Service ORiented Computing EnviRonment (SORCER) for Large Scale, Distributed, Dynamic Fidelity Aeroelastic Analysis & Optimization

Raymond M. Kolonay
Air Force Research Laboratory
Wright-Patterson Air Force Base, Ohio 45433

Michael Sobolewski
Universal Technology Corporation
Dayton, Ohio 45432

I. Abstract

A computational framework that supports dynamic fidelity for aeroelastic analysis and optimization is presented.

The current research represents some of the recent developments in the Service Oriented Computing EnviRonment (SORCER)[1-6]. From an engineering perspective SORCER is a programming and computing environment that enables one to perform large scale system level engineering analysis and design space exploration that may be done on a geographically distributed heterogeneous computing environment. From a computer science view SORCER is a federated service-to-service (S2S) meta-computing environment that treats service providers as network peers with well-defined semantics of a service-object oriented architecture (SOOA). Here we present the evolution of SORCER from an Exertion-Oriented Programming paradigm to a Variable Oriented paradigm and its ability to enable dynamic fidelity of engineering responses and their respective sensitivities. In Exertion-Oriented Programming the SORCER framework supports the development of an engineering analysis or design process by enabling the developer to easily combine services on the network to create a process (Exertion) that performs multidisciplinary analysis or design of an engineering system. Hence, the application developer constructs the Exertions (Tasks and Jobs) and their relationships. In Exertion-Oriented Programming the focus is on engineering applications or services and the combination of these services to produce an analysis or result. Variable Oriented (VO) Programming focuses on a specific engineering quantity and views its calculation as a function or a function of functions as opposed to an engineering application or service. This effort describes the development and capability of the VO programming along with a demonstration of the dynamic fidelity by performing aeroelastic analysis with six different fidelities of induced drag. Euler based calculations with a Trefftz-plane, linear panel methods with a Trefftz-plane, standard one-point approximations with Euler and panel methods, and modified one-point approximations with Euler and panel methods. The fidelity can be selected dynamically by the analysis or optimization algorithm based on either accuracy or efficiency requirements.

II. Background

Over the past thirty years the two primary paradigms that have emerged to perform Multidisciplinary Analysis and Design (MAD) are “Monolithic Approach (MA)” and “Best in Class Approach (BCA)” systems. The monolithic approach consists of a single application, which contains all the necessary functionality or tools to model, analyze, and optimize a given component or an entire configuration. MA systems are usually easy to use, administer, and are robust but tend to be somewhat rigid and do not contain the latest tools or technology for all disciplines. BCA typically uses a scripting language to “glue” together several independent applications that provide the “best” functionality for a given domain. The BCA tends to give more flexibility but at the cost of robustness in the overall system. In recent years companies and organizations have been favoring BCA systems such as *ModelCenter*®[7], iSight[8], Execution Engine(formerly FIPER)[9], VisualDoc[10], OpenMDAO[11], and SORCER.

Companies and institutions are often organized by specialized areas for a given project or product. For example, in the aerospace industry within a project there are the aerodynamics group, controls group, structures group, and the propulsion group. These groups or Centers of Excellence (COE) are often physically separated and the product development process specified for inter-organization communication is “throwing things over the wall”. Historically this was not a problemsince the designs kept the disciplines orthogonal. That is, the mutual influence between aerodynamics and struc-

tures was limited and occurred primarily through loads. With this in mind there was no need and very little to be gained by integrating these two domains. With product performance requirements becoming more stringent and the need for robust, optimum and cost effective designs, organizations are becoming challenged to produce satisfactory designs. That is, in order to produce products that meet the new requirements, it is no longer proper to consider the domains orthogonal. The product needs to be developed from the system level, not at the individual component level followed by the assembly of these components. In addition, more detail is being required within a given domain in order to fully understand the product or predict its behavior. This being recognized by organizations, new processes no longer accept the "over the wall" technique for inter-organizational communication. The processes now requires that the organizations be integrated and collaborate extensively. Concurrently, development increasingly takes place at spatially distributed locations, with vendors and revenue sharing partners involved. Again, the process does not permit "over the wall" communications to these team members. Now, all participants in the design process need consistent real-time access to all current product information and applications adding additional strains on the integration requirements. The new organizational environments along with the demands for competitive products are producing two conflicting requirements: the need for very specialized domain specific expertise/applications and the need for the generalist/multidisciplinary design and analysis that considers the entire design of the product at the system level. One of the key factors in the success of future product development will be the ability to integrate the tools, data, and information from different domains and organizations during the development process[13].

SORCER is being developed with the above organizational structure, design process and resulting challenges in mind. Specifically, it is addressing the ability to integrate the tools, data, and information from different domains and organizations during the development process. It is a service-oriented product development environment providing an open flexible design environment, which allows universal availability and incorporation of existing data, tools/methods, processes and hardware as services distributed on a network and heeds the eight network computing fallacies identified by Deutsch[14]. It provides a common way to model an analysis and design process in conjunction with product data. It is a network-based distributed framework which supports collaboration among geographically distributed engineering and business partners. SORCER federates a series of network services (which may be distributed) in real time and orchestrates the communication between the services based on a user defined control strategy algorithm to execute a desired process and perform the required calculations, thus enabling one to perform large scale system level engineering analysis and design space exploration in a distributed heterogeneous computing environment. Large scale is defined as the integration of tens, hundreds, and possibly thousands of product development software tools and their data distributed across a network. This network may span organizational boundaries, company boundaries, or national boundaries. The product development tools addressed are primarily (but not limited to) engineering tools. These include computer aided design (CAD), computer aided manufacture (CAM) and computer aided engineering (CAE). The computing hardware environment is expected to range from laptops to high performance computing resources that have 10Ks of compute nodes. Run times for the software tools will be from seconds to weeks with input and output data in kilobytes to terabytes represented in various formats. In addition to the ability to perform multidisciplinary analysis other significant requirements for the computing environment are: ability to easily accommodate multiple fidelities/scales not only of the responses but also the response sensitivities with respect to independent variables, ability to perform non-deterministic analyses, and the ability to account for uncertainty quantification within the analyses and design space exploration. SORCER development supports these functional requirements.

III. Service-object Oriented Platform: SORCER

The Service ORiented Computing EnviRonment (SORCER) is a federated service-to-service (S2S) meta computing environment that treats service providers as network peers with well-defined semantics of a federated service-object oriented architecture that is based on the federated method invocation (FMI) [15]. It incorporates Jini semantics of services[16] in the network and the Jini programming model [17]with explicit leases, distributed events, transactions, and discovery/join protocols. While Jini focuses on service management in a networked environment, SORCER is focused on EO programming and the execution environment for exertions (see in Fig. 1). The SORCER programming environment creates the unifying representation for three concrete programming syntaxes: the SORCER Java API described in reference [18], and the graphical form in reference [19], and the functional composition form presented in this paper. The notation of functional composition has been developed for three related lan-

guages: Exertion-Oriented Language (EOL), Var-Oriented Language (VOL), and Var-Oriented Modeling Language (VML) that are complemented with the Java object-oriented syntax. In the following two sections we will describe the basic syntax of these three languages. EOL is fully described by Sobolewski in reference [23].

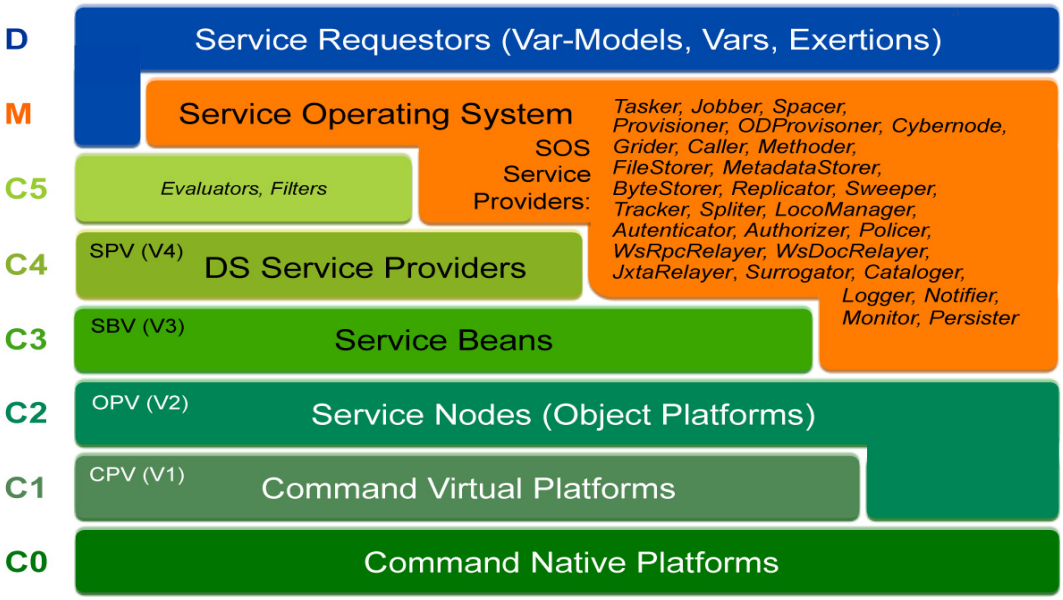

Figure 1. The SORCER layered architecture, where C0C5 (carrier)—the metaprocessor with its service cloud at C4 and C3, platform cloud at C2 and C1, M (management)—SORCER operating system, D (domain)—service requestors; where V1V4 stands virtualization of the corresponding layer

IV. Expressing a Process of Federating Processes: Exertion Oriented Programming

In language engineering, the art of creating languages, a metamodel is a model to specify a language. An exertion is a metamodel to model a connectionist process expression that models a behavioral phenomena as the emergent processes of interconnected networks of service providers. The central exertion principle is that a process can be described by the interconnected federation of simple and often uniform end efficient service providers that compete with one another for activation in the process.

Exertion-oriented programming (EOP) is a service-oriented programming paradigm using service providers and service commands. A service command-exertion-is interpreted by the SORCER Operating System (SOS) and represented by the data structure that consist of a data context, multiple service signatures, and a control context together with their interactions, to design distributed applications as service collaborations. In EOP a service signature determines a service invocation on a provider. The signature usually includes the service type, operation of the service type, and expected quality of service (QoS). While exertion's signatures identify (match) the required collaborating providers (federation), the control context defines for the SOS how and when the signature operations are applied to the data context. Note that the service type is the classifier of service providers with respect to its behavior (interface), but the signature is the classifier of service providers with respect to the invocation (operation in the interface) and service deployment defined by its QoS.

An exertion is an expression of a distributed process that specifies for the SOS how a service collaboration is actualized by a collection of providers playing specific roles used in a specific way[25]. The collaboration specifies a collection of cooperating providers, the exertion federation, identified by the exertion's signatures. Exertions encapsulate explicitly data, operations, and a control strategy for the collaboration. The signatures are dynamically bound to corresponding service providers, members of the exerted collaboration.

The exerted members in the federation collaborate transparently according to their control strategy managed by the SOS. The SOS invocation model is based on the Triple Command Pattern [26] that defines the federated method invocation (FMI).

A task exertion (or simply a task) is an elementary service command executed by a single service provider or its small-scale federation. The task federation is managed by the receiving provider for the same service context used by all providers in the federation. A job exertion is a composite service command defined hierarchically in terms of tasks and other jobs, including control flow exertions [27]. A job exertion is a kind of command script, that is similar conceptually to a UNIX script, but with service commands, to execute a large-scale federation. The job federation is managed by one of two SOS rendezvous providers, (Jobber or Spacer) but the task federation is managed by the receiving provider as previously stated. Either a task or job is a service-oriented program that is dynamically bound by the SOS to all required and currently available or provisioned on-demand service providers.

The exertion's data defined as the data *context* describes the information that tasks and jobs operate on. A data context, or simply a context, is a data structure that describes a service provider's ontology along with related data [27]. Conceptually a data context is similar in structure to a file system, where paths refer to objects instead of to files. A provider's ontology (paths) is controlled by the provider vocabulary that describes the data structures in a provider's namespace within a specified service domain of interest. A requestor submitting an exertion to a provider has to comply with that ontology as it specifies how the context data is interpreted and used by the provider.

The exertion collaboration defines its interaction. The exertion interaction specifies how invocations of signature operations are sent between service providers in a collaboration to perform a specific behavior. The interaction is defined by control contexts of all component exertions. From the computing platform point of view, exertions are entities considered at the programming level, interactions at the operating system level, and federations at the processor level. Thus, exertions are programs that define distributed collaborations on the service processor. The SOS manages these collaborations as interactions on its virtual service processor, the dynamically formed service federations (see Fig. 1).

In SORCER the provider is responsible for deploying the service on the network, publishing its proxy to one or more registries, and allowing requestors to access its proxy. Providers advertise their availability on the network; registries intercept these announcements and cache proxy objects to the provider services. The SOS looks up proxies by sending queries to registries and making selections from the available service types. Queries generally contain search criteria related to the type and quality of service. Registries facilitate searching by storing proxy objects of services and making them available to requestors. Providers use discovery/join protocols to publish services on the network; the SOS uses discovery/join protocols to obtain service proxies on the network. While the exertion defines the orchestration of its service federation, the SOS implements the service choreography in the federation defined by its FMI.

Three forms of EOP have been developed: Exertion-oriented Java API, interactive graphical, and textual programming. Exertion-oriented Java API is presented in [27]. Graphical interactive exertion-oriented programming is presented in [19]. Details regarding textual EOP and two examples of simple EO programs can be found in [23] and [28].

V. Expressing a Process of Converging Processes: Var-oriented Programming and Var-oriented Modeling

In every computing process variables represent data elements and the number of variables increases with the increased complexity of the problems being solved. The value of a computing variable is not necessarily part of an equation or formula as in mathematics. In computing, a variable may be employed in a repetitive process: assigned a value in one place, then used elsewhere, then reassigned a new value and used again in the same way. Handling large sets of interconnected variables for transdisciplinary computing requires adequate programming methodologies.

Var-Oriented Programming (VOP) is a programming paradigm using service variables called "Vars", data structures defined by the triplet <*value, evaluator, filter*>, together with a *Var* composition of evaluator's dependent variables to design var-oriented multi-fidelity compositions. It is based on dataflow principles that changing the value of a var should automatically force recalculation of the values of vars, which depend on its value. That is demand driven calculations. VOP promotes values defined by evaluators/filters to become the main concept behind any processing.

Var-Oriented Modeling (VOM) is a modeling paradigm using vars in a specific way to define heterogeneous multidisciplinary var-oriented models, in particular large-scale multidisciplinary analysis models including analysis and optimization component models. The programming style of VOM is declarative; models describe the desired results of the program, without explicitly listing commands or steps that need to be carried out to achieve the results. VOM focuses on how vars connect, unlike imperative programming, which focuses on how evaluators calculate.

VOM represents models as a series of interdependent var connections, with the evaluators/filters between the connections being of secondary importance.

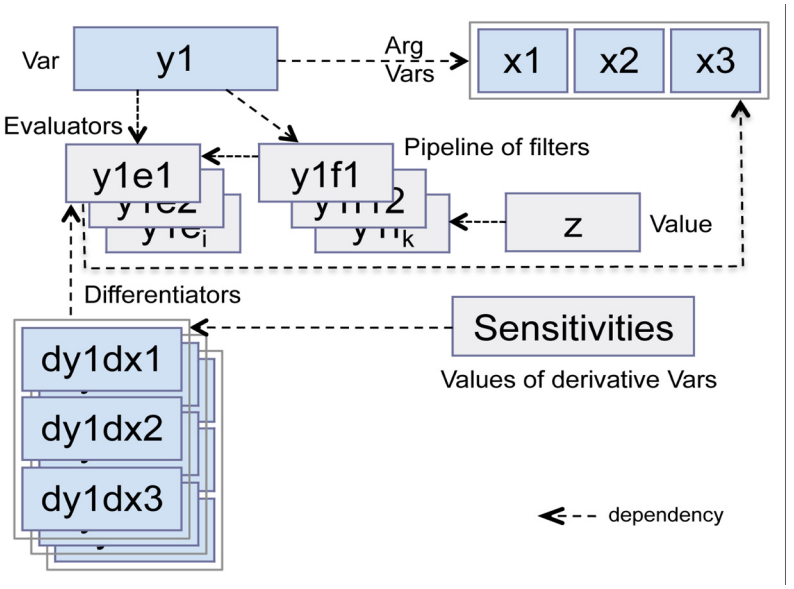

Figure 2. The var structure: value/evaluator/filter. Vars are indicated in blue color. The basic var y1, z=y1(x1, x2, x3), depends on its argument vars and derivative vars.

The SORCER meta modeling architecture [28] is the unifying representation for three concrete programming syntaxes: the SORCER Java API described in [27], the functional composition form[23],[28] and the graphical form described in reference [19]. The functional composition notation has been used for the Var-Oriented Language (VOL) and Var-oriented Modeling Language (VML) that are complemented with the Java object-oriented syntax.

The fundamental principle of functional programming is that a computation can be realized by composing functions. Functional programming languages consider functions to be data, avoid states, and mutable values in the evaluation process in contrast to the imperative programming style, which emphasizes changes in state values. Thus, one can write a function that takes other functions as parameters, returning yet another function. Experience suggests that functional programs are more robust and easier to test than imperative ones.

Not all operations are mathematical functions. In nonfunctional programming languages, "functions" are subroutines that return values while in a mathematical sense a function is a unique mapping from input values to output values. In SORCER the special type of variable called *var* allows one to use functions, subroutines, or coroutines in the same way. A value of var can be associated with a mathematical function, subroutine, coroutine, object, or any data. The concept of var links the three languages VOL, VML, and EOL into a uniform service-oriented programming model that combines federating services (EOP) with other type of process execution.

The semantics of a variable depends on the process expression formalism:

1. A variable in mathematics is a symbol that represents a quantity in a mathematical expression.
2. A variable in programming is a symbolic name associated with a value.
3. A variable in object-oriented programming is a set of object's attributes accessible via operations called getters.
4. A *var* in service-oriented programming is a triplet <*value, evaluator, filter*>, where:

a) a *value* is a valid quantity in an expression; a value is invalid when the current evaluator is changed, its arguments change, or is undefined;

b) an *evaluator* is a service with the var dependents that define the variable composition; and

c) a *filter*: is a getter operation.

Var-oriented programming is the programming paradigm that treats any computation as the triplet: value, filter (pipeline of filters), and evaluator (VFE, see Fig. 2). Evaluators and filters can be executed locally or remotely, sequentially or concurrently. An evaluator may use a *differentiator* to calculate the rates at which the var quantities

change. Multiple associations of evaluator-filter can be used with the same var enabling a var to havemulti-fidelity. The VFE paradigm emphasizes the usage of evaluators and filters to define the value of a var. The semantics of the value, whether the var represents a mathematical function, subroutine, coroutine, or just data, depends on the evaluator and filter currently used by the var. It is important to note that functions of functions can also be represented in the VFE model.

A service in VOL is the work performed by a variable's evaluator. Evaluators for a basic var, that depends on other vars arguments, define:

1. var composition via the var arguments of evaluator (dependent vars)
2. multiple processing services (multi-fidelity)
3. multiple differentiation services (multi-fidelity)
4. evaluators can execute commands (executable codes), object-oriented services (method invocations), and exertion-oriented services).

Thus, in the same process various forms of services (intra and interprocess) can be mixed within the same process expression in VOL. Also, the fidelity of var values can change as it depends on a currently used evaluator.

The variable evaluation strategy is defined as follows: the var value is returned if is valid, otherwise the current evaluator determines the variable's raw value (not processed or subjected to analysis), and the current pipeline of filters returns the output value from the evaluator result and makes that value valid. Evaluator's raw value may depend on other var arguments and those arguments in turn can depend on other argument vars and so on. This var dependency chaining is called the var composition and provides the integration framework for all possible kinds of computations represented by various types of evaluators including exertions via exertion evaluators.

In general, it is perceived that the languages used for either modeling or programing are different. However, both are complementary views of process expression and after transformation and/or compilation both need to be executable. An initial model, for example an initial design of an aircraft engine, can be imprecise, not executable, at a high level with informal semantics. However, its detailed model (detailed design) has to be precise, executable, and low level with execution semantics. Differences between modeling and programming that traditionally seemed very important are becoming less and less distinctive. For example models created with Executable UML[29] are precise and executable.

Data contexts (objects implementing the Context interface) with specialized aggregations of vars are called var-models. Three types of models: response, parametric, and optimization have been developed[23]. These models are expressed in VML using functional composition and/or the Java API for var-oriented modeling.

The modularity of the VFE framework, reuse of evaluators and filters, including exertion evaluators, in defining var-models is the key feature of variable-oriented programming (VOP) and var-oriented modeling (VOM). The same evaluator with different filters can be associated with many vars in the same var-model. VOM integrates var-oriented modeling with other types of computing via various types of evaluators. In particular, evaluators in var-models can be associated with commands (executables), messages (objects), and services (exertions).

Var-models support multidisciplinary and multi-fidelity computing. Var compositions across multiple models define multidisciplinary problems; multiple evaluators per var and multiple differentiators per evaluator define their multi-fidelity. These are called amorphous models. For the same var-model an alternative set of evaluators/filters (another fidelity) can be selected at runtime to evaluate a new particular process (“shape”) of the model and quickly update the related computation appropriately.

VI. Engineering Example: Induced Drag Design Optimization Model Definition, Configuration and Behavior in SORCER

When performing engineering analysis and design space exploration there is often the need to compute engineering quantities at different levels of fidelity. This may be due to different flight conditions or, depending on the search algorithm, one may wish to use various types of surrogate models. Here, the definition of fidelity is associated with the level of accuracy of a computed quantity. Higher fidelity implies greater accuracy but not necessarily greater computational cost. This example will focus on various levels of fidelity for computing induced drag including aeroelastic effects.

First, consider an optimization problem where the objective is to minimize the induced drag(D_I) of a system by selecting a trimmed angle of attack α along with the scheduling of various control surfaces cs_i. This can be stated in the following form:

$$\begin{gathered}\text{Minimize: } D_I(cs_i, \alpha) \\ \text{subject to Constraints} \\ cs^L < cs_i < cs^U \\ \alpha^L < \alpha < \alpha^U \\ L_T(cs_i, \alpha) = C_{L\alpha}\alpha + \sum_{n=1}^{ns} cs_i C_{L\beta_i}\end{gathered} \tag{1}$$

(2)

Where α and the cs_i represent a vector of design variables (x_j) and the equality constraint $L_T(cs_i, \alpha)$ enforces the trimmed state (lift) for the vehicle.

With the Munk[30] displacement theorem, one can calculate induced drag by using a Trefftz-plane located far downstream from the lifting surface. Utilizing this the induced drag, D_I, can be expressed solely in terms of the lift per unit span($Lpus_i$) and the span wise coordinates y .

$$D_I = \frac{1}{4\pi\rho_\infty V_0^2}\sum_{i=1}^{n} Lpus_i \Delta y_i \sum_{k=1}^{n} (Lpus_k - Lpus_{k+1})\left(\frac{1}{y_i - y_k} - \frac{1}{y_i + y_k}\right) \tag{3}$$

The sensitivities of D_I can be obtained by differentiating Equation (3) with respect to the design variables x_j.

$$\begin{aligned}\frac{\partial D_I}{\partial x_j} = \left(-\frac{1}{4\pi\rho_\infty V_0^2}\right)\Bigg\{&\sum_{i=1}^{n} \frac{\partial Lpus_i}{\partial x_j}\Delta y_i \sum_{k=1}^{n} (Lpus_k - Lpus_{k+1})\left(\frac{1}{y_i - y_k} - \frac{1}{y_i + y_k}\right) + \\ &\sum_{i=1}^{n} Lpus_i \Delta y_i \sum_{k=1}^{n}\left(\left(\frac{\partial Lpus_k}{\partial x_j} - \frac{Lpus_{k+1}}{\partial x_j}\right)\left(\frac{1}{y_i - y_k} - \frac{1}{y_i + y_k}\right)\right)\Bigg\}\end{aligned} \tag{4}$$

For the current work the x_j are defined in Figure 3, consisting of 20 trailing edge control surfaces ($cs_1 \ldots cs_{20}$) and the free stream angle of attack (α) of the wing. Since a full span model is considered there are an additional 20 trailing edge control surfaces on the left wing as well. These are shown in Figure 3 as $cs_{21} \ldots cs_{40}$

Inspection of Equation (4) reveals that the calculation of $\frac{\partial Lpus_i}{\partial x_j}$ are required. For the present study $\frac{\partial Lpus_i}{\partial x_j}$ are determined by finite difference making the sensitivities semi-analytic.

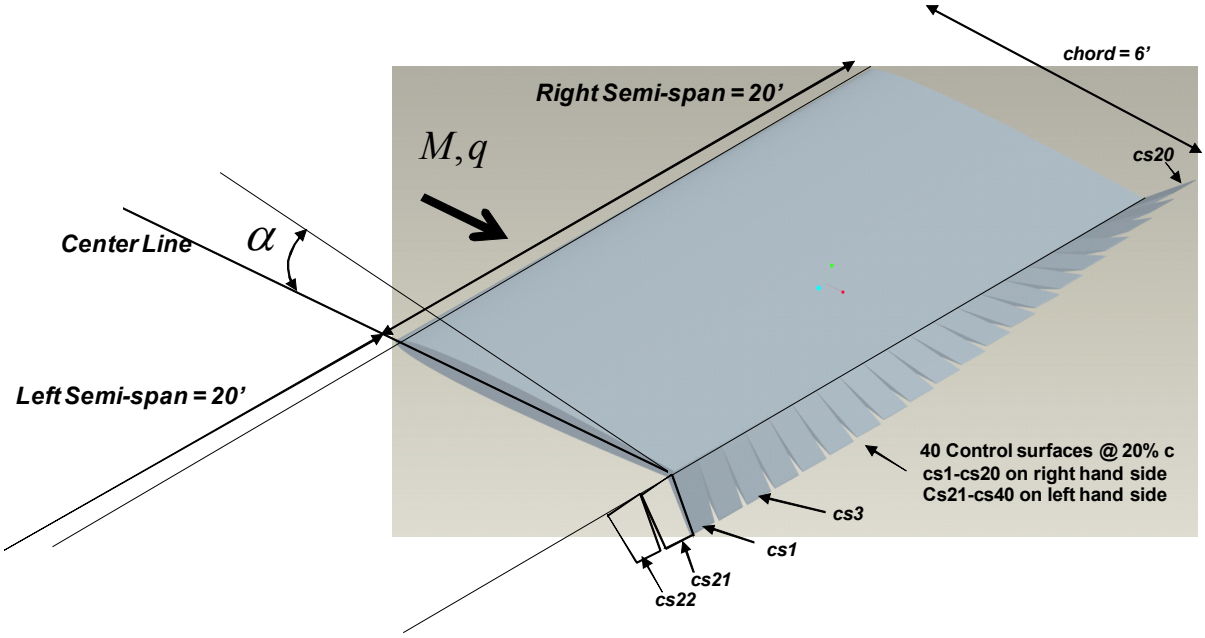

Figure 3. Goland Wing Aerodynamic Configuration

Functionally we see that $Lpus_i(x_j)$ and $D_I(Lpus_i)$ which can be written as

$$D_I(Lpus_i(x_j)). \tag{5}$$

This is the functional composition to be represented in SORCER. In addition, we would like to compute $Lpus_i(x_j)$ and $D_I(Lpus_i)$ with various fidelities. As discussed previously a SORCER *var* consists of three components; *Value, Evaluators, and Filters*. The purpose of this example is to take the engineering functional relationship defined in Equation (5) and create a Model with multiple fidelities in SORCER. Currently SORCER has three types of models; ResponseModel, ParametricModel, and OptimizationModel. Here the development of the Optimization-Model will be demonstrated. This will require the definition of the independent design variables, response variables, objective function, and constraints followed by the configuration of each variable with the appropriate Filters and Evaluators.

For the case depicted in Equation (5) there are a total of 21 design variables, 20 design variables associated with $cs_1 \ldots cs_{20}$ and one design variable associated with α. The remaining control surface variables $cs_{21} \ldots cs_{40}$ will be "linked" (a set relationship) to $cs_1 \ldots cs_{20}$. The response variables consists of the 40 lift per unit span values, $Lpus_i$, the induced drag, D_I, and the total lift, L_T. The different fidelities for the responses $Lpus_i(x_j)$ and $D_I(Lpus_i)$ along with their respective sensitivities that are to be represented by the model can be found in Table 1. Table 1 indicates that $Lpus_i(x_j)$ has four different fidelities for its responses and sensitivities while $D_I(Lpus_i)$ has six fidelities.

The four fidelities used to represent $Lpus_i(x_j)$ are; linear potential method (LPM), standard one point Taylor Series Approximation using linear potential methods (SOA-LPM), Euler method (EM), and Standard One point Taylor Series Approximation using the Euler method (SOA-EM). The sensitivities for the LMP and the EM are computed by forward finite difference (FFD) and used to construct the SOA-LPM and SOA-EM respectively. The six fidelities for $D_I(Lpus_i)$ are; LPM, SOA-LPM, EM, SOA-EM along with a Modified One point Taylor Series Approximation where instead of performing the Taylor Series approximation directly on $D_I(Lpus_i)$ it employes the linear approximation on $Lpus_i(\bar{x})$ and substitutes this into Equation (3) and Equation (4). This results in maintaining the nonlinear behavior of the function and it's respective sensitivities. Details of the SOA and the MOA used can be found in reference [32].

Table 1. Fidelities for $Lpus_i$, $\frac{\partial Lpus_i}{\partial x_i}$, D_I, and $\frac{\partial D_I}{\partial x_i}$

Fidelity	$Lpus_i$	$\frac{\partial Lpus_i}{\partial x_i}$	D_I	$\frac{\partial D_I}{\partial x_i}$
Liner Potential Method (LPM)	X	FFD	X	FFD
Standard One Point Approximation with Linear Potential Method (SOA-LPM)	X	Constant	X	Constant
Euler Method (EM)	X	FFD	X	FFD
Standard One Point Approximation with Euler Method (SOA-EM)	X	Constant	X	Constant
Modified One Point Approximation with Linear Potential Method (MOA-LPM)			X	Semi-Analytic
Modified One Point Approximation with Euler Method (MOA-EM)			X	Semi-Analytic

A. Model Definition

There are two primary steps in constructing a model within the SORCER environment. These are model definition and model configuration. The model definition is essentially a skeleton or meta information about the model. To define the optimization model describe in Equation (1) with its various fidelities defined in Table I the following syntax is used.

```
OptimizationModel omodel = optimizationModel("Induced Drag Optimization Model",
designVars(vars(loop("i",1,20),"cs$i$", 0.2, -10.0, 10.0)),
linkedVars(names(loop("i",21,40),"csl$i$")),
designVars(var("alpha", 5.0, -5.0, 10.0)),


responseVars(loop("i",1,40),"Lpus$i$",
  realization(
          evaluation("Lpus$i$LinearPotentialExact", "LpusAstrosExacte",
               differentiation(wrt(names(loop("k",1,20),"cs$k$"),"alpha","q"),
                    gradient("Lpus$i$AstrosExacteg1"))),
          evaluation("Lpus$i$LinearPotentialSOA","Lpus$i$AstrosSOAe",
               differentiation(wrt(names(loop("k",1,20),"cs$k$"),"alpha","q"),
                    gradient("Lpus$i$AstrosSOAeg1"))),
          evaluation("Lpus$i$EulerExact", "LpusAvusExacte",
               differentiation(wrt(names(loop("k",1,20),"cs$k$"),"alpha"),
                    gradient("Lpus$i$AvusExacteg1"))),
          evaluation("Lpus$i$EulerSOA","Lpus$i$AvusSOAe",
               differentiation(wrt(names(loop("k",1,20),"cs$k$"),"alpha"),
                    gradient("Lpus$i$EulerSOAeg1"))),

responseVar("DI",
  realization(
           evaluation("LinearPotentialExact", "DIAstrosExacte",
              differentiation(wrt(names(loop("i",1,20),"Lpus$i$"),"q"),
                    gradient("DIAstrosExacteg1"))),
           evaluation("LinearPotentialSOA", "DIAstrosSOAe",
              differentiation(wrt(names(loop("i",1,20),"Lpus$i$"),"q"),
                     gradient("DIAstrosSOAeg1"))),
           evaluation("LinearPotentialMOA", "DIAstrosMOAe",
               differentiation(wrt(names(loop("i",1,20),"Lpus$i$"),"q"),
```

```
                    gradient("DIAstrosMOAeg1"))),
          evaluation("EulerExact", "DIAvusExacte",
               differentiation(wrt(names(loop("i",1,20),"Lpus$i$"),"q"),
                    gradient("DIAvusExacteg1"))),
          evaluation("EulerSOA", "DIAvusSOAe",
               differentiation(wrt(names(loop("i",1,20),"Lpus$i$"),"q"),
                    gradient("DIAvusSOAeg1"))),
          evaluation("EulerMOA", "DIAvusMOAe",
               differentiation(wrt(names(loop("i",1,20),"Lpus$i$"),"q"),
                    gradient("DIAvusMOAeg1"))),

responseVar("LT",
  realization(
          evaluation("LinearPotentialExact", "LTAstrosExacte",
               differentiation(wrt(names(loop("i",1,20),"Lpus$i$")),
                    gradient("LTAstrosExacteg1"))),
          evaluation("LinearPotentialSOA", "LTAstrosSOAe",
               differentiation(wrt(names(loop("i",1,20),"Lpus$i$")),
                    gradient("LTAstrosSOAeg1"))),
          evaluation("EulerExact", "LTAvusExacte",
               differentiation(wrt(names(loop("i",1,20),"Lpus$i$"), "q"),
                    gradient("LTAvusExacteg1"))),
          evaluation("EulerSOA", "LTAvusSOAe",
               differentiation(wrt(names(loop("i",1,20),"Lpus$i$"), "q"),
                    gradient("LTAvusSOAeg1")))

objectiveVars(var("DIo", "DI", Target.min )),
constraintVars(var("LTc", "LT", Relation.eq, 1000.0)))
```

This is a single statement with embedded statements. An `OptimizationModel` Object is created called `omodel`, the name of the model is "`Induced Drag Optimization Model`". This name is arbitrary and user specified. The following describes the constructors;

`designVars("cs",20)` - Returns a list of 20 `Variable` objects with the *type* attribute set to "DESIGN" and names "cs1", "cs2",... "cs20".

`designVars("alpha")` - Returns a single Variable object with *type* attribute of "DESIGN" and name "alpha"

`linkedVars("cs", 20,21)` - Returns a list of 20 Variable objects with *type* attribute "DESIGN" and *kind* attribute of "LINKED" with names "cs21", "cs22", ..."cs40"

`responseVars(loop("i",1,40),"Lpus$i$",-` Returns a list of 40 Variable objects with *type* attribute of "RESPONSE" with names "Lpus1", "lpus2", ..."lpus40".

As can be seen each response Var contains multiple evaluation objects and each evaluation contains a single differentiation object. In general the evaluation can contain multiple differentiation objects to allow multi-fidelity for sensitivities as well. Each evaluation specifies a fidelity for each "Lpusi" response and its respective sensitivities. The response variables for Induced Drag "DI" and total lift "LT" are defined in a similar fashion.

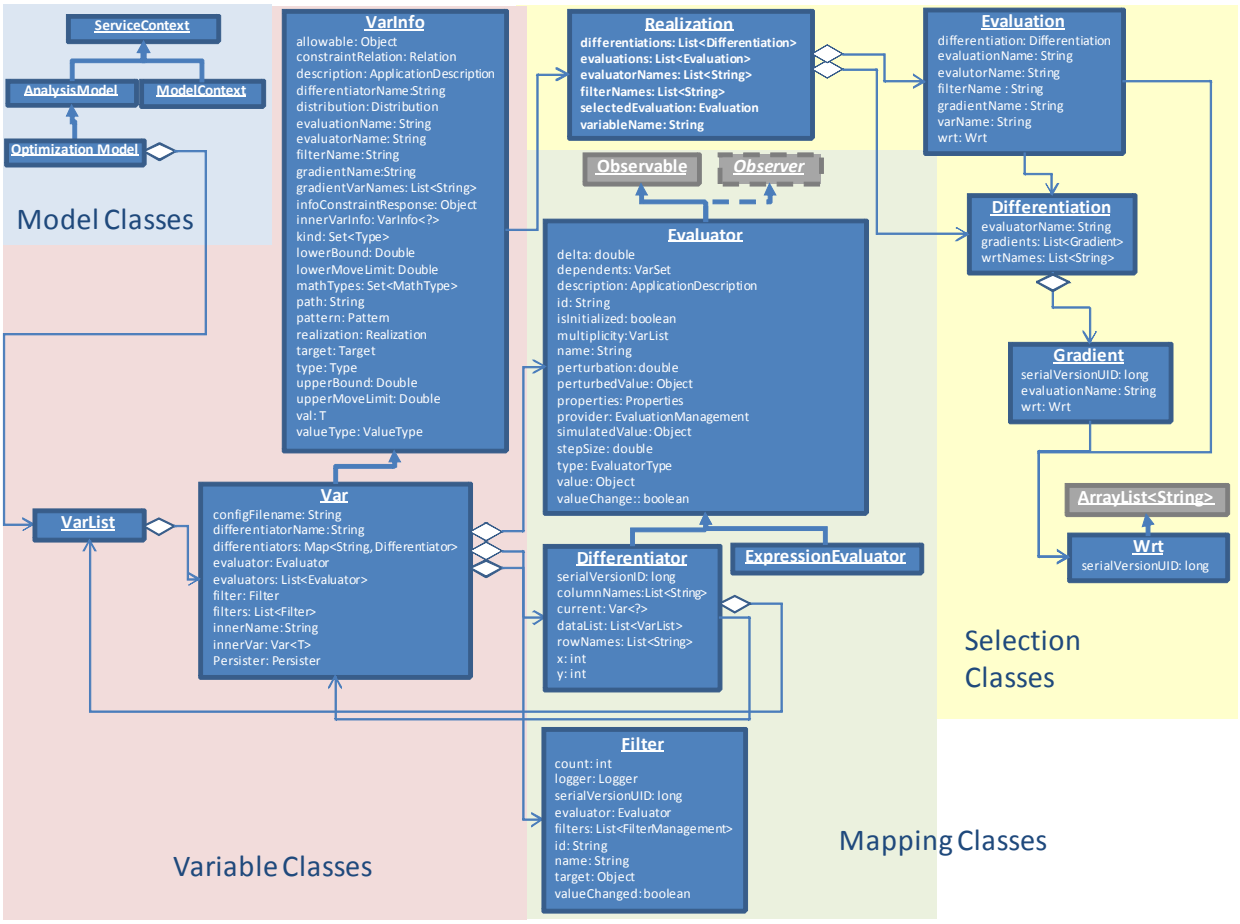

Figure 4. Var, Filter, Evaluation Class Diagram

A. Model Configuration

The next step required is to configure the DesignVariables, LinkedVariables, and the ResposneVariables.

a. configure the Design Variables - Each independent design variable consists of zero, one, or more `Filters`. For this case the independent design variables reside in three locations depending on the fidelity being employed. For the LPM, which uses the ASTROS application to compute the $Lpus_i$, all the independent Vars are in the ASTROS input file. For the EM, which uses the Air Vehicles Unstructured Solver (AVUS), The cs_i variables reside in an ascii text file(AVUS_Boundary_Condition.dat) and α resides in a separate ascii text file (AVUS_Job.job). In order to have the ability to get and set the values in these files from anywhere on the network (we have no idea where AVUS may run) we need to create `Filter` objects and place them into the Variables. Filters take a larger entity and "filter" out a portion of that entity for either reading, writing or passing to another filter. Since we are working with ascii text files a `BasicFileFilter` object will be used. Each `BasicFileFilter` requires a `Pattern` object which is used to locate the desired quantity in the entity. Here our pattern is essentially *line, field, delimiter* information. Other patterns such as regular expressions are available as well. As an example the filters for the EM fidelity are developed. Figure 5 is an excerpt from the AVUS_Boundary_condition.dat file and the AVUS_Job.job file (the line numbers are not actually in the file. They are shown here to indicate the location in the file the excerpt has been taken).

```
187 Upper CS01
188 Transpiration
189 Boundary
190 Yes
191 TBC-Type  Rx  Ry  Rz  Thetax Thetay Thetaz
192 1 4.5 0.0 0.0 0.0 12.0 0.0
193 ######################################################################
194  32
195 Upper CS02
196 Transpiration
197 Boundary
198 Yes
199 TBC-Type  Rx  Ry  Rz  Thetax Thetay Thetaz
200 1 4.5 0.0 0.0 0.0 0.0 0.0
201 ######################################################################
202  33
203 Upper CS03
204 Transpiration
205 Boundary
206 Yes
207 TBC-Type  Rx  Ry  Rz  Thetax Thetay Thetaz
208 1 4.5 0.0 0.0 0.0 0.0 0.0
209 ######################################################################
210  34
211 Upper CS04
212 Transpiration
213 Boundary
214 Yes
215 TBC-Type  Rx  Ry  Rz  Thetax Thetay Thetaz
216 1 4.5 0.0 0.0 0.0 0.0 0.0
217 ######################################################################
```

Figure 5. Excerpt from AVUS_Boundary_Condition.dat

```
94 ****************************************************
95 UNITS (1=MKS, 2=CGS, 3=FOOT-SLUG-SEC, 4=INCH-SNAIL-SEC)
96   3
97 MACH NO.   ANGLE OF ATTACK   ANGLE OF SIDESLIP
98 0.85 5.14159 0.0
99 STATIC PRESSURE    STATIC TEMPERATURE
100   -1.             -1.
101 GAMMA    GAS CONSTANT    PRANDTL NUMBER    GRAVITY
102   -1.       -1.            -1.               0.
103 ****************************************************
104               INITIAL CONDITIONS
```

Figure 6. Excerpt from AVUS_Job.job file

To configure the Var first a Pattern is created and is supplied to construct the Filter. Once the Filter is obtained it can be assigned to a specific Var in the mode.

Below is the syntax for creating the Pattern objects for the filter for cs_1 .

```
// create the patterns for the csi filters
Pattern cs1p= new BasicPattern("cs1", "File", "Double", 192, 6, " ");
Pattern cs2p = new BasicPattern("cs2", "File", "Double", 200, 6, " " );
Pattern cs3p = new BasicPattern("cs3", "File", "Double", 208, 6, " " );
Pattern cs4p = new BasicPattern("cs4", "File", "Double", 216, 6, " " );
.
.
.
Pattern alphap = new BasicPattern("alpha", "File", "Double", 98, 2, " ");
```

The arguments for the BasicPattern construction are the String *patternName* , String *patternTyp*e, String *data-Type*, int *line*, int *field*, String *delimiter*. For the above cases we see that cs1p identifies the value at line 192, field 6, using a "space" delimiter between each field. Hence it references the Thetay value that currently holds a value = 12.0. Twenty patterns need to be created for the cs1-cs20 and a final Pattern created for alpha. Once all of the Patterns are created then the BasicFileFilters can be constructed for each item.

```
BasicFileFilter cs1bff = new BasicFileFilter(bcURL, cs1p);
BasicFileFilter cs2bff = new BasicFileFilter(bcURL, cs2p);
BasicFileFilter cs3bff = new BasicFileFilter(bcURL, cs3p);
BasicFileFilter cs4bff = new BasicFileFilter(bcURL, cs4p);
.
.
.
BasicFileFilter abff = new BasicFileFilter(alphaURL, alphap);
```

The arguments for the BasicFileFilter constructor used here (there are others) are a URL object and a Pattern object. The URL objects are references to the respective files; AVUS_Boundary_Condition.dat for all the cs_i and AVUS_Job.job for α .

The final step is to add the BasicFileFilter objects to each of the design Variables. The below lines first extract the design Variables from the ResponseModel and then adds the Filters to the Variables.

```
omodel.getDesignVariable("cs1").setFilter(cs1bff);
omodel.getDesignVariable("cs2").setFilter(cs2bff);
omodel.getDesignVariable("cs3").setFilter(cs3bff);
.
.
.
omodel.getDesignVariable("alpha").setFilter(abff);
```

This completes the configuration of the Design Variables. At this point the programmer can perform gets and sets on these objects which will read or write values to the respective files. For example if one wishes to set the value for cs1=15.0 the following syntax could be used

```
Variable cs1Var = omodel.getDesignVariable("cs1");
cs1Var.setValue(15.0);
```

or all in one line

```
omodel.getDesignVariable("cs1").setValue(15.0)
```

This would cause the value in the file AVUS_Boundary_Condition.dat at line 192, field 6 to be changed to 15.0. Another useful method available is to have a local copy of the file made (anywhere on the network) and update the value in the local copy of the file.

```
cs1Var.setValueLocal(15.0);
```

b. configure the Linked Variables - Each linked variable consists of an `Evaluator` and one or more `Filters`. For this case the linked variables are the $cs_{21} - cs_{40}$ and reside in the ascii text file AVUS_Boundary_Condition.dat for the EM fidelity and the ASTROS input file for LPM. Patterns and BasicFileFilters need to be constructed in the same fashion as those for the $cs_1 - cs_{20}$ design Variables.

```
Pattern cs21p = new BasicPattern("cs21", "File", "Double", 512, 6, " " );
Pattern cs22p = new BasicPattern("cs22", "File", "Double", 520, 6, " " );
Pattern cs23p = new BasicPattern("cs23", "File", "Double", 528, 6, " " );
.
.
.
Pattern cs40p = new BasicPattern("cs40", "File", "Double", 664, 6, " " );

BasicFileFilter cs21bff = new BasicFileFilter(bcURL, cs21p);
BasicFileFilter cs22bff = new BasicFileFilter(bcURL, cs22p);
BasicFileFilter cs23bff = new BasicFileFilter(bcURL, cs23p);
.
.
.
BasicFileFilter cs40bff = new BasicFileFilter(bcURL, cs40p);
```

Now there is a difference when configuring a linked Variable versus a design Variable. Instead of setting the filter on the Variable we set the Persister (with the Filter) with the following syntax.

```
omodel.getLinkedVariable("cs21").setPersister(cs21bff);
omodel.getLinkedVariable("cs22").setFilter(cs22bff);
omodel.getLinkedVariable("cs23").setFilter(cs23bff);
```

The last task that needs to be performed is the development of the functional relationship between the linked Variable and the design Variable. Here is the simple functional relationships that we wish to enforce

$$\begin{aligned} cs_{21} &= cs_1 \\ cs_{22} &= cs_2 \\ &\cdots \\ cs_{40} &= cs_{20} \end{aligned} \tag{6}$$

It should be noted that any arbitrary functional relationship can be used. To create the functional relationship we have to create Evaluators. Evaluator Objects essentially perform computations given an set of inputs or Variables in this case, and the result of the calculations can be sent to a Variable Filter or Variable Persister. To create an Evaluator (a Java Expression Parser - JepEvaluator is used here, see the sorcer.calculus.evaluator package for a complete list of available evaluators) for the relationship $cs_{21} = cs_1$ the following syntax is used.

```
Evaluator lcs21 = new JepEvaluator("cs21", "cs1");
```

This creates a Jep Expression $cs_{21} = cs_1$. For the remaining linked Variables we have

```
Evaluator lcs22 = new JepEvaluator("cs22", "cs2");
Evaluator lcs23 = new JepEvaluator("cs23", "cs3");
.
.
.
Evaluator lcs40 = new JepEvaluator("cs40", "cs20");
```

Next we define the Evaluator dependencies

```
Evaluator lcs21.addDependent("cs1")
Evaluator lcs22.addDependent("cs2")
```

```
Evaluator lcs23.addDependent("cs3")
.
.
.
Evaluator lcs40.addDependent("cs20")
```

Now that the Evaluator and it's dependencies have been defined the Evaluator can now be assigned to the linked Variable.

```
omodel.getLinkedVariable("cs21").setEvaluator(lcs21,"EulerExact");
omodel.getLinkedVariable("cs22").setEvaluator(lcs22,"EulerExact");
omodel.getLinkedVariable("cs23").setEvaluator(lcs23,"EulerExact");
.
.
.
omodel.getLinkedVariable("cs40").setEvaluator(lcs40,"EulerExact");
```

The behavior that will occur now is the following. If a design Variable, say cs_3, has it's value set, then all linked Variables that depend on cs_3 will be set according to their respective Evaluators. So for our case performing the following statement

```
omodel.getVariable("cs3").setValue(19.5,"EulerExact")
```

Results in not only the value in the file AVUS_Boundary_Condition.dat at line 208, field 6 to be changed to 19.5 but also the value in the file AVUS_Boundary_Condition.dat at line 528, field 6 to be changed to 19.5 as well.

c. Configuration of the Response Variables - The response Variables expose calculated quantities or engineering responses computed within the SORCER environment. Response Variables consists of an `Evaluator` and one or more `Filters`. Evaluators are the actual compute engines and they have potential dependencies. That is they depend on `Vars` (other Response, Design, and indirectly Linked) as their input. Since in SORCER a Response Variable, like all Vars, must reduce to a single scalar object and Evaluators compute essentially "blobs", Filters are used to "filter" the "blobs" to the scalar entity the response Variable represents. First, configure the $Lpus_i$ response Variables. There are 40 response Variables on the semi-span for the $Lpus_i$ and they are dependent on the cs_{1-40}. and α. Functionally this can be written as $Lpus_i(cs_1 \ldots cs_{40}, \alpha)$. First the Evaluators for the $Lpus_i$ are developed. As it turns out all 20 $Lpus_i$ are computed by performing a single AVUS run. Hence all 20 $Lpus_i$ response variables will use the same Evaluator. This Evaluator executes AVUS for a given set of cs_i α and produces an AvusOutput Object that contains all $Lpus_i$ values. The type of Evaluator used is an *ExertionEvaluator*. An ExertionEvaluator can be either a single Task or an entire Job. Each Exertion has a simple method that returns an Evaluator object for itself. The below syntax shows the configuration of the "Lpusi"

```
// get a list of the Design Variables and a list of the Response Variables
// from the model to be used in the Response Variable Configuration
VarList<Variable> dvs = omodel.getDesignVars();
VarList<Variable> rvs = omodel.getResponseVars();

// configure the lpus response variables.
//construct the exertion evaluator for the lift per unit span (lpus)
// response variables

// Construct the avusTask
Task avusTask = getAvusTask();
Evaluator avusEvaluator = avusTask.getEvaluator();

// add the dependencies to the avusEvaluator evaluator (all designVars in
// the omodel model)
```

```
avusEvaluator.addDependents(dvs);
```

Once the Evaluator with it's dependencies has been constructed, which will be used by all of the $Lpus_i$ response variables, each $Lpus_i$ response Variable sets it's Evaluator to the avusEvaluator.

```
// Set the Evaluator for each of the Lpus response variables
rvs.get(1).setEvaluator(avusEvaluator);
rvs.get(2).setEvaluator(avusEvaluator);
.
.
.
rvs.get(20).setEvaluator(avusEvaluator);
```

Now construct the Filters for each "Lpusi" response Variable. The goal is to filter from the result of the Evaluator (Task in this case) to the individual values for the $Lpus_i$. In order to do this it is necessary to see what is in the AvusTask. Figure 7 illustrates the objects in the Task object. Specifically the `MethodSignature` and the `Context`.

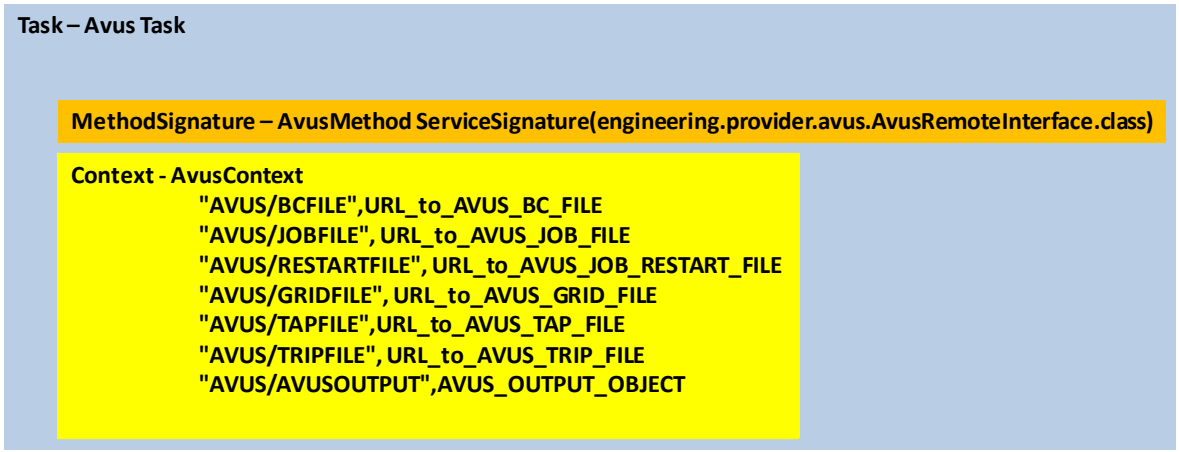

Figure 7. AVUS Task, Method, Context

The `Context` contains the input and output produced by executing the task/Evaluator. Of specific interest here is the last entry in the Context, the entry that has the path "AVUS/AVUSOUTPUT" and the data AVUS_OUTPUT_OBJECT. The AVUS_OUTPUT_OBJECT is an object(AvusOutput.java) that contains much of the information generated from an Avus run and in particular a field named "`windAxisLpus`" this field is an object that contains a field which is a array which in turn contains the $Lpus_i$ values. Hence a set of filters are necessary to extract the AVUS_OUTPUT_OBJECT from the Task, extract the "windAxisLpus" field from the AVUS_OUTPUT_OBJECT and then a final filter that extracts the i th $Lpus$ from the array. Figure 8 depicts this filtering process graphically. One can think of this as "piping" the result of one filter to the next. This is similar to the concept of piping in the unix shell environment.

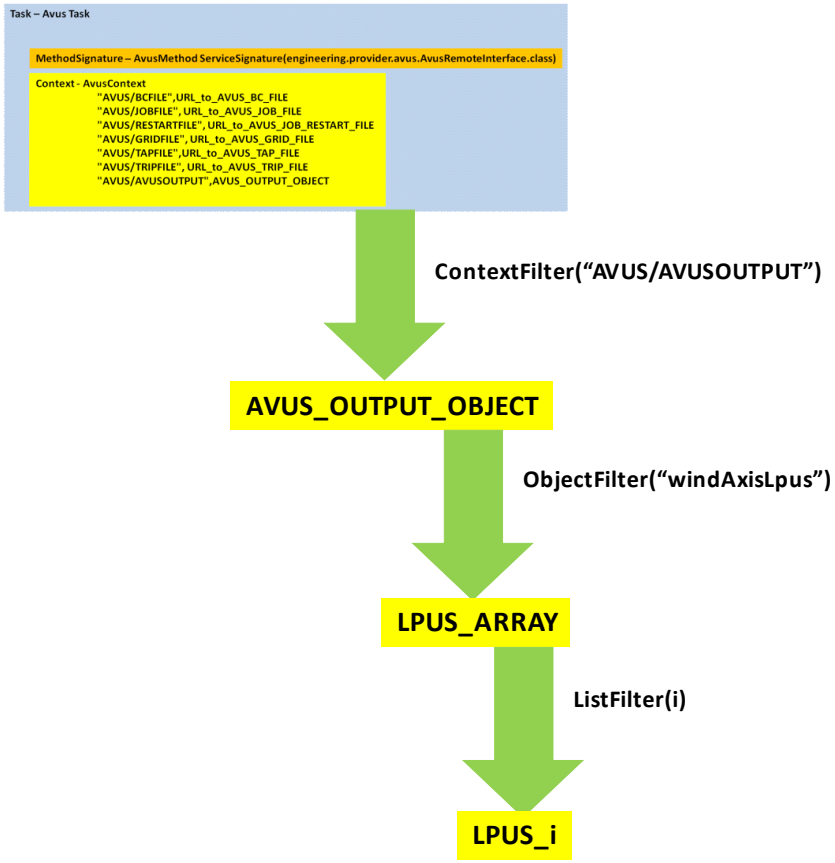

Figure 8. Filter Pipeline for obtaining LPUS_i

The following syntax creates filters shown in Figure 8.

```
// create the filter pipeline for each of the Lpus rvs's
// The first filter is the ContextFilter that extracts the
// AVUS_OUTPUT_OBJECT from the Evaluator(AvusTask) Context at the path
// location of "AVUS/AVUSOUTPUT"

ContextFilter contextFilter = new ContextFilter("avusTaskContext",
"AVUS/AVUSOUTPUT");

// Object Filter to extract the windAxisLpus field from the
// AVUS_OUTPUT_OBJECT

ObjectFilter objFilter = new ObjectFilter("windAxisLpus");

// List filter to extract the ith  Lpus from the windAxisLpus
// Array
ListFilter lf1 = new ListFilter(1);
ListFilter lf2 = new ListFilter(2);
ListFilter lf3 = new ListFilter(3);
.
```

```
.
.
ListFilter lf20 = new ListFilter(20);


// Finally create the pipeline of filters (filtering will occur in this
// order
Filter rvFilter = new Filter(contextFilter, objFilter, lf1);
// Set the filter on the ith(1st here) Response Variable
rvs.get(1).setFilter(rvFilter);
Filter rvFilter = new Filter(contextFilter, objFilter, lf2);
rvs.get(2).setFilter(rvFilter);
Filter rvFilter = new Filter(contextFilter, objFilter, lf3);
rvs.get(3).setFilter(rvFilter);
.
.
.
Filter rvFilter = new Filter(contextFilter, objFilter, lf20);
rvs.get(20).setFilter(rvFilter);
```

This completes the configuration of the $Lpus_i$ response variables. The following is a few examples of the behavior of the environment for the defined design and response variables under different conditions. Note: this assumes that the fidelity for all the Vars is set to "EulerExact"

Example 1

// set cs1=15.0

dvs.get(1).setValue(15.0)

// get value of $Lpus_1$

rvs.get(1).getValue();

The statement set cs_1=15.0 causes cs_1 and cs_{21} (remember they are linked) to be changed in the boundary condition file. The statement rvs.get(1).getValue(); Will cause the Evaluator to "evaluate" the value for $Lpus_1$ since some of it's dependencies (cs_1 in this case) have changed. The Evaluator will execute the AvusTask to compute a new $Lpus_1$.

Example 2

// set cs1=15.0

dvs.get(1).setValue(15.0)

// get value of $Lpus_1$

rvs.get(2).getValue();

The the statement set cs_1=15.0 causes cs_1 and cs_{21} (remember they are linked) to be changed in the boundary condition file. The statement rvs.get(1).getValue(); Will cause the Evaluator to "evaluate" the value for $Lpus_1$ since some of it's dependencies (cs_1 in this case) have changed. The Evaluator will execute the Avus task to compute a new $Lpus_1$. The statement rvs.get(2).getValue() retunes the value for $Lpus_2$. It **does not** cause the Evaluator to "evaluate". Remember all the $Lpus_i$ response variables have the same Evaluator (just different Filters) and none of this Evaluator's dependencies have changed since the dvs.get(1).getValue() command.

Now consider the configuration of the induced drag variable D_I. This requires constructing the functional relationship found in Equation (5) $D_I(Lpus_i(cs_1 \ldots cs_{40}, \alpha))$. A separate Class called InducedDrag.java has been created to compute D_I. This class has a method called *evaluateInducedDrag* which takes as an argument a list of Variables. These Variables are expected to be the Lpus_i response Variables. The method essentially computes Equation (3). For this a *MethodEvaluator* will be used.

```
// Construct the InducedDrag Object
InducedDrag idragObj = new InducedDrag("avusIdrag", yiA);
// Create the Method Evaluator
MethodEvaluator iDragMethodEval = new MethodEvaluator("iDragEvaluator",
 idragObj, "evaluateIDrag");
// Set the arguments for the Method
iDragMethodEval.setArgs(lpusVars);

// set the iDrag evaluator dependencies
iDragMethodEval.addDependents(lpusVars);

// add the Evaluator to the iDrag response Variable
rvs.getVariable("iDrag").setEvaluator(iDragMethodEval);
```

This completes the configuration of the idrag response Variable.

Example behavior assuming fidelity is set as "EulerExact":
// set cs1=15.0
dvs.get(1).setValue(15.0)
// get value of D_I
rvs.getVariable("iDrag").getValue();

The statement set cs_1 =15.0 causes cs_1 and cs_{21} (remember they are linked) to be changed in the boundary condition file. The statement rvs.getVariable("iDrag").getValue(); Will cause the Evaluator to "evaluate" the value for D_I since some of it's implicit dependencies (cs_1 in this case) have changed. It causes the Lpus_i Evaluator (avusEvaluator) to "evaluate" since it is its explicit dependency, cs_1, that has changed. Once this completes, the idragMethodEvaluator evaluates with the updated set of Lpusi's, its explicit dependencies, and produce a new value for induced drag.

d. Model Fidelity Selection & Query - In the previous section an optimization model was defined and configured. In addition, a few examples were given showing how an individual Var could be manipulated and its resulting behavior. Here we will show how one can interact with the model as a whole. Specifically to change the fidelity of the model Vars and to query the model for information such as responses and sensitivities. The interaction with the model is carried out by suppling a *ModelContext* object to the model. This object contains information to reconfigure the model (select fidelities), update the model, make additions to the model, and or query the model for information. As an initial example consider the desire to obtain the objectives, constraints and respective gradients from the model for the values of "cs1" = 2.0 and "cs15"=-0.041. The below syntax shows how this is done.

```
// create ModelContext object
ModelContext modelContext = new ModelContext("Induced Drag Opti Model Query");
//create VarInfoList specifying the desired values for cs1 and cs15
VarInfoList designInfo = varsInfo(varInfo("cs1", 2.0),varInfo("cs15", -0.041)
//
modelContext.setDesignVarsInfo(designInfo);
modelContext.setObjectivesInfo(null);
modelContext.setConstraintsInfo(null);
modelContext.setObjectivesGradientInfo(null);
modelContext.setConstraintsGradientInfo(null);
// Send the modelContext to the model for the query
modelContext = (ModelContext)omodel.evaluate(modelContext);
```

The above results in the model setting the values of "cs1" = 2.0 and "cs15"=-0.041 and evaluating the constraints, objectives, and their respective gradients and returns them in the modelContext. It is worthwhile to note that a "null" passed into the query indicates that all of that information is to be computed. If one desires only a subset then

a *VarInfoList* containing the subset of desired quantities should be suppled. For example if only the responses “Lpus1”, “Lpus2”, and Lpus3” were desired from the model the following query would be used.

```
VarInfoList lpusiInfo=varsInfo(varInfo(“Lpus1”),varInfo(“Lpus2”),varInfo(“Lous3”));
modelContext.setResponseInfo(lpusiInfo);
modelContext = (ModelContext)omodel.evaluate(modelContext);
```

The API for extracting the calculated objective, constraints and respective gradient values from the modelContext are as follows.

```
VarInfoList objInfo=modelContext.getObjectiveInfo();
VarInfoList constInfo=modelContext.getConstraintsInfo();
TableList objGradInfo=modelContext.getObjectivesGradientValues();
TableList constGradInfo=modelContext.getConstraintsGradientValues();
```

As a second example, consider the reconfiguration of the model to a new fidelity of the responses and sensitivities. This is achieved by first creating a *ModelContext* and placing *Evaluation* objects in the *ModelContext*. The Evaluation objects select the given fidelity for a Var or set of Vars. As an example, the changing of the fidelity of the response to “Linear Potential Method” and sensitivity to “Lpus1AstrosExacteg1” for the “Lpus1” variable can be done with the following;

```
// Create the Evaluation
Evaluation eval=new Evaluations(“Lpus1”,“Linear Potential Method”,
                                “Lpus1AstrosExacteg1”);
VarInfoList lpusiInfo=varsInfo(varInfo(“Lpus1”));

// Create the ModelContext
ModelContext modelContext = new ModelContext("Induced Drag Opti Model Query");
modelContext.setSelectEvaluations(eval);
modelContext.setResponsesInfo(lpusiInfo);
modelContext = (ModelContext)omodel.evaluate(modelContext);
```

This query results in the Evaluation for for “Lpus1” to be changed to “Linear Potential Method” which will change the Evaluator for “Lpus1” to “LpusAstrosExacte” and the Gradient to “Lpus1AstrosExacteg1”. Finally, the query requests the values of “Lpus1” be computed with the new Evaluation.

VII.Concluding Remarks

As designers develop complex systems that have very little historical information associated with them, it is becoming evident that new programming languages for transdisciplinary computing are required. These languages should reflect the complexity of meta computing problems we are facing in service-oriented computing, for example, concurrent engineering processes of the collaborative design by hundreds of people working together and using thousands of programs written already in software languages (languages for computers) that are dislocated around the globe. The transdisciplinary design of an aircraft engine or even a whole air vehicle requires large-scale high performance meta computing systems handling dynamically executable codes represented by software languages.

Domain-specific languages (DSL) are for humans, intended to express specific complex problems and related solutions. Three programming languages for transdisciplinary computing are described in this paper: VOL, VML, and EOL. These languages are interpreted by the exertion shell of SOS.

As complexity of problems being solved increases continuously, we have to recognize the fact that in transdisciplinary computing the only constant is change. The concept of the evaluator in the VFE framework provides the uniform service-orientation for all computing and meta computing needs with various applications, tools, utilities, and exertions as services. The SORCER operating system supports the two-way convergence of three programming models for transdisciplinary computing. On one hand, EOP is uniformly converged with VOP and VOM to express a service-oriented computation process in terms of other (inter/intra) process expressions (the network is the computer). On the other hand, VOM and VOP are uniformly converged with EOP to express an advanced multidisciplinary com-

putation model with multi-fidelity compositions in terms of other (intra/inter) process expressions including service federations (the computer is the network).

The SORCER platform with three layers of converged programming: exertion-oriented (for service collaborations), var-oriented (for var-oriented multi-fidelity compositions), and var-oriented modeling (multidisciplinary var-oriented models) has been successfully deployed and tested in multiple concurrent engineering and large-scale distributed applications including Distributed High Fidelity Engineering Design Optimization.

A computational framework that supports dynamic fidelity for aeroelastic analysis and optimization is presented. This effort describes the development and capability of the VO programming along with a demonstration of the dynamic fidelity by performing aeroelastic analysis with six different fidelities of induced drag. Euler based calculations with a Trefftz-plane, linear panel methods with a Trefftz-plane analysis, standard one-point approximation with Euler and panel methods, and a modified one-point approximation with Euler and panel methods. The fidelity can selected dynamically by the analysis or optimization algorithm based on either accuracy or efficiency requirements.

II. References

[1]SORCERsoft. Available at: http://sorcersoft.org. Accessed on: April 24, 2010.

[2]SORCER Research Topics. Available at: http://sorcersoft.org/theses/. Accessed on: April 24, 2010

[3]Sobolewski, M.: Exertion Oriented Programming. IADIS 3(1), 86-109 (2008) ISBN: ISSN: 1646-3692

[4] Sobolewski, M.: Federated Collaborations with Exertions. In: 17th IEEE International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), pp. 127-132 (2008)

[5]Sobolewski, M.: Metacomputing with Federated Method Invocation. In: Akbar Hussain, M. (ed.) Advances in Computer Science and IT, pp. s337-s363 (2009) In-Tech, intechweb.org, ISBN 978-953-7619-51-0, http://sciyo.com/articles/show/title/metacomputing-with federated-method-invocation (accessed on: August 10, 2010)

[6]Sobolewski, M.: Object-Oriented Metacomputing with Exertions. In: Gunasekaran, A., Sandhu, M. (eds.) Handbook On Business Information Systems. World Scientific, Singapore (2010) ISBN: 978-981-283-605-2

[7]http://www.phoenix-int.com/software/phx_modelcenter.phpe

[8]http://www.simulia.com/products/isight2.html

[9]http://www.simulia.com/products/see2.html

[10]http://www.vrand.com/visualdoc.html

[11]http://openmdao.org/

[12]http://www.jini.org/wiki/Main_Page

[13]NATO RTO Report TR-AVT-093 “Integration of Tools and Processes for Affordable Vehicles”. Published 2006

[14]http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing

[15]Sobolewski M. Federated Method Invocation with Exertions, Proceedings of the IMCSIT Conference, PTI Press, ISSN 1896-7094, pp. 765-778

[16]Jini Architecture Specification. Avai-lable at: http://www.jini.org/wiki/Jini_Architecture_Specification. Accessed on: April 24, 2010

[17]Edwards, W.K. Core Jini, 2nd ed., Prentice Hall

[18]Sobolewski, M.. Exertion Oriented Programming, IADIS, vol. 3 no. 1, pp. 86-109, ISBN: ISSN: 1646-3692

[19]Sobolewski M., Kolonay R.. Federated Grid Computing with Interactive Service-oriented Programming, International Journal of Concurrent Engineering: Research & Applications, Vol. 14, No 1, pp. 55-66

[20]Sobolewski, M. SORCER: Computing and Metacomputing Intergrid, 10th International Conference on Enterprise Information Systems, Barcelona, Spain (2008). Available at: http://sorcer.cs.ttu.edu/publications/papers/2008/C3_344_Sobolewski.pdf.

[21]Sobolewski, M. Federated Collaborations with Exertions, 17h IEEE International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), pp.127-132.

[22]Sobolewski, M. "Metacomputing with Federated Method Invocation", Advances in Computer Science and IT, edited by M. Akbar Hussain, In-Tech, in-techweb.org, ISBN 978-953-7619-51-0, s. 337-363. Available at: http://sciyo.com/articles/show/title/metacomputing-with-federated-method-invocation

[23]Sobolewski, M., (2010b, keynote). Exerted Enterprise Computing: From Protocol-Oriented Networking to Exertion-Oriented Networking, R. Meersman et al. (Eds.): OTM 2010 Workshops, LNCS 6428, 2010, Springer-Verlag Berlin Heidelberg 2010, pp. 182- 201.

[24]Sobolewski, M. (2010a) "Object-Oriented Metacomputing with Exertions," Handbook On Business Information Systems, A. Gunasekaran, M. Sandhu (Eds.), World Scientific, ISBN: 978-981-283-605-2

[25]Sobolewski, M. (2008c). Federated Collaborations with Exertions, 17h IEEE International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), pp.127-132

[26]Sobolewski M. (2007). Federated Method Invocation with Exertions, Proceedings of the IMCSIT Conference, PTI Press, ISSN 1896-7094, pp. 765-778

[27]Sobolewski, M. (2008a). Exertion Oriented Programming, IADIS, vol. 3 no. 1, pp. 86-109, ISBN: ISSN: 1646-3692

[28]Sobolewski, M., (2011, keynote), Provisioning Object-oriented Service Clouds for Exertion-oriented Programming. The 1st International Conference on Cloud Computing and Services Science, CLOSER 2011, Noordwijkerhout, the Netherlands, 7-9 May 2011, SSRI, Springer-Verlag.

[29]Mellor, S.J. And Balcer, M.J. (2002). A Foundation for Model-driven Architecture. Boston : Addison-Wesley, 2002.

[30]Munk, Max M., "The Minimum Induced Drag of Aerofoils," NACA Report No.121, 1921, pp 373-390.

[31]Ashley, H. and Landahl, M., *Aerodynamics of Wings and Bodies*, Dover Publications, Inc., New York, 1985, pp135-237.

[32]R.M. Kolonay, R. Roberts, L. Lambe, "A Comparison of Four Approximation Techniques for an Euler Based Induced Drag Function," 12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Victoria, British Columbia, Canada, 10-12 September 2008, AIAA paper 2008-5801.

[33]Kolonay, R.M., Eastep, F.E., Sanders, B. "Optimal Scheduling of Control Surfaces on a Flexible Wing to Reduce Induced Drag,", *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, AIAA Paper 2004-4362, Albany, New York, Sept., 2004.