The Convergence of Three Languages for Transdisciplinary Computing

Michael Sobolewski[1,2], Raymond Kolonay[1]

[1] Air Force Research Laboratory
Wright-Patterson Air Force Base, Ohio 45433

[2] United Technology Corporation
Dayton, Ohio 45433
sobol@sorcersoft.org

Keywords: concurrent engineering, metacomputing, SOA, service-oriented programming

Abstract: Each computing system requires a platform that allows software to run. The computing platform includes a programming environment to create applications with a coherent operating system and processor. Each platform's programming environment reflects a relevant abstraction, and usually the type and quality of the abstraction implies the complexity of problems we are able to solve. The Service ORiented Computing EnviRonment (SORCER) targets service abstractions for transdisciplinary complexity with support for high performance computing. SORCER service-commands are expressed in Exertion-Oriented Language (EOL) in convergence with two other languages: Variable-Oriented Language (VOL) and Variable-oriented Modeling Language (VML). 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 an explicit network-centric service-oriented computation process in terms of other implicit (inter/intra) process expressions. On the other hand, VOM and VOP are uniformly converged with EOP to express an explicit declarative service model with multifidelity and multidisciplinary features in terms of other implicit (intra/inter) process expressions including network-centric service federations.

1. Introduction

In transdisciplinary computing systems each service provider in the collaborative federation performs its services in an orchestrated workflow. Once the collaboration is complete, the federation dissolves and the providers disperse and seek other federations to join. The approach is network centric in which a service is defined as an independent self-sustaining entity—*remote service provider*—performing a specific network activity. These service providers have to be managed by a relevant *operating* *system* with *commands* for expressing interactions of providers in the network.

The reality at present, however, is that transdisciplinary computing environments are still very difficult for most users to access, and that detailed and low-level programming must be carried out by the user through command line and script execution to carefully tailor jobs on each end to the resources on which they will run, or for the data structure that they will access. This produces frustration on the part of the user, delays in the adoption of service-oriented techniques, and a multiplicity of specialized “cluster/grid/cloud-aware” tools that are not, in fact, aware of each other which defeats the basic purpose of the cluster/grid/cloud.

A *computer* as a programmable device that performs symbolic processing, especially one that can process, store and retrieve large amounts of data very quickly, requires a computing platform (runtime) to operate. *Computing platforms* that allow software to run on the computer require a *processor*, *operating system*, and *programming environment* with related runtime libraries and user agents. Therefore, the metacomputer requires a *platform* that describes a kind of networking framework to allow software to run utilizing virtual distributed resources. Different platforms of metacomputers can be distinguished along with corresponding types of virtual network processors.

We consider a metaprogram as the process expression of hierarchically organized collaboration of remote component programs. Its service-oriented operating system makes decisions about where, when, and how to run these components. The specification of the service collaboration is a *metaprogram* a program that manipulates other programs remotely as its data. Nowadays the similar computing abstraction is usually applied to the program executing on a *single computer* as to the program executing in the *network of computers*, even though the executing environments (platforms) are structurally completely different. Most, so called, service-oriented programs are still written using software languages such as FORTRAN, C, C++ (compiled into native processor code), Java, Smalltalk (compiled into intermediate code), and interpreted languages such as Perl and Python, the way it usually works on a single host. The current trend is to have these programs and scripts define remote computational modules as *service providers*.

Instead of moving executable files around the computer networks we can autonomically provision the corresponding computational components (executable codes) as uniform metainstructions of the service metaprocessor. Now we can submit a metaprogram (service command) in terms of metainstructions (services) to the *metacompute OS* that manages dynamic federations of service providers and related resources, and enables the collaboration of the required service providers according to the metaprogram definition with its own control strategy. We treat *services* as *service types* (e.g. a form of the Java interface) and service providers as service instances implementing that service types. A provider can implement multiple service types, so can provide multiple services.

One of the first metacompute platforms was developed under the sponsorship of the National Institute for Standards and Technology (NIST)—the Federated Intelligent Product Environment (FIPER) (Sobolewski, 2002). The goal of FIPER is to form a federation of distributed service objects that provide engineering data, applications, and tools on a network. A highly flexible software architecture had been developed for transdisciplinary computing computing (1999-2003), in which engineering tools like computer-aided design (CAD), computer-aided engineering (CAE), product data management (PDM), optimization, cost modeling, etc., act as both service providers and service requestors.

The Service-ORiented Computing EnviRonment (SORCER) (Sobolewski, 2008-2011) builds on the top of FIPER to introduce a metacomputing operating system with all system services necessary, including service management (rendezvous services), a federated file system, and autonomic resource management, to support service-object oriented metaprogramming. It provides an integrated solution for complex transdisciplinary applications (see Fig. 1) that require multiple complex solutions across multiple disciplines combined at runtime into a transdisciplinary one. The SORCER metacomputing environment adds an entirely new layer of abstraction to the practice of metacomputing—*exertion-oriented* (EO) programming with a *federated method invocation* (FMI). The EO programming makes a positive difference in service-oriented programming primarily through a new metaprogramming abstraction as experienced in many service-oriented computing projects including systems deployed at GE Global Research Center, GE Aviation, Air Force Research Lab, SORCER Lab, and SORCER partners in China and Russia.

The reminder of this paper is organized as follows Section 2 describes briefly the SORCER metacomputing platform; Section 3 describes exertion-oriented programming; Section 4 describes var-oriented programming and var-oriented modeling; finally Section 5 concludes with final remarks and comments.

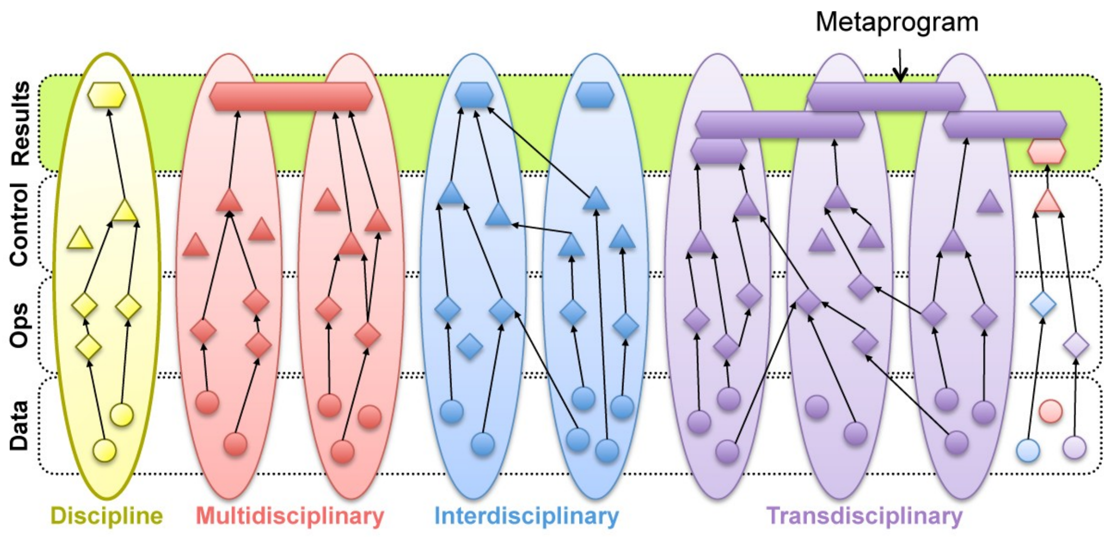

Figure 1. By providing easy-to-use, self-discovering services representing domain knowledge (data), tools (operations), and related technologies (control) with metaprogramming methodology, the SORCER environment reduces integration and deployment costs, facilitate productivity, increases research collaboration, and advances the development and acceptance of secure and fault tolerant transdisciplinary concurrent engineering solutions.

2. Service-object Oriented Platform: SORCER

The Service-ORiented Computing EnviRonment (SORCER) is a federated service-to-service (S2S) metacomputing 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) (Sobolewski, 2007). It incorporates Jini semantics of services (“Jini Architecture”, n.d.) in the network and the Jini programming model (Edwards, 2000) 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. 2). The SORCER programming environment creates the unifying representation for three concrete programming syntaxes: the SORCER Java API described in (Sobolewski, 2008a), and the graphical form described in (Sobolewski and Kolonay, 2006), and the functional composition form presented in this paper. The notation of functional composition has been developed so far for three related languages: Exertion-Oriented Language (EOL), Var-Oriented Language (VOL), and Var-oriented Modeling Language (VML) that are usually complemented with the Java/Groovy object-oriented syntax. In the following two sections we will describe the basic syntax of these three languages. More details on EOL can be found in (Sobolewski, 2010b).

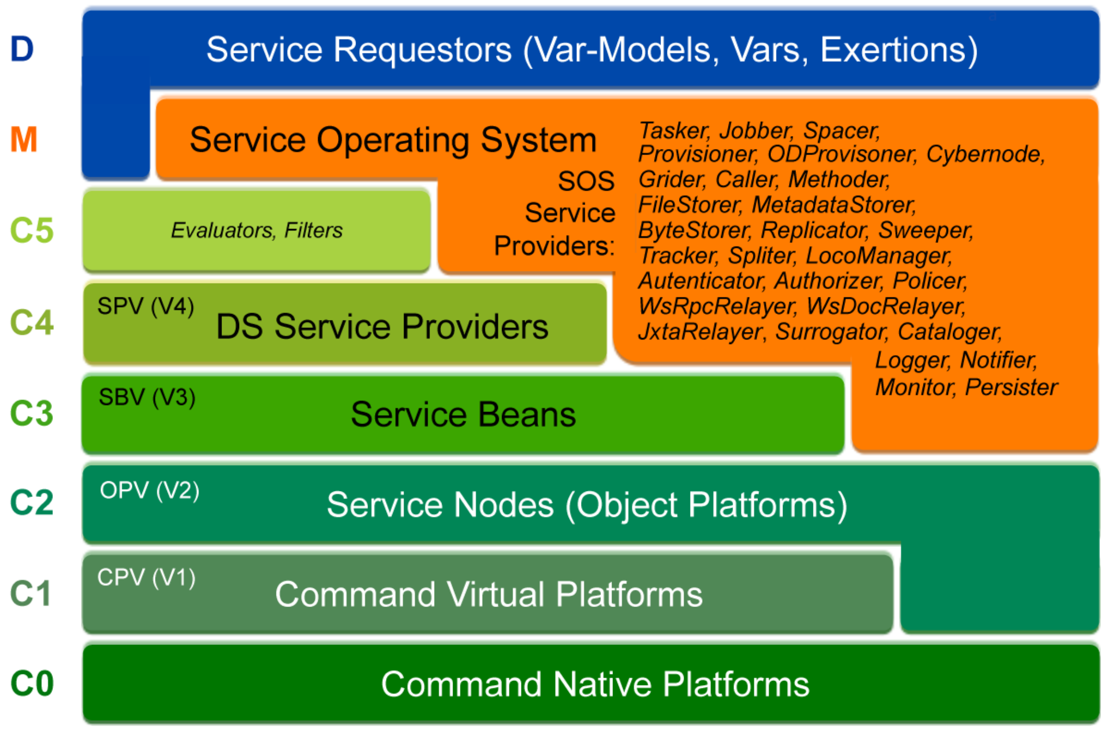
D
Service Requestors (Var-Models, Vars, Exertions)
M
Service Operating System
SOS Service Providers:
Tasker, Jobber, Spacer, Provisioner, ODProvisoner, Cybernode, Grider, Caller, Methoder, FileStorer, MetadataStorer, ByteStorer, Replicator, Sweeper, Tracker, Spliter, LocoManager, Autenticator, Authorizer, Policer, WsRpcRelayer, WsDocRelayer, JxtaRelayer, Surrogator, Cataloger, Logger, Notifier, Monitor, Persister
C5
Evaluators, Filters
C4
SPV (V4)
DS Service Providers
C3
SBV (V3)
Service Beans
C2
OPV (V2)
Service Nodes (Object Platforms)
C1
CPV (V1)
Command Virtual Platforms
C0
Command Native Platforms

Figure 2. The SORCER layered architecture, where C0-C5 (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 V1-V4 stands virtualization of the corresponding layer.

3. 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 connectionist process expression that models 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 to be *exerted* for a provided service in the dynamically created federation.

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. Please 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 (Sobolewski, 2008c). The *collaboration* specifies a collection of cooperating providers—the *exertion federation*—identified by the exertion's signatures. Exertions encapsulate explicitly *data*, *operations*, and *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* (Sobolewski, 2007) 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 (Sobolewski, 2008a). A job exertion is a kind of command script, that is similar conceptually to 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* by the receiving provider. 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 called *data context* describes the data that tasks and jobs work on. A data context, or simply a *context*, is a data structure that describes service provider ontology along with related data (Sobolewski, 2008a). Conceptually a data context is similar in structure to a files system, where paths refer to objects instead to files. A provider‘s ontology (object paths) is controlled by the provider vocabulary that describes 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 context data flows between invocations of signature operations that 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 collaborations as interactions on its virtual service processor—the dynamically formed service federations.

The primary difference between *exertion* and *federation* is *management* and *implementation*. The exertion and the federation distinctions are based on the analogies between the *company management* and *employees*: the top-level exertion refers to the *central control* (the Chairman of company) of the behavior of a *management system* (the Chairman's staff vs. component exertions), while federation refers to an *implementation system* (the company employees vs. the service providers) which operates according to management rules (FMI), but without centralized control.

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 (Sobolewski, 2002 and 2008a). Graphical interactive exertion-oriented programming is presented in (Sobolewski and Kolonay, 2006). Details regarding textual EOP and two examples of simple EO programs can be found in (Sobolewski, 2010b) and (Sobolewski, 2011).

4. 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 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 multifidelity 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. 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 response, parametric, and optimization component models. The programming style of VOM is declarative; models describe the desired results of the program, without explicitly listing command 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.

The SORCER metamodeling architecture (Sobolewski, 2011) is the unifying representation for three concrete programming syntaxes: the SORCER Java API described in (Sobolewski, 2008a), the functional composition form, (Sobolewski, 2010b and 2011), and the graphical form described in (Sobolewski and Kolonay, 2006). The functional composition notation has been used for *Var-Oriented Language* (VOL) and *Var-oriented Modeling Language* (VML) that are usually 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 mathematical function, subroutine, coroutine, object, or any local or distributed 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 or filter is changed, evaluator's arguments change, or the value is undefined;
    b) an *evaluator* is a service with the argument vars that define the variable dependency composition; and
    c) a *filter*: is a *getter* operation.

Var-oriented programming is the programming paradigm that treats any computation as the VFE triplet: *value*, *filter* (*pipeline of filters*), and *evaluator* (see Fig. 3). 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 (multifidelity). The VFE paradigm emphasizes the usage of multiple pairs of *evaluator-filter* (called var evaluations) to define the value of var. The semantics of

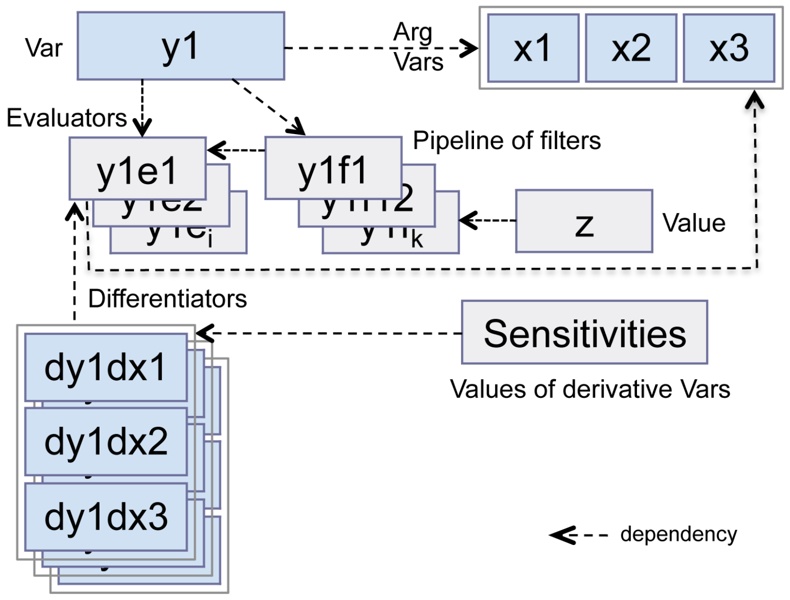

Figure 3. The var structure: value/evaluator/filter. Vars are indicated in blue color. The basic var y1, z=y1(x1, x2, x3), depends on its argument vars and derivative vars.

the *value*, whether the var represents a mathematical function, subroutine, coroutine, or just data, depends on the evaluator and filter currently used by the var.

A *service* in VOL is the work performed by a variable's evaluator-filter pair. Evaluators for dependent vars, that depend on their argument vars, define:

1. a var composition via the var arguments of its evaluator
2. multiple processing services (mutifidelity)
3. multiple differentiation services (mutifidelity)
4. evaluators can execute commands (executable codes), object-oriented services (method invocations), and exertions (exerting service federations).

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. Please note that vars used in data contexts of exertions extend EOP for the flexible service semantics defined by VOP.

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 aircraft engine, can be imprecise, not executable, at high level with informal semantics. However its detailed model (detailed design) has to be precise, executable, 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 (Mellor and Balcer, 2002) are precise and executable.

Data contexts (objects implementing `Context` interface) with specialized aggregations of vars are called *var-models.* Three types of analysis models: *response*, *parametric*, and *optimization* have been studied already (Sobolewski, 2010b). These models are expressed in VML using functional composition and/or 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 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 *multifidelity* traits of transdisciplinary computing. Var compositions across multiple models define multidisciplinary problems; multiple evaluators per var and multiple differentiators per evaluator define their multifidelity. They 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 computations in the right evolving or new direction.

Let's consider the Rosen-Suzuki optimization problem to illustrate the basic VML, VOL, and EOL concepts, where:

1. design vars: *x1*, *x2*, *x3*, *x4*
2. response vars: *f*, *g1*, *g2*, *g3*,
   and
3. *f = x1^2-5.0*x1+x2^2-5.0*x2+2.0*x3^2-21.0*x3+x4^2+7.0*x4+50.0*
4. *g1 = x1^2+x1+x2^2-x2+x3^2+x3+x4^2-x4-8.0*
5. *g2 = x1^2-x1+2.0*x2^2+x3^2+2.0*x4^2-x4-10.0*
6. *g3 = 2.0*x1^2+2.0*x1+x2^2-x2+x3^2-x4-5.0*

The goal is then to minimize *f* subject to
g1 <= 0, *g2 <= 0*, and *g3 <= 0.*

In VML this case is expressed by the following mogram:

```
int designVarCount = 4;
int responseVarCount = 4;
OptimizationModel model = optimizationModel(
    "Rosen-Suzuki Model",
    designVars(vars(loop(designVarCount),
          "x", 20.0, -100.0, 100.0)),
    responseVars("f"),
    responseVars(loop(responseVarCount-1), "g"),
    objectiveVars(var("fo", "f", Target.min)),
    constraintVars(
          var("g1c", "g1", Relation.lte, 0.0),
          var("g2c", "g2", Relation.lte, 0.0),
          var("g3c","g3", Relation.lte, 0.0)));
configureAnalysisModel(model);
```

Response vars `f, g1, g2, g3` are configured by the function `configureAnalysisModel` defined in VOL, for example var `f` is configured as follows:

```
var(model, "f",
    evaluator("fe1",
          "x1^2- 5.0*x1+x2^2-5.0*x2+2.0
               *x3^2-21.0*x3+x4^2+7.0*x4+50.0"),
    args("x1", "x2", "x3", "x4"));
```

The `model` above can be provisioned directly in SORCER as a servicer provider and used by the space exploration provider of the `Exploration` type that also uses the CONMIN optimization (CONMIN, n.d) service provider of the `Optimization` type. The requestor creates the exertion `opti` as follows:

```
// Create an optimization data context

Context exploreContext = exploreContext(
    "Rosen-Suzuki context",
    varsInfo(
          varInfo("x1", 1.0),
          varInfo("x2", 1.0),
          varInfo("x3", 1.0),
          varInfo("x4", 1.0)),
    strategy(new ConminStrategy(
          new File(System.getProperty(
                "conmin.strategy.file")))),
    dispatcher(
          sig(null, RosenSuzukiDispatcher.class,
                Process.INTRA)),
    model(sig("register",
          OptimizationModeling.class,
                "Rosen-Suzuki Model")),
    optimizer(sig("register",
          Optimization.class,
                "Rosen-Suzuki Optimizer")));


// Create a task exertion

Task opti= task("opti",
    sig("explore", Exploration.class,
          "Rosen-Suzuki Explorer"),
    exploreContext);
```

then executes the `opti` exertion:

```
// Execute the exertion and log results

logger.info(">>>>>>>>>>>>> results: " + context(exert(opti));
```

with the exertion's output data context logged as follows:

```
[java] Objective Function fo = 6.002607805900986
[java] Design Variable Values
[java] x1 = 2.5802964087086235E-4
    x2 = 0.9995594642481355
    x3 = 2.000313835134211
    x4 = -0.9986692050113675
[java] Constraint Values
[java] g1c = -0.002603585246998996
    g2c =-1.0074147118087602
    g3c = 4.948009193483927E-7
[java] ITERATIONS
[java] Number of Objective Evaluations = 88
[java] Number of Constraint Evaluations = 88
```

```
[java] Number of Objective
    Gradient Evaluations = 29
[java] Number of Constraint Gradient
    Evaluations = 29
```

The `exploreContext` defines initialization of design vars (`varsInfo`), the optimization strategy, and the exploration dispatcher with two required services—two signatures for: optimizer and model. The context then is used to define the exertion task `opti` with the signature for exploration service named `Rosen-Suzuki Explorer` of the `Exploration` type. For simplicity, signatures above do not specify QoS for the specified providers. To illustrate the provider QoS concept (Rubach and Sobolewski, 2009), for example, the optimizer's signature can be expressed as follows:

```
sig("register", Optimization.class, qosCtx))
```

where the `qosCtx` context may be defined as follows:

```
QosContext qosCtx = qos(
    serviceProvider(entry(
          "Name","Rosen-Suzuki Optimizer"),
          libs(entry("Name","Conmin"),
                entry("Class", NativeLibrarySupport.class),
                      entry("FileName", "conmin.so"))),
    objectPlatform(entry("Name","Java"),
          entry("Class", J2SESupport.class),
          entry("Version", "1.5.*")),
    commandPlatform(
          processor(entry("Available", "2"),
                entry("Architecture", "x86")),

          memory(entry("Capacity", "4G"),
                entry("Available", "2G")),
          disk(entry("Capacity", "20G"),

                entry("Available", "4G"))),
    sla(entry("cost", 200),
          entry("time", 5000),
          entry("CPU", range(0.0, 0.9)),
          entry("Memory", range(0.2, 0.5)),
          entry("ProcAvail_CPU_Util", range(1.5, 2.0),
          metric("ProcAvail_CPU_Util",
                impl("result = Double.parseDouble(proc_avail)
                      * cpu_util",
                      args(var("proc_avail", processor(
                                  entry("Available", ""))),
                            var("cpu_util", sla(
                                  entry("CPU", null)))))),
    authorization(entry("estimatedDuration", 300001),
          entry("priority", range(5,10)),
          execDate("2011-01-10 00:00:00",
                "2011-01-11 12:00:00"),
          project(entry("name","RS"),
                entry("manager","Smith"),
                entry("description", "RS optimization")),
          organization(entry("name", "TTU"),
                entry("department", "CS"),
                entry("description","SORCER Testbed"))));
```

The `qosCtx` context specifies for the optimizer the required QoS: for the provider by operator `serviceProvider`, for the object platform by operator `objectPlatform`, for the command platform by operator `commandPlatform`, for the expected SLA by operator `sla` and `authorization` by authorization for the service requestor.

5. Conclusions

As we move from the problems of the *information era* to more complex problems of the *molecular era*, it is becoming evident that new programming languages for transdisciplinary computing are required. These languages should reflect the complexity of metacomputing 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 metacomputing systems handling anywhere-anytime 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 SOS shell. The essential differences between the UNIX operating system, and the SOS are illustrated in Table 1.

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-filter pair in the VFE framework provides the uniform service-orientation for all computing and metacomputing 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 an explicit network-centric service-oriented computation process in terms of other implicit (inter/intra) process expressions (the network is the computer). On the other hand, VOM and VOP are uniformly converged with EOP to express an explicit declarative transdisciplinary process in terms of other implicit (intra/inter) process expressions including exertions (the computer is the network).

The SORCER platform with three layers of converged programming: *exertion-oriented* (for service collaborations), *var-oriented* (for var-oriented multifidelity 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 large scale, distributed, dynamic fidelity aeroelastic analysis and optimization (Kolonay & Sobolewski, 2011).

Features	**UNIX**	**SOS**
Data	File/File system	Object/Data context
Data flow	Pipes	Data context pipes
Interpreter	e.g. C Shell	SOS shell
Programming language	Shell scripting - procedural	VOL, VML, EOL – service-oriented (VOL) – service modeling (VML) – service-object oriented (EOL) – interactive visual programming –Java API
System (SW) language	C	Java/Jini/Rio

Table 1. UNIX OS vs. SORCER OS. Pipes in UNIX are between processes, in SORCER they are between data contexts; instead of UNIX pipeline SORCER defines a workflow by composition of exertions; the UNIX shell is local but the SOS shell is a network shell.

Acknowledgments

This work was partially supported by Air Force Research Lab, Air Vehicles Directorate, Multidisciplinary Science and Technology Center, the contract number F33615-03-D-3307, Algorithms for Federated High Fidelity Engineering Design Optimization.

References

Edwards, W.K. (2000) *Core Jini*, 2nd ed., Prentice Hall

FIPER: Federated Intelligent Product EnviRonmet. Available at: http://sorcersoft.org/fiper/fiper.html. Accessed on: April 24, 2010.

Foster I.; Kesselman C. & Tuecke S. (2001). The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International J. Supercomputer Applications*, 15(3)

Jini Architecture Specification. Available at: http://www.jini.org/wiki/Jini_Architecture_Specification. Accessed on: April 24, 2010

Kolonay, R.M. & Sobolewski, M. (2011). Service ORiented Computing EnviRonment (SORCER) for Large Scale, Distributed, Dynamic Fidelity Aeroelastic Analysis & Optimization. International Forum on Aeroelasticity and Structural Dynamics 2011 (IFASD2011), 26-30 June, Paris, France.

Linthicum, D.S. (2009). Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide, Addison-Wesley Professional, ISBN-10 0136009220

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

Metacomputing: Past to Present, Retrieved April 24, 2010, from: http://archive.ncsa.uiuc.edu/Cyberia/MetaComp/MetaHistory.html

Rubach, P. & Sobolewski, M. (2009). SERVME: SLA-Based QoS Framework for Federated Computing Environments, to be published in the *Proceedings of the Enterprise Computing Conference - EDOC*, IEEE Computer Society

Sobolewski M. (2002). Federated P2P services in CE Environments, Advances in Concurrent Engineering, A.A. Balkema Publishers, 2002, pp. 13-22

Sobolewski M., Kolonay R. (2006). Federated Grid Computing with Interactive Service-oriented Programming, International Journal of Concurrent Engineering: Research & Applications, Vol. 14, No 1, pp. 55-66

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

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

Sobolewski, M. (2008b) 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.

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

Sobolewski, M. (2009). "Metacomputing with Federated Method Invocation", Advances in Computer Science and IT, edited by M. Akbar Hussain, In-Tech, intechweb.org, ISBN 978-953-7619-51-0, s. 337-363. Available at: http://sciyo.com/articles/show/title/metacomputing-with-federated-method-invocation

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

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.

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.

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

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