CONTENTS

Introduction

Development of the framework

  Understanding analysis module

  Data Server

  Limitations

  Application Programming Interface

  Command line interface to data server

Requirements for MDO framework

  Architectural Design

  Problem Formulation Construction

  Problem Execution

  Information Access

Readme

(Following Appendices are prepared by MR Sankar, CASDE / CABS)

Appendix-I    A Demo System Analysis File in C

Appendix-II   A Demo Sub-system analysis in FORTRAN
Introduction

As part of the initiative taken by Center for Aerospace Systems Design and Engineering (CASDE) in IIT Bombay, framework for Multi-disciplinary Design Optimization (MDO) is being developed. This framework will enable to perform MDO studies with ease and using all the available resources.

The goals for this framework are:

- Integrate available disciplinary analysis modules to set up system analysis
- Integrate optimizers with system analysis
- Develop tools and user interfaces for easy problem setup
- Build a distributed computing architecture to use the heterogeneous computing resources available over the network
- Provide tools for visualizations of the solution space
- Provide easy debugging and maintenance of the framework

The research done worldwide in the area of MDO has enabled us to understand the general requirements for developing such a framework. These requirements have also been taken into consideration while designing the MDO framework. These requirements are discussed in detail later in the document.

Development of the framework

The issues that were focused in the first phase were

- Providing mechanism for exchange of data between different disciplinary analysis modules
- Integrating available batch mode optimizers with system analysis to perform simple design optimization studies

To simplify the initial design of the framework following assumptions/constraints were introduced.

1. Source code availability for analysis modules.
   This constraint allows us to modify the source code for analysis modules for integration.
2. All codes are running on single computer.
   This assumption removes the complexities of network communication and allows focusing on other issues at hand.
**Understanding analysis module**

Once we have identified various disciplinary analysis modules that are to be integrated for performing a system analysis, it is essential to understand the nature of analysis module from the point of view of data exchange. Each analysis module can be split into three components viz. input processing, disciplinary analysis and output processing.

![Figure 1: Anatomy of Analysis Module](image)

Usually in any analysis module the input processing is done at the beginning of the analysis module and output processing at the end of the analysis module. Though this is not a requirement it allows us to design interfaces easily.

To exchange data between two analysis modules it is necessary to setup a link between these analysis modules. This is easy when we know which two analysis modules need to exchange data. When developing a framework such information is not available. Therefore it is required to introduce a third entity, which will handle the data exchange requests and creates a data exchange link between the analysis modules. This entity is designated as **Data Server**.

**Data Server**

Data server is designed as the central repository for storage of data that needs to be exchanged between analysis modules. The design of data server is stateless, so it does not have to store information about analysis modules exchanging the data.

Since data server is a separate process, all the analysis modules have to communicate using Inter Process Communication (IPC). This communication is abstracted into a library with easy to use interface.
As seen from the figure 2, analysis module reads required input from the data server and writes the output back to the data server. So data server holds all the inputs and outputs required for different analysis modules, thus creating an infrastructure for automatic exchange of data.

**Limitations**

1. Each input and output variable is identified with a character string label. This label has to be unique in the context of the system analysis. Which means two analysis modules cannot have the same label for different variables.

2. Since the data exchange is automated by binding the two variables with the same label, it is necessary that variables that are exchanged should have the same label across the system analysis. (If two different system analyses need different labels for the same variable of particular analysis module, then two separate wrappers would have to be written for that analysis module.)

These limitations are due to the current implementation of data server. They would be removed when data server has additional knowledge of the label linking.

**Application Programming Interface**

The data server and associated communication library has been designed to work with any flavor of Unix. It is written in ANSI C and easily portable. Data server interface bindings are provided for C and FORTRAN languages.
Initializing the network communication

subroutine finitdsn [FORTRAN]
void init_dsn(void) [C]

This subroutine initializes the network communication parameters. It should be called once in analysis module before reading input data or writing output data.

Closing the network communication

subroutine fshutdsn [FORTRAN]
void shut_dsn(void) [C]

This subroutine shuts down the network communication. It should be called in the end before exiting the analysis module.

Reading input data

subroutine freadint(label, length, value) [FORTRAN]
character *16 label
integer length
integer value

subroutine freadreal1(label, length, value) [FORTRAN]
character *16 label
integer length
real value

subroutine freadreal2(label, length, value) [FORTRAN]
character *16 label
integer length
double precision value

void read_int(char *label, int *value) [C]
void read_real1(char *label, float *value) [C]
void read_real2(char *label, double *value) [C]

These subroutines are used to read the variables from data server depending on the data type. Label is a character string of up to 16 characters. Length is the length of the character string of the label. Value is the place holder (variable name local to the calling routine for the data represented by the label) for the value returned by the data server.

Writing output data

subroutine fwriteint(label, length, value) [FORTRAN]
character *16 label
integer length
integer value

subroutine fwritesreal1(label, length, value) [FORTRAN]
character *16 label
integer length
real value

subroutine fwritesreal2(label, length, value) [FORTRAN]
character *16 label
integer length
double precision value

void write_int(char *label, int value) [C]
void write_real1(char *label, float value) [C]
void write_real2(char *label, double value) [C]

These subroutines are used to write variables to the data server. Label is
character string of up to 16 characters. Length is the length of the character
string for the label. Value holds is the local variable name that holds the data to
be associated with the label.

The data server is designed to handle different data types and varying
dimension. The current API limits the data exchange of single values of
dimension zero. The new API supporting arrays would be available soon.

Command line interface to data server

There is a command line interface tool provided to query the data server. This
tool can list all the variables held by the data server, read and write data to the
data server. This tool is provided to assist the developer to verify correct usage of
the API. This tool can also be used to supply the initial data required to execute
analysis modules.

Listing all variables from data server
dscommand list

Writing a variable to data server
dscommand write <label> <type> <value>
type := INT | REAL1 | REAL2

Reading a variable from data server
dscommand read <label>
Requirements for MDO framework

It is essential that the framework designed to perform MDO studies should be flexible and extensible to accommodate multiple analysis modules and various computing resources available. Following requirements have been put forth on the MDO framework to ensure smooth functioning of the same.

Architectural Design

• All the framework functions should be controlled by a graphical user interface (GUI). This GUI should be easy to use to facilitate easy learning by the user. Intuitive design helps the user to utilize the full functionality of the framework.
• The framework design should be governed by object-oriented principles.
• The framework should be extensible. It should be easy to define new interface to integrate new programs and optimizers into the framework.
• As a result of the extra layers in the framework, it should not impose unreasonable amount of overheads in the execution.
• Framework should be able to handle large size problems. The number of design variables defines the size. Typically framework should be able to handle few hundreds to thousands of design variables.
• MDO is collaborative effort. Therefore framework should provide mechanisms for easy data exchange between multiple users and provide simultaneous data access to various parts of the framework.
• To be able to sustain the development effort and to ensure the safety of investment, the framework should be designed using standards.

Problem Formulation Construction

• As the design problems are coupled systems, the framework should support complex branching and iterative execution schemes.
• The problem definition is usually quite huge and hence it should be easy to reconfigure the problem to add new disciplines or integrate newer optimization techniques to the existing problem.
• There are a large number of legacy and proprietary codes developed in multiple languages and on multiple systems. To be able to make use of the user familiarity, the framework should be able to use existing programs.
• The best method for optimization is unknown for most of the problems. Therefore it is necessary to be able to utilize different optimization methods and framework should support it.
• As the analysis modules are executing on different computers, it is important to provide debugging support for multiple processes running across the network. It helps identify the problem in problem definition.
Problem Execution

- The framework should automate the execution of analysis modules and movement of data between processes.
- Many of the analysis modules are compute-time-intensive. The framework should use the course-grain parallelism to execute multiple processes in parallel without affecting the final result.
- To be able to use the available computing resources on the network, the framework should run on heterogeneous computers.
- In the problem definition it is essential that user can interact with the system and modify the problem definition, as he seems fit. The framework should support user interaction (steering) during design cycle.
- The framework should support execution of a problem in batch mode. This facilitates exploring the design space by specifying different starting points.

Information Access

- The framework should provide database management features.
- Framework should have the capability to visualize intermediate and final optimization and analysis results.
- Framework should have monitoring capability for viewing the status of an execution and system status.
- Mechanisms for fault tolerance.
This document is supplied along with,
- A zip file containing installable 'dataserver' code
- demo code for a system analysis in 'C' (Appendix-I)
- demo code for one sub-system analysis in 'FORTRAN' (Appendix-II)

To compile dataserver and the interface library.

*type 'make'*

It will build

- dataserver (data server executable)
- libdsnetwork.a (data server interface library)
- dscommand (data server command line interface)

To build analysis module with dataserver interface, it has to be linked with the dataserver network library.

*cc -o analysis analysis.c -L/path/to/dataserver -ldsnetwork*
*f77 -o analysis analysis.f -L/path/to/dataserver -ldsnetwork*
A Demo System Analysis File in C

By
MR Sankar, Graduate Student, Aerospace Engineering, IIT Bombay. Scientist CABS, Bangalore

The system analysis file is implemented (in our case) in ‘C’. However, it can be in any language. This file integrates all subsystem analyses and presents them together. Then defines the sequence or logic on which the subsystem analyses are to be executed (and calls them). The structure of the system file is given below.

Starting data server application, starting and stopping data server communication and format (& options) of read/write functions are given in the accompanying write up on framework and data server.
#include <stdio.h>
#include <interface.h>

void
analysis(char *path)
{
  int status;

  /* fprintf(stderr, "Executing %s ...", path); */
  status = system(path);
  if(status != 0) {
    fprintf(stderr, "Analysis error status=%d\n", status);
    exit(status);
  } else {
    /* fprintf(stderr, "done\n"); */
  }
}

int
main(int argc, char *argv[])
{
  /* declare system level variables */
  double h, v, p, cl, thr;
  double d, t, lift, en;
  int standalone=0;

  /* Pass an argument to run the system analysis in standalone mode */
  if(argc == 2) standalone = 1;

  if(standalone) {
    printf("Enter the values of h, v, p, cl, thr : ");
    scanf("%lf %lf %lf %lf %lf", &h, &v, &p, &cl, &thr);
  }

  /* Initialize the variables */
  init_dsn();

  /* In 'C' format is write_real2("variablename_tobestored",value) */
  /* No need to give string length. */

  /* Common data */
  /* System */
  write_real2("pi",3.1415);
  write_real2("g",9.81);
  write_real2("rpm",6.);

  /* Subsystem 1 */
write_real2("r", 150.);
write_real2("rcs", 2.);
write_real2("f", 3.0e9);
write_real2("snr", 12.5);
write_real2("kt0", -204.);
write_real2("bw", 2.5e6);
write_real2("ls", 10.);
write_real2("nf", 3.);
write_real2("np", 24.);
write_real2("cpi", 8.);
write_real2("an_eff", 0.25);
write_real2("bbf", 1.4);
write_real2("df", 0.07);
write_real2("prf", 5.0e3);
write_real2("w_area_ant", 345.2);

/* Subsystem 2 */
write_real2("w_area_rot", 74.743);

/* Subsystem 3 */
/* Subsystem 4 */

/* Design variables */
if(standalone) {
    write_real2("h", h);
    write_real2("v", v);
    write_real2("p", p);
    write_real2("cl", cl);
    write_real2("thr", thr);
}
shut_dsn();

/* Sequence and call the subsystem routines, each will be an executable */
/* System Analysis */
analysis("./SS1");
analysis("./SS2");
analysis("./SS3");

/* Output */
if(standalone) {
    init_dsn();

    /* Format for reading is read_real2("variablename_in_dataserver",
&variablename_tobestored)*/
    read_real2("d", &d);
    read_real2("t", &t);
    read_real2("lift", &lift);
    read_real2("en", &en);
shut_dsn();

printf("Drag = %e\n", d);
printf("Thrust = %e\n", t);
printf("Lift = %e\n", lift);
printf("Endurance = %e\n", en);

exit(0);
A Demo Sub-system analysis in FORTRAN

By
Scientist CABS, Bangalore

The subsystem analysis file defines the subsystem model and contains all the necessary formulations (accompanying subsystem file is implemented in FORTRAN, since that how it was available, and the language is retained).

Subsystem analyses have as inputs local variables (considered constants at system level), system level variables and coupling variables (values computed by this analysis and required by another analysis as input). If there were no common data storage mechanism in place, then all these data exchanges would have taken place through the system level data sharing mechanism (eg. COMMON BLOCKs). Since a data server has been implemented, the data exchange is done through the same.

Subsystem file reads the all the above variables from the data server, carries out necessary function evaluation either locally or calling necessary codes/software and writes back the results (these are typically coupling variables to be supplied to other subsystems). Data exchange is similar to system level implementation.

Subsystem analysis file

Data server

Read subsystem data (constants), system level variables required for the analysis and coupling variables

Analysis routines executed locally or external codes called

Write back the results to the data server.
Sub-system analysis code

program main

double precision p, l_ant, b_ant, w_ant, w_rsys, q

common data
double precision pi, g, rpm
common /syscommon/ pi, g, rpm
double precision r, rcs, f, snr, kt0, bw, ls, nf, np, cpi,
               1an_eff, bbf, df, prf, w_area_ant
common /sysradar/ r, rcs, f, snr, kt0, bw, ls, nf, np, cpi,
               2an_eff, bbf, df, prf, w_area_ant

start dataserver communication
call finitdsn

--------Input header---------
"---------------------------"
--- read data from dataserver -----
Format freadreal2(variablename_in_dataserver,stringlength,
                   variablename_tobe_stored)

call freadreal2("pi", 2, pi)
call freadreal2("g", 1, g)
call freadreal2("rpm", 3, rpm)
call freadreal2("r", 1, r)
call freadreal2("rcs", 3, rcs)
call freadreal2("f", 1, f)
call freadreal2("snr", 3, snr)
call freadreal2("kt0", 3, kt0)
call freadreal2("bw", 2, bw)
call freadreal2("ls", 2, ls)
call freadreal2("nf", 2, nf)
call freadreal2("np", 2, np)
call freadreal2("cpi", 3, cpi)
call freadreal2("an_eff", 6, an_eff)
call freadreal2("bbf", 3, bbf)
call freadreal2("df", 2, df)
call freadreal2("prf", 3, prf)
call freadreal2("w_area_ant", 10, w_area_ant)

This is design variable (system level variable)
call freadreal2("p", 1, p)

--------Analysis routines---------------
Call the subsystem module with necessary arguments
In this code it is appended below. Otherwise the necessary
analysis routines can be placed here itself
To keep the old format intact we have used 'call by subroutine' using common construct

```
call radar(p, l_ant, b_ant, w_ant, w_rsys, q)
```

Output module

```
c Format fwritereal2(variablename_in_analysis,stringlength, variablename_tobe_in_dataserver)
```

```
call fwritereal2("l_ant", 5, l_ant)
call fwritereal2("b_ant", 5, b_ant)
call fwritereal2("w_ant", 5, w_ant)
call fwritereal2("w_rsys", 6, w_rsys)
call fwritereal2("q", 1, q)
```

shutdown communication with dataserver

```
call fshutdsn
```

```
stop
end
```

radar analysis subroutine

```
subroutine radar (p_l_ant,b_ant,w_ant,w_rsys,q)
```

```
t_sc = 60./rpm
lamda=c/f
pri=1/prf
cr = pri * df * bw
par1 = 10.*((3.*dlog10(4*pi)+4.*dlog10(r*1000.))+(dlog10(bw))
1 +kt0+snr+ls+nf
par2 =10.*(dlog10(p*1000.)+2.*dlog10(10^000.)+(2.*dlog10(bw)+
2 dlog10(np)+dlog10(cr))
par=par1-par2
```
par=par/20.
ag=10.*par/an_eff
a_ant=ag*(lamda**2.)/(4.*pi)
theta=np*cpi*pri*360./t_sc
l_ant=50.76*lamda*bbf/theta
b_ant=a_ant/l_ant
p_avg = p*1000.*df
w_ant = w_area_ant*a_ant
w_rsys = 9.81*(11.4*p_avg**.575-1.6*p_avg**.63)
q=5.*p_avg/1000.
return
end