Int. J. Communications, Network and System Sciences, 2012, 5, 579-592
http://dx.doi.org/10.4236/ijcns.2012.529068 Published Online September 2012 (http://www.SciRP.org/journal/ijcns)

Unified Mogramming with Var-Oriented Modeling and Exertion-Oriented Programming Languages

Michael Sobolewski[1,2], Raymond Kolonay[1]
[1]Air Force Research Laboratory, WPAFB, USA
[2]Polish Japanese Institute of IT, Warsaw, Poland
Email: sobol@sorcersoft.org

Received June 26, 2012; revised July 21, 2012; accepted August 2, 2012

ABSTRACT

The Service ORiented Computing EnviRonment (SORCER) targets service abstractions for transdisciplinary complexity with support for heterogeneous service-oriented (SO) computing. SORCER's models are expressed in a top-down Var-oriented Modeling Language (VML) unified with programs in a bottoms-up Exertion-Oriented Language (EOL). In this paper the introduction to mogramming (modeling and programing), which uses both languages, is described. First, the emphasis is on modeling with service variables that allow for computational fidelity within VML. Then, seven types of service providers, both local and distributed, are described to form collaborative federations described in EOL. Finally, the unified hybrid of SO modeling and SO programming is presented. Fourteen simple mogramming examples illustrate the syntax and usage of both VML and EOL.

Keywords: Metacomputing; Service-Oriented Mogramming; Var-Oriented-Modeling; Exertion-Oriented Programming; SOA; SORCER

1. Introduction

A transdisciplinary computational model requires extensive computational resources to study the behavior of a complex system by computer simulation. The large system under study that consists of thousands or millions of variables is often a complex nonlinear system for which simple, intuitive analytical solutions are not readily available. Usually experimentation with the model is done by adjusting the parameters of the system in the computer. The experimentation, for example aerospace models with multi-fidelity, involves the best of the breed applications, tools, and utilities considered as heterogeneous services of the model. The modeling services are used in local/distributed concurrent federations to calculate and/or optimize the model across multiple disciplines fusing their domain-specific services running on laptops, workstations, clusters, and supercomputers.

Services are autonomous (acting independently), local or distributed units of functionality. Elementary services have no calls to each other embedded in them. Compound services are compositions of elementary and other compound services. Each service implements multiple actions of a cohesive (well integrated) service type, usually defined by an interface type. A service provider can implement multiple service types, and thus can provide multiple services. Its service type and operation complemented by its QoS parameters (service signature) are used to specify functionality of a provider. Instances of a service provider are equivalent units of functionality identified by the same signature.

In transdisciplinary computing systems each local or distributed 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 service centric in which a service provider is defined as an independent self-sustaining entity performing a specific local or network activity. These service providers have to be managed by a relevant *service-centric operating system* with *commands* for executing interactions of providers in dynamic virtual federations [1].

The reality at present, however, is that metacomputing environments [2] 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 static interactions on each end to the distributed 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 (SO) techniques, and a multiplicity of specialized "server/cluster/grid/cloud-aware" tools [3-5] that are not, in fact, aware of each other which defeats the basic purpose of the metacomputing.

Both *computing* and *metacomputing platforms* that allow software to run on the computer require a *processor*, *operating system*, and *programming environment* with related runtime libraries and user agents. We consider a SO model or program (mogram) as the process expression of hierarchically organized services executed by an aggregation of *service providers*—the virtual SO processor. Its SO Operating System (SOOS) makes decisions about where, when, and how to run these service providers. The specification of the service collaboration is a SO mogram that manipulates other executable codes (applications, tools, and utilities) locally or remotely as its data. Three types of mograms are considered in the paper: var-models, exertions and hybrid mograms that use both of them.

Instead of moving executable files around the computer network we can autonomically provision [6,7] the corresponding computational components (executable codes) as uniform services (metainstructions) of the virtual SO processor. Now we can invoke a SO mogram as a command of the SOOS that exerts its dynamic federations of service providers and related resources, and enables the collaboration of the required service providers according to the SO moogram definition with its own data and control strategy.

One of the first SO platforms developed under the sponsorship of the National Institute for Standards and Technology (NIST) was the Federated Intelligent Product Environment (FIPER) [8]. The goal of FIPER was 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 (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 SORCER environment [5,6,9-11] builds on the top of FIPER to introduce a SOOS with all system services necessary, including service management (rendezvous services), a federated file system, and autonomic resource management, to support service-object oriented programming. It provides a SOOS for complex network-centric applications that require multiple solutions across multiple disciplines combined at runtime into a transdisciplinary collaboration of service providers in the global network. The SORCER environment adds two entirely new layers of abstraction to the practice of SO computing—SO models expressed in a Var-oriented Modeling Language (VML) in concert with SO programs expressed in an Exertion-Oriented Language (EOL) verified and validated in projects at the General Electric Global Research Center, GE Aviation, Air Force Research Lab, SORCER Lab at TTU [12].

The remainder of this paper is organized as follows Section 2 describes briefly var-oriented modeling; Section 3 describes exertion-oriented programming; Section 4 describes var-oriented programming and var-oriented modeling for design optimization; Section 5 introduces the SORCER SOOS; finally Section 6 concludes with final remarks and comments. The basic concepts of service-oriented mogramming [13] (modeling and programming) are illustrated with simple and easy to follow examples.

2. Var-Oriented Modeling

A computation is a relation between a set of inputs and a set of potential outputs. There are many ways to describe or represent a computation and a composition of them. Two types of computations are considered in this paper: *var-oriented* and *exertion-oriented*. A *service* is the work performed in which a service provider *exerts* acquired abilities to execute a computation. A service variable, called a *var* and an *exertion* are expressions of a service in the *Var-Oriented Language* (VOL) and the *Var-Oriented Modeling Language* (VML), respectively.

The first one is drawn primarily from the semantics of a variable the second one from the semantics of a routine. Either one can be mixed with another depending on the direction of the problem being solved: top down or bottom up. The top down approach usually starts with var-oriented modeling in the beginning focused on relationships of vars in the model with no need to associate them to services. Later the var-model may incorporate relevant services (evaluators) including exertions. In var-oriented modeling three types of models can be defined (response, parametric, and optimization) and in exertion-oriented programming seven different types of elementary exertions (tasks) and two types of compositional exertions (jobs) are defined.

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 a var allows one to use functions, subroutines, or coroutines in the same way. A value of a var can be associated with a mathematical function, subroutine, coroutine, object, or any local or distributed data. The functional composition notation has been used for the *Var-Oriented Language* (VOL) and the *Var-Oriented Modeling Language* (VML) that are usually complemented with the Java object-oriented syntax. The concept of vars and exertions as expression of services combines the three languages VOL, VML, and EOL into a uniform SO programming model.

2.1. Var-Orientd Programing (VOP)

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.

A *service variable* (*var*) is a structure defined by the triplet <*value*, {*evaluator*}, {*filter*}>. VOP is a programming paradigm that uses *service variables* to design var-oriented multifidelity compositions. An evaluator-filter pair is called a var fidelity. It is based on dataflow principles where changing the value of any argument var should automatically force recalculation of the var's value. VOP promotes values defined by an evaluator-filter pairs in the var and its dependency chain of argument vars to become the main concept behind any processing.

The semantics of a variable depends on the process expression formalism [14]:

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 an object's attributes accessible via operations called getters.

4) A service variable is a triplet: <*value*, {*evaluator*}, {*filter*}>, where:

a) a *filter* is a getter operation;

an *evaluator* is a service with the argument vars that define the var dependency chain; and

b) a *value* is a quantity filtered out from the output of the current evaluator; the value is invalid when the current evaluator or its filter is changed, current evaluator's arguments change, or the value is undefined.

VOP is the programming paradigm that treats any computation as the VFE triplet: *value*, *filter* (*pipeline of filters*), and *evaluator* as illustrated in **Figure 1**. Evaluators and filters can be executed locally or remotely. An evaluator may use a differentiator to calculate the rates at which the var quantities change with respect to the argument vars. Multiple associations of an evaluator-filter pair can be used with the same var allowing var's fidelity. The VFE paradigm emphasizes the usage of multiple pairs of *evaluator-filter* (called var fidelities) to define the value of var. The semantics of the *value*, whether the var represents a mathematical function, subroutine, coroutine, or data, depends on the evaluator and filter currently used by the var.

A *service* in VOP is the work performed by a var's evaluator-filter pair. Evaluators and filters of the var define:

1) the var arguments and their dependency chain (composition);

2) multiple processing services (output multifidelity);

3) multiple differentiation services (differentiation multifidelity);

4) evaluators can execute any type of local or distributed processing (connectivity and net heterogeneity); and

5) filters provide postprocessing (interoperability).

Thus, in the same process various forms of services (local and distributed) can be mixed within the same uniform process expression. Also, the fidelity of vars can be changed at runtime as it depends on the currently selected evaluator-filter pair.

The variable evaluation strategy is defined as follows: the var value is returned if it 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 var's value valid. The evaluator's

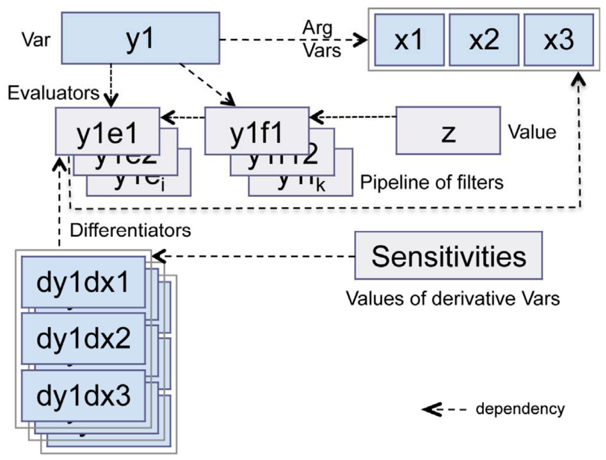

Figure 1. The var structure: <value, {evaluator}, {filter}>. Vars are indicated in blue color. The basic var y1, z = y1 (x1, x2, x3), depends on its argument vars and derivative vars in differentiators.

raw value may depend on other var arguments and those arguments in turn can depend on other var arguments and so on. This var dependency chaining provides the integration framework for all possible kinds of computations represented by various types of evaluators including exertions described in Section 3. To illustrate the basic VOL syntax a few simple examples will be given. First, using VOL the output var y is created with four argument vars x1, x2, x3, and x4 and then is evaluated.

Example 1. "Hello Arithmetic" y = (x1 ∗ x2) − (x3 + x4)

The argument vars:

```
Var x1 = var ("x1", 10.0); Var x2 = var("x2", 50.0),
Var x3 = var ("x3", 20.0), Var x4 = var ("x4", 80.0);
```

The output var y with an expression evaluator is defined as follows:

```
Var y = var("y",
    expr("(x1 * x2) - (x3 + x4)",
        args(x1, x2, x3, x4)));
```

Evaluate (value) and test (assertEquals) the var y:

```
assertEquals (value(y), 400.0);
```

A var with its referencing environment (substitution) for the free argument vars, evaluators, and filters is called a *var closure*. A variable is a *free* if its value is not defined.

The var y with free x1, x2, x3, and x4 can be defined in VOL as follows:

Example 2. Closing y over x1, x2, x3, and x4

```
Var y = var("y",
   expr("(x1 * x2) - (x3 + x4)",
     args("x1", "x2", "x3", "x4")));
```

Closing y over x1, x2, x3, and x4 can be done as follows:

```
Object val = value(y,
   entry("x1", 10.0), entry("x2", 50.0),
   entry("x3", 20.0), entry("x4", 80.0));
assertEquals(val, 400.0);
```

The example below illustrates two var closures of z over x1, x2 and one of its evaluators.

Example 3. Closing over var fidelities

```
Var z = var("z", evaluators(
   expr("e1", "x1 * x2", args("x1", "x2")),
   expr("e2", "x1 * x2 + 0.1", args("x1", "x2"))));
assertEquals(value(z,
   entry("x1", 10.0), entry("x2", 50.0), eFi("e1")), 500.0)
assertEquals(value(z,
   entry("x1",10.0), entry("x2",50.0), eFi("e2")), 500.1);
```

where the operator eFi stands for evaluator fidelity; eFi selects the evaluator by a given name.

2.2. Var-Orientd Modeling (VOM)

Var-Oriented Modeling is a modeling paradigm using vars in a specific way to define heterogeneous var-oriented models, in particular large-scale multidisciplinary models including response, parametric, and optimization models. The programming style of VOM is declarative; models describe the desired results of the output vars, without explicitly listing instructions or steps that need to be carried out to achieve the results. VOM focuses on how vars connect (compose) in the scope of the model, 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.

A *var-oriented model* or simply *var-model* is an aggregation of related vars. A var-model defines the lexical scope for var unique names in the model. Three types of models: *response*, *parametric*, and *optimization* have been studied to date [15,16]. These models are declared in VML using the functional composition syntax with VOL and possibly with EOL and the Java API to configure the vars.

The *input var* is typically the variable representing the value being manipulated or changed and the *output var* is the observed result of the input vars being manipulated. If there is a relation specifying output in terms of given inputs, then output is known as an "output var" and the var's inputs are "argument vars". Argument vars can be either output or input vars.

Returning to the "Hello Arithmetic" example used to illustrate SO concepts in this paper, we will define a function composition subtract(multiply(x1, x2), add(x3, x4) instead of subtract(x1, x2, x3, x4) defined as var y in Example 1. Thus, we are decomposing y(x1, x2, x3, x4) to f3(f4(x1, x2), f5(x3, x4)). In reality, the arithmetic functions corresponding to operators: multiply (f4), add (f5), and subtract (f3), can be replaced by any type of domain-specific services.

Example 4. "Hello Arithmetic" Model

```
VarModel vm = model("Hello Arithmetic",
  inputs(
    var("x1"), var("x2"),
    var("x3", 20.0), var("x4", 80.0)),
  outputs(
    var("f4", expression("x1 * x2",
      args( "x1", "x2"))),
    var("f5", expression("x3 + x4",
      args("x3", "x4"))),
    var("f3", expression("f4 - f5",
      args("f4", "f5")))));
```

Take into account that two output vars f4 and f5 are free arguments with respect to f3, and two input vars x1 and x2 are free arguments with respect to f4. Closing the var f3 in the model vm over x1, x2, and subsequently over f4 and f5 can be stated as follows:

```
assertEquals(value(var(put(vm,
    entry("x1", 10.0), entry("x2", 50.0)), "f3"), 400.0);
```
where the operator put makes a substitution in the model vm for given entries and returns the initialized model.

The modularity of the VFE framework, composition of argument vars, reuse of evaluators and filters in defining var-models is the key feature of 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).

3. Exertion-Oriented Programming

In language engineering—the art of creating languages —a *metamodel* [13] is a model to specify a language. EOL is a metamodel to model connectionist process expressions that model behavioral phenomena as the emergent processes of interconnected federations of service providers. The central exertion principle is that a computation can be expressed and actualized [14] by the interconnected federation of simple, often uniform, and efficient service providers that compete with one another to be *exerted* for their services in the dynamically created federation. Each service provider implements multiple actions of a cohesive (well integrated) service type, usually defined by an interface type. A service provider implementing multiple service type provides multiple services. Its service type complemented by its QoS parameters can identify functionality of a provider. In EOL an exertion can be used as a closure over free variables in the exertion's data and control contexts.

Exertion-oriented programming (EOP) is a SO programming paradigm using *service providers* and *service commands*. Service commands are executed by the network shell nsh of the SORCER Operating System (SOS). In particular, the shell interprets exertion scripts called *netlets*. There is a helper Java class called ExertShell that has a few methods for running exertions and vars with the Java runtime. In SOS, an exertion is the expression of a structure that consists of a *data context*, a *control context*, and *component exertions* to design hybrid (distributed/local) service collaborations. Conceptually a control context comprises of a *control strategy* and multiple *service signatures*, which define the service invocations on federated providers. The signature usually includes the *service type*, *operation* within the *service type*, and expected *quality of service* (QoS) [15]. An exertion's signatures identify the required providers, but the control strategy for the SOS defines how and when the signature operations are applied to the data context in the federated collaboration. Please note that the service type is the classifier of service providers with respect to their behavior (interface), but the signature is the classifier of service provider instances with respect to the invocation and its service quality defined by its QoS.

From the SOS point of view a netlet is the interpreted exertion (script) but from EOL point o view the exertion is an *expression of a process* that specifies for the SOS how service *collaboration* is actualized by a collection of providers playing specific roles used in a particular way [17]. The *collaboration* specifies a collection of cooperating providers identified by the exertion's signatures. Exertions encapsulate explicitly *data*, *operations*, and a *control strategy* for the collaboration. The SOS dynamically binds the signatures to corresponding service providers—members of the exerted federation. The exerted members in the federation collaborate transparently according to the exertion's *control strategy* managed by the SOS. The SOS invocation model is based on the *Triple Command Pattern* [10] that defines the federated method invocation (FMI).

Herein the service-oriented computing philosophy defines an exertion as a mapping with the property that a single service input context is related to exactly one output context. A context is a dictionary composed of path-value pairs—associations—such that each path referring to its value appears at most once in the context. Everything, which has an independent existence, is expressed in EOL as an association, and relationships between them are modeled as data contexts. Additional attributes with a context path can be specified giving more specific meaning to the value referred by its path. The context attributes form a taxonomic tree, similar to the relationship between directories in file systems. Paths in the taxonomic tree are names of implicit exertion's arguments (context free variables). Each exertion has a single data context as the explicit argument. Paths of the data context form implicit domain specific inputs and outputs used by services providers. Context input associations are used by the providers to compute output associations that are returned in the output context.

The context mapping is defined by an exertion signature that includes at least the name of operation (selector) and the service type defining the service provider. Additionally, the signature may also specify the exertion's return path, the type of returned value, and QoS. Seven signature types are distinguished and are created with the sig operator as follows:

```
  1) sig(<selector>, <code>)              command sig
  2) sig(<selector>, Class|Object)        object sig
  3) sig(<selector>, <service type>)      net sig
  4) sig(Evaluation)                      evaluator sig
  5) sig([<selector>,] Filter)            filter sig
  6) sig(Fidelity, Var)                   var sig
  7) sig(<selector>, Modeling)            model sig
```
where keywords with the first letter capitalized are Java

interfaces or classes.

A selector of a signature may take the expanded form to indicate its data context scope by appending a context prefix after the proper selector with the preceding # character. The part of the selector after the # character is a prefix of context paths specifying the subset of input and output paths for the prefixed signature.

The operator provider returns a service provider defined by a service signature:

```
provider(Signature):Object
```

An exertion specifies the collection of service providers including dynamically federated providers in the network. Its primary service provider is defined by the primary signature marked by the SRV type. An exertion can be used as a closure with its context containing free variables (for example free paths). An *upvalue* is a path that has been bound (closed over) with an exertion. The exertion is said to "close over" its upvalues by exerting service providers. The exertion's context binds the free paths to the corresponding paths in a scope at the time the exertion is executed, additionally extending their lifetime to at least as long as the lifetime of the exertion itself. When the exertion is entered at a later time, possibly from a different scope, the exertion is evaluated with its free paths referring to the ones captured by the closure. There are two types of exertions: service exertions and control flow exertions. Two types of service exertions are distinguished: tasks and jobs. The srv operator defines service exertions as follows:

```
srv(<name> {, <signature> } , <context>
  {, <exertion> }):T <T extends Exertion>
```

For convenience tasks and jobs are also defined with the task and job operators as follows:

```
task(<name>, { <signature> },    <context>):Task
job(<name> [, <signature> ], <context>, <exertion>
  {, <exertion> }):Job
```

A job is an exertion with a single input context and a nested composition of component exertions each with its own input context. Tasks do not have component exertions but may have multiple signatures, unlike jobs that have at least one component exertion and a signature is optional. There are eight interaction operators defining control flow exertions. An interaction operator could be one of: alt (alternatives), opt (option), loop (iteration), break, par (parallel), seq (sequential), pull (asynchronous execution), push (synchronous). The interaction operators opt, alt, loop, break have similar control flow semantics as those defined in UML sequence diagrams for combined fragments. A job represents a mapping that describes how input associations of job's context and component contexts relate, or interact, with output associations of those contexts.

A task is an exertion with a single input context as its parameter and returns the calculated output context. It may be defined with a single signature or multiple signatures (batch). A batch task represents a concatenation of tasks sequentially processing the same-shared context. Processing the context is defined by signatures of PRE type executed first, then the only one SRV signature, and at the end POST signatures if any. The provider defined by the task's SRV signature manages the coordination of exerting the other batch providers. When multiple signatures exist with no type specified, by default all are of the PRE type except the last one being of the SRV type. The task mapping can represent a function, a collection of functions, or relations actualized by collaborating service providers determined by the task signatures.

There are two ways to execute exertions, by exerting the service providers or evaluating the exertion. Exerted service federation returns the exertion with output data context and execution trace available from collaborating providers:

```
exert(Exertion {, entry(path, Object }) : Exertion
```

where entries define a substitution for the exertion closure.

Alternatively, an exertion when evaluated returns its output context or result corresponding to the specified result path either in the exertion's SRV signature or in its data context:

```
value(Exertion {, entry(path, Object) } ) : Object
```

The following getters return an exertion's signature and context:

```
sig(Exertion):Signature
context(Exertion):Context
```

A context of the exertion or its component exertion is returned by the context operator:

```
context(Exertion [, path ] )
```

where path specifies the component exertion. The value at the context path or subcontext is returned by the get operator:

```
get(Context, path {, path}) :Object
or assigned with the put operator:
put(Context {, entry(path, Object) }):Context
```

As an example consider the following object and net tasks.

Example 5 Exertion net task

```
task("net- multiply ",
  sig("multiply", Multiplier.class, result("result/y")),
  context(
    in("arg/x1", 10.0),
    in("arg/x2", 50.0));
```

Example 6. Exertion object task

```
task("obj-multiply",
  sig("multiply", MultiplierImpl.class,
    result("result/y")),
  context(
    in("arg/x1", 10.0),
    in("arg/x2", 50.0));
```

Replacing the service type in the task signature from interface Multiplier.class (Example 5) to its implementation MultiplierImpl.class (Example 6) changes task execution from local to remote. For component services, free signatures in exertion closures allow for reconfiguration of networking (local/distributed) at runtime. As an additional example consider another version of "Hello Arithmetic" with multiple signatures.

Example 7. "Hello Arithmetic" batch task

```
task("Hello Arithmetic",
    sig("multiply", Multiplier.class,
      result("subtract/arg/x1")),
  sig("add", Adder.class,
      result("subtract/arg/x2", Direction.IN)),
  sig("subtract", Subtractor.class,
      result("result/y", Direction.IN)),
  context(in("multiply/arg/x1", 10.0),
      in("multiply/arg/x2", 50.0),
      in("add/arg/x1", 20.0), in("add/arg/x2", 80.0)));
```

In the above batch task its common context defines a scope for each signature by the path prefix being its selector name. This way the service providers can select their inputs or outputs paths in the shared context accordingly. That restriction does not apply to an exertion job as its component exertions have their own data contexts.

An exertion can be described through its relationship with other exertions. Another important operation defined on exertions is exertion composition, where the output from one exertion becomes the input to another exertion. By analogy with our arithmetic subtraction composition, it is possible to define a compound exertion. The "Hello Arithmetic" job by analogy to the "Hello Arithmetic" batch task can be described in EOL by three services:

f3 = x5 – x6; f4 = x1 * x2; and f5 = x3 + x4

that implement three interfaces: Subtractor, Multiplier, and Adder, respectively. We want to program a distributed service that mimics a function composition:

f3(f4(x1, x2), f5(x3, x4))

and calculate: f3(f4(10.0, 50.0), f5(20.0, 80.0))

Example 8. "Hello Arithmetic" exertion net job

```
Task f4 = task("f4", sig("multiply", Multiplier.class),
  context("multiply",
    in("super/arg/x1"), in("arg/x2", 50.0),
    out("result/y")));
Task f5 = task("f5", sig("add", Adder.class),
  context("add", in("arg/x3", 20.0), in("arg/x4", 80.0),
    out("result/y")));
Task f3 = task("f3", sig("subtract", Subtractor.class),
  context("subtract", in("arg/x5"), in("arg/x6"),
    out("result/y")));
Job f1 = job("f1",
  context(in("arg/x1", 10.0), result("f3/result/y")),
    job("f2", t4, t5,
      strategy(Flow.PARALLEL, Access.PULL) ),
    t3,
    pipe(out(f3, "result/y"), in(f5, "arg/x5")),
    pipe(out(f4, "result/y"), in(f5, "arg/x6")));
  assertEquals(get(exert(f1), "f1/f3/result/y"), 400.0);
```

Above, five exertions are declared, three tasks f4, f5, and f3 and two jobs: f1 and f2. A few EOL operators are used in the program to define services: sig defines the *service operation* by its name in the requested service type, e.g., the operation "subtract" defined by the Java interface Subtractor.class in f3; operators in, out specify service input and output parameters by paths in the *data context*. The expressions that start with the operator task or job are *exertions*. Jobs f1 and f2 specify service compositions and define its control *strategy* expressed by the strategy operator. Service jobs define virtual services created from other services. Tasks are *elementary exertions* and jobs are *compound exertions* in exertion-oriented programming.

The task f4 requests to multiply its arguments arg/x1 and arg/x2 by the service Multiplier.class. The value of arg/x1 comes from the data context of its parent job j1 as indicated by the prefix super in the path super/arg/x1. The task f5 requests to add its arguments arg/x1 and arg/x2 by the service Adder.class. The task f3 requests to subtract arg/x2 from arg/x1 by the service Subtractor.class where input values are not yet defined. The job f2 requests execution of both f4 and f5 with its control strategy strategy(Flow.PAR, Access.PULL)). This means that the component exrtions f4 and f5 of f2 are executed in parallel and the corresponding service providers will not be accessed directly by the SOS. In this case the corresponding service providers will process their tasks via the SORCER shared exertion space (PULL) when they are available to do so [17]. The default control strategy is sequential (SEQ) execution with PUSH access, which is applied to job f2. The job f1, executes the nested job f2 and then via data pipes (defined with the pipe operator in f1) passes the results of tasks f4 and f5 on to task f3 for arg/x1 and arg/x2 correspondingly. The last statement in the above program exerts the collaboration exert(f1). Exerting means executing the service collaboration and returning the exertion with the processed contexts of all component exertions along with operational details like execution states, errors, exceptions, etc. Then the get operator returns the value of the service collaboration f1 at the path f1/f3/result/y, which selects the value 400.0 from the context of executed task f3 at the path result/y. The invocation exert(f1), creates at runtime a dynamic federation of required collaborating services by SOS with no network configuration. This type of process is referred to as "federated".

Please note that the program above defines a function composition f3:

```
f3(f4(x1, x2), f5(x3, x4))
```

as a service composition f1:

```
f1(f2(f4(x1, x2), f5(x1, x2)), f3(x4, x5))
```

with two jobs f1 and f2 that are implicit in the function composition f3 since the output from one function becomes the input to another function directly. That's not the case in the exertion composition since each exertion has a single explicit argument of the Context type, thus two implicit free paths of the task f3, arg/x4 and arg/x5, have to be closed over pipes by the job f1 handling f3 and before handling the job f2 with two task f4 and f5 in parallel. On the one hand, the job composition allows specifying the control strategy for executing component exertions. For example, the job f2 is run in parallel and providers pull the component exertion from the network when they are available at their own pace (asynchronous execution). On the other hand, the job composition allows for simplicity and flexible data integration by hiding details of arguments (always one exertion argument with free context paths—hidden arguments) and data flow between component exertions over context pipes. Since the service composition is explicit and the execution control strategy along with its state is embedded in service exertions they can be rerun from the last state if they have been interrupted.

So far we have analyzed service-orientation models and exertions. Let's replace the task f3 in Example 8 with the var task to get a hybrid EOL/VML net job below.

Example 9. Hybrid "Hello Arithmetic" job

```
Task f4 = task("f4", sig("multiply", Multiplier.class),
  context("multiply",
    in("super/arg/x1"), in("arg/x2", 50.0),
    result ("result/y")));
Task f5 = task("f5", sig("add", Adder.class),
  context("add", in("arg/x3", 20.0), in("arg/x4", 80.0),
    result ("result/y")));
Task f3 = task("f3", sig(var("vf3",
    expression("vf3-e", "x5 - x26", vars("x5", "x6"))),
    result(path("result/y"));
Job f1 = job("f1",
  context(in("arg/x1", 10.0), result("f3/result/y")),
  job("f2", t4, t5,
    strategy(Flow.PARALLEL, Access.PULL) ),
  t3,
  pipe(out(f3, "result/y"), in(f5, "arg/x5")),
  pipe(out(f4, "result/y"), in(f5, "arg/x6")));
assertEquals(get(exert(f1), "f1/f3/result/y"), 400.0);
```

Now, consider the "Hello Arithmetic" from Example 4 with var f5 as the hybrid of var modeling and exertion programming with a net task instead of an expression evaluator.

Example 10. Hybrid "Hello Arithmetic" model

```
VarModel vm = model("Hybrid Hello Arithmetic",
  inputs(
    var("x1"), var("x2"), var("x3", 20.0), var("x4")),
  outputs(
    var("f4", expression("x1 * x2",
      args(vars("x1", "x2")))),
    var("f5", task("t5",
      sig("add", Adder.class),
      context("add",
        in("arg/x3", var("x3")),
        in("arg/x4", var("x4")), result("result/y")))),
    var("f1", expression("f4 - f5",
      args(vars("f4", "f5"))))));
```

Note that the same vars appear in the exertion task t5 and in the model as well. Evaluate and test the var f1 in the model vm over x1, x2, and x4:

```
assertEquals(value(var(put(vm,
  entry("x1", 10.0), entry("x2", 50.0), entry("x4", 80.0),
    "f1")), 400.0);
```

Three forms of EOP have been developed: Exertion-oriented Java API, interactive graphical, and EOL textual programming. The exertion-oriented Java API is presented in [8,9] and the graphical interactive exertion-oriented programming is presented in [18]. Details regarding textual EOP along with two examples of simple EO programs can be found in [5,15].

3.1. How to Create an Application Service Provider?

To complete the "Hello Arithmetic" job declared in Example 8, let's implement one of the arithmetic services, for example Adder that can be used by the SOS as expected. A plain old Java object (POJO) becomes a SORCER service provider, when injected into a standard SORCER service container called ServiceTasker. Such an object, called a service bean, implements its service type (Java interface that does not have to be Remote), with the following characteristics:

1) Defines the service operations you'd like to call remotely;

2) The single parameter and returned value of each operation is of the type sorcer.service.Context;

3) Each method must declare RemoteException in its throws clause. The method can also declare application-specific exceptions; and

4) The class implementing the interface and local objects must be serializable.

The interface for the Adder bean can be defined as follows:

```
interface Adder {
  Context add(Context context)
    throws RemoteException;
}
```

The Adder interface implementation:

```
public class AdderImpl implements Adder {
  public Context add(Context context)
    throws RemoteException {
    double result = 0;
    List<Double> inputs = context.getInValues();
    for (Object value : inputs)
      result += value;
    context.putValue(context.getOutPath(), result);
    return context;
  }
}
```

Finally, starting the ServiceTasker with the following configuration file:

```
sorcer.core.provider.ServiceProvider {
  name = "SORCER Adder";
  beans = new Class[] {
    sorcer.arithmetic.AdderImpl.class };
}
```

registers the Adder provider dynamically with the SOS network processor. In the same fashion one can implement and deploy the Multiplier and Subtractor providers necessary to execute Examples 8-10.

4. SO Optimization Models

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 a var's multifidelity. These are called *amorphous* models. For the same var-model an alternative pair of evaluator-filter (new fidelity) can be selected or added at runtime to evaluate a new particular process ("shape") of the model and quickly update the related computations in an evolving or new direction.

Consider the Rosen-Suzuki optimization problem, where:

design variables: $x1$, $x2$, $x3$, $x4$

response variables: f, $g1$, $g2$, $g3$, and

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

$g1$ = $x1$^2+$x1$+$x2$^2-$x2$+$x3$^2+$x3$+$x4$^2-$x4$-8.0

$g2$ = $x1$^2-$x1$+2.0*$x2$^2+$x3$^2+2.0*$x4$^2-$x4$-10.0

$g3$ = 2.0*$x1$^2+2.0*$x1$+$x2$^2-$x2$+$x3$^2-$x4$-5.0

The goal is to minimize f subject to

$g1 <= 0$, $g2 <= 0$, and $g3 <= 0$.

In VML this problem is expressed by the following var-model:

Example 11. Optimization model in VML

```
int inputsCount = 4;
int outputsCount = 4;
OptimizationModel rsm = model("R-S Model",
  inputs(loop(inputsCount), "x", 20.0, -100.0, 100.0)),
  outputs("f"),
  outputs (loop(outputsCount -1), "g"),
  objectives(var("fo", "f", Target.min)),
  constraints(
    var("g1c", "g1", Relation.lte, 0.0),
    var("g2c", "g2", Relation.lte, 0.0),
    var("g3c", "g3", Relation.lte, 0.0)));
configureModel(model);
```

All vars in the model are configured with needed evaluators/filters and differentiators by the method configureModel, 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"));
```

Having the rsm model declared and configured we can set values of input vars:

```
put(rsm, entry("x1", 1.1), entry ("x2", 2.2),
   entry ("x3", 3.3), entry ("x4", 4.4));
```

and get the output value of f:

```
assertEquals(value(rsm, "f"), 42.190000000000005));
```

or the value of constraint var g2c:

```
assertEquals(value(rsm, "g2c"), false));
```

Alternatively we can evaluate vars using var closures as illustrated in Example 4. The Rosen-Suzuki Model can be used locally as the Java object rsm or that object can be deployed directly in SORCER as a service bean (see Section 3.1) and used by the design space exploration provider of the Exploration type in combination with an optimizer provider of the Optimization type, e.g., the CONMIN code or DOT [19].

A var from a var-model can be accessed with var (<model>, <varName>). However, in multidisciplinary modeling a var from one remote model can be used in another remote model. Var proxying allows for building transdisciplibary-distributed models. In the example below the pf var is a proxy to the f var in the net Rosen-Suzuki Model declared in Example 11.

Example 12. Proxy var in the Rosen-Suzuki Model

```
Var pf = var("f", sig(OptimizationModeling.class,
   "Rosen-Suzuki Model"));
assertEquals(value(pf), 1570.0);
```

More complicated modeling tasks like parametric analysis or optimization can be complemented with EOP presented in Section 3. Evaluators for vars can be defined as exertions and vice versa exertions can use vars and var-models as their modeling components as well (see Examples 9 and 10).

Returning to Example 11, the Rosen-Suzuki Model, we create the parametric analysis and optimization exertion tasks. In the parametric task mt the model is specified by the net signature msig within the signature of the task mt. The model reads the parametric table at inURL and writes the response parametric table at outURL in the format specified by inputs and outputs operators. The response table is also returned in the task context at the path table/out as the requested result.

Example 13. Parametric analysis of Rosen-Suzuki problem

```
Signature msig = sig(ParametricModeling.class,
    "Rosen-Suzuki Model");
  String outURL = Sorcer.getWebsterUrl()
    + "/rs-model/rs-out.data";
  String inURL = Sorcer.getWebsterUrl()
    + "/rs-model/rs-in.data";
ModelTask mt = task(sig("calculateOutTable", msig),
    context(parametricTable(inURL),
      responseTable(outURL, inputs("x1", "x2"),
        outputs("f", "g1", "g2")),
      result("table/out"),
      par(queue(20), pool(30))));
  Table responseTable = value(mt);
```

For the optimization opti task, first, define the data context, then the opti task that takes it as an argument, and finally we exert the Exploration net provider named Rosen-Suzuki Explorer specified by the task's signature.

Example 14. Nonlinear optimization problem

```
// Create an optimization data context
Context exploreContext = exploreContext(
  "Rosen-Suzuki context",
  inputs(
    entry("x1", 1.0), entry ("x2", 1.0),
    entry ("x3", 1.0), entry ("x4", 1.0)),
  strategy(new ConminStrategy(
    new File(System.getProperty(
      "conmin.strategy.file")))),
  dispatcher(
    sig(null, RosenSuzukiDispatcher.class)),
  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",
      result("exploration/results")),
    exploreContext);
// Execute the exertion and log the optimal solution
logger.info("Rosen-Suzuki exploration results:" +
  value(opti));
```

The exertion's output solution is logged as follows:

```
Rosen-Suzuki exploration results:
Objective Function fo = 6.002607805900986
Design Values
x1 = 2.5802964087086235E-4
    x2 = 0.9995594642481355
    x3 = 2.000313835134211
    x4 = -0.9986692050113675
Constraint Values
    g1c = -0.002603585246998996
    g2c =-1.0074147118087602
    g3c = 4.948009193483927E-7
Exploration statistics
    Number of Objective Evaluations = 88
    Number of Constraint Evaluations = 88
    Number of Objective Gradient Evaluations = 29
    Number of Constraint Gradient Evaluations = 29
```

In the above program the exploreContext defines initialization of the input vars (x1, x2, x3, and x4) of the optimization model, its optimization strategy, and the exploration dispatcher with two required network services—specified by two signatures for the optimizer and required optimization model. The instance of custom RosenSuzukiDispatcher is specified by the signature sig(RosenSuzukiDispatcher.class) to be used by the generic exploration service. The context is then used to define the exertion task opti with the signature for the exploration provider named Rosen-Suzuki Explorer of the Exploration type. The function value(opti) executes the opti task in the network for event-driven collaboration between the optimizer and the model that is managed by the explorer customized by the given dispatcher. The presented schema is generic as the explorer can use multiple hierarchically organized dispatchers to implement its optimization strategy provided by the task. Also, the task can specify a custom model manager that is responsible for complex updates and reconfigurations of the model during collaborative optimization. For simplicity, the signatures do not specify QoS [6,7] for the specified providers.

5. The SORCER Operating System (SOS)

In SORCER the provider container (ServiceTasker) is responsible for deploying services in the network, publishing their proxies to one or more registries, and allowing requestors to access its proxies. Providers advertise their availability in 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 the SOS. Providers use discovery/join protocols [1,20,21] to publish services in the network and the SOS uses discovery/join protocols to obtain service proxies in the network. While an exertion defines the *orchestration* of its service federation, the SOS implements the service *choreography* in the federation defined by its FMI [10].

The SOS allows execution of *netlets* (interpreted mograms containing exertions) by exerting the specified federation of service providers. The overlay network of the service providers defining the functionality of SOS is called the *sos-cloud* (see **Figure 2**) and the overlay network of application providers is called the *app-cloud*—service processor [5] (see **Figure 2**). The *instruction set* of the SOS service processor consists of all operations offered by all service providers in the app-cloud. Thus, an exertion is composed of instructions specified by service signatures with its own control strategy per service composition and data context representing the shared data for the underlying federation. The signatures (instances of Signature type) returned by the sig operator specify participants of collaboration in the app-cloud.

A provider signature is defined by its service type, operation in that interface, and a set of optional QoS attributes. A SRV signature—of which there is only one allowed per exertion—defines the dynamic late binding to a provider that implements the signature's interface. The data context describes the data that tasks and jobs operate on. The SOS allows for an exertion to create a service federation and coordinate the execution of all nested signatures bound to providers in the federation. The exertion-oriented computing concepts are defined differently than those in traditional grid/cloud computing where a job is just an executing process for a submitted executable code—the executable becomes the single service itself that can be parallelized on multiple processors, if needed. Herein a job is the expression of collaborating service providers within the federation that is formed by the SOS for the job as specified by all its nested signatures (see various signature types in Section 3).

An exertion object of the Exertion type is created by either task or job operators in EOL. Then, the ExertShell can execute the exertion as follows:

ExertShell#exert(Exertion, Transaction):Exertion

where a parameter of the Transaction type is required when transactional semantics is needed for the participating service providers within the collaboration defined by the exertion. Thus, EO programming allows one to execute an exertion and invoke exertion's signatures on collaborating service providers indirectly, but where does the service-to-service communication come into play? How do these services communicate with one another if they are all different? Top-level communication between services, or the sending of service requests, is done through the use of the generic Servicer interface and the operation service that SORCER providers are required to implement:

Servicer#service(Exertion, Transaction):Exertion.

This top-level service operation takes an exertion object as an argument and gives back an exertion object as the return value.

So why are exertion objects used rather than directly calling on a provider's method and passing data contexts? There are two basic answers to this. First, passing exertion objects helps to aid with the network-centric messaging. A service requestor can send an exertion object implicitly out onto the network, ExertShell#exert(Exertion), and any service provider can pick it up. The receiving provider can then look at the signature's interface and operation requested within the exertion object, and if it doesn't implement the desired interface and provide the desired method, it can continue forwarding it to another service provider who can service it. Second, passing exertion objects helps with fault detection and recovery. Each exertion object has its own completion state

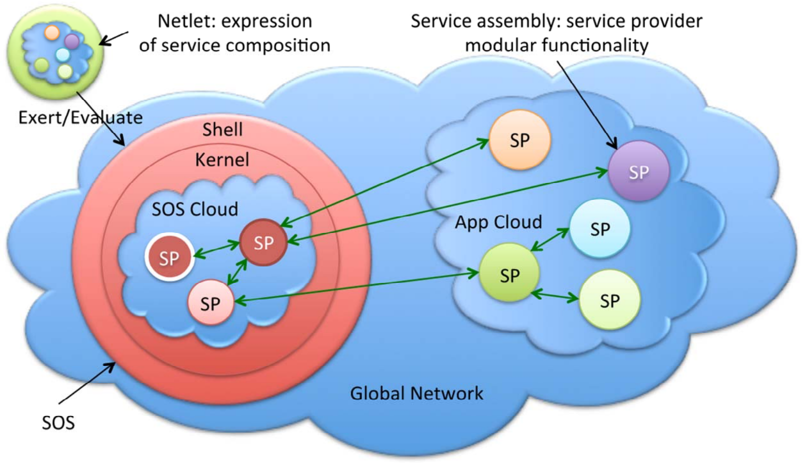
Netlet: expression of service composition
Service assembly: service provider modular functionality
Exert/Evaluate
Shell
Kernel
SOS Cloud
SP
SP
SP
App Cloud
SP
SP
SP
SP
SP
Global Network
SOS

Figure 2. The modules of the SOS kernel are service providers (SPs), the same kind as the application domain-specific SPs. Both the SOS federation in the sos-cloud and application federation in the app-cloud consist of dynamically federated net SPs by the SOS for its executing netlet. Local (not shown) SPs run within the SOS shell and/or net SOS/App SPs. A service composition is defined by the user's netlet; in contrast a service assembly is developed and configured by a developer of service provider.

associated with it to specify if it has yet to run, has already completed, or has failed. Since full exertion objects are both passed and returned, the user can view the failed exertion to see what method was being called as well as what was used in the data context input that may have caused the problem. Since exertion objects provide all the information needed to execute the exertion including its control strategy, the user would be able to pause a job between component exertions, analyze it and make needed updates. To determine where to resume an exertion, the executing provider would simply have to look at the exertion's completion states and resume the first one that wasn't completed yet. In other words, EOP allows the user, not programmer to update the metaprogram on-the-fly, which practically translates into creating new interactive collaborative applications at runtime.

Applying the inversion principle, the SOS executes the exertion's collaboration with dynamically found, if present, or provisioned on-demand service providers [7]. The exertion caller has no direct dependency to service providers since the exertion uses only service types they implement.

Despite the fact that any servicer can accept any exertion, SOS services have well defined roles in the S2S platform [15]:

1) Taskers—accept exertion tasks; they are used to create application services by dependency injection (service assembly from service beans and related components) or by inheritance (subclassing ServiceTasker and implementing required service interfaces) [9];

2) Jobbers—manage service collaboration for PUSH service access [17];

3) Spacers—manage service collaboration for PULL service access using space-based computing [17];

4) Contexters—provide data contexts for APPEND signatures;

5) FileStorers—provide access to federated file system providers [11];

6) Catalogers—SOS registries, provide management for QoS-based federations;

7) ExertMonitors—monitor execution of running exertions;

8) SlaMonitors—provide monitoring of SLAs [7];

9) Provisioners—provide on-demand provisioning [22];

10) Persisters—persist data contexts, tasks, and jobs to be reused for interactive EO programming;

11) Relayers—gateway providers; transform exertions to native representation, for example integration with Web services and JXTA;

12) Authenticators, Authorizers, Policers, KeyStorers—provide support for service security;

13) Auditors, Reporters, Loggers—support for accountability, reporting, and logging

14) Griders, Callers, Methoders—support for a conventional compute grid (managing and running executable codes in the network);

15) Notifiers—use third party services for collecting provider notifications for time-consuming programs and disconnected requestors.

Both sos-providers and app-providers do not have mutual associations prior to the execution of an exertion; they come together dynamically (federate) for all nested tasks and jobs in the exertion.

Domain specific servicers within the app-cloud—taskers—execute task exertions. Rendezvous peers (jobbers, spacers, and catalogers) manage service collaborations. Providers of the Tasker, Jobber, and Spacer type are basic service containers. In the view of the P2P architecture [17] defined by the Servicer interface, a job can be sent to any servicer. A peer that is not a Jobber or Spacer type is responsible for forwarding the job to one of the available rendezvous peers in the SORCER environment and returning results to the requestor. Thus implicitly, any peer can handle any exertion type. Once the exertion execution is complete, the federation dissolves and the providers in the federation disperse to seek other exertions to join.

The functional notation to execute exertions—exert (Exertion):Exertion and value(Exertion):Object—used in the programming examples relies the ExertShell described earlier. Once the SORCER runtime is installed you can also run netlets like any other script. In direct interpretation, the command:

```
nsh -f myNetlet.xrt [arguments]
```

invokes the SORCER network shell (nsh) to interpret the netlet contained in the file myNetlet.xrt and passes the created exertion object onto the ExertShell for execution. The -f option says that the file is interpreted (see **Figure 2**). In indirect interpretation, the first line of the file should be in the format:

```
#!/usr/bin/env nsh -f
```

Then an exertion script can be invoked in the same way as any other command, *i.e.*, by typing the script name on the command line. For a command line interacttive shell type nsh and at the prompt execute: exert myNetlet.xrt. The interactive nsh allows for booting or destroying service providers, looking up providers, monitoring running exertions in the network, etc.

You can write netlets and execute them directly on the command line as if they were normal Unix shell scripts. The following netlet, which defines the exertion in Example 4 can be executed directly:

```
#!/usr/bin/env nsh -f
import sorcer.arithmetic.provider.Multiplier;
import sorcer.service.Strategy.Monitor
import sorcer.service.Strategy.Wait
task("net-multiply",
  sig("multiply", Multiplier.class),
    context(
        input("arg/x1", 10.0d),
        input("arg/x2", 50.0d),
        output("result/y")),
    strategy(Monitor.YES, Wait.NO));
```

6. 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 complex adaptive systems are required. These languages should reduce the complexity of metacomputing problems we are facing in SO computing, for example, the collaborative design by hundreds of people working together and using thousands of programs written already in software languages that are dislocated around the globe. The multidisciplinary design of an aircraft engine or even a whole air vehicle requires large-scale high performance metacomputing systems handling anywhere-anytime collaborations of various executable codes in the form of applications, tools, and utilities. Domain specific languages are mainly for humans, unlike software languages for computers, intended to express domain specific complex problems and related solutions. Three programming languages for SO computing are described in this paper: VOL, VML, and EOL. The network shell (nsh) interprets mograms in these languages and the SOS manages related service federations.

As complexity of problems being solved increases continuously, we have to recognize the fact that in SO computing the only constant is change. The concept of the evaluator-filter pair in the VFE framework combined with exertions provides the uniform modeling technique for SO integration and interoperability with various applications, tools, utilities, and data formats.

The SORCER operating system supports the two-way convergence of modeling and programming for SO computing as presented in hybrid programming examples. On one hand, EOP is uniformly converged with VOP and VOM to express an explicit network-centric computation process emphasizing that *the network of service providers is the computer*. On the other hand, VOM and VOP are uniformly converged with EOP to express an explicit declarative SO model emphasizing that *the computer is the network of vars*. The evolving SORCER environment with its SO computing model has been successfully verified and validated in multiple concurrent engineering and large-scale distributed applications [16,23-25].

7. Acknowledgements

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

REFERENCES

[1] "Jini Network Technology Specifications v2.1," 2012. http://www.jiniworld.com/doc/spec-index.html

[2] "Metacomputing: Past to Present," 2011. http://archive.ncsa.uiuc.edu/Cyberia/MetaComp/MetaHistory.html

[3] I. Foster, C. Kesselman and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of Supercomputer Applications*, Vol. 15, No. 3, 2001.

[4] D. S. Linthicum, "Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide," Addison-Wesley Professional, 2009.

[5] M. Sobolewski, "Object-Oriented Service Clouds for Transdisciplinary Computing," In: I. Ivanov, *et al.*, Eds., *Cloud Computing and Services Science*, Springer Science + Business Media, New York, 2012.

[6] M. Sobolewski, "Provisioning Object-Oriented Service Clouds for Exertion-Oriented Programming," *Proceedings of CLOSER* 2011*—International Conference on Cloud Computing and Services Science*, 2011, pp. IS-11-IS-25.

[7] P. Rubach and M. Sobolewski, "Autonomic SLA Management in Federated Computing Environments," *International Conference on Parallel Processing Workshops*, Vienna, 22-25 September 2009, pp. 314-321. doi:10.1109/ICPPW.2009.47

[8] M. Sobolewski, "Federated P2P Services in CE Environments," *Advances in Concurrent Engineering*, A. A. Balkema Publishers, Taylor and Francis, 2002, pp. 13-22.

[9] M. Sobolewski, "Exertion Oriented Programming," *IADIS*, Vol. 3, No. 1, 2008, pp. 86-109.

[10] M. Sobolewski, "Metacomputing with Federated Method Invocation," In: M. A. Hussain, Ed., *Advances in Computer Science and IT*, In-Tech, 2009, pp. 337-363. http://sciyo.com/articles/show/title/metacomputing-with-federated-method-invocation

[11] M. Sobolewski, "Object-Oriented Metacomputing with Exertions," In: A. Gunasekaran and M. Sandhu, Eds., *Handbook on Business Information Systems*, World Scientific Publishing Co. Pte. Ltd., 2010.

[12] SORCERsoft. http://sorcersoft.org

[13] A. Kleppe, "Software Language Engineering," Pearson Education, London, 2009.

[14] K. M. Fant, "A Critical Review of the Notion of Algorithm in Computer Science," *Proceedings of the* 21*st Annual Computer Science Conference*, Indianapolis, February 1993, pp. 1-6.

[15] M. Sobolewski, "Exerted Enterprise Computing: From Protocol-Oriented Networking to Exertion-Oriented Networking," In: R. Meersman, *et al.*, Eds., *OTM* 2010 *Workshops*, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 182-201. http://river.apache.org/

[16] R. M. Kolonay and M. Sobolewski, "Service Oriented

Computing EnviRonment (SORCER) for Large Scale, Distributed, Dynamic Fidelity Aeroelastic Analysis & Optimization," *International Forum on Aeroelasticity and Structural Dynamics*, Paris, 26-30 June 2011.

[17] M. Sobolewski, "Federated Collaborations with Exertions," 17*th IEEE International Workshop on Enabling Technologies*: *Infrastructures for Collaborative Enterprises* (*WETICE*), Rome, 23-25 June 2008, pp. 127-132.

[18] M. Sobolewski and R. Kolonay, "Federated Grid Computing with Interactive Service-Oriented Programming," *International Journal of Concurrent Engineering*: *Research & Applications*, Vol. 14, No. 1, 2006, pp. 55-66.

[19] CONMIN User's Manual, 2012. http://www.eng.buffalo.edu/Research/MODEL/mdo.test.orig/CONMIN/manual.html

[20] W. K. Edwards, "*Core Jini*," 2nd Edition, Prentice Hall, Upper Saddle River, 2000.

[21] Apache River, http://river.apache.org/

[22] Rio Project, http://www.rio-project.org/

[23] S. Goel, S. S. Talya and M. Sobolewski, "Mapping Engineering Design Processes onto a Service-Grid: Turbine Design Optimization," *Concurrent Engineering*, Vol. 16, No. 2, 2008, pp. 139-147. doi:10.1177/1063293X08092487

[24] R. M. Kolonay, E. D. Thompson, J. A. Camberos and F. Eastep, "Active Control of Transpiration Boundary Conditions for Drag Minimization with an Euler CFD Solver," AIAA-2007-1891, 48*th AIAA/ASME/ASCE/AHS/ASC Structures*, *Structural Dynamics*, *and Materials Conference*, Honolulu, 2007.

[25] W. Xu, J. Cha and M. Sobolewski, "A Service-Oriented Collaborative Design Platform for Concurrent Engineering," *Advanced Materials Research*, Vol. 44-46, 2008, pp. 717-724. doi:10.4028/www.scientific.net/AMR.44-46.717