True Service-Oriented Metamodeling Architecture

Michael Sobolewski[1,2(✉)]

[1] Air Force Research Laboratory, WPAFB, Dayton, OH 45433, USA
sobol@sorcersoft.org
[2] Polish Japanese Academy of IT, 02-008 Warsaw, Poland

Abstract. True service-oriented metamodeling architecture provides a set of guidelines and the Service-oriented Mogramming Language (SML) for structuring and expressing of service specifications. SML is an executable language in the SORCER platform based on service abstraction (everything is a service) and three pillars of service-orientation: contextion (context awareness), multifidelity, and multityping. Contextion is related to parametric polymorphism, multifidelity to ad hoc polymorphism, and multityping is a form of net-centric type polymorphism. SML allows for defining complex polymorphic services that can express, reconfigure, and morph service-oriented processes at runtime. In this paper the metaprocess modeling architecture applicable to service-orientation is presented with five types of service-oriented processes. Its runtime environment is introduced with the focus on actualization of emergent service processes expressed in SML with the corresponding Service Virtual Machine (SVM).

Keywords: True service orientation · Contextion · Multifidelities · Multityping · Service Mogramming Language (SML) · Emergent systems · SORCER

1 Introduction

Service-oriented architecture (SOA) emerged as an approach to combat complexity and challenges of large monolithic applications by offering cooperations 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 and combine some functionality from service providers, while it doesn't know where those service providers are or even how to communicate with them?

In service-oriented *mogramming* - modeling or programming, or both - three types of services are distinguished: *operation services*, and two types of *request services*. An operation service, in short *opservice*, invokes a service provider operation. An *elementary request service* asks a service provider for output data given input data. A *combined request service* asks cooperation of service providers for output data and utilizes obtained from multiple service providers output data. A *service consumer* utilizes output data of aggregated request services. The end user that creates request services and utilizes the actualized service partnership becomes the coproducer and the consumer of created service cooperation in the network. Software developers develop service provider provisioned in the network and end users develop service partnerships.

A *compute service* is the work performed in which a service provider (one that serves) exerts acquired abilities to execute a computation. The *true compute service* needs the computation and *net-centric* service providers to be expressed and executed under condition that service consumers should never communicate directly to service providers. The combined request service realized by opservices represents the *dynamic partnership* of service providers in the network. In contrast the elementary request service represents the opservice of the selected provider in the network.

Many people think they are doing or talking about SOA, but most of the time they are 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 service abstractions and an architecture that loosely-couples the providers of capability from the consumers of capability. It is not possible to have direct reliable communication if variability exists in the network and provided service capabilities evolve over time. Second, if we are relying on black-box middleware and often proprietary technology to manage service communication differences, it simply shifts all the complexity and work from the endpoints to an increasingly more complex, expensive, and brittle middle point. Reworked middleware, what often is done and named as SOA, is not the solution for a dynamic, net-centric communication and architecture.

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 engineering systems-of-systems involving multiple coupled domains and multiple evolving disciplines. The formulation of MADO problems has become increasingly complex as the number of engineering 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 [6]. Therefore, MADO is an appropriate domain for studying real-world service-oriented architectures and systems.

This is achieved by the use of reference architectures and their ability to place information views on multidisciplinary data and integration of heterogeneous tools, applications, and utilities used frequently by distributed engineering teams. There are several trends that are forcing system architectures to evolve due to complexity of engineering problems being solved presently [14]. Users expect a rich, interactive and dynamic user experience on a wide variety of friendly user agents and highly modular and dynamic systems. Systems must be highly scalable, highly available and run locally or in the network, 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 with statically connected modules. When the dynamic system changes frequently the static user agent cannot catch-up with the backend changes, so it becomes obsolete in evolving and complex highly dynamic MADO systems. In a dynamic system when its backend is morphing constantly to emergent solution [1], the user agent has to support emergent nature of its backend. An emergent system means net-centric to refer to 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. An emergent system means also service oriented (SO) and scalable with multiple computational fidelities of underlying services so the communication network of services can be scaled up and down dynamically, from a single computer to a large number of computers with relevant computational fidelities [15, 16].

Top-down and bottom-up problem solving describes two different methods of reasoning: working at the top is considered strategic and declarative, while working at the bottom is tactical and imperative. How a given situation is actually perceived and processed will vary with the person, experience, process expression and actualization chosen. However, the approach is to do whatever is best for managing complexity of the solution by a combination of programming paradigms in designing processes. There is no universal programming paradigm that works well for all situations. Different paradigms are more applicable to different classes of problems and solutions. One should always carefully choose the right paradigm to match the particular subproblem, or component service at hand. Programming paradigms are not language-specific; therefore, basic paradigms should be available in a service-oriented language as well.

An algorithm is a process expression for solving a problem in the form of a self-contained step-by-step set of statements to be performed with an explicit control flow defined. Statements often refer to a subroutine as a sequence of instructions designed to perform a frequently used task within an algorithm. The emphasis on an explicit control flow distinguishes an imperative programming language [9] from a declarative programming language.

In declarative programming a process is expressed by the logic of computation without describing its control flow. In particular, the logic of computation in functional programming is defined by a function composition. The result of execution of a function composition depends only on inputs and the function composition. There is no a shared state that the execution of function composition depends on. A functional program is stateless but imperative programs usually take advantages of a shared state in an executing algorithm.

Object-oriented programming is a convenience and ability to reason about implemented object operations as subroutines, called methods, with a shared state represented by instance and class variables encapsulated in objects. Being able to hide details of subroutines and their data structures can help reason about the logic of object cooperation such that each object in cooperation manages its own state by own implementation of its subroutines (methods).

Service semantics can be declarative, imperative, or object-oriented depending how multi-machine subroutines, corresponding to executable codes, can be combined into service providers in the network. Therefore, a blend of declarative, imperative, and object-oriented programming should be supported by SO programming languages intended for solving complex problems and building heterogeneous distributed SO systems.

Each programming paradigm introduces distinguishing principles of its programming model but also depends on its lower level paradigm. The pillars of SO programming introduced in this paper are layered on pillars of object-orientated, procedural, and functional programming as illustrated in Fig. 1. The pillars of SO programming are focused on context awareness of services, management of service multifidelities, and multitype management of services for registering, looking up, and referencing both a single service provider and cooperation of service providers. Each paradigm abstraction based on: functions, procedures, objects, and services is the foundation of corresponding pillars. The ceilings: FP, PP, OOP, and SOM correspond to functional, procedural, object-oriented programming, and service-oriented mogramming, respectively. SO mogramming is not a replacement for any programming paradigm, it just inherits programming styles from the layers below and complements them with higher-level service abstractions.

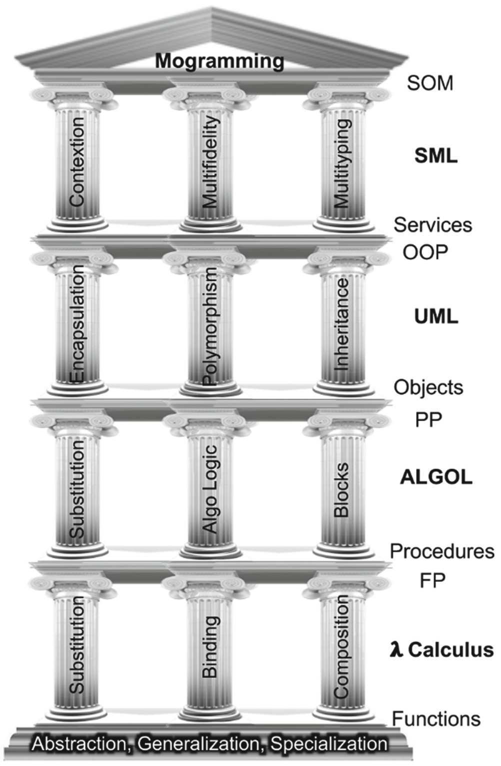

Fig. 1. The service mogramming gate.

Mogramming [4] that combines multiple programming paradigms uniformly [12]. A *service mogram* is an expression of cooperation of routines and models as component services that in turn comprise of operation services, all represented in the Service Mogramming Language (SML). Mogram exhibit hierarchically organized net-centric executable codes represented by its operation services, a of the *net-centric service processor*.

The Service-ORiented Computing EnviRonment (SORCER) [14, 19]) adheres to the true SO architecture based on formalized service abstractions and the three pillars of SO programming presented in Sect. 2. Evolution of the presented approach started with the FIPER project [10] funded by NIST ($21.5 million) at the beginning of this millennium then continued at the SORCER/TTU Laboratory [19], and maturing for real world aerospace applications at the Multidisciplinary Science and Technology Center, AFRL/WPAFB [2, 4–7, 13].

The remainder of this paper is organized as follows: Sect. 2 relates to a service-oriented conceptual framework called Meta-Service Facility (MSF); Sect. 3 describes service semantics in SORCER; Sect. 4 gives introduction to SML; Sect. 5 illustrates introduced concepts of SML with an example of multifidelity model; Sect. 6 describes briefly the object-oriented platform of SORCER; then we conclude with the final remarks and comments.

2 Meta-Service Facility (MSF)

The Meta-Object Facility (MOF) is the Object Management Group (OMG) standard for object-oriented model-driven engineering [18]. Its purpose is to provide a type system for entities in the CORBA (Common Object Request Broker Architecture) architecture and a set of interfaces through which those types can be created and manipulated. Similarly, the Meta-Service Facility (MSF) is a reference service-oriented methodology that focuses on creating and exploiting service models, which are conceptual models of all the topics related to specific structures of request services in SML. Hence, it highlights and aims at abstract representations of the knowledge and activities that govern a particular domain service, rather than the computing concepts in that domain. Its purpose is to provide a type system and semantics for entities in the SORCER (Service-Oriented Environment) architecture. MSF is a metamodel defined by the Multifidelity Service System (MSS, defined in Sect. 2.1, 12) that specifies how the SML model should conform to the conceptual MSF framework.

A *computing service* is the work performed in which a service provider (one that serves) exerts acquired abilities to execute a computation. A *service provider* corresponds to actualization of a *request service*. A single service provider actualizes *elementary request service*, but a *combined request service* is actualized by a cooperation of service providers. Therefore, a *request service* may represent a process expression realized by cooperation of service providers. In this Section, we assume that a mathematical function represents a request service to be actualized by a corresponding actualization, executable code or a combination of executable codes. An elementary request service is actualized by an executable code, but a combined request service (function composition) is actualized by a combination of executable codes.

2.1 Multifidelity Function Systems

A function is a prescription that assigns to every entity of one set X an entity of another (or the same) set Y what is declared by stating its domain X and codomain Y as follows:

$$f : X \to Y \tag{1}$$

such that exists a relation $R \subseteq X \times Y$ and each pair $<x, f(x)> \in R$. A relation R is called a realization of a function f. So, a function f is like a process $f = (X, Y, R)$. Each input x that is in the set X of inputs is paired with one output y in the set Y of outputs:

$$y = f(x) \tag{2}$$

A function of two or more variables is considered to have a domain consisting of ordered pairs or tuples of argument values. The arity of a relation R is the dimension of the domain in the corresponding Cartesian product. A function of arity n thus has arity $n+1$ considered as a relation.

A set F of interrelated multivariable functions is called a functional system FS

$$FS = <X, Y, F> \tag{3}$$

with domain X and codomain Y, such that for each function $f_i \in F$, there exist a realization $R_i \subseteq X^n \times \mathrm{Y}$ where X^n is the Cartesian power of a set X.

A multifidelity function f

$$f = <X, Y, R_f, mFi_f> \tag{4}$$

is a mapping with multiple realizations $mFi_f = \{R_i\}, i = 1, 2, \ldots, m$ with a selected realization $R_f \in mFi_f$. A selected realization R_f is said to be a fidelity of function f.

Let's denote a fidelity of a function f as $fi(f)$, then each input tuple x of X is paired with one output tuple y of the set Y according to its fidelity $fi(f)$ where n is the arity of function f, provided $(x, f(x)) \in fi(f)$

$$y = f(x, fi(f)) \tag{5}$$

A *multifidelity function* f is a dynamic process $f = (X, Y, fi(f), mFi_f)$, with a substitutable fidelity $fi(f) \in mFi_f$.

A fidelity substitution fp, called a projection in FS, is a mapping:

$$fp : F \rightarrow FR \tag{6}$$

where F is a set of multifidelity functions, FR is a set of all realizations of functions F in FS, such that for each function $f \in F, fp(f) \in mFi_f$ and $fp(f) = fi(f))$, for $mFi_f \subseteq FR \subseteq P(X \times Y)$.

A *fidelity morpher fm* is a mapping that defines fidelities of functions in association with the inputs and outputs of functions, as follows:

$$fm : F \times X \times Y \rightarrow FR \tag{7}$$

where for each $f \in F$, and $x \in X$ and $y \in Y$, $fm(f, x, f(x)) \in mFi_f$ and $mFi_f \subseteq FR$.

A multifidelity function system is a triplet:

$$MFS = <F, FP, FM> \tag{8}$$

where F is a set of interrelated functions with a related set of fidelity projections FP and a set a fidelity morphers FM. Realizations of functions F under fidelity management defined by FP and FM are called functional multifidelities. Note that a single projection $fp \in FP$ defines a realization of multifidelity functions F in MFS while a set of multiple fidelity projections for the same F is a metasystem – a system of projected systems. A set FP of projections allows for a reconfiguration of MFS. A set of morphers FM defines self-morphing of MFS based on runtime inputs and outputs interpreted by morphers applying accordingly fidelity projections FP.

Multifidelity functions are polymorphic functions. Multifidelity is a kind of ad hoc polymorphism in which a polymorphic function can denote a number of distinct and potentially heterogeneous realization (implementations) depending on the type of arguments to which it is applied. The term ad hoc in this context refers to the fact that this type of polymorphism is not a fundamental feature of the type system.

A *total fidelity* R_0 of a multifidelity function f is a fidelity such that $R_0 \in mFi_f$ and for each $R_i \in mFi_f, R_i \cap R_0 \subset R_0$. If $R_i \cap R_0 \subset R_j \cap R_0$ for $R_i \in mFi_f$ and $R_j \in mFi_f$, then R_i is said to be lower fidelity than R_j or R_j is higher fidelity than R_i. A total fidelity of function f can be considered as a realization of a total function f and any lower fidelity as a realization of a partial function of function f. Lower fidelities of a function $f \in F$ are often used when the exact domains of its realizations in mFi_f, are not known or they are proper subsets of the domain R_0.

Note that a fidelity of a multifidelity function is not related directly to a fuzzy concept (like in fuzzy sets or rough sets) of which the membership boundaries of a set $R_i \in mFi_f$ can vary considerably according to current context or conditions. Here boundaries of all sets R_i are fixed once and for all fidelities. It means we are not concerned with a vague or imprecise definition of function; we might have a precise (analytic) realization as well. We can consider lower fidelities as good approximations of R_0 under some conditions. In such situation, a lower fidelity can be more beneficial, "cheaper" to compute than a higher fidelity, while a higher fidelity is also available but considered not preferred all the time due to, for example, time-consuming realization.

Let's specialize a multifidelity process f in MFS defined by a realization R_f (4) as a multifidelity functionality or subroutine $fi(f) = R_f$ (function, procedure, method, service). Let's also specialize both a domain and a codomain to a set C of all tuples of elements from sets X and Y. A tuple in C is called a *tuple context* of the set C of all tuple contexts. If $<x, f(x)> \in R, x \in X, f(x) \in Y$ and $X \subseteq C$ and $Y \subseteq C$ then the function f such that:

$$f : C \to C \tag{9}$$

is called *context aware* or *contextion function* (in short, *contextion*). Later we consider all request services as contextions, unless otherwise stated.

Depending on the programming semantics of function $f \in F$, defined by a programming subroutine, the multifidelity function system MFS with the applied pillars of programming (see Fig. 1) can be specialized as functional, procedural, object-oriented, service-oriented, or a programming system with any combination of programming paradigms needed. Despite many efforts, there is a lack of good consensus on what is the proper semantic of true service and how to do true SOA well. In the following Subsection a conceptual multifidelity service-oriented system is proposed with multifidelities and context awareness as two pillars of service orientation. All pillars are revisited later with the third one, multityping, defined in Sect. 3.

2.2 Multifidelity Service Systems

Service-oriented architecture (SOA) is an architectural approach in which applications make use of services available in the network. It emerged as the approach to combat complexity and challenges of large monolithic applications by offering cooperation of replaceable local/remote component services with one another at runtime, as long as the semantics of the component service is the same. A service network is a structure that brings together local/remote *service providers* to deliver service cooperation represented by the net-centric *request services* – expressions of the hierarchically organized cooperations of service providers. The net-centricity of request services and replaceability of local/remote service providers is defined in SML using multitypes of *provider services* also called *operation services*, in short *opservices*, as explained in Sect. 3. In a conceptual multifidelity service system MSS, the semantics of local/remote service providers (delivering executable codes) is generalized to replaceable multifidelity realizations of *service functions – service contextions*. Semantically request services are like cooperation activities but opservices like service provider actions.

In a multifidelity function system *MFS* system defined in (8) both a domain and a codomain of functions are abstracted to a set *C* of tuple contexts. However, in *MSS* data used by request services is embedded in *service contexts* – collections of hierarchically organized attributed tuples, a kind of service taxonomy or ontology. Combined request services use an evolving shared context while executing cooperative problem solving and return the result service context that contains outputs of all participating services. The design principle for aggregating data into service contexts and processing shared contexts by all cooperating services working in unison is called *service context awareness*. Service context awareness, also called service *contextion*, is a form of parametric polymorphism. In particular, a service contextion is a mapping from input service context to output service context. Using contextion, a function or a data type can be expressed generically so that it can handle inputs and outputs identically without depending on their type. Request services (multifidelity service contextion) and service context types (data types) are *generic services* and *generic datatypes* and form the basis of *generic service-oriented programming*.

A *service context* is a collection of related *named entries* such that each name is uniquely associated with a constant, calculated, or undefined value. Names of entries create a namespace of the context in terms of domain attributes. A sequence of attributes associated with a context value is called a *path*. Attributed paths of context entries specify the semantics of context data. Note that a tuple context is an ordered collection of input/output data while a service context is unordered semantic map (ontology) that associates values with context paths shared in the network for cooperating service providers.

Given the set *ES* of all entries, the set of all service contexts *CS* is equal to P(*ES*), the powerset of *ES*. Contexts with constant values are called *data contexts* or *data models* and denoted by *DC*. Contexts that contain evaluated entries are called *context models* and denoted by *CM*. Therefore, the set of contexts *CS* is the union

$$CS = DC \cup CM \tag{10}$$

A *request service*, called a *contextion function* or simply *contextion,* is a mapping

$$c : DC \to DC \tag{11}$$

such that $c(dc_{in}) = dc_{out}$ for $dc_{in} \in DC$ and $dc_{out} \in DC$. A context dc_{out} is an output context of the service request $c \in RS$ for an input context dc_{in}.

A multifidelity function system *MFS* defined in (8) with a set *F* of functions replaced by the set *RS* of multifidelity request services with the set *CS* of service contexts is called a *multifidelity service system* defined as follows:

$$MSS = <CS, RS, SP, SM> \tag{12}$$

where *SP* is a set of service projections, *SM* is a set a service morphers. Realizations of service requests *RS* under fidelity management defined by *SP* and *SM* are called service multifidelities.

A multifidelity from the computing perspective refers to a computing environment with multiple implementations for a given computing process, meaning there are different computing processes to choose from [15–17]. When selecting fidelities for a complex computing process, it is important to appropriately balance the fundamental tradeoff between cost and computability of total versus partial service realizations at runtime. Such tradeoff in complex systems can be part of the computational process itself with fidelity management based on analysis of intermediate input and output service contexts at runtime with *morph-fidelities*, fidelities associated with service morphers from *SM*. Morphers based on contextion inputs and outputs reconfigure fidelities of contextions *RS* in *MSS* by applying corresponding projections from *SP* [15].

In SML various types of request services are distinguished with two main categories: elementary and combined request services along with five types of contextions described in Sect. 3. On the one hand, a multifidelity function system *MFS*, as defined in (7), is a conceptual framework for multifidelities in SML. On the other hand, a multifidelity service system *MSS* defines context awareness as parametric polymorphism for input/outputs of service contextions in *MSS*.

2.3 Multitypes of Provider Services

A *service provider* is a multifidelity realization of an *elementary request service* in the multifidelity service system *MSS* (12). In contrast, an *operation service*, in short, an *opservice*, is an expression of a direct service provider in SML, then request services use opservices directly. This allows to distinguish service operations for service *actualization* from various combinations of request services for service *cooperation*.

A *service type* is an attribute of service provider which tells the request service how its service consumer intends to use a required service provider. An association $<op, tp>$ of a service type *tp* and its operation *op* that represents a contextion is called a *service signature*. A service provider may implement multiple service types each with multiple operations. Therefore, a signature type can be generalized to a *multitype* that serves as a classifier of service providers in the network. A multitype signature $<op, tp_1, tp_2, \ldots, tp_n>$ is an association of a service operation *op* and a multitype in the form of the list of service types $tp_1, tp_2, \ldots, tp_n$ implemented by a service provider. The service type of a multitype associated with an operation is called a *primary service type*, usually the first service type in the list of service types of the signature. If all service types of a signature are of interface type, then such signature is called *remote*. If a primary type of a signature is a class type, then a signature is called *local*.

An instance of a service provider actualized by a signature is called a *providlet*. Note that binding to a service provider in the network is dynamic, so the identity of a service provider instance is undetermined in a request service. Service signatures in request services are free variables to be bound to providlets - redundant service provider instances available or provisionable in the network.

Multityping is a form of subtype polymorphism in which a service provider multitype (subtype) is related to another multitype (supertype) by dynamic binding the service multitype to local/remote service provider instances, meaning that service signatures of request services can also operate on subtypes of service providers. Net-centric multityping leads to *multitype management* - coordination of the service activities: registration, discovery, provisioning, and lookup - all based on a service multitype implemented by service providers. However, *multi-multitype management* is the organization and coordination of provisioning and binding a combined request service with multiple service signatures, using its *multi-multitypes*, to a group of service providlets. A multi-multitype of a combined request service is the classifier of a service providlet group in the network as the instruction set of the dynamic *service processor* for the request service. Net-centric multi-multitype grouping for combined service requests is oblivious of implementation, location, and invocation protocols of participating self-contained service providlets.

Multityping defines the inheritance hierarchy of service providers in the network. A multitype *N* is assignable from a multitype *M*, if *N* and *M* are the same, or each service type of *N* is assignable from a service type of multitype *M*. If *N* is assignable from *M,* then *N* is said to be a *supermultitype* of *M*. If *M* is a *submultitype* of *N*, then multityping relation is defined, as *N* is assignable from *M*, to mean that any signature of type *M* can be safely used in a context where a signature of type *N* is expected. Therefore, if a providlet of multitype *N* is required while in the network exists a providlet of type *M* and *N* is assignable from *M*, then the signature of multitype *N* can be bound to the providlet of multitype *M*. The same applies to multi-multitypes that define the inheritance of service providlet groups in the network.

3 Service Semantics in SORCER

Service semantics can be either declarative, imperative, OO, SO, or a blend of them. A blend of relevant combinations of request services should be supported by SO languages intended for solving complex problems and building distributed heterogeneous SO systems. Therefore, elementary and combined services should be expressed in a programming language with adequate semantics and syntax. 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, procedural, and functional programming (see Fig. 1). The pillars of true SO programming: contexting, multifidelity, and multityping describe the basic traits of request services as described in Sect. 2. The presented metafidelity service system *MSS* (12), is a metamodel for the SML service semantics described in this Section, the SML syntax (Sect. 4) and the reference SORCER architecture (Sect. 6). The term semantics reflects the need to not only model something in the real world, but to model the meaning that this something has for the purpose of the metamodel – service-oriented computing. SML expressions are executed with a Service Virtual Machine (SVM) presented in Sect. 6.

A *service consumer* is a combination of request services, but a *service provider* delivers executable codes (as executable applications, tools, or utilities) to be actualized via *operation services* (of Signature and Evaluator types, called *exec-opservices*) used by *elementary request services* (of Task and Entry types), see Fig. 2. SML *request services* say what to do, but *service providers* run executable codes that are expressed by opservices. *Combined request services* represent SO processes by hierarchically organized elementary and other request services that in turn run cooperations of executable codes expressed by opservices. In other words, in SML exec-opservices are *provider services* – service specifications or contracts, but service providers are implementations of them. A set of opservices of a combined request service binds at runtimes to the collection of service providers called the *service provider partnership*. The executable codes of the partnership form the instruction set of the dynamic *net-centric service processor*.

A *pipeline service* is a set of opservices connected in series, where the output of one opservice is the input of the next one. The opservices of a pipeline can be executed sequentially or in parallel. A pipeline is a combined opservice of Evaluator type that can be used with looping and branching evaluators to form structured algorithms. Exec-opservices can be concatenated with cxt-opservices that preprocess/postprocess service contexts used by exec-opservices and request services.

A *domain service,* in programming dialect called a *mogram* [4], is either a routine (imperative domain) or model (declarative domain), or both. It provides for declarative/imperative transitions within a model across both component models and routines (*transmodel*) and for transitions within a routine across both component routines and models (*transroutine*). The *Transroutine* and *Transdomain* types are subtypes of the *Transdomain* type along with the *Collaboration* type as shown in Fig. 2.

A *domain service* is either *declarative* – a model, or *imperative* – a routine. Models are collections of functional compositions of entries, but routines are either *structured blocks* of routines or *workflow jobs* comprised of component routines and elementary routines called *tasks*. In principle, a model is a hierarchically composed domain of entries, but a routine is a domain of a hierarchically structured tasks.

Subordinated domain of a transroutine or a transmodel contribute directly to its transdomain responses - the output context of the transdomain. However, a service *collaboration* is the transdomain focused on cooperation of subordinated domains toward collaboration driven by an explorer/optimizer. It means that direct results of component domains are used by the exploration/optimization process returning an indirect result.

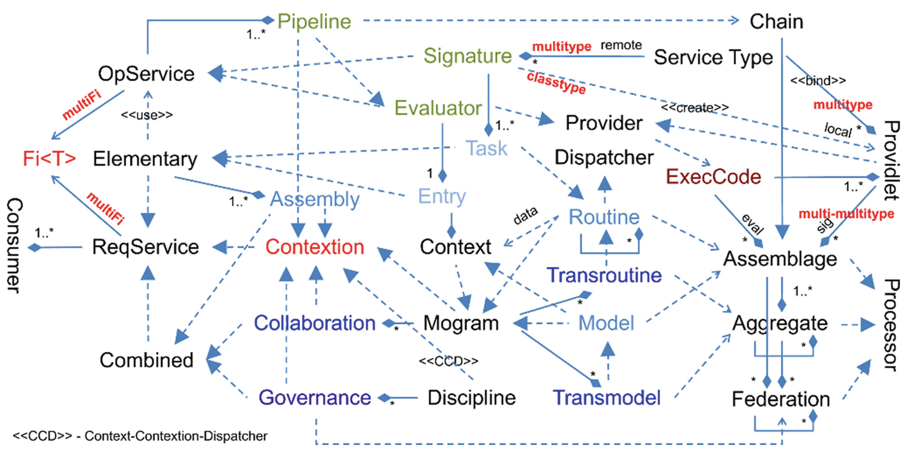

Fig. 2. The service relationships in SML for process expression and actualization. By the same color of Domain its subtypes Routine and Model are indicated. The same coloring convention applies to color of Transdomain and its subtypes Collaboration, Transroutine, and Transmodel.

The *Domain* type is the direct supertype of *Context* and the subtype of *Contextion*. Subtypes relationships of the Domain type are shown in Fig. 2 by the same color with its subtypes: *Routine* and *Model* and direct generalizations as well. The coloring convention applies to the color of *Transdomain* and its subtypes (*Collaboration*, *Transmodel*, *Transroutine*) as well.

A *discipline service* is a triplet: a <*context*, *contextion*, *dispatcher*> (CCD), such that a context is the input data, a contextion is the process expression (request service), and a dispatcher is the controller (routine service) of the discipline. A dispatcher configures and dispatches its contextion to be executed in the network then returns the proper result. CCD is the architectural service pattern for developing and deploying disciplinary services as self-contained services with multifidelity components (contexts, contextions, and dispatchers) for constructing runtime triplets to be used in federated service with centralized governance. Therefore, any contextion can be used to create a discipline fidelity of multifidelity disciplines. Next, multiple heterogeneous local/remote disciplines can be combined into a multidisciplinary service under control of the shared central governance.

A governance service is a specification of transdisciplinary supervised cooperation of federated disciplines. Federated disciplines can be seen as a cooperation of heterogeneous contextions (states) unified under common governance to be realized by a supervisor (governor).

Service provider partnerships are runtime collections of service providers that realize service cooperations expressed by request services. From the service actualization point of view, a *pipeline* depends on the Chain of service providers bound to the concatenation of opservices, but a combined request service depends hierarchically on actualization of component elementary request services, which in turn depend on actualization of opservices executing corresponding executable codes of corresponding service providers. Partnerships represent dynamic cooperations of Chain, Assemblage, Aggregate, and Federation type – see Fig. 2. An assemblage refers to grouping providers of elementary request services used in domains as routines or models. An aggregate refers to grouping assemblages for transdomains as transroutines, transmodels, and collaborations. Finally, the federation refers to governing of discipline services, that in turn federate partnerships of discipline contextions specified by the governance service as shown in Fig. 2.

To illustrate the structure of service federalism with five types of combined request services, let us consider governance service A depicted in Fig. 3. It is the functional cooperation A(B(E, F(I, J)), C, D(G(K), H)) of three disciplines (two with collaborations B and D and one with a pipeline C), two additional collaborations (F and G), and five assemblies (E, I, J, K, H). The governance A binds hierarchically to service federation FS of three disciplines at runtime with 23 service providers by 21 elementary services and 2 opservices.

SO federalism is a model of net-centric governance – a federal (central) contextion with federated contextions of disciplines (like states), and an opservices corresponding to providlets (citizens). The rules of governance are realized by the service operating system (SOS – a kind of federal government). SOS coordinates execution of federated disciplines downstream from the governance via request services to opservices. It does so by hierarchically executing service providers referenced by multitypes of opservices at runtime. The main purpose of SOS is to satisfy interests of service consumers and to fulfill their needs using capabilities of hierarchical service partnerships for request services - from federations, aggregates, assemblages, chains, and opservices down to service providers of executable codes.

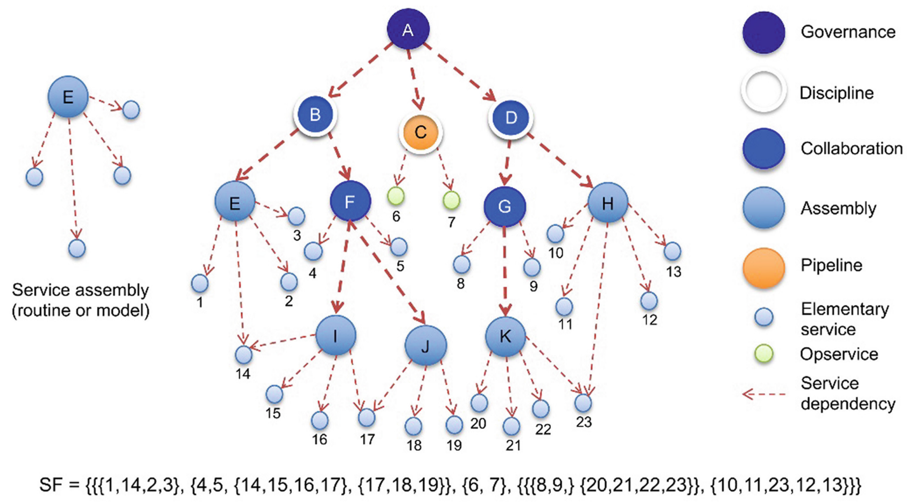

Fig. 3. Governance service – multidisciplinary service as a cooperation of seven types of services.

Entries and tasks depend on operation services: evaluators and signatures, respectively. Entries use various types of multifidelity evaluators, to invoke executable codes. A signature is a multitype provider service (a net-centric handle) to be bound at runtime to the remote/local service providlet to execute a signature operation. The unique signature-based architecture allows for configuration and execution of distributed dependencies of combined request services by uniform handling of local and remote service providers at various levels of granularity and fidelity. When dealing with net-centric complexities, you have a case to distribute services, otherwise create a modular monolith with locally executable services. Later, when complexity of the system becomes unmanageable you can deploy almost instantly the existing local providers as network providers on as-needed basis, and then run updated services of the original monolith in the network. In SORCER it is done by changing the primary service type of signatures from the class type to the interface type, or just selecting the remote service fidelity. When using the signature-based approach, service providers never communicate directly with each other. The signature with the primary service type as an *interface type* is called a *remote signature* and with a *class type* is called an *local signature*. When executing a combined request service, SOS creates the hierarchical service partnership with the relevant network connectivity at runtime and executes the exec codes of the partnering providers.

Governance request services allow for creating large scale multidisciplinary federated systems. However, most discipline processes (contextions) are expressed by mograms required for constructing effective heterogenous discipline service to be federated. A *domain* is a contextion composition expressing a service combination by one of the five design patterns:

1. *entry model* – is a declarative expression of interrelated higher-order entries (contentions) in a context model.
2. *service block* – is an expression of concatenated subroutines with branching and looping tasks as a block-structured subroutine.
3. *service job* – is an object-oriented composite (workflow) of subroutines with a control strategy for each component job to be executed sequentially or in parallel, synchronously or asynchronously, with context pipes between component subroutines.
4. *service transroutine* – is a service block or service job (transroutine) comprised of both subroutines and service models
5. *service transmodel* – is context model comprised of both models and subroutines.

The presented above mogramming abstractions reduce representational complexity of typical SO processes, so it makes easer to comprehend a computing paradigm of each service design pattern: functional (1), procedural (2), and object composite (3 and 4). Therefore, each mogram abstraction exposes the details which really matter to the domain-specific users from the perspective of preferred programming paradigm and hide the other details (service types, exec codes of evaluators, providlets implementing multitypes) regarding development and deployment of service providers implemented with lower level programming abstractions and languages used by software developers. The above four service design patterns reflect corresponding programming styles shown in Fig. 1. The mogram design patterns allow to blend multiple programming styles within a single combined requested service.

The presented service semantics of request service in SORCER allows to summarize the three SO pillars (see Fig. 1) as follows:

1. *Contextion* allows for a mogram to be specified generically, so it can handle context data uniformly with required data types of context entries to be consistent with ontologies of service providers. Contextion as the form of parametric polymorphism is a way to make a SO language more expressive with one generic type for inputs and outputs of all request services.
2. Morphing a request service is affected by the initial fidelities selected by the user and morphers of morph-fidelities. Morphers associated with morph-fidelities use heuristics provided by the end user that dependent on the input service contexts, and subsequent intermediate results obtained from service providers. *Multifidelity management* is a dispatch mechanism, a kind of ad hoc polymorphism, in which fidelities of request services are reconfigured or morphed with fidelity projection at runtime.
3. Service multityping as applied to service signatures and providers is a form of subtype polymorphism with the goal to find a remote instance (providlet) of the service provider by the range of service types that a service provider implements and registers for lookup. It also allows a multifidelity opservice to call an operation of a primary service type implemented by the service provider as a different service fidelity. With respect to service providers to be provisioned for service signatures of a request service – multi-multityping of the request service specifies which service providers have to be additionally provisioned to complement existing service providers in the network.

4 Introduction to SML

A language can be specified by its metamodel with a great flexibility [4], as shown in Fig. 4. A language can be also specified by a grammar, for example the Java language in EBNF. The primary responsibility of the metamodel layer is to define languages that describe semantic domains to allow users to model a wide variety of different problem domains. The presented approach to true service-oriented metamodeling architecture is based on three abstract service categories: operation services (signatures and evaluators), elementary request services (task and entries) and combined request services (domain, discipline, transdomain, and transdiscipline) used with three pillars of service-orientation: contextion, multifidelity, and multityping described in Sect. 2.

Therefore, MSF for SML, is like MOF [18] for UML. It is a metamodel defined by the multifidelity service system *MSS* (12) that specifies how the SML model should conform to the conceptual *MSS* system. The SML metamodeling hierarchy along with the UML metamodeling hierarchy is depicted in Fig. 4 to explain the relationship of SML (MSF/M2) to the object-oriented SORCER runtime (MOF/M0). The SORCER operating system manages request services that comprise of hierarchically structured operation services and runs the corresponding net centric service provider partnership (MSF/M0) bound to operation services, that serve as the instruction set (MOF/M1-) of the service processor at MSF/M1-.

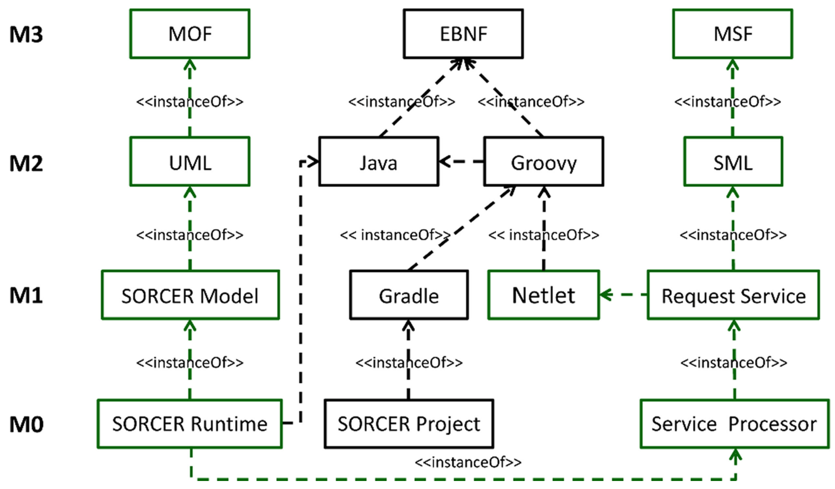

Fig. 4. The UML/SML specific five-layer MOF-MSF metamodel hierarchy.

A request service, called a context model CM in SML conceptually corresponds to a multifidelity service system *MSS* (12) with a collection *RS* of interrelated request services as functional entries in CM used as its domain and codomain. A multifidelity contextion f = (CM, *fi*(f), mFi_f) in CM is declared in SML as a service entry as follows:

$$func\ \mathrm{f} = ent(\text{“f”}, mFi_f, args(\text{“f}_1\text{”}, \text{“f}_2\text{”}, \ldots, \text{“f}_k\text{”}))$$

where “f” is a name (a path in CM) of the function *f* declared by the operator *ent*; “f_1”, “f_2”, …, “f_k” are argument paths of function f in CM, and mFi_f is the multifidelity of function f. By default, a fidelity of function f, *fi*(f) – an *entry evaluator*, is the first realization in the ordered set mFi_f. The argument paths “f_1”, “f_2”, …, “f_k” in CM bind to values of corresponding entries in CM to create a subcontext of CM as the argument of contextion f.

The *ent* operator defines a generic functional expression declared in a context model CM. Functional entry form higher-order functions – responses of the model. If *ent* declares a constant function, then a model with all such entries is called a *data model* or *data context*.

A *service signature* in SML is an operation service representing either the local or remote operation of a service provider. It declares a service type of provider tp with its operation op, to be invoked in the scope of its service context. A signature association <op, tp> is denoted in SML by *sig*(op, tp). A functional request service f defined by an operation op to be executed by the service provider implementing a service type tp is declared as follows:

$$\textit{func}\ \mathrm{f} = \textit{ent}(\text{“f”}, \textit{sig}(\mathrm{op}, \mathrm{tp}, \textit{inPaths}(\text{“}\mathrm{x}_1\text{”}, \text{“}\mathrm{x}_2\text{”}, \ldots, \text{“}\mathrm{x}_s\text{”}), \\ \textit{outPaths}(\text{“}\mathrm{y}_1\text{”}, \text{“}\mathrm{y}_2\text{”}, \ldots, \text{“}\mathrm{y}_t\text{”})))$$

or as a multifidelity service entry with multiple signatures:

$$\textit{func}\ \mathrm{f} = \textit{ent}(\text{“f”}, \textit{entFi}(\textit{sig}(\mathrm{op}_1, \mathrm{tp}_1), \ldots, \textit{sig}(\mathrm{op}_n, \mathrm{tp}_n)))$$

where the operator *entFi* declares a multifidelity of entry f and the operators *inPaths* and *outPaths* specify subcontexts determined by input and output paths in CM as the argument and return values of the operation op.

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 $\mathrm{tp}_1, \ldots, \mathrm{tp}_s$, 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 $(\mathrm{tp}_1, \mathrm{tp}_2, \ldots, \mathrm{tp}_s)$, an operation op_1 of a type tp_1, and an optional service provider name myService takes the following expanded form:

$$\textit{sig}\ \mathrm{s} = \textit{sig}(\mathrm{op}_1, \mathrm{tp}_1, \mathrm{tp}_2, \ldots, \mathrm{tp}_s, \textit{prvName}(\text{“myService”}))$$

Note that a signature does not refer to a particular instance of a service provider; its multitype is used for binding to an available instance (providlet) at runtime. Multi-typing is used to manage unpredictability of the unreliable network comprised of replaceable remote service providlets with one another at runtime, as long as the multitype semantics of the service providlets is the same. The local/remote semantics of a service in SML is based on the concept of multityping. If the primary type tp_1 is a 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 providlet implementing the required multitype.

A *value entry* x (constant function) equal to y, is declared in CM as follows:

$$\textit{val}\ \mathrm{x} = \textit{val}(\text{“x”}, \mathrm{y}), \text{ for value y} \in \mathrm{Y}$$

or a multifidelity variable x

$$\mathit{val}\ \mathrm{x} = \mathit{val}(\text{“x”}, \mathit{entFi}(\mathit{val}(\text{“}x_1\text{”}, y_1), \ldots, \mathit{val}(\text{“}x_k\text{”}, y_k)))$$

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

$$\mathit{cxt}\ \mathrm{dc} = \mathit{context}(\mathit{val}(\ldots), \ldots, \mathit{val}(\ldots))$$

and valuation of the entry x in dc as follows:

$$\mathit{Object}\ \mathrm{y} = \mathit{val}(\mathrm{dc}, \text{“x”})$$

where “x” is a name of variable (a path) in a data context dc.

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

$$\mathit{setValue}(\mathrm{dc}, \text{“x”}, \mathrm{v})$$

A context model mdl (of mog type) as an unordered collection of value entries and functional entries is declared as follows:

$$\mathit{mog}\ \mathrm{mdl} = \mathit{model}(\mathit{val}(\ldots), \ldots, \mathit{ent}(\ldots), \ldots)$$

Note that multivariable functional entries of a context model may take other functional entries as arguments to create higher-order functions while function multi-variability is bound to the corresponding subcontext of the underlying model.

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

$$\mathit{Object}\ \mathrm{y} = \mathit{exec}(\mathrm{mdl}, \text{“f”})$$

or

$$\mathit{Object}\ \mathrm{y} = \mathit{exec}(\mathrm{mdl}, \text{“f”}, c_{in})$$

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

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

$$\mathit{cxt}\ c_{out} = \mathit{eval}(\mathrm{mdl})$$

or

$$\mathit{cxt}\ c_{out} = \mathit{eval}(\mathrm{mdl}, c_{in})$$

where c_{out} is a data context - the result of evaluation of responses for an input context c_{in}. Model evaluations are defined by function compositions of response entries with no explicit strategy for altering the function compositions of the model. However, function compositions can be altered by execution dependencies specified for entries that depend on execution of other entries in the model.

A subset of responses of a model (paths of response entries in the model) can be part of the model declaration by inlining responses “f_1”, “f_2”, …, “f_k” as follows:

response(“f_l”, “f_2”, …, “f_k”)

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

responseUp(mdl, “f_l”, “f_2”,…, “f_k”)

and to decrease responses:

responseDown(mdl, “f_l”, “f_2”,…, “f_k”)

No paths provided for responseDown removes all responses and responseUp may append new responses of the model.

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

$$\mathit{Object}\,\mathrm{out} = \mathit{exec}(\mathrm{sr}, \mathrm{arg}_1, \ldots, \mathrm{arg}_\mathrm{n})$$

where arg_i is an SML argument of the Arg type. For example, signatures, fidelities, contexts, and models 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 service type (interface) Adder. Here, the signature *sig*(“add”, Adder.-class) binds to an instance of service provider - providlet - 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 for a given context on the locally created instance. Therefore, a change of the primary type of signature from interface type to a class type changes a remote call to a local one and vice versa.

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

$$\textit{mog}\ \mathrm{y} = \textit{task}(\text{“y”},\ \textit{sig}(\mathrm{op}, \mathrm{tp}), \textit{context}(\ldots))$$

where “y” is a name of the task y with a given signature and an input context.

A multifidelity task is declared in SML as follows:

$$\textit{task}(\text{“y”}, \textit{sigFi}(\textit{sig}(\text{“fi}_1\text{”}, \mathrm{op}_1, \mathrm{tp}_1), \ldots, \textit{sig}(\text{“fi}_n\text{”}, \mathrm{op}_n, \mathrm{tp}_n)), \textit{context}(\ldots))$$

where the operator *sigFi* declares a 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 executable codes, represented by exec-operations services, interconnected and replaceable at runtime. In SML interconnections of functional entries and service tasks (see Fig. 2) are declared by a combined request service (models and subroutines) that binds operation services (evaluators and signatures) to remote/local executable codes at runtime.

In SML a service subroutine is a request for a procedural (block) or workflow (job) service type. A service task is an elementary subroutine used in combined subroutine. A combined subroutine is a collection of subroutines and/or mograms grouped together within the scope of SML operators, either block or job. A *subroutine block* is a concatenation of component mograms along with flow-control tasks: conditional (opt, alt) and loop (loop) tasks. The SML semantics of opt, alt, and loop is the same as the corresponding UML operators used with interaction frames (combined fragments) in sequence diagrams. A *subroutine job* (service workflow) is an object-oriented composite of component subroutines and/or mograms, optionally with an explicit control strategy and service pipes for interprocess communication between components of the workflow.

Subroutines can be used as evaluators of entries in context models, but responses of evaluated context models can be used as data contexts in subroutines. That way, either a subroutine blended with models, or a model blended with subroutines, creates a service combination of models and/or subroutines – a service mogram. The SML *ent* operator, in most obvious cases, declares a service entry of the type according to its evaluator type. However, specialized SML entry operators, for example: *val*, *prc*, *lmb*, *snr*, and *srv* correspond to entry subtypes: value, procedure, lambda, service neuron, and service, respectively; can be used to indicate directly requested entry subtypes.

A mogram m_{in} to be executed by exerting cooperating service providers is declared as follows:

$$\textit{mog}\ \mathrm{m}_{\mathrm{out}} = \textit{exert}(\mathrm{m}_{\mathrm{in}})$$

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

$$\mathit{cxt}\ c_{out} = \mathit{result}(m_{out})$$

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

$$\mathit{Object}\ y = \mathit{value}(c_{out}, \text{“x”})$$

or from the exerted mogram directly:

$$\mathit{Object}\ y = \mathit{exec}(mog_{out}, \text{“x”})$$

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

$$\mathit{cxt}\ c_{out} = \mathit{eval}(m_{in})$$

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

A mogram is a collection of interacting request services (entries, tasks, models, and subroutines) that bind at runtime to a cooperation of service providers via mogram opservices. Multifidelity mograms can morph during execution under control of the fidelity mangers and related morphers with the goal to return the emerged result of the evolving net-centric cooperation of service providers - a morphing system of systems. A mogram, is also called an *exertion* [11] due to *exert* operator applied to mograms.

To illustrate SML in action we refer the reader to the examples, in the open source SORCER project [20], in the module *examples*, in particular multifidelity test cases: at sml/src/test/main/java/mograms/ModelMultiFidelities.

5 An Example of a Multifidelity Model in SML

To illustrate the introductory SML syntax presented above in action, a simple context model is declared in SML with four multifidelity entries (mFi1, mFi2, mFi3 and mFi4), four metafidelities (sysFi2, sysFi3. sysFi4, sysFi5), four morphers (morpher1, morpher2, morpher3, morpher4) as lambda expressions, and five provider services used in entries and tasks of the model mdl below. Signatures in entries are remote and in tasks local.

```
// multifidelity model with four morph-fidelities
// (mphFi) and corresponding morphers
mog mdl = model(inVal("arg/x1", 90.0),
      inVal("arg/x2", 10.0), inVal("morpher3", 100.0),
    ent("mFi1", mphFi(morpher1, add, multiply)),
    ent("mFi2", mphFi(entFi(ent("ph2", morpher2),
      ent("ph4", morpher4)), average, divide, subtract)),
    ent("mFi3", mphFi(average, divide, multiply)),
    ent("mFi4", mphFi(morpher3, t5, t4)),
    fi2, fi3, fi4, fi5,
    response("mFi1", "mFi2", "mFi3", "mFi4", "arg/x1",
      "arg/x2", "morpher3"));

// signatures used in multifidelity entries in mdl above
sig add = sig("add", Adder.class,
    result("y1", inPaths("arg/x1", "arg/x2")));
sig subtract = sig("subtract", Subtractor.class,
    result("y2", inPaths("arg/x1", "arg/x2")));
sig average = sig("average", Averager.class,
    result("y3", inPaths("arg/x1", "arg/x2")));
sig multiply = sig("multiply", Multiplier.class,
    result("y4", inPaths("arg/x1", "arg/x2")));
sig divide = sig("divide", Divider.class,
    result("y5", inPaths("arg/x1", "arg/x2")));

// two service tasks used as fidelities of mFi4 in mdl
mog t4 = task("t4", sig("multiply", MultiplierImpl.class,
           result("result/y",
             inPaths("arg/x1","arg/x2"))));

mog t5 = task("t5", sig("add", AdderImpl.class,
        result("result/y",
           inPaths("arg/x1", "arg/x2"))));

// four morphers used with morph-fidelities
Morpher morpher1 = (mgr, mFi, value) -> {
    Fidelity<Signature> fi = mFi.getFidelity();
    if (fi.getSelectName().equals("add")) {
        if (((Double) value) <= 200.0) {
            mgr.morph("sysFi2");
        } else {
            mgr.morph("sysFi3");
```

```
                }
            } else if (fi.getPath().equals("mFi1")
                && fi.getSelectName().equals("multiply")) {
                    mgr.morph("sysFi3");
            }
        };

        Morpher morpher2 = (mgr, mFi, value) -> {
            Fidelity<Signature> fi = mFi.getFidelity();
            if (fi.getSelectName().equals("divide")) {
                if (((Double) value) <= 9.0) {
                    mgr.morph("sysFi4");
                } else {
                    mgr.morph("sysFi3");
                }
            }
        };

        Morpher morpher3 = (mgr, mFi, value) -> {
            Fidelity<Signature> fi = mFi.getFidelity();
            Double val = (Double) value;
            if (fi.getSelectName().equals("t5")) {
                if (val <= 200.0) {
                    ((EntModel)mgr.getMogram())
                      .putValue("morpher3", val + 10.0);
                    mgr.reconfigure(fi("t4", "mFi4"));
                }
            } else if (fi.getSelectName().equals("t4")) {
                // t4 is a mutiply task
                ((EntModel)mgr.getMogram())
                    .putValue("morpher3", val + 20.0);
            }
        };

        Morpher morpher4 = (mgr, mFi, value) -> {
            Fidelity<Signature> fi = mFi.getFidelity();
            if (fi.getSelectName().equals("divide")) {
                if (((Double) value) <= 9.0) {
                    mgr.morph("sysFi5");
                } else {
                    mgr.morph("sysFi3");
                }
            }
        };
```

```
// metafidelities used by morphers
fi fi2 = metaFi("sysFi2", mphFi("ph4", "mFi2"),
  fi("divide", "mFi2"), fi("multiply", "mFi3"));
fi fi3 = metaFi("sysFi3", fi("average", "mFi2"),
  fi("divide", "mFi3"));
fi fi4 = metaFi("sysFi4", fi("average", "mFi3"));
fi fi5 = metaFi("sysFi5", fi("t4", "mFi4"));
```

Let's evaluate mdl subsequently with specified multifidelities and morphers with default fidelities and later with the requested fidelity fi("mFi1", "multiply").

```
// evaluate mdl with default fidelities
cxt out = eval(mdl);
assertTrue(value(out, "mFi1").equals(100.0));
assertTrue(value (out, "mFi2").equals(9.0));
assertTrue(value (out, "mFi3").equals(900.0));
assertTrue(value (out, "mFi4").equals(110.0));

// evaluate mdl the fidelity mFi1
out = eval(mdl, fi("mFi1", "multiply"));
assertTrue(value (out, "mFi1").equals(900.0));
assertTrue(value (out, "mFi2").equals(50.0));
assertTrue(value (out, "mFi3").equals(9.0));
assertTrue(value (out, "mFi4").equals(920.0));
```

Let's restrict morphing of the multifidelity model mdl until the value of entry "morpher3" in mdl is less than 900.0. It is implemented with a service block mdlBlock executing a loop with the condition in the form of lambda expression where cxt is the current context of mdlBlock. The morph fidelity of the entry mFi1 in mdl is selected to multiply when exerting mdlBlock.

```
Block mdlBlock = block(
   loop(condition(cxt ->
        (double) value(cxt, "morpher3") < 900.0),
        mdl));

mdlBlock = exert(mdlBlock, fi("multiply", "mFi1"));
assertTrue(value(context(mdlBlock),
        "morpher3").equals(920.0));
```

The above examples can be found in the SORCER-multiFi project [20] in the module examples at sml/src/test/java/sorcer/sml/mograms/ModelMultiFidelities, test cases morphingFidelities and morphingFidelitiesLoop.

6 The SORCER Platform

Computing requires a platform (runtime system) to operate. Computing platforms that allow programs to run require a processor, operating system, and programming environment with supporting tools to create and run programs. SORCER is the platform driven by the three pillars of SO: contextion, multifidelity, and multityping. The SORCER programming environment is based on SML and Java APIs with its unique service-oriented operating system (SOS) that manages the net-centric service processor for executing request services. Technically, the service processor comprises of local/remote objects implementing evaluators and service types of signatures. SORCER remote objects (providlets) are deployed with dynamic small-footprint dynamic service containers called service exerters. A service exerter can run concurrently multiple providlets in the network.

The relationship of the basic SORCER types required to implement multifidelity services is depicted in the diagram in Fig. 2 with UML relationships. Services of the Request type are instances of two elementary subtypes: Entry and Task, and the basic request service type Contextion with five subtypes: Pipeline, Domain, Discipline, Collaboration, and Governance. All request services are instances of the common Service type with uniform execution of local and remote services at runtime. Top-level interfaces of the SORCER system that refer to the SO concepts: Fi<T>, Signature, Evaluator, Request, Entry, Task, Contextion, Context, Model, and Provider, all are subtypes of the common Service type.

From the SO point of view creation of user-centric request services – mogramming – is the primary objective assuming that service providers implement multitypes and their operations can be incorporated into net-centric service processor managed by SOS. Note, that multifidelities are used in request services only. A combined request service hierarchically combines elementary requests (entries and tasks) that bind dynamically to executable subroutines of evaluators and service providers, respectively.

Each service provider implements a multitype of service types. Each service type may have multiple implementations in the network. SOS does not know location of service provider instances in the network; it requires only their service types to be implemented in the network. The question is, how to find a required implementation in the network. The answer is, by matching a multitype of the service signature to the multitype of the implementation available in the network. To differentiate 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 signatures is the concept of finding providers of the same multitype from redundant instances (providlets) available in the network.

In systems theory emergence is a process whereby larger entities and regularities arise through interactions among smaller or simpler entities that themselves do not exhibit such properties. An emergent SO behavior can appear when a number of simple services operate in an environment, forming more complex behaviors as a service collective (partnership). It can commonly be identified by patterns of accumulating change used by morphers. Emergent behavior is hard to predict since the number of interactions between components of a system increases exponentially with the number of components, thus potentially allowing for many new and subtle types of behavior to emerge. Emergence is often a product of particular patterns of interaction. Negative feedback introduces constraints that serve to fix structures or behaviors. In contrast, positive feedback promotes change, allowing local variations to grow into global patterns. Multifidelity services can be observable and observed. Therefore, the positive or negative feedback received by morphers regarding applied system fidelities from observable multifidelity services can be used to update fidelities, upstream to the metamodel level and downstream for new projected and created instances of the metamodel. The projected and new instances are created by the fidelity management system to form emergent properties of the morphing multifidelity model as illustrated in Fig. 5.

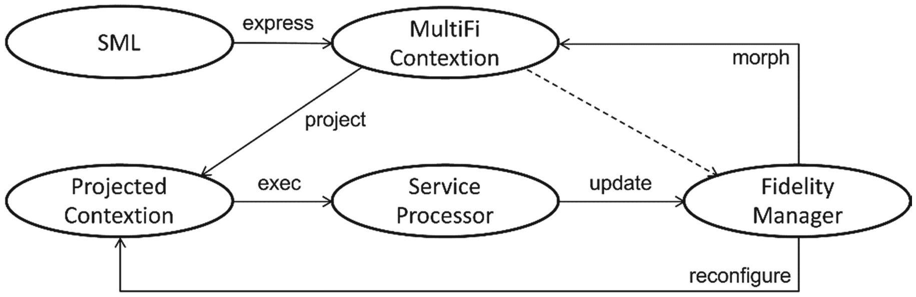

Fig. 5. Morphing and reconfiguring multifidelity service mograms expressed in SML

An emergent modeling platform requires the ability to express a SO system with a given fidelity projection as the instance of the metasystem with multiple fidelity projections. In SML a projected contextion is an instance of a multifidelity contextion – a metasystem. Also, the computing platform requires the ability to execute and morph the evolving system with updated projections managed by the metasystem. SOS enables quick and effective SO communication with net-centric services and allows for evolving updates such that each new instance of the multifidelity system is a new projection of the metasystem.

SML defines two types of multifidelities in contextions: select-fidelities and morph-fidelities. Select-fidelities allow for system reconfiguration, but morph-fidelities allow for self-morphing the structure of the multifidelity request service. Morph-fidelities are observed by a fidelity manager of contextion. Therefore, the positive or negative feedback received from computed 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 contextion fidelity projection. Morphers associated with morph-fidelities form emergent properties in the morphing multifidelity system.

A projected contextion, that defines the service cooperation actualized and managed by SOS, is an instance of the metasystem expressed by a multifidelity contextion. To reconfigure or morph a multifidelity contextion its fidelity manager uses projection functions and morphers. Both reconfiguration and morphing allow for adaptivity of system and metasystem respectively, when updates of fidelities and metafidelities are under control of the fidelity manager at runtime. Morphers of morph-fidelities in request services managed by fidelity managers may reconfigure the current contextion or morph to a new projected contextion as shown in Fig. 5.

Adaptive SO systems with morph-fidelities are emergent systems. This type of systems exhibits three types of adaptivities called system-of-system, system, and service agility [15]. Metasystem agility refers to system reinstantiation with metafidelities, system agility refers to updating system fidelity projections, and service agility refers to updating fidelities of elementary request services at runtime.

Virtual machines are based on computer architectures and provide functionality of a physical computer. Service Virtual Machine (SVM) is a network process virtual machine designed to run cooperating executable codes in in the network expressed by combined request services presented in Sect. 3. SVM serves as an abstraction layer for SML in SORCER. Thus, it becomes a multifidelity processor architecture for SML with basic operations corresponding to six categories of opservices: evaluator, signature, getter (filter), setter, appender, and connector. The first two opservices run executable codes locally/remotely (exec-opservices) and the remaining ones (context opservices in short cxt-opservices) preprocess service contexts used as inputs and outputs by request services. With two exec-opservices request services may executed unlimited number of executable codes in the network. Therefore, the network of providlets becomes the native processor for SVM with custom instruction set for request services.

The architecture of SVM with the basic internal components: the thread stack for executing request services, multifidelity projection area, combined request service area, elementary request service area, and opservice area is shown in Fig. 6. The opservice area comprises of two categories of exec-opservices (signatures, evaluators), and four categories of cxt-opservices (setters, getters, connectors, and appenders). Each thread has its own stack that holds a frame for each request service executing on that thread. A new frame is created and added to the top of stack for each component request services to be executed. The frame is removed when the request service returns normally or if an uncaught exception is thrown during the service execution. SVM supports Java methods that call back from JVM into SVM and invoke a request service.

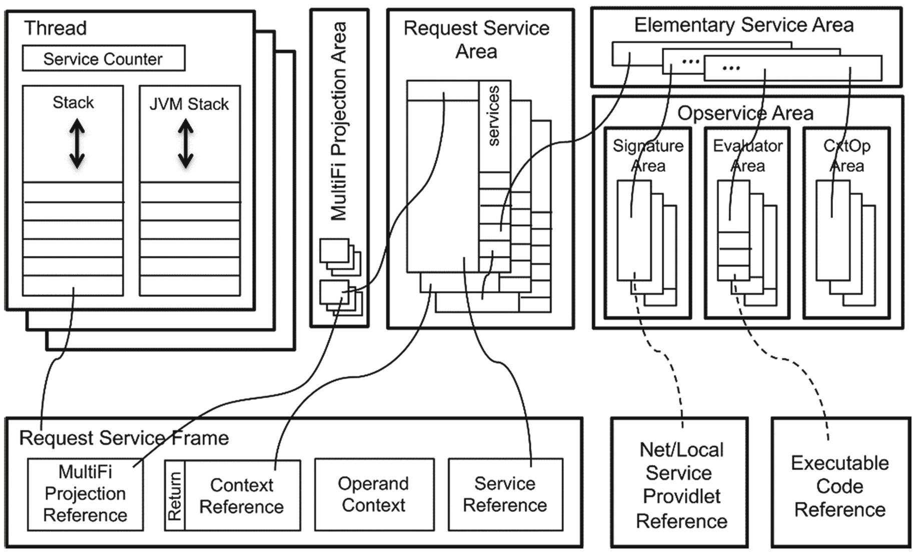

Fig. 6. SML Service virtual machine (SVM).

Each frame contains: multiFi projection, input context with context return, operand context, and a request service reference. The operand context is used during the execution of services in a similar way the general-purpose registers are used in a native CPU. While combined request services comprise of request services, only elementary request services comprise of exec-opservices. Most SVM exec-opservice spends its time manipulating the operand context by operations that produce or consume context values by calling remote/local providlets or executable codes of evaluators.

JVM used by SVM is a kind of native processor for SVM but opservices specify the instruction set of the SVM processor. Remote exec-opservices of SVM may call providlets in the network to execute request service as well. Therefore, via propagation of remote signatures originated from an SVM, the SVM expands itself into the network of cooperating SVMs executing opservices on multiple machines in the network on behave of the originating SVM. Such distributed SVM executing concurrently opservices in the network forms the instruction set of the network multiprocessor. The network multiprocessor becomes a collection of remote and local executable codes bound at runtime to signatures and evaluators executed concurrently by multiple SVNs.

In comparison to an object-oriented virtual machine, for example JVM, SVM request services and opservices correspond to methods and opcodes (bytecodes), respectively. In JVM two types of methods, instance and class methods, are distinguished but in SVM two types of request services, elementary and combined service. Opcodes used by elementary services specify the machine internal operations only, however opservices may execute locally (local signatures) and remotely (remote signatures). The fact that SVM can execute machine instructions in the network at runtime is the primary distinction between SVM and JVM. This essential service-oriented computational feature makes SVM a network-centric virtual machine.

The opservice that is assigned to the runtime data area of the SVM via a request service is executed by the SVM execution engine. The execution engine executes the SVM opcode in the unit of service instruction. It is like a CPU executing the machine command one by one. Each command of the opservice consists of an operand context. The execution engine gets one exec-opservice and execute the executable code, associated either with the evaluator or the providlet, with the operand context, and then executes the next opservice. An SVM request service is written in SML that a human can understand as service-oriented cooperation (pipeline, domain, discipline, collaboration, and governance), rather than in the programming language used to implement the execution engine. The SOS uses the SVM execution engine and manages local and remote service providers for opcodes to be executed and provides common functionalities for handling request and provider services, fidelity management, and context management for SVM.

7 Conclusions

Markov tried to consolidate all work of others on effective computability. He has introduced the term of algorithm in his 1954 book Teoriya Algorifmov [8]. The term was not used by any mathematician before him and reflects a limiting definition of what constitutes a computational process: a mathematical mapping from various initial data to the desired result. The mathematical view of process expression has limited computing science to the class of processes expressed by algorithms. 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.

A *service* is the work performed in which a service provider (one that serves) exerts acquired abilities to execute a computation. To be the *true service* resulted from the performed computation, both the computation and the service providers have to be expressed then realized under condition that service consumers should never communicate directly to service providers. Asserted cooperations of service providers represented by operation services are called request services. This way, in SML everything is a service. *Request services* represent cooperations of *opservices* bound at runtime to service providers to execute computations. In this paper, service-orientation is proposed as the approach with five types of emergent net-centric multifidelity request service representing the following service request services: pipelines, assemblies, collaborations, disciplines, governances.

The “everything is a service” semantics of SML is introduced for request services to be actualized by dynamic cooperations of service providers in the network. A multifidelity request service is considered as a dynamic representation of a net-centric emergent process defined by the end user. In SORCER, a rectified contextion – a service request embedded into a service provider container, becomes a service providlet – a process expression becomes an executable service provider.

To express emergent processes consistently and flexibly, the actualization of SML by SOS is based on three pillars of services orientation (contextion, multifidelity, and multityping) and on generalization of the pillars of functional, procedural, and object-orient programming (see Fig. 1). Generalization of the existing programming paradigms leads to five types of service combinations (pipeline, domain, discipline, collaboration, and governance). Request services are multifidelity services, but provider services are multitype services. By multitypes of service signatures used in contextions a multi-multitype of service cooperation is determined. Therefore, multitype of a signature and a multi-multitype of contextions are classifiers of instances of service providers (providlets) and cooperations of service providers in the network, respectively. To the best of our knowledge there is no comparable true service-oriented system, programming language based the three pillars of service-orientation and its SVM.

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 request and opservices [15].

The SORCER architectural approach represents five types of net-centric multifidelity service cooperations expressed by request services created by the end users and executable codes of service providers by software developers. It elevates combination of contextions into the first-class elements of the SO federated process expression. The essence of the approach is that by making specific choices in grouping hierarchically provider services for contextions, we can obtain desirable dynamic properties from the SO systems we create with SML.

Thinking more explicitly about SO languages, as domain specific languages for humans than software languages for computers, may be our best tool for dealing with real world complexity. Understanding the principles that run across process expressions in SML and appreciating which language features and service virtual machines (SVMs) are best suited for which type of processes, bring these process expressions (request services in SML) to useful life. No matter how complex and polished the individual process operations are, it is often the quality of the operating system (SORCER) that determines the power of the computing system. The ability of presented metamodeling architecture with SML and SVM with its execution engine to leverage network resources as services is significant to real-world applications in two ways. First, it supports multi machine executable codes via opservices that may be required by SO applications; second, it enables cooperation of variety of computing resources represented by request services that comprise of opservices actualized by the multi machine network at runtime.

The software as a service (SaaS) approach spreads rapidly because it makes end users more productive. However, lack of service-oriented integration frameworks, forces end users to go back and forth endlessly between the component services (applications) they need and like, is disruptive because it corrodes productivity of complex service-oriented systems. The more services you have, the trickier it gets to move swiftly and meaningfully between them and integrate reliably into large distributed systems.

Embedded service integration in the form of *combined request services* in SML solves a problem for both system developers and end users. Embedded service integration is a transformative development that resolves the stand-off between system developers who need to innovate service integrations and end users who want their services to be productive in their integrated systems, not hold them back. Service integration is key to this, but neither system developers nor end-users want to be distracted by time-consuming integration projects.

The first rule of service-orientation in SORCER: 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 platform with SML and SVM supports the two-way convergence of modeling (top-down problem solving with context models) and programming (bottom-up problem solving with service pipelines and routines) – mogramming. The 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 [2, 4–7, 13].

Acknowledgments. 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-2019-4488. The effort is also partially supported by the Polish Japanese Academy of Information Technology.

References

1. Aziz-Alaoui, M., Cyrille Bertelle, C. (eds.): Emergent Properties in Natural and Artificial Dynamical Systems (Understanding Complex Systems). Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-34824-7
2. Burton, S.A., Alyanak, E.J., Kolonay, R.M.: Efficient supersonic air vehicle analysis and optimization implementation using SORCER. In: 12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSM AIAA 2012-5520 (2012)
3. Kao, J.Y., White, T., Reich, G., Burton, S.: A multidisciplinary approach to the design of a low-cost attritable aircraft. In: 18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA Aviation Forum 2017, Denver, Colorado (2017)
4. Kleppe A.: Software Language Engineering, Pearson Education (2009). ISBN: 978-0-321-55345-4
5. Kolonay, R.M., Sobolewski M.: Service ORiented Computing EnviRonment (SORCER) for large scale, distributed, dynamic fidelity aeroelastic analysis & optimization. In: International Forum on Aeroelasticity and Structural Dynamics, IFASD 2011, Paris, France, 26–30 June 2011 (2011)
6. Kolonay, R.M.: A physics-based distributed collaborative design process for military aerospace vehicle development and technology assessment. Int. J. Agile Syst. Manag. **7**(3/4), 242–260 (2014)
7. Kolonay, R.M.: MSTC Engineering - A distributed and adaptive collaborative design computational environment for military aerospace vehicle development and technology assessment. In: AIAA 2019-2992, AIAA Aviation Forum 2019, Dallas, Texas (2019)

8. Markov, A.A.: Theory of Algorithms. Keter Press (1971). (trans. by Schorr-Kon, J.J.)
9. O'Hearn, P.W., Tennent, R.D. (eds.): Algol-Like Languages (Progress in Theoretical Computer Science), vol. 1. Birkhäuser (1997). ISBN-10: 0817638806
10. Sobolewski, M.: Federated P2P services in CE environments. In: Advances in Concurrent Engineering, pp. 13–22. A.A. Balkema Publishers (2002)
11. Sobolewski, M.: Object-oriented meta-computing with exertions. In: Gunasekaran, A., Sandhu, M. (eds.), Handbook on Business Information Systems. World Scientific (2010). https://doi.org/10.1142/9789812836069_0035
12. Sobolewski, M., Kolonay, R.: Unified mogramming with var-oriented modeling and exertion-oriented programming languages. Int. J. Commun. Netw. Syst. Sci. **5**(9) (2012). http://www.scirp.org/journal/PaperInformation.aspx?paperID=22393(2012). Accessed 28 Oct 2019
13. Sobolewski, M.: Service oriented computing platform: an architectural case study. In: Ramanathan, R., Raja, K. (eds.) Handbook of Research on Architectural Trends in Service-Driven Computing, pp. 220–255. IGI Global, Hershey (2014)
14. Sobolewski, M.: Technology foundations. In: Stjepandić, J., Wognum, N., Verhagen, W.J.C. (eds.) Concurrent Engineering in the 21st Century, pp. 67–99. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-13776-6_4
15. Sobolewski, M.: Amorphous transdisciplinary service systems. Int. J. Agile Syst. Manag. **10** (2), 93–114 (2017)
16. Sobolewski, M.: Service-oriented mogramming with SML and SORCER. In: Proceedings of 9th International Conference on Cloud Computing and Services Science, Greece, 2–4 May, pp. 331–338. SCITEPRESS (2019). ISBN 978-989-758-365-0
17. Stults, I.C.: A multifidelity analysis selection method using a constrained discrete optimization formulation, School of Aerospace Engineering, Georgia Institute of Technology, Dissertation (2009). https://smartech.gatech.edu/handle/1853/31706. Accessed 28 Oct 2019
18. The MetaObject Facility Specification. https://www.omg.org/mof/. Accessed 28 Oct 2019
19. SORCER/TTU Projects. http://sorcersoft.org/theses/index.html. Accessed 28 Oct 2019
20. SORCER Project. https://github.com/mwsobol/SORCER-multiFi. Accessed 28 Oct 2019