20th ISPE International Conference on Concurrent Engineering
C. Bil et al. (Eds.)

doi:10.3233/978-1-61499-302-5-381

Parametric Mogramming with Var-oriented Modeling and Exertion-Oriented Programming Languages

Michael Sobolewski[a, b], Scott Burton[a, c], and Raymond Kolonay[a]
[a] *Air Force Research Laboratory, WPAFB, Ohio 45433*
[b] *Polish Japanese Institute of IT, 02-008 Warsaw, Poland*
[c] *American Optimization LLC, Liberty Township, Ohio 45044*

Abstract. The Service ORiented Computing EnviRonment (SORCER) targets service abstractions for transdisciplinary concurrent engineering with support for true service-oriented (SO) computing. SORCER's models are expressed in a top-down Var-oriented Modeling Language (VML) unified with programs in a bottom-up Exertion-Oriented Language (EOL). In this paper the basic concepts of *mogramming* are presented. On the one hand, modeling with service variables allows for computational fidelity within multiple types of evaluations. On the other hand, any combination of local and remote services can be described in EOL as a collaborative federation of engineering applications, tools, and utilities. An example of aircraft conceptual design application is given to illustrate how parametric models can participate in service-oriented engineering analyses.

Keywords. transdisciplinary concurrent engineering, service-oriented mogramming; var-oriented modeling; exertion-oriented programming; SOA; SORCER

Introduction

A transdisciplinary computational model [4] requires extensive computational resources to study the behavior of a system by computer simulation. The large system under study that consists of thousands or millions of variables is often a complex adaptive system for which simple, intuitive analytical solutions are not readily available. Usually adjusting the parameters of system in the computer network does experimentation with the distributed model. 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 service collaboration to calculate and/or optimize the model across multiple disciplines fusing their domain-specific services running on laptops, workstations, clusters, and supercomputers.

An elementary *service* is the work performed in which a service provider (one that serves) *exerts acquired abilities* to execute a computation. Elementary services are autonomous units of functionality and can be either local or distributed. Elementary services have no calls to each other embedded in them. By contrast a compound service is a composition of elementary and other compound services that *exerts acquired abilities* of collaborating service providers. Each elementary service implements multiple actions of a cohesive (well integrated) service type, usually defined by an

interface type in an underlying programming language, e.g. a Java interface. A service provider can implement multiple service types, and thus can provide multiple elementary services to be offered. An elementary service reference with operation of its service type, complemented by its QoS parameters is called a *service signature*. Service signatures are used to reference local or remote service providers. Different instances of a service provider are equivalent units of functionality identified by the same signature.

In most service systems the focus is on back-end aggregation of services into a single provider, thus having more services performed by the same provider or by the same provider node, e.g., an application server. In either case theses new services are still elementary services to the end user. This type of back-end aggregation, done by software developers and deployers, is called *service assembly* in contrast to the aggregation of services at the front-end accomplished by the end user. The front-end aggregation is called *service composition* and requires service-oriented (SO) languages to express actualization of compound services. Service compositions are called *exertions* and use service signatures to bind at runtime to corresponding service providers. A dynamic collection of service providers requested for the actualization of exertion is called a *service federation*.

In concurrent engineering computing systems each local or distributed service provider in the collaborative federation performs its services in an orchestrated interaction of applications, tools, and utilities. Once the service collaboration is complete, the federation dissolves and the providers disperse and seek other federations to join. These service providers have to be managed by a relevant *service-centric operating system* with programming environment to express complex interactions of providers in dynamic virtual federations [1].

The SORCER platform [4][5][6] (open source project [7]) introduces front-end mogramming languages [3][8] with a modular SO Operating System (SOOS). It 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). The unification of VML and EOL has been verified and validated in research projects at Air Force Research Lab and SORCER Lab at TTU [1][9][11][12][13].

The remainder of this paper is organized as follows: Section 1 describes briefly var-oriented modeling; Section 2 describes exertion-oriented programming; Section 3 introduces the SORCER SOOS; Section 4 demonstrates parametric modeling for aircraft conceptual design; finally Section 5 concludes with final remarks and comments.

1. Var-oriented Modeling

A computation is a relation between a set of inputs and a set of corresponding 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 *front-end* service composition with its own control strategy created by the end user in *Exertion-Oriented Language* (EOL) is called an *exertion*. A service variable, called a *var* and *var-model* are *front-end* modeling services in the *Var-Oriented Language* (VOL) and the *Var-oriented Modeling Language* (VML), respectively.

The exertions are drawn primarily from the semantics of a routine. The vars and var-models are drawn primarily from the semantics of a variable and function composition. Either one of these process expressions 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, getters, and setters) including exertions as evaluators. In var-oriented modeling three types of models can be defined (response, parametric, and optimization). EOL distinguishes three types of exertions: elementary exertions (*tasks*), batch tasks (*batches*), and hierarchical exertions (*jobs*). The functional composition notation has been used for expressions in VOL, VML, and EOL that is usually complemented with the Java API.

1.1. Var-Oriented 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 collection of triplets: { <*evaluator*, *getter*, *setter*> },

where:

1. an *evaluator* is a service with the argument vars that define the var dependency chain;
2. a getter is a *pipeline of filters* processing the result of evaluation; and
3. a *setter* assigns and returns a *value* that is a quantity filtered out from the output of the current evaluator.

The var value is invalid when the current evaluator, getter, or setter is changed, current evaluator's arguments change, or the value is undefined. VOP is a programming paradigm that uses vars to design var-oriented multifidelity compositions. An <evaluator, getter, setter> triplet 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 selectable var fidelities and their dependency chains of argument vars to become the main concept behind any processing.

Evaluators, getters, and setters 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 <evaluator, getter, setter> can be used with the same var allowing var's fidelity. The semantics of the *value*, whether the var represents a mathematical function, subroutine, coroutine, or data, depends on the evaluator, getter, and setter currently used by the var. The var dependency chaining provides the integration framework for all possible kinds of computations represented by various types of evaluators including exertions described in Section 2.

1.2. Var-Oriented 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, getters, and setters 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 [9]. In the model hierarchy, optimization models are parametric and response, and parametric are response ones as well. 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 [3]. 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:

```
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, getters, setters, and differentiators by the method configureModel. 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));`

Var-models with no constraints and objective are parametric models. A parametric task (see Section 2) allows for specifying a parametric table with rows of values of input vars and calculate the corresponding output table as illustrated in Fig 1.

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 triplet <evaluator, getter, setter> (new fidelity) can be selected or added at runtime to evaluate an updated analysis ("shape") of the model and quickly update the

related computations in an evolving or new direction. Var-models can be used as local object models or as network service providers. In either case modeling tasks (exertions) are used to specify modeling services as illustrated in Section 2.

2. Exertion-oriented Programming

The central exertion principle is that a computation can be expressed and actualized 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 types provides multiple services. Its service type complemented by its QoS parameters can identify functionality of a provider. In an exertion-oriented language (EOL) a *service exertion* can be used as a closure over free variables in the exertion's data and control contexts. In exertion-oriented programming everything is a service. Exertions can be used directly as service providers as well.

In EOL service providers are uniformly accessed through two types of references: *class* and *interface* signatures. *Class* and *interface* signatures are also called *object* and *net* signatures correspondingly. Exertion-oriented programming (EOP) is a SO programming paradigm using *service providers* and *exertions*. Exertions can be created with textual language (*netlets*), API (*exertlets*), and user agents that behind visual interactions create exertlets. Netlets are interpreted scripts and executed by the network shell `nsh` of the SORCER Operating System (SOS). Invoking the *exert* operation on the exertlet (Java object) returns the collaborative result of the requested service federation. Netlets are executed with a SORCER network shell (nsh) the same way Unix scripts are executed with any Unix shell.

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* that defines the federated method invocation (FMI) [5].

Three types of service exertions are distinguished: tasks, batches and jobs. The `srv` operator defines service exertions as follows:

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

For convenience tasks, batches, and jobs are also defined with the `task`, `batch`, and `job` EOL operators as follows:

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

Consider a parametric task `mt` that specifies a parametric model in the network by a service type `ParametricModeling.class` named `"Rosen-Suzuki Model"` with a parametric and response tables indicated by `inURL` and `outURL` correspondingly.

```
ModelTask mt = parametricTask(
  sig("evaluateResponseTable", ParametricModeling.class, "Rosen-Suzuki Model"),
  context(responseTable(outURL, responses("f", "g1", "g2")),
    parametricTable(inURL, rows(11, 16), parameters("x1", "x2")),
```

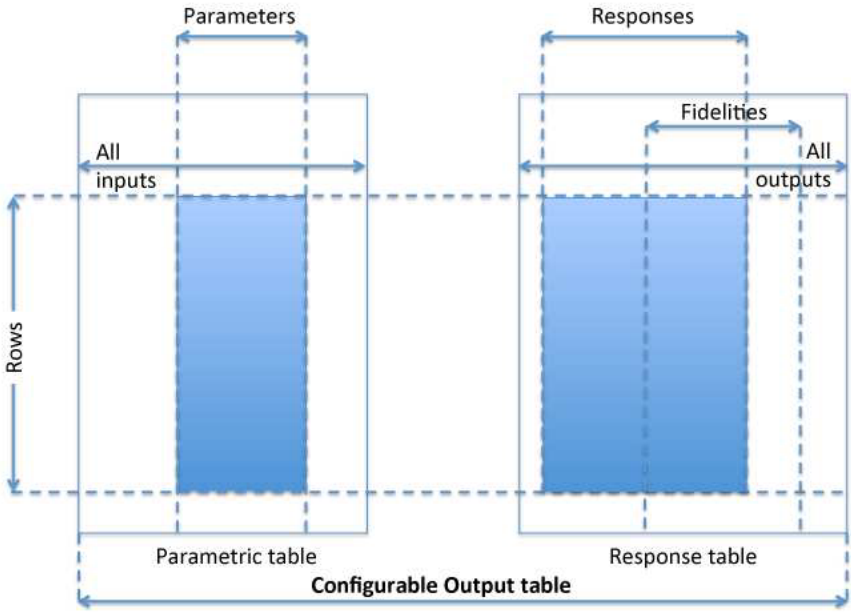

Figure 1. Output table of parametric analysis with configurable parameters and responses.

```
fidelities(entry("f", eFi("fe"))), result("table/out"), par()));
```

The returned output `table` is specified in the task according to a structure shown in Fig. 1 and it is calculated by calling: `value(mt)`. At the same time the output table is written into `outURL`.

3. 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 [2] 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 [5].

The SOS allows execution of *netlets* (interpreted mograms) 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* and the overlay network of application providers is called the *app-cloud*—service processor [4]. 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 specify participants of collaboration in the app-cloud.

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—synchronous service coordination, spacers—asynchronous service coordination [10], and catalogers—dynamic network service catalogs) manage service collaborations. Providers of the Tasker, Jobber, and Spacer type are basic service containers.

4. Aircraft Conceptual Design Application using SORCER

The Air Force Research Lab's (AFRL) Multidisciplinary Science and Technology Center (MSTC) is investigating conceptual design processes and computing frameworks that could significantly impact the design of the next generation efficient supersonic air vehicle (ESAV). To make the technological advancements required of a new ESAV, the conceptual design process must accommodate both low- and high-fidelity multidisciplinary engineering analyses. These analyses may be coupled and computationally expensive, which poses a challenge since a large number of configurations must be analyzed. In light of these observations, the ESAV design process was implemented using the SORCER Operating System (SOS) to combine propulsion, structures, aerodynamics, performance, and aeroelasticity in a multidisciplinary analysis (MDA). The SORCER platform provides the MDA automation and tight integration to distributed computing resources necessary to achieve the volume of analyses required for conceptual design.

The MDA is a blend of conceptual and preliminary design methods from propulsion, structures, aerodynamics, performance, and aeroelasticity disciplines. The analysis process and data flow is shown in the ESAV N^2 diagram in Fig. 2. The process begins by parametrically generating discretized geometry suitable for several different analyses at varying fidelities. The geometry is used as input to compute several figures of merit of the aircraft, which include the aircraft drag polars, design mass, range, and aeroelastic performance. The different responses are evaluated for several flight conditions and maneuvers. These responses are then used to construct the objective and constraints of the multidisciplinary optimization (MDO) problem.

MDO generally require a large number of MDAs be performed. This significant computational burden is addressed by using the SORCER platform. The network-centric approach of SORCER enables the use of heterogeneous computing resources, including a variety of operating systems, hardware, and software. Specifically, the ESAV studies performed herein use SORCER in conjunction with a mix of Linux-based cluster computers, desktop Linux-based PCs, Windows PCs, and Macintosh PCs. The ability of SORCER to leverage these resources is significant to MDO applications in two ways: 1) it supports platform-specific executable codes that may be required by an MDA; and 2) it enables a variety of computing resources to be used as one entity (including stand-alone PCs, computing clusters, and high-performance computing facilities). The main requirements for using a computational resource in SORCER are network connectivity and Java platform compatibility. SORCER also supports load

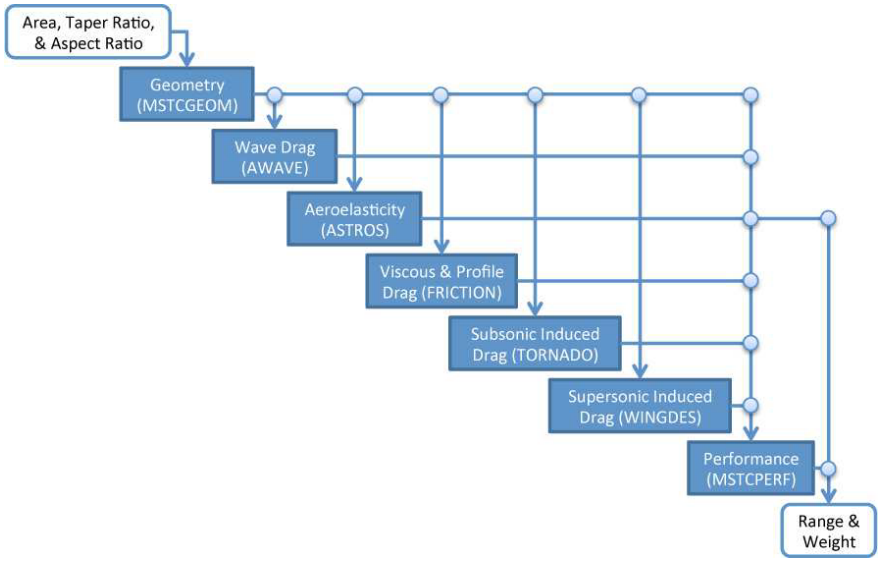

Figure 2. The ESAV MDA N^2 diagram includes geometry generation, aerodynamic analysis, aeroelastic analysis, and performance analysis. (Each box in the figure represents a SORCER Provider).

balancing across computational resources using space computing, making the evaluation of MDO objective and constraint functions in parallel a simple and a dynamically scalable process.

SORCER employs Jini [1] technology with its JavaSpaces service [14] to implement loosely coupled space-based service federations. The SOS via its Spacer providers [10] enables different processes on different computers to communicate asynchronously in a reliable manner. Using Spacer services, SOS implements a self-load balancing service cloud that can dynamically grow and shrink during the course of an optimization study, see Fig 3-left.

An exertion space—or "space"—is a exertion storage in the network that is managed by SOS (its Spacer providers). The space provides a type of shared memory where requestors, e.g., vars, can put exertions they wish to be processed by service providers. Service providers, in turn, find spaces in the network and monitor them for exertion tasks with their service type. If a service provider sees a task it can operate on in a space, and the task has a flag indicating it has not been processed, the provider takes the task from the space. The provider then performs the requested service and returns the task to the space with a flag indicating the task has been processed. Once the task has been returned to the space, the Spacer that initially wrote the task to the space detects the returned task and checks to see if it has been processed. If the task indicates it has been processed, the Spacer removes the task from the space and returns it to the submitting service requestor.

To achieve the load balancing across multiple computers, a service provider may be configured to have a fixed number of worker threads. The number of worker threads determines the number of tasks from the space a provider can process in parallel. By configuring the number of worker threads for a specific service provider on a specific computer, the provider can self-load balance the computer it is hosted on.

The SORCER platform is then used with an external optimization program to optimize an ESAV for range. The results from the optimization are shown in Fig 3-

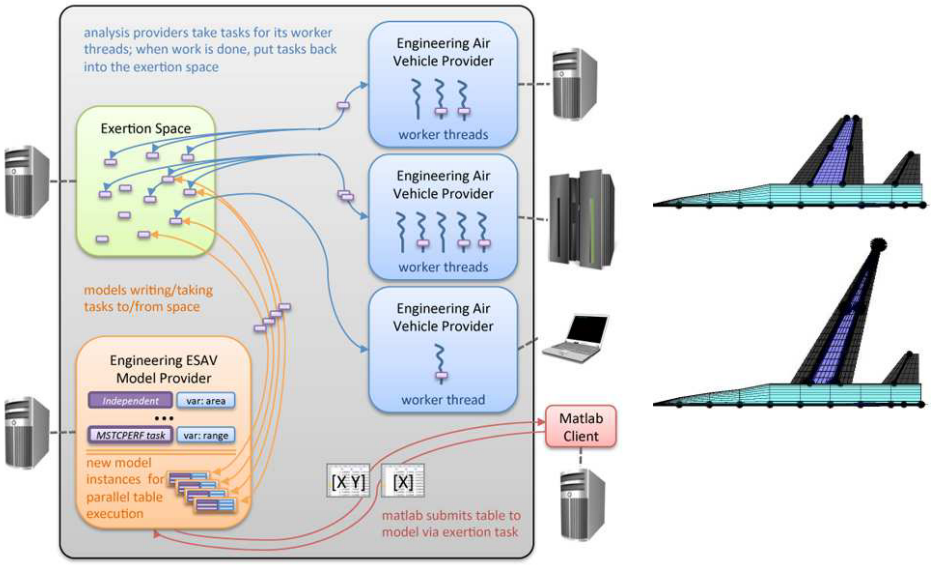

Figure 3. Left: SORCER uses Exertion Space to provide a flexible, dynamic space computing facility for ESAV optimization studies. Right: the ESAV optimization result half-span planforms: baseline 1550 mi range (top); optimized 2500 mi range (bottom).

right. The optimized design has a higher aspect ratio than the baseline design. This feature is consistent with historical aircraft design trends for long-range aircraft. The results provide a degree of validation of the implementation of the optimization code, the SORCER ESAV parametric model, the SORCER providers, and the SOS.

The use of the space computing proved reliable and efficient. It was a straightforward process to add computers to the SORCER service cloud as needed during the course of the two-optimization studies. This flexibility proved valuable as the number of computers available varied from day-to-day.

5. Conclusions

As we move from computing of the *information era* to advanced computing of the *service era*, it is becoming evident that new SO mogramming languages are required. By higher-level abstractions, these languages reduce the complexity of transdisciplinary designs performed by hundreds of people working together and using thousands of services (programs) written already in legacy languages that are dislocated in the global network. Domain specific SO languages are for humans, unlike software languages for computers, intended to express domain specific complex processes and related solutions. Three programming languages for SO computing are described in this paper: VOL, VML, and EOL. The network shell (nsh) interprets netlets in these languages and the SOS manages corresponding service federations.

The concept of the var fidelities in the EGS framework combined with exertions provides the uniform modeling technique for SO interoperability and integration 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 the ESAV parametric model. On one hand, EOP is uniformly converged with VOM to express a front-end SO procedural federation. On the other hand, VOM is uniformly converged with EOP to express a front-end declarative SO modeling. Both front-end exertions and var-models can be used as service providers directly. The evolving SORCER platform (open source project [7]) with its SO computational model has been successfully verified and validated in ESAV and other concurrent engineering distributed applications [1][9][11][12] [13].

6. 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 and by SMT Software S.A., the contract number POIG.01.04.00-14-062/12 Engineering Toolkit.

References

[1] Burton, S.A., Alyanak, E.J., and Kolonay, R.M. (2012). Efficient Supersonic Air Vehicle Analysis and Optimization Implementation using SORCER, 12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSM AIAA 2012-5520

[2] Jini Network Technology Specifications v2.1. Available at: http://www.jiniworld.com/doc/ spec-index.html. Accessed 5 June 2013

[3] Sobolewski, M. and Kolonay, R., 2012. Unified Mogramming with Var-Oriented Modeling and Exertion-Oriented Programming Languages, Int. J. Communications, Network and System Sciences, 2012, 5, 9. Published online http://www.scirp.org/journal/PaperInformation.aspx?paperID=22393

[4] Sobolewski, M., 2012, Object-Oriented Service Clouds for Transdisciplinary Computing, in I. Ivanov et al. (eds.), *Cloud Computing and Services Science*, DOI 10.1007/978-1-4614-2326-3_1, Springer Science + Business Media New York 2012

[5] Sobolewski, M., 2010. Object-Oriented Metacomputing with Exertions, *Handbook On Business Information Systems*, A. Gunasekaran, M. Sandhu (Eds.), World Scientific Publishing Co. Pte. Ltd, ISBN: 978–981–283–605–2

[6] SORCERsoft.org. Available at: http://sorcersoft.org. Accessed 5 June 2013

[7] SORCER Project. Available at: http://sorcersoft.github.io. Accessed 5 June 2013

[8] Kleppe A., 2009. *Software Language Engineering*, Pearson Education, ISBN: 978–0–321– 55345–4

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

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

[11] Goel, S.; Talya, S. S. and Sobolewski, M., 2008. Mapping Engineering Design Processes onto a Service-Grid: Turbine Design Optimization, *International Journal of Concurrent Engineering: Research & Applications*, Concurrent Engineering, Vol.16, pp 139–147

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

[13] Xu, W., Cha, J., Sobolewski, M., 2008. A Service-Oriented Collaborative Design Platform for Concurrent Engineering, Advanced Materials Research, Vols. 44–46 (2008) pp. 717–7224

[14] Freeman, E., Hupfer, S., and Arnold, K., 1999. *JavaSpaces Principles, Patterns, and Practice*, Addison Wesley Longman, Inc.