_FAZB_=$2 \
       -D_WI_=$3 \
       -D_SREF_=$4 \
       $POSTDIR/$1.tmpl > $POSTDIR/$1.inp

# run POST
/usr/bin/rsh $RSH_MACHINE "cd $POSTDIR; $POSTDIR/p3d5 $1"

# Use these lines if using the custom POST output block
str1=`grep 'time ..* veli ..* gdalt' $POSTDIR/$1.out | tail -1`
str2=`grep 'xmin4..* xmin5..*' $POSTDIR/$1.out | tail -1`
wi=$3                                      # initial weight
wf=`echo $str1 | awk '{print $10}'`        # final weight
LOXused=`echo $str1 | awk '{print $12}'`   # LOX used
weicon=`echo $str2 | awk '{print $2}'`     # weight consumed
xmin4=`echo $str2 | awk '{print $2}'`      # max wing normal force
xmin5=`echo $str2 | awk '{print $4}'`      # max wing normal force

# Calculate mass ratio
mr=`$SCRIPTDIR/calc_mr.pl $wi $wf`

# Calculate mixture ratio
r=`$SCRIPTDIR/calc_r.pl $LOXused $weicon`

#calculate max normal force ratio
wnRatio='$SCRIPTDIR/calc_wn.pl $wi $xmin4 $xmin5'

# Write output results (for command-line execution)
echo "Initial weight  = $wi"
echo "Final weight    = $wf"
echo "Weight consumed = $weicon"
echo "LOX used        = $LOXused"
echo " "
echo "Mass ratio      = $mr"
echo "Mixture ratio   = $r"
echo "Max wing normal force ratio = $wnRatio"
```

Figure 10. Korn shell script used to automate the setup, execution, and post-processing of POST.

An "agent wrapper" was designed for the POST program that automates most of the procedures described above. The wrapper takes the form of a Unix Korn Shell (ksh) script, executed at the command line with several arguments. The ksh script is convenient since it makes use of Unix commands and syntax that most users are already familiar with. Drawbacks include limited capabilities, especially with math functions and code reuse. Figure 10 shows the key portions of this script, while Figure 11 shows a schematic of the execution process. An input deck template is populated with the input variables provided on the command-line (initial weight, reference area, etc.). The resulting file is given a ".inp" extension and submitted to POST on an SGI machine. Once POST has finished, the output file is parsed for the last instances of certain key variables. These values are used to calculate the desired output quantities (mass ratio, mixture ratio, etc.), which are then sent to the user terminal or piped to an external file.

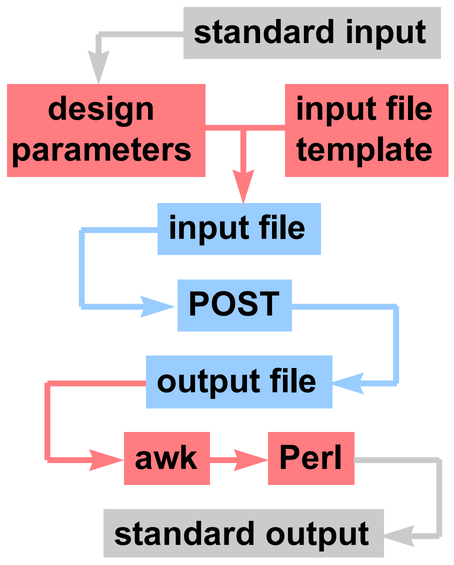


Figure 11. Execution process for an automated Unix command-line analysis.

This same procedure may be used for a variety of input/output file analysis codes. For example, the Georgia Tech-developed RBCC propulsion code SCCREAM is wrapped similar to POST. In SCCREAM's case, however, the script is written in Perl, and the input files are generated in full by the script instead of by a template.

The problem that arise when automating a program such as POST is that errors during runtime are difficult to catch. POST is notorious for its ambiguous run failures. If a vehicle does not achieve orbit, the user is given little indication of why not, and the calculated mass and mixture ratio may be meaningless. The script itself has essentially no intelligence with how to deal with problems that arise. Research that is being conducted on expert systems as SSDL may help to alleviate this problem in the future.

5.2 Wrapping PC-based codes

Many of the mechanisms for communicating between Unix codes has been established over the years. However, the problem of integrating a personal computer, whether a Macintosh or Windows-based machine, presents a much larger challenge. As an example of a Macintosh-based analysis tool, we consider the Cost And Business Analysis Module (CABAM), developed in the SSDL. This Microsoft Excel spreadsheet takes a description of a particular launch system, along with assumptions about the launch market, cash handling, and production and operations systems. It provides estimates for vehicle development costs, recurring costs, cash-flow, and rate of return on investment.

A wrapping script was developed to automate, and ultimately integrate the CABAM spreadsheet into a design environment. In order to provide the communications link between the Unix environment and the Macintosh operating system, two key pieces of pre-existing software were used. Peter’s Scripting Daemon provides a simple Telnet protocol server to allows a user to connect to the Macintosh. Once connected, the user may issue commands to execute AppleScript procedures. AppleScript is an automating language built into most Macintosh computers. Therefore, once the appropriate AppleScript scripts have been written, a user can sit at a Unix terminal, and interact with any number of programs on the Macintosh. In this example, AppleScript is used to open Microsoft Excel with the CABAM spreadsheet, change approximately 15 cells corresponding to vehicle component weights, read the resulting estimate of internal rate of return (IRR), then close the spreadsheet.

Defines the type of script to be interpreted (expect).

```
#!/usr/local/bin/expect -f
```

Header block gives version and usage info.

```
#
# Script to automate the iteration of the Excel CABAM spreadsheet.
# Usage:  iterate <infile>
#    infile    file containing program and iteration parameters
#
# Written by:  David Acton, November 4, 1997
```

Command-line arguments are read in, and the iteration parameters are read from the input file (cabam.in).

```
# ******** Set program parameters ************************

log_user 0
set infile "[lrange $argv 0 0]"
if ![string length $infile] then {
       send_user "Error: No input file specified \n"
       exit
}
source $infile
```

The Telnet process is spawned, and the user is logged in using Expect’s “expect/send” syntax.

```
# ******** Start up Telnet ************************

# Telnet to Macintosh host designated by $host
set env(TERM) vt100
spawn -noecho telnet $ipAddress
# Wait for login prompt, then send username
expect {Username*}
send "$user\n"
# Wait for password prompt, then send password
expect {Password*}
expect -timeout 1
send "$password\n"
# Wait for > prompt
expect {>*}
```

Peter’s Scripting Daemon commands are sent to initiate Microsoft Excel with the desired spreadsheet. $scriptfile1 contains the AppleScript commands required to interact with Excel.

```
# ******** Get Excel ready ************************

# Send the /exec command and wait for ]
send "/exec\n"
expect {]}
# Get the contents of the AppleScript file
source $scriptfile1
# Send . to conclude exec
expect {]}
send ".\n"
# Wait for > prompt
expect {>*}
send "\n"
expect {>*}
```

The contents of $scriptfile2 tell AppleScript to insert vehicle component weight values in their corresponding cells.

```
# ******* Iterate the spreadsheet ************************

# Send the /exec command and wait for ]
send "/exec\n"
expect {]}
# Get the contents of the AppleScript file
source $scriptfile2
# Send . to conclude exec
expect {]}
send ".\n"
```

(Continued… See Appendix)

Figure 12. ‘Expect’ shell script used to automate the setup, execution, and post-processing of CABAM.

The procedure described above is fine when a user is sitting at the Unix terminal issuing the AppleScript commands, but what about automating the process. Telnet is a program designed for user interaction, so a scripting language called "Expect" was utilized to automate the Telnet session. Expect simulates user interaction by "expecting" certain characters to be sent back to the terminal, then "sending" the other characters back. It can be used to automate "ftp" sessions, and a host of other Unix programs that rely on user keystrokes.

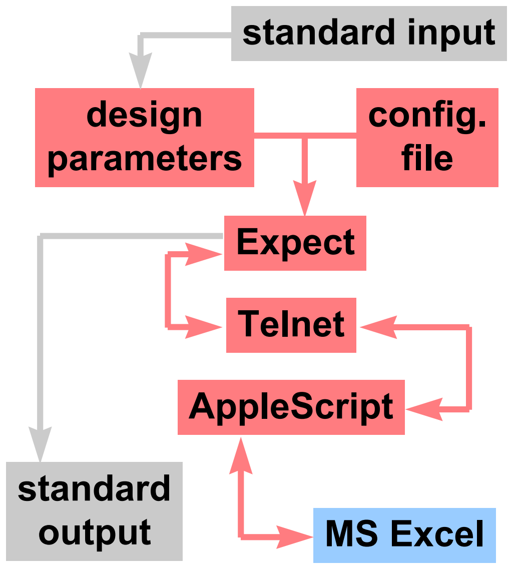


Figure 13. Execution process for an automated Macintosh-based analysis.

Figure 12 shows the first portion of the CABAM wrapping script, written in the Expect scripting language. The general process for automating an Excel-based tool on a Macintosh is shown in Figure 13.

5.3 Developing web-based interfaces

The benefits of Web execution of analysis tools has already been discussed in Section 4.1.2. The precise techniques to developing a web interface to a tool are explained in the current section. The Web interfaces developed as a part of this research essentially build on top of the scripting techniques described in the previous two sections. These scripting techniques were originally designed to be used from the command-line, but now it makes sense to create a web interface to the tools so that other users from the launch vehicle design community can make use of them.

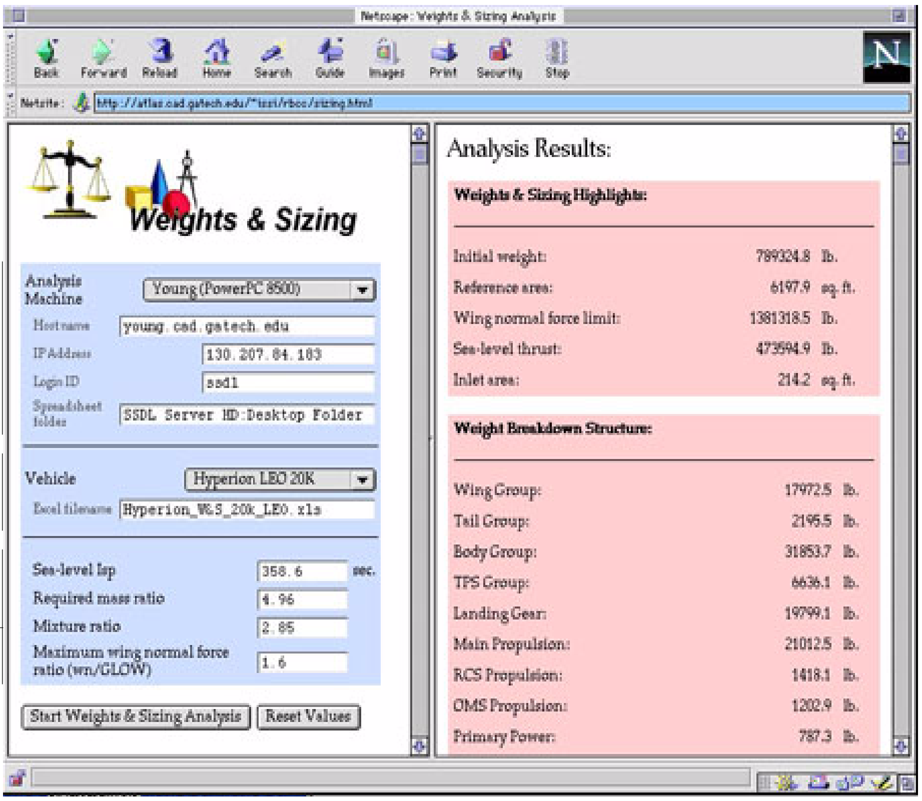
As an example of an analysis tool that had a Web front-end applied to it, we choose the Weight & Sizing Excel spreadsheet tool (WATES). This spreadsheet is used to calculate the various component weights of a conceptual launch vehicle. It uses a combination of physics-based and empirically-based calculations, and relies on information from a variety of different disciplines including Trajectory, Propulsion, and Configuration & Layout. Since the WATES spreadsheet resides on a Macintosh computer, it was wrapped in a manner similar to CABAM, using the Expect and AppleScript languages.

In order to transition such a wrapper to a Web-based interface, an HTML input form was generated (See Figure 14). This form provides the user with pull-down menus and input boxes to enter the required inputs for the analysis. The user may select which of several Macintosh machines to connect to, as well as the name of the file to be loaded. Behind the input form lies some relatively simple JavaScript code that sets certain default values and performs range checking on the user inputs. When the user submits the HTML form by clicking the “Submit” button, a Perl CGI script parses through the variable string generated by the HTML form, picking out the relevant input values. The Perl script writes out the appropriate AppleScript commands to temporary files. Perl then executes a modified version of the Expect WATES wrapper, which in turn spawns a Telnet process to the Macintosh to perform the AppleScript commands. When the Expect script closes the Telnet connection and sends output results to the user, it sends them embedded in HTML code directly to the right frame of the browser window. In the case of WATES, the output is simply a series of numbers defining the weight-breakdown of the vehicle, but this by no means limits the types of output that may be displayed. Some analyses may output such items as image files containing plots, and these are easily displayed in a Web browser window.

The user may select which Macintosh to connect to along with the name of the folder to be accessed.

The name of the spreadsheet to be opened is also user selectable.

Input variables assume a default value when the page is loaded, but the user is free to change any of them to suit his or her particular design.


Netscape: Weights & Sizing Analysis
Back
Forward
Reload
Home
Search
Guide
Images
Print
Security
Stop
Netsite:
http://atlas.cad.gatech.edu/~lzri/rbcc/sizing.html
Weights & Sizing
Start Weights & Sizing Analysis
Reset Values

Analysis Machine	Young (PowerPC 8500)
Host name	young.cad.gatech.edu
IP Address	130.207.84.183
Login ID	ssdl
Spreadsheet folder	SSDL Server HD:Desktop Folder
Vehicle	Hyperion LEO 20K
Excel filename	Hyperion_W&S_20k_LEO.xls
Sea-level Isp	358.6 sec.
Required mass ratio	4.96
Mixture ratio	2.85
Maximum wing normal force ratio (wn/GLOW)	1.6

Analysis Results:

Weights & Sizing Highlights:

Initial weight:	789324.8	lb.
Reference area:	6197.9	sq. ft.
Wing normal force limit:	1381318.5	lb.
Sea-level thrust:	473594.9	lb.
Inlet area:	214.2	sq. ft.

Weight Breakdown Structure:

Wing Group:	17972.5	lb.
Tail Group:	2195.5	lb.
Body Group:	31853.7	lb.
TPS Group:	6636.1	lb.
Landing Gear:	19799.1	lb.
Main Propulsion:	21012.5	lb.
RCS Propulsion:	1418.1	lb.
OMS Propulsion:	1202.9	lb.
Primary Power:	787.3	lb.

When the form is completed, the user clicks this button to submit the analysis for execution.

Results are displayed in a convenient and attractive HTML frame. The content need not contain only textual data, but also image data.

Figure 14. Web-interface to the WATES Weight & Sizing analysis.

The HTML form/JavaScript interface method was used to wrap four important analyses in the SSDL. These include the trajectory code POST, the propulsion code SCCREAM, the economics and cost spreadsheet CABAM, and the weights & sizing spreadsheet WATES. Figure 15 shows the general scheme in which a stand-alone code may be wrapped into a Web-based resource for use by a single user or a Web-based system executive.

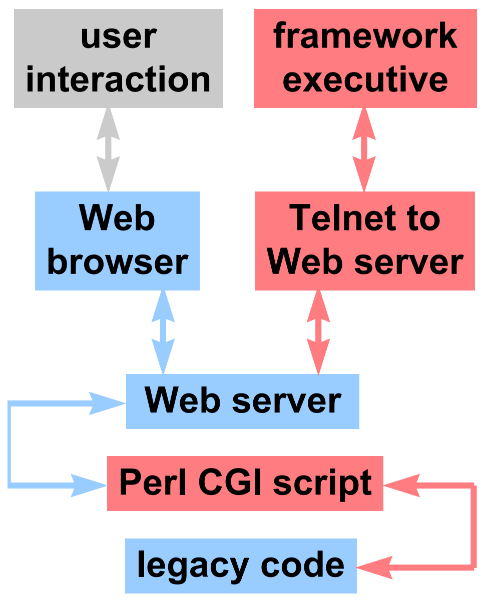


Figure 15. User interaction and framework integration of a Web-based analysis.

6. Proofs of Concept

Of course, the research that has been performed in automation and integration techniques will be of no good unless we prove that they aid in the process of engineering design. To that end, several demonstration problems were set up that make use of the techniques described in Section 5.

6.1 RBCC Demonstration (Unix)

The IMAGE architecture was the basis of a four-code demonstration of the design process for a reusable launch vehicle based on a Rocket-Based Combined-Cycle (RBCC) engine. The vehicle chosen was a vision vehicle being studied in the SSDL named *Hyperion*. The four disciplines involved were Propulsion, Trajectory, Weights & Sizing, and Economics & Cost, with the corresponding analysis tools SCCREAM, POST, WATES, and CABAM. The purpose of the demonstration was to show how agents can be integrated in the IMAGE environment, automatically executed according to a specified design process, and iterated to convergence without any user interaction. A diagram of the setup and execution process for this particular problem is shown in Figure 16. The underlying wrapping technologies utilized in this demonstration were those discussed in Sections 5.1 and 5.2, to enable both Unix and Macintosh-based analysis tools.

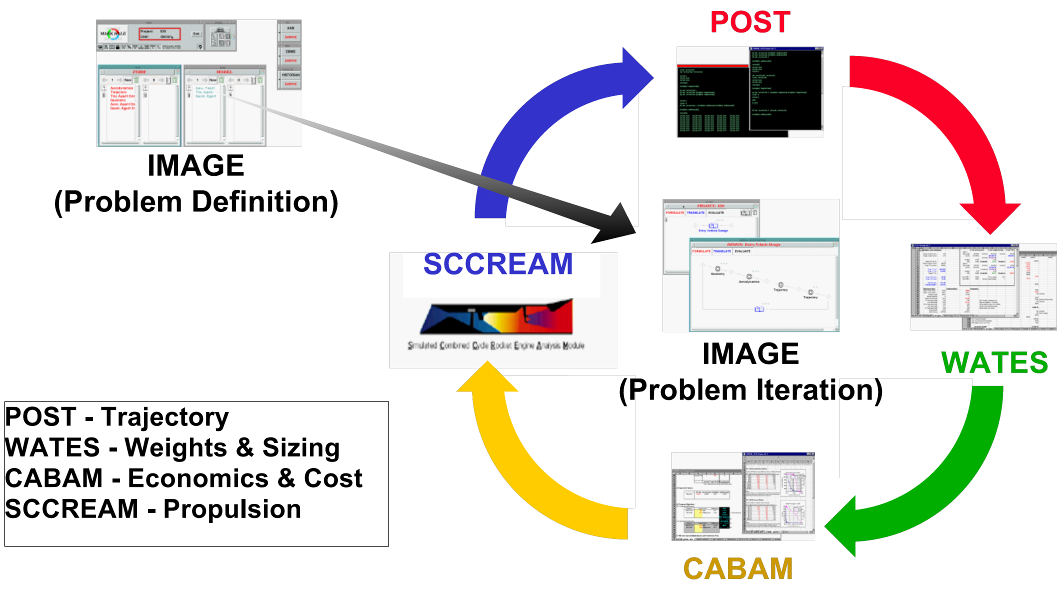


Figure 16. Setup and execution process for the four-code RBCC demonstration problem.

This exercise was conducted as part of a cooperative study between Georgia Tech and a Huntsville, Alabama firm named International Space System Incorporated (ISSI). The study is funded through NASA's Marshall Space Flight Center, where the final demonstration will be presented. As of the writing of this paper, the demonstration had not yet been given.

6.2 *RBCC Demonstration (Web)*

The *Hyperion* RBCC demonstration described above was also transitioned into a Web-based format. The user is presented with a main selection page that allows the user to perform each of the four analyses (See Figure 17). When the use clicks on one of the images, the corresponding analysis is spawned as a Web-based HTML form interface to the code (as described in Section 5.3).

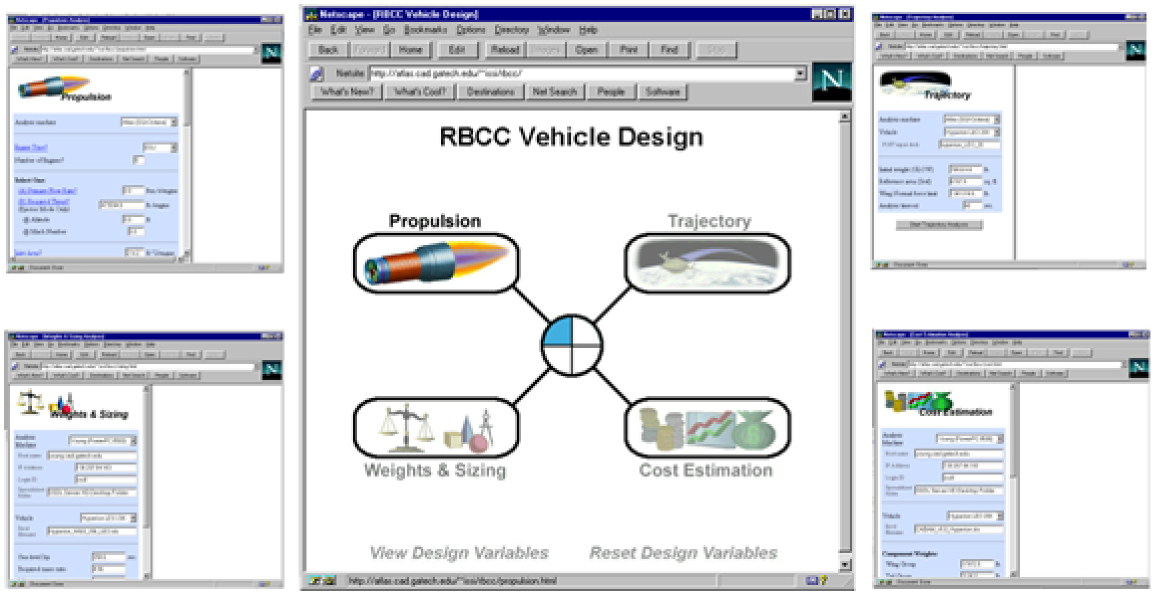


Figure 17. Web-based RBCC demonstration problem, showing main interface, and corresponding analysis pages.

It would appear that this demonstration is simply a collection of the previously developed Web interfaces, but the key difference here is the true interaction between the analyses. When each HTML form is generated, the default input variable values are read from an external database file. When each analysis is completed, the output variable values are written to this same database file, so that subsequent analyses may use the updated numbers. In this way, all four analyses may be iterated consecutively until convergence is reached. Currently, the environment does not allow for optimization of the design. The major drawback to this interface is the requirement that the user be present to initiate each analysis. So in a sense, we have achieved one of our major goals for a design framework, that is information exchange, but we have not achieved the complete autonomy phase yet.

In this RBCC Web demonstration, the IMAGE architecture was essentially replaced by a single main selection page. Of course, the Web-interface should not be compared directly with IMAGE at this stage, since it contains very few of the characteristics that would qualify it as a true architecture. After all, IMAGE is not only a mechanism for executing analyses, but rather an environment in which the entire design problem is set up. Form, Model, and Process schemas, as well as the design process, are set up within IMAGE itself. For the Web demonstration, the input forms, design variables, and data exchange were all set up by hand in the basic Unix filesystem environment. With future research into Web-based system executives, however, a more meaningful comparison could be made.

6.3 Planetary Demonstration (Unix vs. Web)

The third demonstration that was conducted compared the IMAGE architecture itself with a pre-existing Web-based design environment developed at NASA Langley Research Center. The Integrated Design System (IDS) originated by NASA provides for the conceptual design of a planetary entry vehicle using HTML form interfaces, Perl scripting behind the scenes, and SGI executables for its three primary disciplines, Geometry, Aerodynamics, and Trajectory. The IDS system is quite similar to SSDL's own RBCC Web demonstration problem in how it is executed. The user is presented with a series of pages, where he selects appropriate input parameters. Perl CGI scripts execute the Unix codes and present the output data to the screen. This data consists of 2-D and 3-D geometry descriptions, aerodynamic coefficient plots, and trajectory performance plots (See Figure 18). The IDS system is a "once-through" design process, and does not currently allow for any sort of iteration.

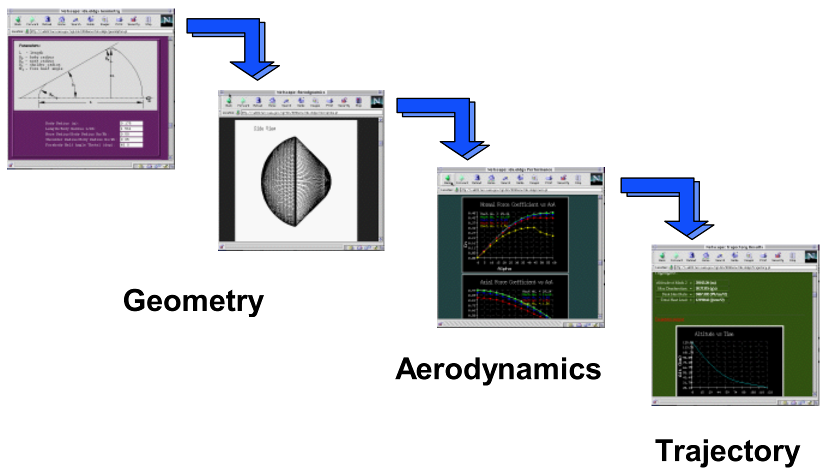


Figure 18. Web-based IDS planetary entry vehicle design process.

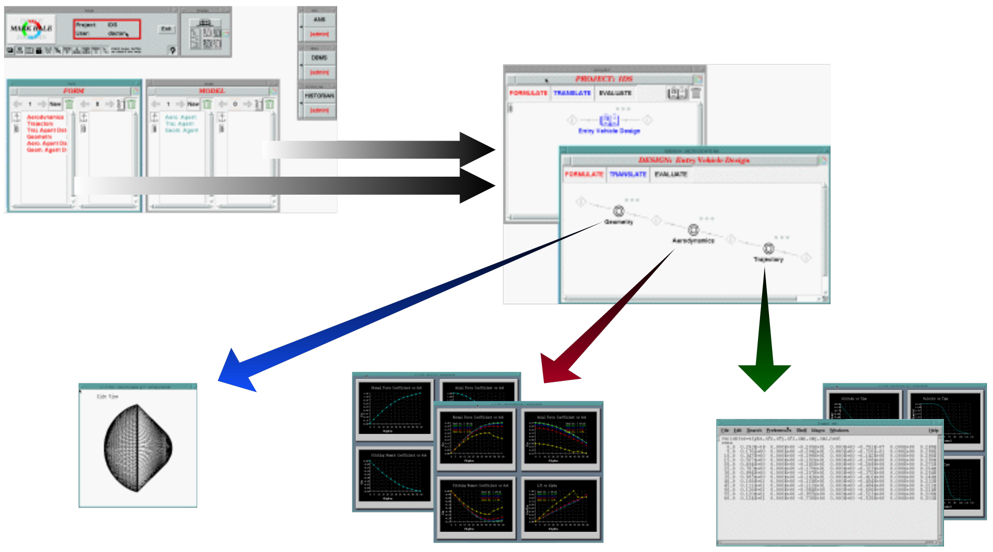


Figure 19. Planetary entry vehicle design process implemented in IMAGE.

As a part of this research, the IDS executable codes and CGI scripts were obtained and modified to perform within the IMAGE architecture. The Form, Model, and Process schemas were constructed for the planetary entry problem, and a sequential execution process was defined (See Figure 19). The intention was to show how the exact same process could be duplicated within an alternate architecture, and compare the two approaches on the following basis:

〈 Ease-of-setup - how quickly and efficiently can one set up an engineering design problem in each architecture?
〈 Ease-of-use - once the problem is set up, which is more convenient to actually execute?
〈 Robustness of environment - how stable is the architecture when codes go awry?
〈 Expandability - how easy is it to change the problem once you've set it up, and how easy is it to add new agents?
〈 Automatability - once the problem is set up, does the user still need to be present?

IMAGE certainly has the edge when it comes to problem setup. The built-in object-oriented data model, while sometimes cumbersome to work with, certainly outperforms setting up and linking individual scripts at the Unix level. Regarding ease-of-use and automatability, IMAGE requires little supervision if set up properly, and allows for relatively easy visualization of past results. However, the IDS interface feels more friendly to the designer since he can actually see what is happening with his analyses. He may also choose to modify a certain portion of his design mid-process based on data from a previous analysis. In our IMAGE version, the user cannot make design decisions while the process is executing. As a research architecture, IMAGE is not the most robust piece of software, and it requires a decent level of Unix knowledge simply to get an entire problem running. During execution itself, however, the system is quite reliable. Presumably, the IDS system took a good deal of time to set up properly, so most if not all bugs are worked out. Therefore the system runs quite smoothly. When it comes to expandability, however, both architectures fall short. In the current version of IMAGE, if you want to make a simple change to the design process, or add another analysis, you must completely redefine the process from the beginning. Future versions should allow easier modification however. The IDS system was really not intended to be completely expandable, as it would require deep editing of Perl scripts and HTML forms to generate a new analysis interface or restructure the design process.

7. Conclusions and Future Research

The IMAGE architecture provides a convenient and accessible architecture within which to deploy integration technologies such as those developed in this research. The detailed process model should also allow for more complicated design structure matrices to be implemented, and possibly the use of Multidisciplinary Design Optimization (MDO) techniques. Applied to a vehicle such as *Hyperion*, such a system could save countless hours of tedious code execution, and may ultimately arrive at a more optimal design.

From the feedback received regarding our own Web-based interfaces, and the results of the IDS vs. IMAGE comparison, it appears obvious that significant research should be focused on continuing the development of design architectures deployed on the Web. Even if the problem itself is developed offline in an IMAGE sort of environment, the familiar and content-rich Web interface will appeal to many engineers. The Web also provides easy access to distributed platforms, and enables distributed design teams. Mark Hale, developer of IMAGE, is currently investigating extending IMAGE to export a design process to the Web.

Additional technologies should be developed to build the library of tools accessible in a tightly-integrated design framework. Currently, we can automate and integrate Unix-based command-line codes, and Macintosh-based analysis tools. Further study should be performed to look at communicating with Windows-based PCs, as well as complicated Unix tools such as finite-element analyses and CAD environments. As the design methodologies developed here for conceptual design begin to take hold, they will inevitably trickle down into the preliminary and detail design phases. In these phases, there will be many analyses and tools that require special wrapping techniques that have not yet been developed. Obviously, the area of computational framework research is one that has a strong future ahead of it.

References

1. Hale, M.A., Craig, J.I., “Techniques for Integrating Computer Programs into Design Architectures,” Sixth AIAA / NASA / USAF / USSMO Symposium on Multidisciplinary Analysis and Optimization, Bellevue, WA, September 4-6, 1996.

2. Hale, M.A., “An Open Computing Infrastructure that Facilitates Integrated Product and Process Development from a Decision-Based Perspective,” Doctoral Thesis, School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA. July 1996.

3. NASA Langley Vehicle Analysis Branch IDS System, URL: “http://vab02.larc.nasa.gov/IDSDemo”

4. Olds, J. R. “System Sensitivity Analysis Applied to the Conceptual Design of a Dual-Fuel Rocket SSTO,” AIAA 94-4339, 5th AIAA / NASA / USAF / USSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, September 1994.

Acknowledgments

This research is funded both by the Vehicle Analysis Branch at NASA Langley Research Center, and by NASA Marshall Space Flight Center. The author would like to thank Dr. John Olds for his guidance and support throughout this research, and the members of the Space Systems Design Laboratory at the Georgia Institute of Technology for their assistance.

10.