Service-oriented Mogramming with SML and SORCER

Michael Sobolewski
Air Force Research Laboratory, WPAFB, Ohio 45433 Polish Japanese Academy of IT, 02-008 Warsaw, Poland

Keywords: Service Orientation, Service Consumers, Service Providers, Multifidelities, Multityping, Service Mogramming Language (SML), Emergent Systems, SORCER.

Abstract: Service-oriented Mogramming Language (SML) is designed for service-orientation as UML was considered for object-orientation. SML is an executable language in the SORCER platform based on service abstraction (everything is a service) and three pillars of service-orientation: context awareness (contexting), multifidelity, and multityping. Context awareness is related to parametric polymorphism, multifidelity is related to ad hoc polymorphism, and multityping is a form of net-centric type polymorphism. SML allows for defining polymorphic service systems that can reconfigure and morph service federations at runtime. In this paper the basic concepts of SML are presented with three ted design patterns of service federations. Its runtime environment is introduced with the focus on the presented service abstractions.

1 INTRODUCTION

Service-oriented architecture (SOA) emerged as an approach to combat complexity and challenges of large monolithic applications by offering collaborations of replaceable functionalities by remote/local component services with one another at runtime, as long as the semantics of the component service is the same. However, despite many efforts, there is a lack of good consensus on semantics of a service and how to do true SOA well. The true SOA architecture should provide the clear answer to the question: How a *service consumer* can consume or compose some functionality from *provider services*, while it doesn't know where *service providers*, implementing that functionality, are or even how to communicate with them.

Many people think they are doing or talking about SOA, but most of the time they're really doing point-to-point integration projects with APIs, web services, or even just point-to-point XML (REST). The reason why this approach is deficient is because service consumers should never communicate directly to service providers. First, the main concept of SOA is that we want to deal with frequent and unpredictable change by constructing an architecture that loosely-couples the providers of capability from the consumers of capability. It is not possible to have direct reliable communication if continuous variability exists in the network and provided service capabilities evolve over time. Second, if we are relying on a black-box middleware and often-proprietary technology to manage service communication differences we will simply shift all the complexity to end-points of services and increasingly more complex, expensive, and brittle middle point. Reworked middleware, what often is done and named as SOA, isn't the solution for a dynamic net-centric service communication.

There are several trends that are forcing system architectures to evolve due to complexity of problems being solved presently (Sobolewski, 2015). Users expect a rich, interactive and dynamic experience on a wide variety of friendly user agents and highly modular and dynamic backend systems. Systems must be highly scalable, highly available and run locally or remotely, or both. Organizations often want to frequently roll out updates, even multiple times a day. Consequently, it's no longer adequate to develop simple, monolithic applications. In a dynamic system when its backend is morphing constantly to emergent solution (Aziz-Alaoui and Bertelle, 2006), the user agent has to support emergent nature of its backend. Emergent system means *net-centric* to refer to participating in distributed problem solving as a part of a continuously evolving complex community of people, devices, information and services interconnected by a communication network to achieve optimal benefit of resources and better synchronization of flowback events and their consequences to the users. Emergent system means

Sobolewski, M.
Service-oriented Mogramming with SML and SORCER.
DOI: 10.5220/0007717903310338
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 331-338
ISBN: 978-989-758-365-0

also *service-oriented* (SO) and *scalable* with multiple computational fidelities of services so your communication network can be scaled up and down dynamically, from a single computer to a large number of computers by adjusting fidelities of collaborating service (Sobolewski, 2017).

Computer-aided engineering is the broad usage of heterogeneous computer software for both standalone and distributed systems to aid in engineering complex analyses and optimization tasks. Multidisciplinary Analysis and Design Optimization (MADO) is a domain of research that studies the application of numerical analysis and optimization techniques for the design of dynamic systems of systems involving multiple coupled disciplines. The formulation of MADO problems has become increasingly complex as the number of disciplines and design variables included in typical studies has grown from a few dozen to thousands when applying high-fidelity physics-based modeling early in the design process (Kolonay, 2014). Therefore, the complex MADO domains have been used for studying the presented true service-orientation. First in the FIPER project funded by NIST ($21.5 million) at the beginning of this millennium (Sobolewski, 2002) then continued at the SORCER/TTU Laboratory (SORCER/TTU Projects, n.d.; Sobolewski, 2010), and maturing for real world aerospace applications at the Multidisciplinary Science and Technology Center, AFRL/WPAFB (Burton, Alyanak and Kolonay, 2012; Kolonay, 2014; Sobolewski, 2014, 2017).

The remainder of this paper is organized as follows: Section 2 describes SML service semantics; Section 3 describes the basic syntax and semantics of SML; Section 4 relates SML to the OO implementation in SORCER; then we conclude with final remarks and comments.

2 SERVICE SEMANTICS

Service semantics can be either declarative, imperative, or OO. A blend of all programming paradigms should be supported by SO languages intended for solving complex problems and building heterogeneous SO systems. Therefore, component services should be expressed using adequate programming styles. Each programming paradigm introduces distinguishing principles of its programming model but also depends on its lower level-supporting paradigm. Therefore, the pillars of SO programming introduced in this paper are layered on pillars of OO, structured, and functional programming. The pillars of true SO programming are focused on contexting, multifidelity, and multityping for frontend and backend services.

A service consumer is a composition of frontend request services and a service provider is an implementation of service types (interface types) using subroutines as shown in Fig. 1. A consumer is expressed in a SO language but a provider is actualized as the OO remote/local counterpart implementing multiple service types. Frontend services are references to backend services. Provider services are service specifications – contracts but service providers are implementations of contracts. A federated request service, called a *federal mogram* (Sobolewski, 2017), corresponds to a union of service mograms such that each component mogram (exertion or model) represents governance for own *collaboration of service providers*. A federated collection of cooperating collaborations defined by a federal mogram is called a *service federation*. Service federalism is a system of federal service governance in which constituent governances (component mograms) share governing power with the central governance (parent mogram) to utilize *federated collaborations* of service providers and subroutines as a service federation. The rules of federal governance are realized by a *SO operating system* (a kind of federal government). The main purpose of the SO operating system is to satisfy interests of service consumers and to fulfill their needs using capabilities of service federations.

Mograms are structured from *elementary request services* (entries and tasks) and other mograms. Entries and tasks depend on operation services called *evaluators* and *signatures*, respectively. Entries use various types of *subroutine services*, called evaluators, to invoke subroutines. A signature is a *reference* to a remote/local operation of service provider. The unique signature-based architecture is about both request service configuration complexity and execution complexity that allows treating local and remote service providers implementing subroutines uniformly at various levels of granularity and fidelity. When dealing with both complexities, you have a case to distribute services, otherwise create a modular monolith with locally executable modules as local services. Later, when complexity of the system becomes unmanageable you can deploy almost instantly the existing local service providers as network services on as-needed basis, and then run changed services of the original monolith in the network. In SORCER that is done by changing the service type of signatures from class to interface, or just selecting the remote service fidelity. Service providers never communicate directly with each other

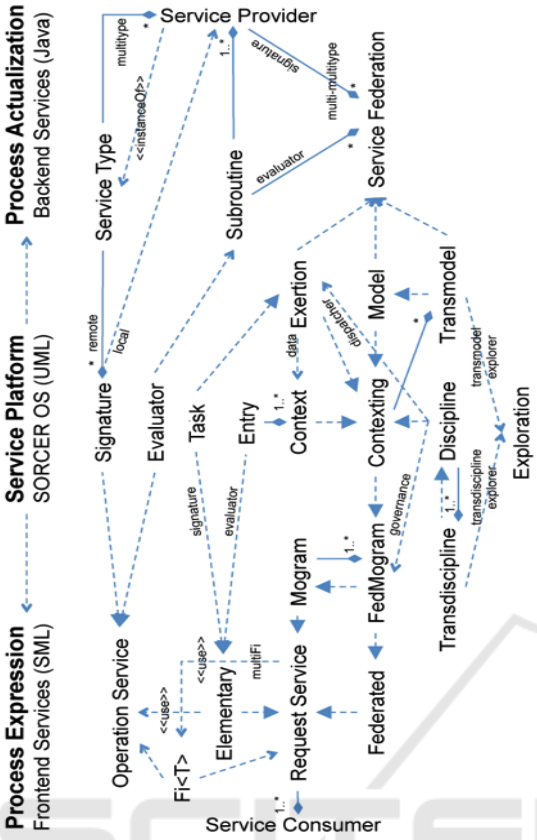

Figure 1: The service semantics in SML.

in SORCER. For executing mograms its operating system creates communication networks of service federations at runtime, as its dynamic net-centric processor.

A federated request service – *federal mogram* – is an expression of a service federation by one of the three service *governance patterns*:

1. *Entry Model* – is a declarative expression of interrelated multiple higher-order entries (responses) composed functionally of dependent service entries in the model.
2. *Exertion Block* – is an expression of concatenated mograms with branching and looping exertions as block-structured contexting.
3. *Exertion Job* – is a hieratically organized workflow of mograms with a control strategy for each federal service activity to be executed sequentially or in parallel, synchronously or asynchronously with context pipes between service activities.

The presented service abstractions reduce representational complexity at each layer, so it makes easer to comprehend federalism of service-orientation at all three layers: service mogram (governance), SO operating system (federal government), and actualization of federal mogram (service federation). Therefore, the presented service abstractions expose the details which really matter from the user perspective (frontend services) and hide the other details regarding development and deployment of backend services (service types, service providers, and subroutines) implemented with lower level programming abstractions.

The three service-oriented pillars can be summarized as follows:

Contexting allows for a mogram to be specified generically, so it can handle encapsulated data uniformly with subtypes of contexts and data types of entries to be specified by service providers. Contexting as the form of parametric polymorphism is a way to make a language more expressive with one primary type for inputs and outputs.

Morphing a service federation for a given mogram is affected by the initial fidelities selected by the user, input service contexts, and subsequent intermediate results obtained from service providers. Morphers associated with the mogram update fidelities using heuristics provided by the end user, usually closures dependent on the current mogram context. *Multifidelity management* is a dispatch mechanism, a kind of ad hoc polymorphism, in which component fidelities of the mogram are selectable at runtime.

Service multityping as applied to service providers is a form of subtype polymorphism with the goal to find a remote instance of the service provider by the range of service types that a service provider may implement and register for lookup. It also allows a request service to call on any implemented service type. With respect to a service federation to be provisioned, multi-multityping specifies which service providers have to be actualized to complement existing service providers in the network.

3 INTRODUCTION TO SML

The presented approach to service-orientation is based on two abstract service categories (see Fig. 1): frontend services (operation services and request services) and backend services (subroutines, providers, and service federations) with three pillars of service-orientation: *contexting*, *multifidelity*, and *multityping* that all together constitute the Meta-Service Facility (MSF) by analogy to MOF used to specify UML (The Meta-Object Facility Specification, n.d.). Therefore MSF for SML, like MOF for UML, is a metamodel that specifies how the SML model should conform to the MSF service

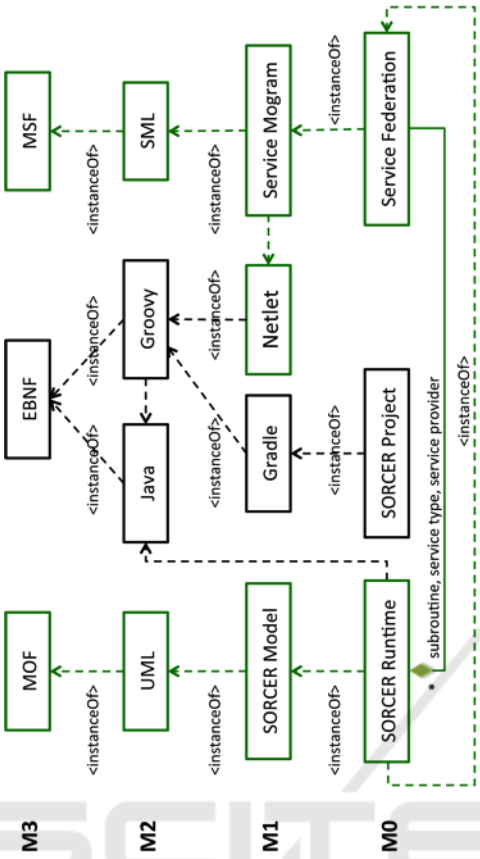

Figure 2: The metamodel hierarchy of MOF/MSF.

semantics. The SML metamodeling hierarchy along with the UML metamodeling hierarchy is depicted in Fig. 2 to explain the relationship of SML (M2) to the object-oriented SORCER runtime (M0).

A service model *SM* in SML conceptually corresponds to a multifidelity functional system. Multifidelity from the computing perspective refers to a computing environment with multiple fidelity levels for a given computing process, meaning there are different implementations of computing process to choose from. Fidelity and cost (or similarly accuracy and time) are positively correlated; this represents a fundamental trade in design.

A multifidelity function $f = (X, Y, fi(f), mFi_f)$ (see definition in Section 3) is declared in SM as follows*:*
func f = *ent*(*"f", mFi*$_f$, *args*(*"f*$_1$*", "f*$_2$*", ..., "f*$_k$*"*))
where *"f"* is a name of the function *f* declared by the operator *ent*; *"f*$_1$*", "f*$_2$*", ..., "f*$_k$*"* are argument identifiers of *f* , and mFi_f is the multifidelity of *f*. By default a fidelity of *f*, *fi*(*f*), is the first realization in the ordered set mFi_f . The identifiers *"f*$_1$*", "f*$_2$*", ... , "f*$_k$*"* refer to other functional entries in *SM*. The entry *f* binds the free identifiers *"f*$_1$*", "f*$_2$*", ..., "f*$_k$" to the corresponding entries in *SM*.

The *ent* operator defines a generic functional expression declared in a service model *SM*. A *service model* is a collection of functional entries that form higher-order functional compositions – *responses* of the model. If *ent* declares a constant function then a model with all such entries is called a *data context*. In SML an entry is a higher-order function .

A *service signature* in SML is an *operation service* referencing an operation of a service provider. It declares a type of provider *tp* and its operation *op*, to be invoked in the scope of a current service context. An association <*op, tp*> is called a *service signature* and is denoted in SML by *sig*(*op, tp*). A return value of an operation *op* executed by a service provider implementing a type *tp* is declared as follows:
func f = *ent*("*f*", *sig*(*op, tp*))
or a multifidelity service entry
func f = *ent*("*f*", *entFi*(*sig*(*op*$_1$, *tp*$_1$), …, *sig*(*op*$_n$, *tp*$_n$))
where the operator *entFi* declares a multifidelity of entry *f*.

A service provider may implement multiple service types used to classify its instances in the network by its *multitype*. In that case a service provider multitype, as a list of implemented service types $tp_1, \ldots, tp_s$, specified in a signature is the service provider's net-centric identity. Optionally a service provider name with additional attributes can be used as well. Thus, a signature *s* with a multitype ($tp_1, tp_2, \ldots, tp_s$), an operation op_1 of type tp_1, and service name *myService* takes the following expanded form:

sig s = *sig*(*op*$_1$, *tp*$_1$, *tp*$_2$, …, *tp*$_s$,
srvName("*myService*"))

Note that a signature *s* does not refer to a particular instance of a service provider; its multitype is used for binding to an available instance at runtime. Multityping is used to manage complexity and unpredictability of the network comprised of replaceable remote service providers with one another at runtime, as long as the multitype semantics of the service providers is the same.
The network-centric semantics of SML is based on the concept of multitype subtyping. If the type tp_1 is of class type then the signature works as a service provider constructor – creates an instance at runtime when the service provider needs to be executed, otherwise SOS finds in the network a remote proxy of the service provider implementing the required multitype.

A *value entry x* (constant function) in *SM* is declared by a value entry (variable) *x* as follows:

val x = *val*("*x*", *y*), $y \in Y$
or a multifidelity value entry
val x = *val*("*x*", *entFi*(*val*("*x*$_1$", *y*$_1$), …,

val("x_k", y_k))).

A *data context dc* (of *cxt type*) is an unordered collection of *val* entries defined as follows:

cxt dc = context(*val*(…), …, *val*(…))

and valuation of the entry *x* in *dc* as follows:

Object y = *value*(*dc*, "*x*")

where "*x*" is a name of variable in a data context *dc*.

A value of an entry *x* in *cxt* can be set to *v* as follows:

setValue(*dc*, "*x*", *v*)

A mogram *mdl* (of *mog type*) as an unordered collection of value entries *val* and multivariable functional entries *ent* is called a *context model* and is declared as follows:

mog mdl = model(*val*(…), …, *ent*(…), …)

Note that multivariable functional entries of a model may take other functional entries as arguments to create higher-order functions.

Evaluation of an entry *f* in a model *mdl* is declared as follows:

Object y = exec(mdl, "f")

or

Object y = exec(mdl, "f", c_{in}*)*

where $y \in Y$ is an output value and c_{in} is a context used for substitution of value entries in *mdl.*

Evaluation of a model *mdl* for its responses is declared as follows:

cxt c_{out} *= eval(mdl)*

or

cxt c_{out} *= eval(mdl,* c_{in}*)*

where c_{out} is a data context - the result of evaluation of response entries for an input data context c_{in}. Model evaluations are defined by functional compositions of response entries with no explicit strategy for altering the functional compositions of the model. However, execution dependencies can be specified for entries that require other entries to be executed beforehand at runtime.

Responses of a model (names of response entries) can be part of the model declaration by inlining responses *"*f_1*", "*f_2*", … , "*f_k*"* as follows:

response(*"*f_1*", "*f_2*", …, "*f_k*"*)

Alternatively, responses can be updated as required. To increase responses:

responseUp(*mdl, "*f_1*", "*f_2*",…, "*f_k*"*)

and *t*o decrease responses:

responseDown(*mdl, "*f_1*", "*f_2*",…, "*f_k*"*)

When names of entries are absent then *responseDown* removes all responses and *responseUp* makes given output entries as additional responses of the model.

So far, we have defined in SML, operational services of *sig* type, elementary services of *ent* and *val* types, and request services of *context* and *model* types. The following statement executes any service *sr:*

Object out = exec(*sr, arg*$_1$*,…, arg*$_n$)

where arg_i is an SML argument of the *Arg* type. For example, signatures, contexts, fidelities, and mograms are of *Arg* type.

The statement executing the operation *add* of service type *Adder* takes the form:

exec(*sig*(*"add", Adder.class*), *context*(

val("*x1", 3.0*), *val*("*x2", 1.0*), *val*("*x3*", *7.0*))

and returns *11.0* by an instance of a service provider found in the network that implements the interface *Adder.* Here, the signature *sig*(*"add", Adder.class*) binds to an instance of service provider - remote object - implementing the service type *Adder.* If the class *AdderImpl* implements the interface *Adder* then the execution:

exec(*sig*(*"add", AdderImpl.class*), *context*(

val(*"x1", 3.0*), *val*(*"x2", 1.0*), *val*(*"x3", 7.0*))

creates an instance of *AdderImpl* at runtime and calls the method *add* with a given context on the locally created instance.

A *service task* is an elementary request service defined by a signature with a data context as follows:

mog y = task("y", sig(op, tp), context(…))

where "*y"* is a name of the task *y* with a given signature and data context.

A multifidelity task is declared in SML as follows:

task("*y*", *sigFi*(*sig*(*op, tp*),…), *context*(…))

where the operator *sigFi* declares multifidelity of task *y* with the first signature as a default fidelity. A selected fidelity can be preselected or declared as an argument when executing a task or set by the fidelity manager of its containing mogram at runtime.

At its heart, *service-orientation* is the act of uniform decomposition into self-contained local and/or remote subroutines implementations interconnected and replaceable at runtime. In SML interconnections of entries and service tasks (see Fig. 1) are declared by a mogram that binds multifidelity signatures to remote/local subroutines of service providers/evaluators at runtime.

In SML an *exertion* is a request for a procedural/workflow service type (Sobolewski, 2010). A service task is an elementary service exertion used in composite exertions. A composite exertion is a collection of exertions and/or mograms grouped together within the scope of either *block* or *job* SML operators. An exertion *block* (service procedure) is a concatenation of component mograms along with flow-control exertions: conditional (*opt, alt*) and loop (*loop*) task. The SML semantics of *opt, alt*, and *loop* is the same as the UML operators used with interaction frames (combined fragments) in sequence diagrams. An exertion *job* (service workflow) is an object composite of component

exertions and/or mograms, optionally with an explicit control strategy and service pipes for interprocess communication between components of the workflow.

Exertions can be used as functionalities of entries in models and evaluated models can be used as data contexts in exertions. That way, either an exertion blended with models, or a model blended with exertions creates a service aggregation of *models* and/or *exertions* – a *service mogram* (model and/or program). The SML *ent* operator, in most obvious cases, declares a service entry of the corresponding subtype according to given arguments to *ent*. However, specialized SML entry operators: *val, call, lambda, neu, srv, and svr* that correspond to entry subtypes: *value, call unit, lambda, neuron, service, and service*; can be used as well along with new introduced subtypes.

A mogram m_{in} to be executed – *exerting* its corresponding service federation is declared as follows:

mog m_{out} = *exert*(m_{in})

An exerted mogram m_{out} contains the result of execution and all net-centric information regarding the execution. The *result* operator returns the output context of the exerted mogram mog_{out} as follows:

cxt c_{out} = *result*(m_{out})

The value *y* of variable *x* in c_{out} is specified by the operator *value* as follows:

Object y = value(c_{out} , "*x*")

or from the mogram directly:

Object y = exec(mog_{out}, "*x*")

An evaluation result c_{out} of a mogram m_{in} is a data context declared as follows:

cxt c_{out} = *eval*(m_{in})

Note, that the *eval* operator returns an output context c_{out} but the *exert* operator an executed mogram m_{out}.

A *federal mogram* is a collection of interacting request services (*entries, tasks, models, and exertions*) that bind at runtime to a federation of subroutine collaborations via mogram signatures and evaluators. Multifidelity federations can morph during execution under control of the mogram morphers and theirs fidelity managers with the goal to return the best result of the evolving net-centric configuration - a morphing system (mogram) of systems (fidelity projections of mogram).

To illustrate SML in action we refer the reader to the examples, in the core of the open source SORCER project, its *multiFi* branch, as described at: http://sorcersoft.org/project/site/
in the module *examples*, in particular a multifidelity test case: *sml/src/test/main/java/mograms/ModelMultiFidelities*

4 THE SORCER PLATFORM

The relationship of the main SORCER types required to implement multifidelity services is depicted in the diagram in Fig. 1. Services of the *Request* type are instances of two elementary subtypes: *Entry* and *Task*, and the federated request type *Mogram*. All frontend entities are instances of the common *Service* type with uniform execution of local and remote services at runtime. Top-level types of the SORCER system refer to the architectural OO view of key SO concepts (*Fi<T>*, *Signature, Evaluator, Request, Entry, Task, Mogram*, and *Provider*) all of the common *Service* type.

In general, a mogram is an expression of collaboration of remote and/or local subroutines. A service model is a declarative representation of interrelated functional entries but a service exertion is an imperative aggregation of component services. Both entries and tasks bind to subroutines via service evaluators and signatures, correspondingly. Therefore a federal mogram is a request service for a *federation* of provider and subroutine collaborations managed by SOS. Signatures by using service multitypes provide for indirect referencing of local/remote service providers but evaluators are called directly. A service consumer runs an aggregation of request services that bind to the hierarchically organized service collaborations.

We distinguish three main categories of frontend services: operation, elementary, and federated services. From the SO point of view creation of user-centric request services – mogramming – is the primary objective assuming that service providers implement multitypes and can be incorporated into service federations as subroutines bound via operation services at runtime. Note, that multifidelities are used in request services only. A mogram is a frontend service that hierarchically aggregates elementary requests (entries and tasks) that bind dynamically to executable subroutines of evaluators and service providers, correspondingly.

Each service provider implements a multitype of service types. Each service type may have multiple implementations (provider services) in the network. We do not know location of service provider instances in the network; we require only their service types to be implemented. The question is, how to find a required implementation in the network. The answer is, by matching a multitype of the signature to the multitype of an implementation available in the network. To differentiae from each other, service providers may implement complementary service types, for example, tag interfaces corresponding to

implementation details. Complementary types can be registered with primary service types, then both used in signatures when looking up a service provider. Multityping of a signature is the concept of finding a provider of the same multitype from redundant instances available in the network.

Morph-fidelities are observables and observed by a fidelity manager. Therefore, the positive or negative feedback received from executing fidelities can be used to update fidelities, upstream of already executed services and downstream for new looked up services. The fidelity manager, as the observer of morph-fidelities, updates associated morphers to reconfigure a mogram's fidelity projection. Morphers associated with morphed fidelities form emergent properties in the morphing multifidelity system.

An emergent modeling platform requires the ability to express a service system with a given fidelity projection as the instance of the multifidelity metasystem with multiple fidelity projections. Also, the computing platform requires the ability to execute and morph the evolving system with updated projections managed by the metasystem. A multifidelity metasystem defined in SML enables quick and effective communication with other team members and allows for evolving updates such that each new instance of the system is a new multifidelity projection of the metasystem.

SML defines two types of multifidelities in mograms: select-fidelities and morph-fidelities. Select-fidelities allow for system reconfiguration but morph-fidelities allow for self-morphing the structure of the mogram. A system mogram, that defines the service federation actualized and managed by the SORCER operating system, is an instance of a metasystem – multifidelity mogram. To reconfigure and morph a mogram its fidelity manager uses projection functions and morphers. Both reconfiguration and morphing allow for adaptivity of system and system-of-systems correspondingly, when updates of fidelities and metafidelities are under control of the fidelity manager at runtime. Adaptive federated SO systems with morph-fidelities are SO emergent systems. This type of systems exhibits three types of adaptivities called system-of-system, system, and service agility (Sobolewski, 2017). Metasystem agility refers to system reinstantiation, system agility refers to updating system projections, and service agility refers to updating fidelities of elementary request services at runtime.

5 CONCLUSIONS

From experience in the past decades it becomes obvious that in computing science the common thread in all computing disciplines is process expression; that is not limited to algorithm or actualization of process expression by a single computer. In this paper, service-orientation is proposed as a class of distributed emergent processes with federated multifidelity and multityped services.

The "everything is a service" semantics is introduced with federated multifidelity services – mograms – as SO process expressions, to be actualized by dynamic federations of service collaborations in the network. A multifidelity mogram is considered as a dynamic representation of a net-centric emergent adaptive process defined by the end user. In SORCER, a rectified mogram, embedded into a service provider container, becomes a service provider – a frontend request becomes a backend provider.

To express emergent processes consistently and flexibly, the actualization of SML by the SORCER platform is based on three pillars of services orientation that incorporate pillars of functional, structured, and object-orient programming. Request services are multifidelity services but provider services are multitype services. By multitypes of signatures used in mograms a multi-multitype of service federation is determined. Therefore, multitype of a signature and multi-multitype of mograms are classifiers of instances of service providers and service federations in the network, correspondingly. To the best of our knowledge there is no comparable true service-oriented system and programming language based the three pillars of federated service-orientation as defined in this paper.

Emergent systems exhibit three types of adaptivities called system-of-systems (metasystem), system, and service agilities. Metasystem agility refers to updating metafidelities (system reinstantiation), system agility refers to updating fidelities of a mogram (system projection), and service agility refers to selecting fidelity of elementary request services (Sobolewski 2017).

The first rule of service-orientation: do not morph and do not distribute your system until you have an observable reason to do so. First develop the system with no fidelities and no remote services. Later introduce must-have distribution and multifidelities. Doing so step-by-step you will avoid the complexity of modeling with multifidelities and distribution all at the same time.

The SORCER architectural style represents a federal governance of net-centric multifidelity service consumers expressed by mograms created by the end users and service providers by software developers. It elevates governance of federated mograms into the first-class elements of the SO federated process expression. The essence of the approach is that by making specific SML choices, we can obtain desirable dynamic properties from the SO frontend system we create. The SORCER platform has been successfully deployed and tested for design space exploration, parametric, and optimization mogramming in multiple projects at the Multidisciplinary Science and Technology Center AFRL/WPAFB (Sobolewski, 2014, 2017).

ACKNOWLEDGEMENTS

This effort was sponsored by the Air Force Research Laboratory's Multidisciplinary Science and Technology Center (MSTC), under the Collaborative Research and Development for Innovative Aerospace Leadership (CRDInAL) - Thrust 2 prime contract (FA8650-16-C-2641) to the University of Dayton Research Institute (UDRI). This paper has been approved for public release, case number: 88ABW-2018-4737. The effort is also partially supported by the Polish-Japanese Academy of Information Technology.

REFERENCES

Aziz-Alaoui, M. and Cyrille Bertelle, C. (eds) (2006) Emergent Properties in Natural and Artificial Dynamical Systems (Understanding Complex Systems), ISBN-13: 978-3540348221, Springer

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

Kleppe A. (2009) Software Language Engineering, Pearson Education, ISBN: 978–0–321– 55345–4

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

Kolonay, R. M. (2014) A physics-based distributed collaborative design process for military aerospace vehicle development and technology assessment, International Journal on Agile Systems and Management, Vol. 7, Nos. 3/4

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

Sobolewski, M. (2010) Object-Oriented Meta- computing with Exertions. In A. Gunasekaran, and M. Sandhu (Eds.), Handbook on Busi- ness Information Systems. World Scientific. doi:10.1142/9789812836069_0035

Sobolewski, M. (2014) Service oriented computing platform: an architectural case study. In: Ramanathan R, Raja K (eds) Handbook of research on architectural trends in service-driven computing, IGI Global, Hershey, pp 220-255

Sobolewski, M. (2015) Technology Foundations. In: J. Stjepandić et al. (eds.) Concurrent Engineering in the 21st Century, ISBN 978-3-319-13775-9, Springer International Publishing Switzerland, pp 67-99

Sobolewski, M. (2017) Amorphous transdisciplinary service systems. Int. J. Agile Systems and Management, Vol. 10, No. 2, 2017, Int. J. Agile Systems and Management, Vol. 10, No. 2, 2017, pp. 93-114

SORCER/TTU Projects, (n.d.) Available at http://sorcersoft.org/theses/index.html (Accessed: January 3, 2019)

SORCER Project, (n.d.) Available at http://sorcersoft.org/project/site/ (Accessed: January 3, 2019)

The MetaObject Facility Specification (n.d.) Available at https://www.omg.org/mof/ (Accessed: January 3, 2019)