

Project Integration Architecture: Architectural Overview

Dr. William Henry Jones

National Aeronautics and Space Administration
John H. Glenn Research Center at Lewis Field
Cleveland, OH 44135
216-433-5862
William.H.Jones@lerc.nasa.gov

Keywords:

PIA; PRICE; Application Integration; Interapplication Propagation of Information;
Semantic Encapsulation; Multifidelity Analysis; Multidisciplinary Analysis

ABSTRACT: *The Project Integration Architecture (PIA) implements a flexible, object-oriented, wrapping architecture which encapsulates all of the information associated with engineering applications. The architecture allows the progress of a project to be tracked and documented in its entirety. By being a single, self-revealing architecture, the ability to develop single tools, for example a single graphical user interface, to span all applications is enabled. Additionally, by bringing all of the information sources and sinks of a project into a single architectural space, the ability to transport information between those applications becomes possible. Object-encapsulation further allows information to become in a sense self-aware, knowing things such as its own dimensionality and providing functionality appropriate to its kind.*

1 Introduction

In the late 1980's, the Integrated CFD and Experiments (ICE) project [1, 2] was carried out with the goal of providing a single, graphical user interface (GUI) and data management environment for a variety of CFD codes and related experimental data. The intent of the ICE project was to ease the difficulties of interacting with and intermingling these disparate information sources. The project was a success on a research basis; however, on review it was deemed inappropriate, due to various technical limitations, to advance the effort beyond the successes achieved.

A re-engineering of the project was initiated in 1996. The effort was first renamed the Portable, Redesigned Integrated CFD and Experiments (PRICE) project and then, as the wide applicability of the concepts came to be appreciated, the Project Integration Architecture (PIA) project. Two key re-engineering decisions were made: the C language used by the ICE project would be abandoned in favor of the now-available C++ object-oriented extension to that language and the graphical user interface would be eliminated as a product element of the project.

During the intervening years, work has proceeded and an operational demonstration of the PIA project has been achieved.

1.1 Goals

Put in simple terms, the basic goal of the PIA effort is to capture in its entirety the usage of any engineering application in a single, useful, well-defined form. This capturing is not limited to the simple output of the application itself, but further includes coordinating information and the engineer's own insights into the meaning of the information.

The nature of an 'engineering application' is, by project design, nebulous: it is considered to be any computer-realized 'thing' which provides or generates useful information about an engineering project. This may include CAD information, design code predictions, experimental data, engineering analysis and simulation, and more.

By bringing all of the information of the engineering process into one architectural design, a number of advantages are expected to (and, it is believed, do) accrue.

1. The information about the information (referred to in some conceptions as the meta-data) is encapsulated with the information itself.
2. A common tool set for the use of engineering applications is possible.
3. Browsers and search engines may be implemented to

peruse information in detail and convert it into information in general for end consumers.

4. Object encapsulation allows information to become self-aware. For example, a measurement may know its dimensionality and provide its value appropriately converted to whatever system of measurement a request is made in.
5. By wrapping every application in a well-defined architecture, it is now possible to code the knowledge to acquire information automatically from other applications. Such coded knowledge is based upon the kind of information desired, rather than upon the application generating that kind of information.
6. Wrapped applications coded to obtain information based upon its kind may then be combined in a directed application graph to build, in effect, super-applications. Applications of differing fidelities and disciplines may be mixed together as appropriate to the project.
7. The building of super-applications enables project-wide optimization, sensitivity analysis, and other such techniques to be conducted.

1.2 Self Revelation

Perhaps the key technology that enables the goals above is that of self revelation, the ability of a thing to reveal to others its own nature. Such a capacity can be implemented by many different techniques; however, it is a natural element of object-oriented technology.

1.2.1 Self Revelation of Kind

The revelation of kind identifies the essential character of the revealing entity. In the object-oriented implementation of PIA, this sets expectations as to the kind of information and functionality a particular object has.

The revelation of kind is effected in two ways: an interrogative form and a declarative form. In the interrogative form, a predicate of kind is posed to the object and either affirmed or denied. In the declarative form, an inquiry is made of the object and a simple coded value is returned declaring the type of the object.

Because of the derivational nature of object technology, both of these revelational forms support the concept of depth. That is, an object may be of a particular kind at some derivational depth but, because of further derivation,

may not appear to be of that kind on its surface. The examination of such layers of meaning is referred to (within this project at least) as ecdysiastical revelation. (From the Greek *ekdysis*, from *ekdyein*, to get out of, strip off.)

1.2.2 Self Revelation of Content

The revelation of content identifies the extent to which expectations based upon the revelation of kind are, in fact, fulfilled. In such situations the expectation of content is nebulous by design and a well-defined method of further exposition is defined.

As an example of self revelation of content, consider the **PacAppl** application class which is to be discussed shortly. That class is defined as having a map of **PacOp**-derived operations objects; however, the map may be empty and, if it is not empty, it is unpredictable exactly which operations objects will, in fact, populate it. Thus, an expectation of content is defined, but the precise extent of content is left open with a defined method for further exploration.

2 Application Architecture

Building upon the concept of self revelation, an application architecture as depicted in Figure 2.1 has been devised.

An application presented in the image of PIA begins with a central application object, labeled **PacAppl** in the upper center of the figure, which is the root structure from which all further components emanate. Four principal components are currently provided by the **PacAppl** object:

1. A set of operations that the application is willing to perform,
2. A mass of data which the application currently contains,
3. A structure by which the contained data is identified, and
4. An ecdysiastical sorting of the information-bearing objects in the application.

The first three components are depicted in the figure in the upper left, the upper central to lower left diagonal, and the central right, respectively. The fourth component is not depicted due to its structural complexity. Each of these components is taken up in its own subsection below.

2.2 Operating Context

The parameter identification and application operation objects both operate in the context of a selected parameter configuration, a concept shown in the figure by the sweeping curves from those kinds of object to a single configuration. These object kinds offer `IsVisible` and `IsEnabled` member functions, respectively, which indicate whether or not the functionality is active within the context of the selected parameter configuration. This feature allows, for example, the internal parameters of an optional analysis component to be made invisible when that component has been disabled in a particular configuration, even though those parameters might exist in, or be inherited by, that configuration.

3 The Base Object

Very nearly all of the object classes involved in implementing the PIA application architecture described above are derived from a common base class, **PacBObj**. Among other things, this base class provides several key features:

1. The ability to be described,
2. The ability to transmit declared events, and
3. The ability to traverse upward through the application structure.

3.1 Descriptive Capabilities

The ability to be described brings a good deal of useful function to the entirety of the architecture. A wide variety of descriptions, such as access controls, descriptive texts, dimensionality, and others, are implemented. Indeed, virtually any sort of descriptive element can be devised and added to the repertoire. Because not all objects will necessarily have descriptive elements and certainly not all objects will have all descriptive elements, the descriptive system is one of minimal overhead.

The only limitation of the descriptive system is that, within a descriptive set, at most one instance of any particular controlling descriptive type is permitted. This limitation is offset by the fact that the descriptive system is implemented in a layered, hierarchial manner in parallel with the derivational class hierarchy and that distinct descriptive sets may exist at each such level. Within this descriptive hierarchy, the top-most description (that is, the description of the most derived class level) of any kind is considered to be the preferred description; however, the facilities exist to reveal the entirety of the descriptive hierarchy should it be desired.

3.1.1 Engineering Logs

A multi-line, descriptive text form is provided as the possible basis of an engineering log facility. Here an indefinite, expandable number of text string elements in an ordered list can be associated with any **PacBObj** derived object.

3.1.2 Change Histories

A change history descriptive form is provided. It provides an implicitly time-stamped, ordered, multi-line descriptive form. This form is implicitly used by the **PacPara** (parameter) base class to record parameter modifications as previous value texts. All currently implemented parameter classes utilize this capability in their corresponding Set-Value services.

3.1.3 Access Controls

An access control descriptive complex is implemented. Any **PacBObj**-derived object can attach access control descriptions throughout its descriptive hierarchy. This allows per-user read, write, execute, and delete controls to be applied on an object-by-object basis.

3.2 Declared Events

A simple subscribe/publish event facility is provided by the **PacBObj** base class. This facility allows utilizing entities to attach one or more objects of **PacEvent** derivation to a **PacBObj**-derived object. The **PacEvent** base class defines a number of different event types but, as a base class, provides only a default response should the event be declared. The **PacBObj** base class implements corresponding event functions which, if invoked, will identify each attached event object and transmit the event declaration to it.

Utilizing code is responsible for developing and attaching derived event class forms which do something meaningful should an event be declared. There is no artificial limitation on the number of event objects that may be attached to a particular **PacBObj** object, nor upon the number of **PacBObj** objects that may be attached to a particular event object.

3.3 Upward Reference

The basic direction of the application architecture is down from encompassing components to more specific components. In implementation, though, it is frequently necessary

to traverse in the opposite direction. This need is met by a pointer member and supporting code in the **PacBObj** class which references the next higher level element of the application structure. Implemented functionality allows traversal of this upward chain of reference until an object of a particular, specified kind is found. Thus, code can be insensitive to the exact placement of its presenting object in the architectural hierarchy.

4 Parameters

Information is encapsulated in a derivationally-rich set of parameter objects. The parameter derivational structure is based upon the **PacBObj** base class and, thus, all of the capabilities of that foundation are inherited.

4.1 Dependent Parameters

The parameter base class utilizes the directed graph capabilities inherited by it to implement a dependent parameter mechanism. Each successor (dependent) in such a graph is informed of any change in any of its predecessor (benefactor) parameters. This dependency mechanism is accounted for during the act of parameter replication, causing the replication of dependent parameters as needed.

4.2 Infusion of Semantic Meaning into Parameter Objects

The self revelation of kind mechanism is used by the parameter object hierarchy to infuse semantic meaning into parameters [3]. The first derivations of parameter objects specialize parameters by their basic structural forms; scalar, vector, matrix, organizational, and the like. Further derivation then associates an atomic kind, long, double, Boolean, string, and the like, with these forms as appropriate.

A further specialization of double parameter kinds declares them to be dimensional in nature, that is being a measurement in a specified system of measurement. (The concepts of dimensionality are discussed in greater detail in [4].) From this point a great majority of engineering parameters may then be derived.

By setting the dimensional elements to null values, dimensional parameters are further specialized to be non-dimensional. By doing this, non-dimensional parameters may be freely intermixed with their dimensional counterparts in dimensionally-sensitive operations. From this base, non-dimensional engineering parameters are then derived.

4.2.1 Related Parameters

A related-parameter descriptive mechanism is utilized to associate other parametric information with semantically-defined parameter objects. For example a grid of information may identify a parameter giving the positions of its grid coordinates through the related parameter mechanism.

5 Persistence

Saving the state of an encapsulated application was a self-evident requirement. To achieve this, an object serialization capability was implemented. Object contents are written out to or read back from an archive file under the control of a `Serialize` function. An archive object keeps track of what objects have been serialized or de-serialized so that redundant references to an object are appropriately handled.

The drawback of this serialization is also its strong point: a `Serialize` function must be manually coded for every class that might participate in such an operation. For the most part such coding amounts to mere tedium; however, it is in this coding that the groundwork for evolutionary change can be laid. If one takes the precaution of serializing an archive version number as the very first step of object serialization, then conditional code for the de-serialization of old object versions can be generated when object revisions are defined. By this means, old objects may be recovered even as object coding evolves.

6 Information Propagation

The infusion of semantic meaning into parameter objects through derived class specialization and self revelation forms the basis for the interapplication transfer of information by allowing one application to 'look' at parameters of another application and discern the semantic nature of the observed parameter objects. This basic technology enables a number of interapplication information transfer modes.

The propagation of parametric information throughout an application graph is the first form of information transfer implemented by the PIA project [5]. The goal is to arrange disparate applications into a cooperative graph whose operation carries out all of the analyses relevant to an engineering project as a whole.

The graph of applications is currently constructed by operations of the testbed GUI. The first application created through the GUI is always made the initial node of an application graph. A right click on that application (or upon any application subsequently added to the graph) pops up a

menu with a pick for adding a successor application to the right-clicked application. The standard select-application dialog is then executed and the application-graph member added.

A basic conceptual view behind the arrangement of an application graph is that there always exists some source definition of a proposed configuration of the project which then feeds as input to various analyses of that configuration. Those analyses then produce results with two potential aspects: intermediate values which are of use for further forms of analysis and final answers contributing toward a judgement of the engineering merit of the design.

Another aspect implicit in this view of information propagation throughout directed application graphs is that such applications operate in a mode in which a single operation (from an external viewpoint) reliably turns input information into output information, as opposed to a scenario in which the propagation act must be iteratively performed until parametric values, in some sense, reach a balance or converge. Note, though, that such an iterative propagation form is not precluded by the PIA architecture, but is only in opposition to the assumptions of the information propagation form first implemented for study.

Information propagation as presently implemented also forces the synchronization of parameter configurations. By this, the concept of a project configuration is made more real and, it is expected, the problem of mismatched configurations within a project analysis will be eliminated.

The propagation implementation also recognizes that not all applications are entirely reliable in their operation. To deal with this, the support utilizes the event mechanism built into the **PacBObj** base class to allow inappropriate operations to alert supposedly corrective entities. There is, thus, the ability to apply corrective measures and re-attempt a particular operation in the overall propagation activity. Failing such corrective actions, the information propagation system will mark the affected parameter configurations as being defective and will prevent further propagative acts based upon those configurations.

The process of information propagation as currently implemented proceeds in the following general manner.

1. The process is begun by delivering a propagation command citing a parameter configuration to an application object which is, typically, the initial node of an application graph. The event is typically delivered from an external, organizing entity, for example a person controlling the overall process through an inter-

acting GUI.

2. The application object converts inputs to outputs after its manner.
3. The application then passes the propagation operation on to each of its immediate successors in the graph. The configuration is passed on in this act.
4. Each receiving application establishes or creates a corresponding configuration.
5. Each receiving application then examines each of the parameter objects available in the source configuration and, based upon the semantic meanings revealed by those parameters, acquires such information as it may. Each receiving successor application is free to examine the extended predecessor applications of its own propagating immediate predecessor application for input information.
6. When each successor has received a propagation act from each of its own immediate predecessors, it then operates to convert its own inputs into outputs and then passes the propagation act on to its immediate successors.
7. The propagation of information continues in this manner throughout the graph until terminal nodes of the graph are reached and return the propagation act back up the graphical chain to the originator of the act. Ultimately, this rolls up to a single return of control to the original initiator of the propagation act, indicating that the operation is complete.

It should be remembered in all of this that it is the technology of self-revelation exposing infused semantic meaning that makes the implementation of information propagation tenable. Applications wrappers need only be coded to look for the kinds of information they desire to acquire during propagation. It is not necessary to code for connection to a specific source application to obtain an expected kind of information, nor is it necessary to code for specific topological arrangements of applications.

7 Documentation

Complete, class-by-class, member-by-member documentation has been generated in Hyper-Text Markup Language (HTML) format and placed on a central server at the Glenn Research Center. The root URL for this documentation is

<http://www.lerc.nasa.gov/WWW/price000/index.html>

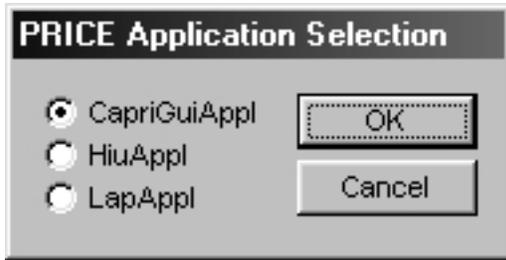


Figure 9.1: GUI Opening Application Query

It must be strongly emphasized that these pages are the informal generation of the researcher involved and do not, in any way, shape, or form, represent an official statement of the Government of the United States.

8 Experience

To date, three applications have been wrapped in the C++ implementation of the PIA Application Architecture: a presentation of experimental data from an inlet unstart experiment for the High Speed Research project (known as HIU), a presentation of flowpath geometry information from Computer Aided Design (CAD) sources accessed through the Computational Analysis Programming Interface (CAPRI) cross-vendor package [6], and an operational wrapping of the Large Perturbation Inlet (LAPIN) analysis code [7]. Propagation of geometry information from the CAPRI/CAD wrapper to the LAPIN wrapper is currently being demonstrated.

9 Testbed GUI

Although a GUI is not considered to be a product of the overall project, such a tool is nevertheless necessary for testing purposes. Indeed, a GUI is the most expedient way to see that the concepts described above do, in fact, work.

The first demonstration of the architecture is shown in Figure 9.1. This is a screen capture of the application selection dialog box implemented by the GUI. The dialog allows the user to select one of the available application types through a mutually-exclusive radio button interaction.

The remarkable thing about the application selection dialog is that it is generated on-the-fly by the GUI, rather than by a static coding of the dialog. At the time of dialog initialization, a scan is done of all PIA classes, isolating those that are derived from the type **PacAppl**. (The class **PacAppl** itself is excluded from this set.) A radio button is generated for each such identified application class, drawing the name

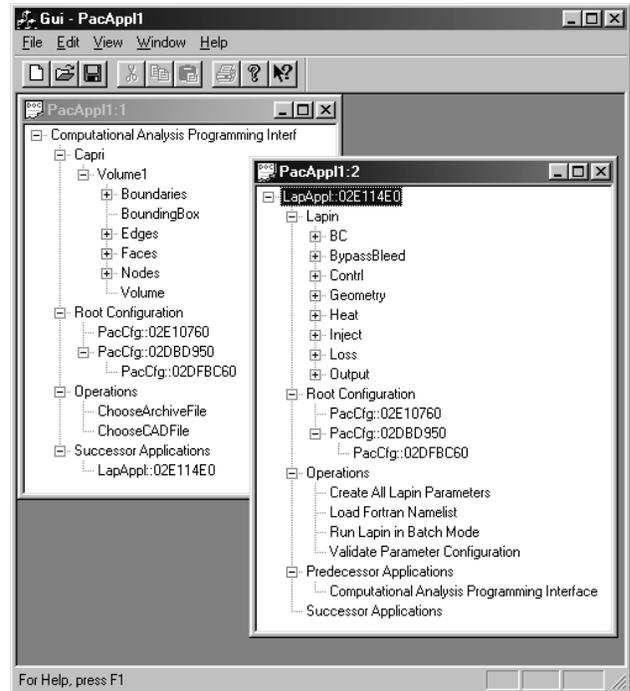


Figure 9.2: Display of Two Independent Applications Connected in a Graph

text from the supporting class information. Thus, the figure shows that, at the time this dialog was captured, three applications were supported: the CAPRI cross-vendor CAD geometry (**CapriGuiAppl**) application, the HSR Inlet Unstart (**HiuAppl**) application, and the LAPIN (**LapAppl**) application.

Figure 9.2 illustrates nearly all the rest of the features of the architecture as exercised by the testbed GUI. The window labeled **PacAppl:1** views a CAPRI application chosen from an application selection dialog in the course of GUI startup. The window labeled **PacAppl:2** views a second application, in this case a LAPIN application, that has been created as a successor to the CAPRI application in the application graph.

A window viewing an application lists from top to bottom

1. The parameter identification tree (which is only partially expanded in either window of the figure for reasons of space),
2. The parameter configuration graph,
3. The operation tree,
4. The application graph predecessor list (except in the case of the **PacAppl:1** window which views the ini-

tial node of the graph which, by definition, has no predecessors), and

5. The application graph successor list.

Comparing the differences between the two windows, for example the different operations lists, shows the self-revealing nature of applications within the architecture. The **PacAppl1:1** window lists the two CAPRI operations; the selection of a CAD file from which an internal CAPRI archive of information will be produced, and the selection of such an archive from which a geometry parameter set in the selected configuration will be produced. Meanwhile, the **PacAppl1:2** LAPIN window lists an entirely different set of operations; the creation of LAPIN parameters, the loading of LAPIN parameters from traditional Fortran namelist input, the running of LAPIN, and the validation of the parameter set as input to a potential LAPIN run.

The two viewing windows also reveal the connection between the two applications as participants in an application graph. The **PacAppl1:1** window views the initial node of the application graph, as witnessed by the absence of a Predecessor Applications element in its view. The Successor Applications list of that view shows the LAPIN application viewed by the **PacAppl1:2** window as its successor. The LAPIN application, in turn, lists the CAPRI application of the **PacAppl1:1** window as its predecessor.

The effect of information propagation throughout the application graph is also illustrated by Figure 9.2. The parameter configuration graph of the CAPRI application viewed through the **PacAppl1:1** window has been expanded beyond its default single-patriarch form to include two child configurations and a grandchild configuration attached to the second child. By the act of information propagation citing the root parameter configuration of the CAPRI application (effected by first selecting the root parameter configuration of that application and then double-clicking the application element of the viewing window), that parameter configuration graph is replicated in the successor LAPIN application. This is further confirmed by the fact that the default parameter configuration object names generated in the CAPRI application as the configuration graph was expanded (for example, PacCfg:02E10760) are, in fact, replicated in the configuration graph of the LAPIN application. This is precisely as expected by the act of information propagation as a result of its effort to keep parameter configurations synchronized between cooperating applications.

This illustration of information propagation is somewhat overextended in order to demonstrate features of the architecture that reach beyond current realities. In fact, no CAPRI/CAD data source with multiple configurations ex-

isted at the time of this writing. Thus, the descendent configurations are, in fact, empty. They exist to demonstrate the configuration bookkeeping aspects of the architecture.

Another aspect of this information propagation figure is also not fulfilled at the time of writing in that no actual information was transferred between applications as a result of the act that produced the display. This is due to two factors. First, there were still bugs in the program which were being tracked down and corrected at the time of writing. Second, no semantically correct CAD file for the sample problem, the inlet flow path of the Rocket Based Combined Cycle engine being developed at the Glenn Research Center, was available at the time of writing. It is believed and represented in good faith that both these difficulties will be overcome in the near future.

10 Future Directions

At this point, the road ahead for the PIA project seems relatively clear. The key technology of self-revelation and its ability to enable common tools, information propagation throughout an application graph, and the like can be considered well demonstrated. Further work now must center on two areas: making the application architecture practicable by moving it to a distributed object environment, and filling in the many semantic gaps of the parameter object set so as to have a fully populated set of information forms.

Work has already begun on the migration to a distributed object environment based upon the Common Object Request Broker Architecture (CORBA). At this point, all foundation classes have been migrated to this environment and operationally demonstrated. Work on application-level classes and concepts is now beginning.

11 Summary

An abstract, highly flexible, object-oriented application architecture has been defined. The architecture has been implemented in C++ and real applications have been wrapped according to that architecture. Dimensional parameter objects wrapped in this architecture are self-aware of their dimensionality and automatically convert their values to a requested unit system. Applications wrapped in this architecture have been connected into directed application graphs and the automatic propagation of information from source application to consumer has been demonstrated.

Author Biography

DR. WILLIAM HENRY JONES is a Computer Engineer at the National Aeronautics and Space Administration, John H. Glenn Research Center at Lewis Field, in Cleveland, Ohio. He has been the sole architect for the PIA Application Architecture since the initiation of the project in 1996. Dr. Jones also has wide ranging interests in the computer field including direct computer control of manufacturing machinery, specialized CAD/CAM systems, electronic hardware design and implementation, compiler theory and implementation, object-oriented design, computer systems security, computational fluid mechanics, and others, as well as considerable experience in aeropropulsion and the practicalities of flight.

Final Report. Contractor Report NASA CR-174676, National Aeronautics and Space Administration, Lewis Research Center, 21000 Brookpark Road, Cleveland, OH 44135, September 1984.

References

- [1] The American Society of Mechanical Engineers. *Integrated CFD and Experiments Real-Time Data Acquisition Development*, number 93-GT-97, 345 E. 47th St., New York, N.Y. 10017, May 1993. Presented at the International Gas Turbine and Aeroengine Congress and Exposition; Cincinnati, Ohio.
- [2] James Douglas Stegeman. Integrated CFD and Experiments (ICE): Project Summary. Technical memorandum Number not yet assigned, National Aeronautics and Space Administration, Lewis Research Center, 21000 Brookpark Road, Cleveland, OH 44135, December 2001.
- [3] William Henry Jones. Project Integration Architecture: Formulation of Semantic Parameters. Draft paper available on central PIA web site, January 2000.
- [4] William Henry Jones. Project Integration Architecture: Formulation of Dimensionality in Semantic Parameters. Draft paper available on central PIA web site, March 2000.
- [5] William Henry Jones. Project Integration Architecture: Inter-Application Propagation of Information. Draft paper available on central PIA web site, December 1999.
- [6] Robert Haimes. *Computational Analysis PROGRAMMING Interface (CAPRI): A Solid Modeling Based Infrastructure for Engineering Analysis and Design*. Cambridge, MA, November 1999. Web Reference: <http://raphael.mit.edu/capri/>.
- [7] M. O. Varner, W. R. Martindale, W. J. Phares, K. R. Kneile, and J. C. Jr. Adams. Large Perturbation Flow Field Analysis and Simulation for Supersonic Inlets: