Development of a Multi-Disciplinary Computing Environment (MDICE)

Gerry Kingsley
John M. Siegel, Jr.
Vincent J. Harrand
Charles Lawrence*
Joel J. Luker**

CFD Research Corporation, Huntsville, AL
[gmk,jms,vjh]@cfdrc.com (205) 726-4800

*NASA Lewis Research Center, Cleveland, OH
Charles.Lawrence@lerc.nasa.gov (216) 433-6048

**AFRL/VAAC, Wright Patterson Air Force Base, Dayton OH
lukerjj@fim.wpafb.af.mil (937) 255-4522

ABSTRACT

The growing need for and importance of multi-component and multi-disciplinary engineering analysis has been understood for many years. For many applications, loose (or semi-implicit) coupling is optimal, and allows the use of various legacy codes without requiring major modifications. For this purpose, CFDRC and NASA LeRC have developed a computational environment to enable coupling between various flow analysis codes at several levels of fidelity. This has been referred to as the Visual Computing Environment (VCE), and is being successfully applied to the analysis of several aircraft engine components.

Recently, CFDRC and AFRL/VAAC (WL) have extended the framework and scope of VCE to enable complex multi-disciplinary simulations. The chosen initial focus is on aeroelastic aircraft applications. The developed software is referred to as MDICE-AE, an extensible system suitable for integration of several engineering analysis disciplines. This paper describes the methodology, basic architecture, chosen software technologies, salient library modules, and the current status of and plans for MDICE. A fluid-structure interaction application is described in a separate companion paper.

1 Introduction

The Visual Computing Environment (VCE) [1] has enabled the analysis of aircraft engine components in support of the Numerical Propulsion System Simulation (NPSS) project involving several organizations (e.g. NASA LeRC, AlliedSignal, Pratt & Whitney, CFDRC). VCE has enabled engineers to perform these analyses by coupling several CFD codes (e.g. ADPAC, NPARC, CORSAIR, NASTAR, NISTAR, CFD-ACE, etc.) into one large CFD simulation.

VCE has also enabled the development of MDICE, a multi-disciplinary version of VCE, which initially places significant emphasis on structural analysis (e.g. modal and finite element methods), and its interaction with advanced (3D) flow simulations and geometry modeling / grid generation. Critical issues include moving geometries and conservative and consistent interpolation of forces, moments, and virtual work along fluid-structure interfaces.

In order to provide state-of-the-art technologies and to furnish support of the highest quality to both (propulsion and aircraft) communities, the overall system will be referred to as MDICE.

This paper describes the MDICE software system. Some background information is presented in section 2. The MDICE methodology is discussed in section 3. Architectural issues are described in section 4. Current status is discussed in section 5, and some concluding remarks are made in section 6.

2 Background

A major obstacle in the analysis of engineering systems has been the lack of integration among the various software modules involved in a multi-disciplinary analysis. Simulations of engineering systems in the past involved many independent computer programs, run manually in a sequential (i.e. one at a time) fashion. For example, an engineer wishing to perform a CFD analysis would often interact with a CAD package to define the geometry, a grid generator to obtain the mesh, a pre-processor to apply boundary conditions and to set up an input file, a flow solver to actually perform the analysis, and a visualization tool to examine the results.

The existence of many different file formats exacerbates the problem. An engineer wishing to use tools that do not share a common file format is often required to write a personal file format converter. Moreover, use of binary data files often means that performing parts of the simulation on different kinds of computers is difficult or impossible. Using a workstation with good graphics abilities for the CAD or visualization portion and a dissimilar computationally powerful machine such a vector super-computer or a massively parallel machine for the fluid flow or structural analysis requires a great deal of skill, training, and experience on the part of the engineer.

The goal of MDICE is to aid the engineer in solving these problems by providing a distributed object-oriented system that achieves a high degree of interoperability between the essential analysis tools in a user-friendly environment. MDICE provides an environment where application programs create and manipulate a common set of objects selected from a rich library of general and specialized objects, and makes a series of functions available that the engineer utilizes to run the simulation.

3 Approach

The MDICE software system enables engineers to carry out multi-disciplinary simulations consisting of a set of interoperating computer programs. Code integration is obtained via dynamic sharing of data, execution control and synchronization. Furthermore, multi-disciplinary interfacing technology plays an important role within MDICE.

The MDICE approach is to provide a computing environment consisting of many computer programs operating concurrently and cooperatively to solve a set of problems. Execution of each computer program is controlled by the engineer via the MDICE graphical user interface. Once running, these programs communicate with each other and with MDICE via a set of standard function calls. As with execution control, management of the communication occurring among the application programs is fully controlled by the user.

Integration of a particular computer program into the environment is accomplished by making the application MDICE compliant rather than hardwiring the code to communicate with a small set of other predetermined programs. Once so integrated, the MDICE compliant program is able to run under the control of MDICE and communicate with any other MDICE compliant application.

There are many advantages to the MDICE approach. Using this environment one can avoid giant monolithic codes that attempt to provide all the needed services in a single large computer program. Such large programs are difficult to develop and maintain and by their very nature cannot contain up to date technology. The MDICE method allows (and even relies upon) the reuse of existing, trusted codes that have themselves already been validated. The flexibility of exchanging one application program for another enables each engineer to select and apply the technology best suited to the task at hand. Efficiency is achieved by utilizing a parallel distributed heterogeneous computing environment. Extensibility is provided by allowing additional disciplines to be added without modifying or breaking the disciplines already in the environment.

4 Architecture

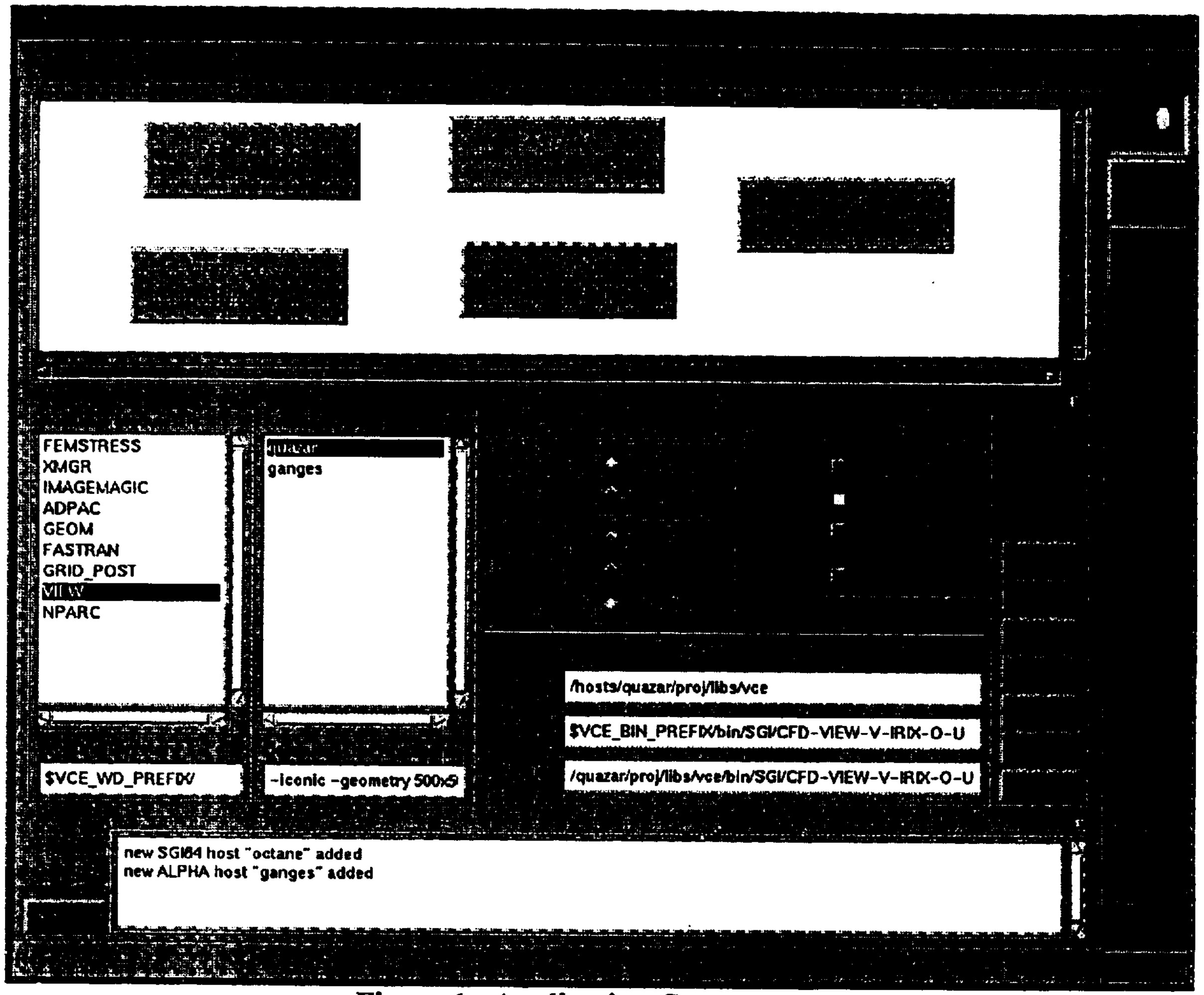
Major features of MDICE include the ability to synchronize application programs, manipulate objects, handle events, carry out remote procedure calls, and a execute a script written in an MDICE script language. MDICE also provides for parallel execution of participating application programs and has a full Fortran interface for those codes written in Fortran 77 or in Fortran 90 (of course, C and C++ are supported as well).

MDICE is broken down into several major components. The first, MDICE proper, is a central controlling process that provides the features described above. Users interact with this central process via a graphical user interface. The second is a library of functions that application programs use to communicate with each other and with MDICE. The environment also encompasses a comprehensive set of MDICE compliant application programs.

4.1 Application Control

The MDICE Graphical User Interface includes facilities for the control of application programs, a drawing area where a visual representation of the simulation is rendered, an object clipboard where information about the various objects created and manipulated by the application programs is displayed, a script editor that allows the engineer to dictate how the simulation should be run, and a status window displaying informational messages.

The graphical user interface is used by engineers to set up a simulation. Application programs are selected; for each, the computer on which the program is to be run is chosen. Other information is provided, such as specifying a directory to run the program in and any command line arguments the program might require. The control panel used to set up and control a simulation is shown in figure 1.



```
FEMSTRESS
XMGR
IMAGEMAGIC
ADPAC
GEOM
FASTRAN
GRID_POST
VIEW
NPARC
```

```
quazar
ganges
```

```
$VCE_WD_PREFIX/
-iconic -geometry 500x5
/hosts/quazar/proj/libs/vce
$VCE_BIN_PREFIX/bin/SGI/CFD-VIEW-V-IRIX-O-U
/quazar/proj/libs/vce/bin/SGI/CFD-VIEW-V-IRIX-O-U
```

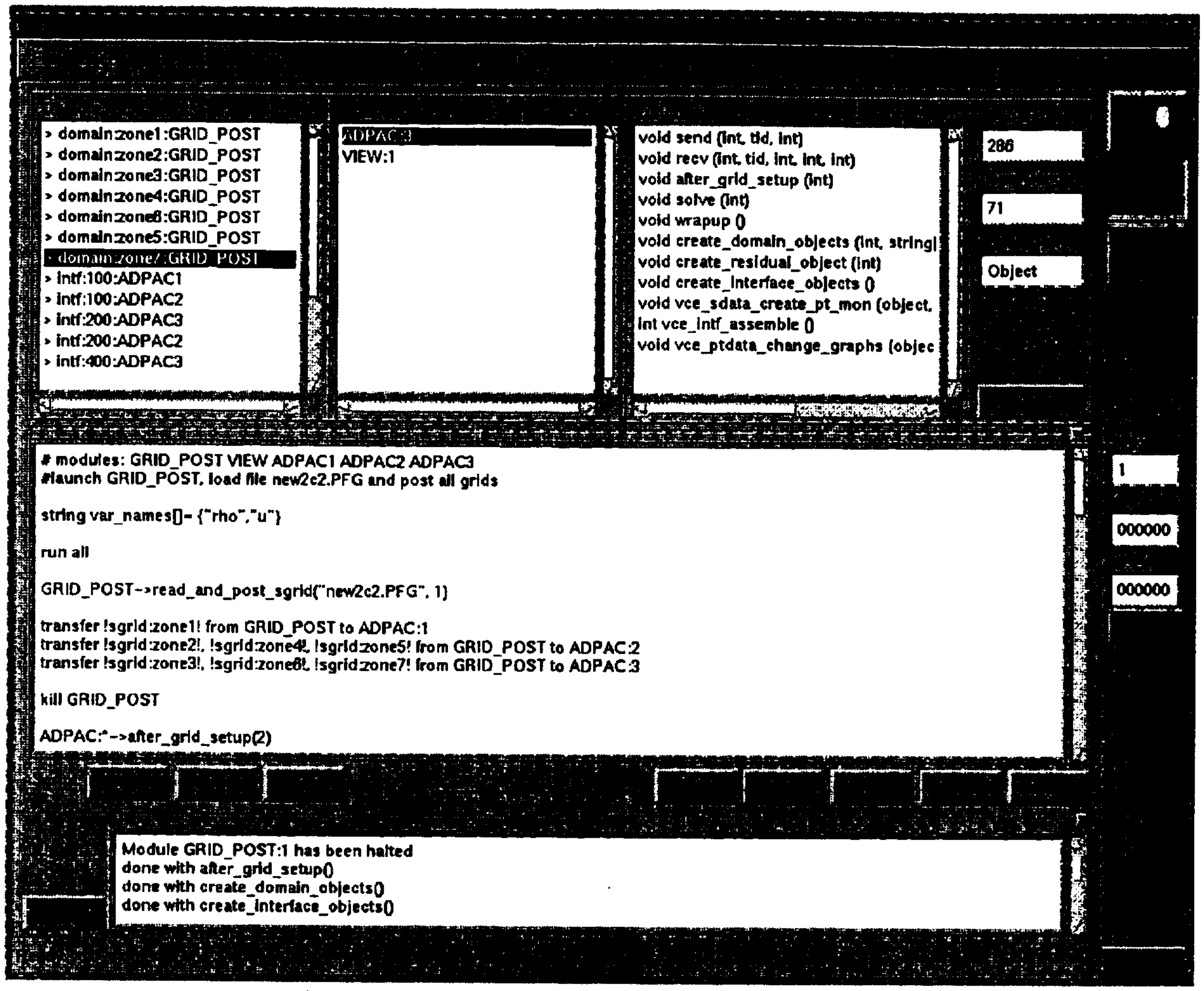
```
new SGI64 host "octane" added
new ALPHA host "ganges" added
```

Figure 1. Application Control Panel

Once the simulation has been set up, it is run and controlled using MDICE. The script panel used to achieve this control is shown in figure 2. The script used by MDICE contains all the conveniences found in most common script languages. In addition, the MDICE script supports remote procedure calls and parallel execution of the application programs being used for a given simulation. These remote procedure calls are the mechanism by which MDICE controls the execution and synchronization of the participating applications. Each application posts a set of available functions and subroutines. These functions are invoked from the MDICE script, but are executed by the application program who posted the function.

4.2 Objects

MDICE supports objects at several levels. On the lowest level, objects are simply named collections of scalar data, arrays, and other objects. These objects are called data objects, array objects, and general objects, respectively. An application program can create an object using one of the creation functions provided by MDICE. An integer handle to the object is returned by the creation function. These handles are used in all subsequent operations on the object.



```
> domain:zone1:GRID_POST
> domain:zone2:GRID_POST
> domain:zone3:GRID_POST
> domain:zone4:GRID_POST
> domain:zone8:GRID_POST
> domain:zone5:GRID_POST
> domain:zone7:GRID_POST
> intf:100:ADPAC1
> intf:100:ADPAC2
> intf:200:ADPAC3
> intf:200:ADPAC2
> intf:400:ADPAC3
```

```
ADPAC:3
VIEW:1
```

```
void send (int, tid, int)
void recv (int, tid, int, int, int)
void after_grid_setup (int)
void solve (int)
void wrapup ()
void create_domain_objects (int, string[
void create_residual_object (int)
void create_interface_objects ()
void vce_sdata_create_pt_mon (object,
int vce_intf_assemble ()
void vce_ptdata_change_graphs (objec
```

```
286
71
Object
```

```
# modules: GRID_POST VIEW ADPAC1 ADPAC2 ADPAC3
#launch GRID_POST, load file new2c2.PFG and post all grids

string var_names[]= {"rho","u"}

run all

GRID_POST->read_and_post_sgrid("new2c2.PFG", 1)

transfer !sgrid:zone1! from GRID_POST to ADPAC:1
transfer !sgrid:zone2!, !sgrid:zone4!, !sgrid:zone5! from GRID_POST to ADPAC:2
transfer !sgrid:zone3!, !sgrid:zone8!, !sgrid:zone7! from GRID_POST to ADPAC:3

kill GRID_POST

ADPAC:*->after_grid_setup(2)
```

```
1
000000
000000
```

```
Module GRID_POST:1 has been halted
done with after_grid_setup()
done with create_domain_objects()
done with create_interface_objects()
```

Figure 2. Object Repository and Script Editor

Each object is completely self describing. It may be assigned a group name, user name, and application name at creation time. In the case of data and array objects, a data type is assigned, and for array objects the length of the array is specified. Using these features, applications may examine a new object that is sent to it from an external application to determine how to deal with it. For example, the recipient application may a query an incoming "grid" object to determine whether the object contains blanking information, or it may query an incoming "flow data" object to see which flow variables are included.

Applications may convert data and array objects from one type to another, or provide updated values for the scalar data or array elements. In this way, an application may convert a newly received array object from double to single precision. Any subsequent receives of this object are automatically converted into single precision regardless of the type being used by the sending application.

Since objects may contain other objects, a hierarchical object tree may be built by an application, as shown in figure 3. Once constructed, the object tree may be registered with MDICE, making the object available for transmission to another application. When an application registers an object, the structure of the object tree (but not the data associated with data or array objects) is sent to the MDICE GUI.

In this example, Domain is a general object containing two sub-objects, Grid and Flow Data, each itself a general object. The Grid object contains three array objects (x, y, and z coordinate arrays) and three data objects (i, j, and k, the extents of the grids in each dimension), while the Flow Data object contains four array objects, one for metric conversion factors and one each for the three flow variables rho (i.e. density), pressure, and temperature.

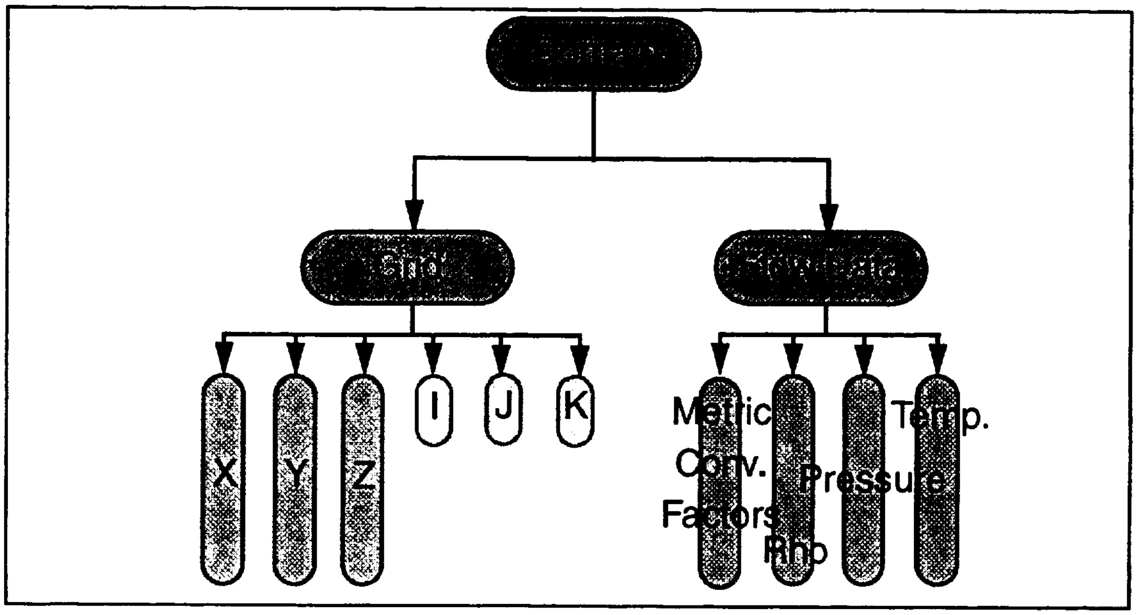


Figure 3. A Hypothetical Domain Object

Applications may themselves create and access the low level objects (data, array and general objects). However, the preferred method of operation is to define a set of higher level objects such as grid objects, flow data objects, and interface objects and place functions that create, access, and manipulate these objects into a library. MDICE provides such a high level library (described in section 4.8); applications may create these high level objects using MDICE-provided convenience functions without concerning themselves with the low level objects they are composed of. In either case, the application simply receives a handle to the object for use in subsequent operations.

4.3 Event Handling

Most window-based programs are event driven programs. These include graphics or design programs that run on UNIX workstations as well as word processors and spreadsheets that run on personal computers. These event driven programs spend most of their time waiting for the user to do something interesting, such as pressing a button, choosing an option from a menu, or moving or clicking the mouse.

Programs integrated into MDICE must become event driven. Subroutines will be called from an MDICE script rather than from PROGRAM MAIN. Object related events such as create, send, receive, and destroy must be handled.

Table 1 illustrates certain events and the actions that one might desire, based on the event.

Table 1: Events and Desired Actions

Event	Desired Action
A CFD flow solver receives a grid object	Compute new cell volumes, Interpolate old flow data to the new grid
A program connected to a structural analysis code halts unexpectedly	Replace changing boundary condition information with static (unchanging) data
The user transmits interface information between a CFD code and a structural analysis code	Interpolate data, Compute deformations, Apply these deformations to the respective grids
A new set of residuals is sent to a line plotter	Add these points to the graph, Redraw the screen

Setting up an application to handle an event is a four step process. We will illustrate this using the first example from table 1, namely a CFD flow solver receiving a grid object.

1. **Prepare for the event**. Typically, the application will want to have some subroutine or function called when an event takes place. In this case, the program wants to have a subroutine called `recv_grid` executed. This accomplished by adding the subroutine as a callback. The name of the object, the type of the event, and the desired subroutine are specified in a special MDICE-provided subroutine call.
2. **Wait for the event**. Once the application has specified all its callback functions, it transfers control to MDICE using a special MDICE-provided control function. This function serves as an event loop, monitoring the environment for events (and other commands such as remote procedure calls, discussed in section 4.4). Once called, this control function never returns.
3. **Handle the event**. MDICE calls the `recv_grid` subroutine on behalf of the application.
4. **Wait for the next event**. No action is required. When the `recv_grid` subroutine from step 3 returns, control is automatically returned to the MDICE event loop, and MDICE continues monitoring the system for future events.

4.4 Remote Procedure Calls

Remote procedure calls are the mechanism by which MDICE invokes subroutines or functions that are executed by a participating application program. The user of the system writes such a function call using the MDICE script editor.

In order to carry out the remote procedure call, MDICE evaluates each expression in the function's argument list and packs the result of the expression into a message buffer. This is called marshalling the arguments. Next, the message buffer is transmitted to the application, which has previously called MDICE's special control function and is in its event loop waiting for something to happen, such as an object related event or (as in this case) an incoming remote procedure call command. The MDICE library intercepts the incoming message, unpacks the arguments, and calls the application's function. After the function completes, the return value (in this example, a boolean value that indicates whether the solution has converged yet) is packed into a new message buffer and sent back to MDICE. MDICE places the return value of the function into the expression containing the original function call, and the script resumes.

The internal processing required by MDICE to make these remote procedure calls work is rather complex. First, MDICE must know about the procedures that the application programs wish to make available. This requires that a detailed type system be implemented. MDICE must also be able to call procedures that are internal to an application without knowing *a priori* what the type signature of the procedures are. This requires that MDICE call wrapper functions whose type MDICE *does* know. These wrapper functions must be linked with the particular application whose procedure is being invoked. In order to spare the application programmer from writing them, MDICE provides a stubber that automatically generates these wrappers.

4.5 The MDICE Script

MDICE allows full user control by means of a script. The MDICE script language is easy to learn yet powerful, making full customization of multi-disciplinary simulations possible.

The language allows all the standard conveniences found in most script languages: local and global variables, a rich set of data types including integers, real numbers, strings, objects, and arrays of these base types, decisions (i.e. if statements), and loops. In addition, the MDICE language provides for remote procedure calls (discussed in section 4.4), execution of portions of the script in parallel (discussed in section 4.6), application control in form of `run` and `kill` commands, and debugging tools such as `print` (which prints a string to the MDICE status window) and `pause` (which pauses the script until the continue button is pressed.

Unlike most shell script languages, the MDICE script is a compiled language. It is read in its entirety, converted into internal data structures (e.g. an abstract syntax tree, control flow graph, and symbol table are all built by MDICE), and the resulting code is executed. Space for variables and the results of expressions are allocated before a script is run and freed when it is complete.

The script is strongly typed. Each expression is fully checked for compatibility of its operands before the script begins running. The return value of each function is also checked, statically if the function has been posted by the named application before script execution (in which case MDICE already knows the type of the return value), dynamically if the function is posted during the run.

4.6 Parallelism

An additional complexity is introduced by the requirement that MDICE be able to call distinct procedures simultaneously. MDICE solves this problem by implementing the notion of threads.

Since threads are not implemented on all the platforms that we wish to run MDICE on, this is implemented directly inside MDICE. Neither application programs nor users need be aware that this is taking place. MDICE has a thread class that represents a portion of the script to be executed in parallel with other portions. A list of runnable threads is maintained by MDICE. When one thread blocks (when a remote procedure is being called, for example), the thread is placed on a list of blocked threads and a new runnable thread is selected. If no more runnable threads remain, MDICE simply waits for one or more remote procedure calls to complete.

As results (i.e. return values) from the remote procedure calls are delivered to MDICE, the appropriate blocked thread is moved from the "blocked" list to the "runnable" list. Each remote procedure call is tagged with a reference number to match these incoming return values with the thread that called the function in the first place. When no more incoming messages are left, one of the runnable threads is chosen for execution (using a round-robin scheduling algorithm) and the script resumes.

For example, if several CFD flow solvers are being used to solve a problem, it is necessary that each perform an iteration over the flow domain concurrently. Each remote procedure to be called simultaneously is carried out by a separate thread. Of course, the end user need not be aware that this is taking place. Using a special `parallel` script command, one can write blocks of code, each of which is executed simultaneously. Within a given block, each statement is executed in parallel.

A shorthand notation is available if each block contains a single expression. This shorthand uses MDICE's semicolon operator. This operator is similar to the comma operator in the C programming language in that each semicolon-separated expression is evaluated; the value of the entire expression is the value of the rightmost expression. The difference is that each expression (i.e. all the remote procedure calls) are evaluated simultaneously. The entire expression completes when all the function calls complete.

4.7 Fortran Interface

The MDICE libraries are written in the C programming language. In order to call the functions from a Fortran program, a complete Fortran interface is provided. This interface allows all the common Fortran data types, including integers, single and double precision reals, character strings, and arrays to be passed into the library functions.

4.8 MDICE Libraries

MDICE provides four libraries of functions that application programs may be linked with when they are integrated into the computing environment. These are

1. A low level MDICE library. This library implements low level objects (data, array, and general objects), communication (sending and receiving messages between applications or between applications and MDICE), and control (event handling and remote procedure calls).
2. An object library containing a rich set of pre-defined objects. These objects include

• Arrays	
• Boundary Conditions	Structured and Unstructured
• Grids	Structured, Unstructured, and Polyhedral
• Flow Data	Structured and Unstructured
• Domains	A Combination of Grids and Flow Data
• Interfaces	Fluid-Fluid and Fluid-Structure
• Plotting Data	Line Data and Point Data

3. An interpolation library that provides interpolation of flow field data between different CFD flow solvers or between a CFD flow solver and a structural analysis code.
4. A memory allocation library that can be used to dynamically check the correctness of the application's use of dynamically allocated memory.

4.9 MDICE Compliant Applications

There are several applications which are already MDICE compliant. These include most of the CFDRC proprietary codes such as:

• CFD-VIEW	Post Processing, Visualization, and Animation
• CFD-GEOM	Geometry Modeling and Grid Generation
• CFD-ACE	Multi-Purpose CFD Flow Solver
• CFD-FASTRAN	External Aerodynamics Flow Solver
• CFD-FEMSTRESS	Structural Analysis

In addition, the following codes (some of them public domain) have been integrated into MDICE:

• ADPAC	NASA Lewis turbomachineryflow solver
• NPARC	NPARC Alliance Inlet/Duct/Nozzle flow solver
• GCNS	Northrop-Grumman flow solver
• WIND	Previously NASTD, a Boeing flow solver
• Cobalt	AFRL/VAAC (WL) flow solver
• Corsair	Pratt & Whitney and NASA LeRC Combustion flow solver
• Xmgr	Two dimensional graphing package
• ImageMagic	Used for capturing screen dumps

5 Current Status and Future Work

The MDICE system is fully functional and is currently being used heavily by CFDRC for several large projects. These include the Numerical Propulsion System Simulation (NPSS), the Multi-disciplinary Aero-Structural Environment, the Generic Remeshing Environment, and simulation of Magnetic Resonance Imaging. An application of this environment for fluid-structure interaction is given in the companion paper [2]

Researchers at the NASA Lewis Research Center are using MDICE to couple different types of CFD flow solvers to perform a coupled inlet-engine analysis [3] and to couple codes in the National Combustion Code project [4]. Pratt & Whitney is using the environment for turbomachinery analysis [5], and AlliedSignal Engines is using the environment for several engine analyses [6]. Engineers at AFRL (WL) are using the environment to couple CFD and structural analysis codes for aeroelasticity problems. At least four industry partners are currently integrating their proprietary codes into MDICE. These are Pratt & Whitney, AlliedSignal, Northrop Grumman, and Boeing.

Future work includes providing a point-and-click script builder, including fault tolerance libraries enabling MDICE compliant applications to checkpoint themselves, and adding more application programs to the environment.

6 Conclusion

We have integrated VCE, a joint CFDRC / NASA LeRC project, and extensions developed by CFDRC and AFRL (WL) into MDICE, a multi-disciplinary computing environment, for running simulations that involve many dissimilar yet interoperating application programs. A central program allows the engineer using the system to fully control and steer the simulation. A rich and robust collection of libraries allow application programs to be integrated into the system by becoming MDICE compliant. Once done, each such application is able to communicate with all other MDICE compliant applications.

We provide several tools to aid the application programmer in this task. These include a reference manual, a user's guide, and a code generator that lifts much of the burden from the programmer. In addition, a complete Fortran interface allows large legacy codes to be integrated using only native Fortran programming constructs.

A significant number of engineering analysis codes from a variety of disciplines such as grid generation, CFD, structural analysis, visualization, etc., have been integrated into the environment by several organizations. Integration of more codes and an extension to more disciplines is planned for the near future.

7 Acknowledgments

The authors would like to gratefully acknowledge the support of the NASA High Performance Computing And Communication Program. We also acknowledge Scott Townsend and Ambady Suresh and the NASA Lewis Research Center, whose support and assistance has been considerable.

The authors would also like to acknowledge the work being done by Steve Connolly at Pratt & Whitney and by Wolfgang Sandel at AlliedSignal. Their feedback has contributed a great deal to the success of the project.

General Support and guidance of several others have also been very useful, including: Russ Clauss of NASA LeRC, Ken Moran, Don Kinsey, and Larry Huttsell of WL, and Andrzej Przekwas and Ashok Singhal of CFDRC.

References

[1] Gerry Kingsley, Vincent Harrand, and Charles Lawrence, "A Visual Computing Environment for Computational Aerosciences", Proceedings of the 1996 Computational Aerosciences Workshop, NASA Ames Research Center, August, 1996, pp. 313-318.

[2] John M. Siegel, Jr., Gerry Kingsley, Paul J. Dionne, Joel J. Luker, and Vincent J. Harrand, "Application of a Multi-Disciplinary Computing Environment (MDICE) for Loosely Coupled Fluid-Structural Analysis", Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis & Optimization.

[3] Gary Cole, "VCE Applications at NASA Lewis: Inlet-Engine Simulation", Proceedings of the First VCE Workshop, NASA Lewis Research Center, November, 1997.

[4] Nan-Suey Liu, "VCE Application to National Combustion Code", Proceedings of the First VCE Workshop, November, 1996, NASA Lewis Research Center

[5] David Edwards, 'VCE Applications at Pratt & Whitney", Proceedings of the First VCE Workshop, NASA Lewis Research Center, November, 1997.

[6] Wolfgang Sandel, "VCE Applications at AlliedSignal Engines", Proceedings of the First VCE Workshop, NASA Lewis Research Center, November, 1997.