Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 8, Number 1, pages 1–8, January 2019

True Service Orientation with SORCER

Michael Sobolewski
Air Force Research Laboratory, WPAFB, Ohio 45433
Polish Japanese Academy of IT, 02-008 Warsaw, Poland
Email: sobol@sorcersoft.org

***Abstract*—Service-oriented Mogramming Language (SML) is designed for service-orientation as UML was considered for object-orientation. SML is also an executable language in the SORCER platform based on service abstraction (everything is a service) and three d of service-orientation: *context awareness (contexting)*, *multifidelity*, and *multityping*. Context awareness is related to parametric polymorphism, multifidelity is a form of ad hoc polymorphism, and multityping is a net-centric form of type polymorphism. SML allows for defining polymorphic service systems that can reconfigure and morph service collaborations at runtime to definite an emergent form with distinct constraints and heuristics. Here, emergence of service system refers to the appearance of higher-level properties and behaviors of collaborating service federations that come from the collective dynamics of that collaborating networked services and activity of service morphers that manage multifidelities at runtime. In this paper the basic concepts of SML with the three design patterns of service collaborations are presented. Its runtime environment is introduced with the focus on the three pillars of service-orientation.**

Keywords*—true service orientation, consumer services, provider services, service mogramming language (SML), multifidelities, emergent systems, mograms, mogramming, SORCER

I. 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. Why? 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 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 communication and architecture.

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 [6]. Therefore, MADO is an appropriate domain for studying SOA [4, 6, 7].

There are several trends that are forcing system architectures to evolve due to complexity of problems being solved presently [8]. 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 [3], 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 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 service providers [9].

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 functional composition. A functional program is stateless but imperative programs usually take advantages of a shared state in an executing *subroutine* - a set of instructions that implement a subprocess within a program. Object-oriented (OO) programming is a convenience and ability to reason about operations implemented by methods (subroutines) with a common shared state represented by encapsulated variables. Being able to hide details of algorithms and their data structures can help reason about the logic of object collaboration such that each object manages its own state by own implementation of methods.

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 effective programming styles. Each programming paradigm introduces distinguishing principles of its programming model but also depends on its lower level paradigm. Therefore, the pillars of SO programming introduced in this paper are layered on pillars of OO, structured, and functional programming. The pillars of SO programming are focused on context awareness, multifidelity, and multityping for both service providers and service federations.

The Service-ORiented Computing EnviRonment (SORCER [7]) adheres to the true SO architecture based on well-defined service abstractions and three pillars of SO programming. Its unique SO approach distinguishes frontend (process expression) services from backend (process actualization) provider services (interface types), service providers, and federations.

A service consumer is a composition of frontend request services and a service provider is a composition of provider services 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 provider services. 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 *service mogram*, corresponds to a union of service collaborations, each represented by a component mogram. The union under governance of the federated mogram represents an actualized *federation* of service providers.

SO federalism is a compound model of governance with a central (containing) mogram, component mograms, and a network of service providers (citizens). The rules of federated mogram governance are realized by a *SO operating system* (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 federated service providers.

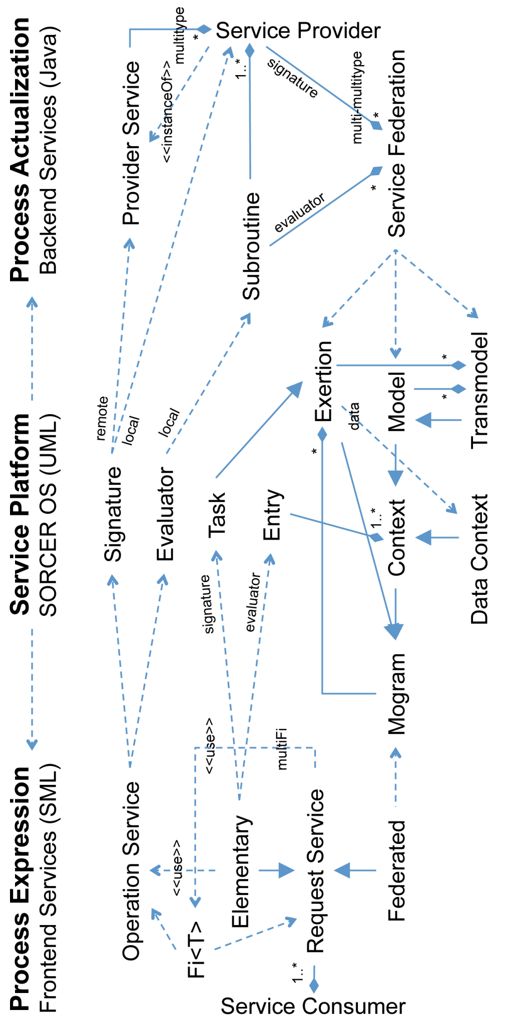

Fig. 1. The semantics of services in SML.

Mograms are structured from *elementary request services* (entries and tasks) and other mograms. Entries and tasks depend on operation services called *signatures*. Entries can also use various types of *evaluators* to invoke local subroutines. A signature is a *reference* to a remote/local operation of service provider. The unique signature-based architecture is about both 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 or just selecting the service fidelity from local to network when requesting a service provider. Service providers never communicate directly with each other in SORCER. For executing mograms its operating system creates communication networks of service federations at runtime, as the operating system's dynamic processor.

The remainder of this paper is organized as follows: Section 2 relates a SO computing platform to mogramming; Section 3 describes the basic syntax and semantics of SML with a mogramming example in Seciton 4; Section 5 relates SML to the OO implementation in SORCER; then we conclude with final remarks and comments.

II. Platforms, Operating Systems, and Mograms

A computing platform is an expression of a runtime process defined by programming language, operating system, and processor. An operating system (OS) is system software that manages a processor for the platform, executable codes of applications, and provides common services including a shell for executing and monitoring user applications. All user applications require a kind of operating system to run. With respect to various abstractions and granularity of user applications, various platforms have been developed over time. Granularity of computational units (instructions, commands, callable units, objects, services) and various forms of program compositions may differ but from the mathematical point of view we can consider them conceptually as generic functions - transformations or processes that map input data to output data. Using the abstraction of generic function as a *subroutine*, SO programming semantics can be generalized and differentiated from functional, structured, and OO programing as illustrated in Fig. 1. Hence, SO programming requires a corresponding operating system and programming environment (OS/language) as other programming paradigms. For example UNIX/C for structured programming and JVM/Java for OO programming.

A granularity of programs increases from a list of instructions, via collaboration of objects to a network-centric federation of service providers. A network-centric federation of remote/local services is the actualization of federated a request service (of *Request* type) that bind operation services (of *Signature* type) to remote service providers (of *Provider* type) and local objects created at runtime. Writing code as a list of low-level machine instructions is a terrible way for humans to reason and instruct computers. Objects have more humane semantics but still low granularity for dynamic network-centric service federations. Service providers are more like tools, and utilities that can be aggregated into service collaborations and expressed by the end users as higher-level instruction in service federations.

Two types of frontend services are distinguished: operation and request services. Operation services of *Signature* or *Evaluator* types refer to remote/local or local subroutines, correspondingly. Evaluators invoking subroutines are called *subroutine services*. A request service is either *elementary* or *federated*. Elementary ones use operation services but federated ones specify how provider and subroutine services are federated to collaborate. Note that an instruction in structured programming or a method in OO programming is an intrinsic component of programs. In SO programming a request service is a program but an operation service is a service handle that binds to a subroutine at runtime - no static dependencies to remote subroutines. Request services can be created with multifidelity signatures and entries to allow for selecting preferred subroutine fidelities at runtime. That type of service-orientation requires a relevant service modeling and/or programming (mogramming [5]) language and a corresponding operating system.

SORCER is a SO computing platform (implemented with objects – see the UML relationships in Fig. 1 and in Fig. 2), which provides SO mogramming with its *service operating system* (SOS) that interprets and runs frontend request services and dynamically manages corresponding backend federations of remote/local service providers as its SO processor. In SORCER, request services bind at runtime to created local objects or to proxy objects that are *created*, *registered*, and owned by service providers. SOS can provision missing service providers at runtime if specified so by mogram signatures [8].

In summary, a request service is either elementary or federated. An elementary service invokes a requested subroutine but federated one invokes the federation of providers and/or subroutines managed by SOS. A federated request service – *mogram* – is an expression of a service federation by one of the three federated design patterns:

1. *entry model* – is a declarative expression of interrelated multiple service entries (responses) composed functionally of dependent service entries in the model.
2. *exertion* – is an expression of hierarchically organized exertions and mograms. An elementary exertion (*Task*) executes a remote/local subroutine of a service provider. An exertion block is a service procedure and exertion job is an exertion composite. *transmodel* (*madomodel*) – is a model that hierarchically aggregates subordinate models as dependent disciplines.
3. *transmodel* (*madomodel*) – is a model that hierarchically aggregates subordinate models and exertions as potentially *coupled* disciplines.

Since either, a model or an exertion may comprise of component mograms, therefore, a mogram is a service expression of a multifidelity system of systems (mogram of mograms) that federates various dependent subsystems (exertions, models, and transmodels) into multiple fidelity projections of hierarchically nested service federations created and managed by SOS at runtime. Thus, each fidelity projection is an instance of the system of systems.

The primary challenge of the SO architecture is to allow the end user to use existing subroutines and service providers exposing service types in the network. The secondary one to create executable declarative and/or imperative federated request services. In other words, instead of invoking statically standalone service providers and subroutines or integrating them with APIs, the computing environment should allow the end users to create and execute directly in SML net-centric service federations. Therefore, SO operating system is required for mograms to manage and execute dynamic service federations - its SO federated processor.

III. Basics of Service Mogramming Language (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 (provider, and federation services) with three pillars of service-orientation: contexting, *multifidelity*, and *multityping.*

Contexting is a property of request services to aggregate and exchange *uniformly* the state of collaborating services with a generic data structure, called a *data context*. It is a form of parametric polymorphism for services – the *Context* type as the generic collection type of argument and return values for all provider services.

Multifidelity is the morphing activity making decisions about choices of service fidelities to be used by services at runtime. It is a form of ad hoc polymorphism that defines hierarchical control flow of executing services in mograms.

A *multitype* of a signature is a classifier of service providers in the network, but a multi-multitype of mogram is a classifier of service federations in the network. A set-theoretic subtyping of multitypes defines an inheritance hierarchy of service providers. *Multityping* is a net-centric form of type polymorphism with set-theoretic subtyping of multitypes. It defines the inheritance hierarchy of service providers and federations 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 the member 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*, the multityping relation is defined, as *N* is a subset of *M*, to mean that any signature of type *M* can be safely used in a context where a signature of type *N* is expected. The same applies to multi-multitypes that define the inheritance of service federations.

Conceptually, if a service mapping *f* produces an output *y*, when given an input *x* and there exist multiple implementations $fi_1, fi_2, \ldots, fi_n$ of *f* called fidelities to produce $y = f(x, fi_j, mFi_f)$, $j = 1, 2, \ldots, n$ such that $y = fi_j(x)$, for a multifidelity $mFi_f = <fi_1, fi_2, \ldots, fi_n>$, then *f* is called a *multifidelity function* with selectable fidelities in mFi_f. In SML, a multifidelity service is defined with domain specific multifidelities, potentially with morphers and a fidelity manager that use provided strategies (heuristics) to select the adequate fidelities implied by the current results and service data context.

A signature binds dynamically by multiple service types (*multitype*) to a service provider but a federated service binds (by multiple multitypes of signatures (*multi-multitype*) to a federation of service providers that can be also provisioned by a multi-multitype. Service providers can be considered as remote or local objects exposing implemented service types that are used by signatures as a multitype used for binding to a service provider in the network. In SML details of the input *x* are hidden by embedding all formal parameters into a single argument called *data context* used by all frontend services.

A service model *SM* in SML conceptually corresponds to a multifidelity functional system. A multifidelity function

$$f = (X, Y, fi(f), mFi_f)$$

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 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 if it does at least one of the following:

- takes one or more functional entries as arguments
- returns a functional entry as its result

A *service signature* in SML is an *operation service* referencing an operation of a provider service. It is declared by a type of provider *tp* and its operation *op*, to be evaluated in the scope of a current context of the service. An association <*op, tp*> is called a *service signature* and is denoted in SML by *sig*(*op, tp*). A return value of 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 all implemented service types $tp_1, \ldots, tp_s$, can be specified in a signature as the provider implementation 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 an instance of a service provider directly its multitype determines instances of service providers at runtime. Multityping is used to manage and reduce complexity and unpredictability of network comprised of replaceable remote component services with one another at runtime, as long as the multitype semantics of the component service is the same.

The network-centric semantics in SML is based on the concept of service multitype. If the operation type tp_1 is a class type then the signature works as a service provider constructor – creates an instance at runtime when the referenced service provider needs to be executed, otherwise SOS finds in the network the remote proxy of the service provider declared by its multitype.

A value $y \in Y$ of constant function (variable) *x* in SML is declared by a value entry *x* as follows:

val x = *val*(“*x*”, *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*) as an unordered collection of *val* entries is defines as follows:

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

and valuation of the entry *x* in a data context *c* as follows:

Object y = *value*(*dc*, “*x*”)

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

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

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

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

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

Note that multivariable entries of models that take functional entries as arguments create functionals (higher-order functions).

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

Object y = eval(mdl, "f")

or

Object y = eval(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 responses defined in the model 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 configuration dependencies (functional composition) 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 to decrease responses:

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

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

So far, we have defined in SML, elemntary services of *ent* and *sig* types and federated services of *model* type. 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 common arguments. The statement executing the operation *add* of service type *Adder*:

exec(*sig*("*add*", *Adder.class*), *context*(*val*("*x1*", *3.0*), *val*("*x2*", *1.0*), *val*("*x3*", *7.0*))

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

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. When a list of signatures is specified then a task is called a *batch task*.

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 subroutine modules 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 at runtime.

An exertion is a procedural/object-composite mogram – federated request service in SML [7]. A service task is an elementary service exertion used in compound exertions. A compound exertion is a set of exertions and/or mograms grouped together within the scope of SML operators: *block* or *job*. The exertion *block* is a concatenation of component mograms along with flow-control exertions such as conditional (*opt*, *alt*) and loop (*loop*) exertions. A *job* exertion is a hierarchically structured exertion (object composite) from component exertions and/or mograms, optionally with a provided control strategy. The SML semantics of *opt*, *alt*, and *loop* is the same as the UML operators used with interaction frames (combined fragments) in sequence diagrams.

Exertions can be used as values of functional entries in models and entry 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 generic *ent* functional operator, in most obvious cases, declares a service entry of the corresponding type according to used arguments for *ent*. Specialized entry operators (*val, call, lambda, neu, srv, and svr correspond to: value, call unit, lambda, neuron, service, and service variable*) can be used along with new *ent* subtypes.

A mogram m_{in} to be executed, is said to be *exerted*, by its corresponding service federation. Exerting is declared as follows:

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

The exerted m_{out} contains the result of evaluation and all net-centric information regarding its 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 executing directly:

Object y = *result*(mog_{out}, “x”)

An evaluation result c_{out} of 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 *service mogram* is a collection of interacting request services (*entries*, *tasks, models, and exertions*) that bind at runtime to a federation of service providers via mogram signatures. Multifidelity federations can morph during execution under control of the mogram morphers and its fidelity manager with the goal to return the best result in the evolving net-centric configuration - a morphing system (mogram) of systems (mogram’s fidelity projections).

IV. An Example of a Multifidelity Model in SML

To illustrate SML in action a simple 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), and five provider services (signatures in entries remote and in tasks local) referenced by service signatures used in entries and tasks of the model mdl.

```
// four entry multifidelity model with four morphed fidelities (mphFi)
// and corresponding morphers
mog mdl = model(inVal("arg/x1", 90.0), inVal("arg/x2", 10.0),
        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", mogFi(morpher3, t5, t4)), fi2, fi3, fi4, fi5,
        response("mFi1", "mFi2", "mFi3", "mFi4", "arg/x1",
"arg/x2"));

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")));

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"))));

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();
   if (fi.getSelectName().equals("t5")) {
      Double val = ((Double) value(context(value), "result/y"));
      if (val <= 200.0) {
         putValue(context(value), "result/y", val + 10.0);
         mgr.reconfigure(fi("mFi4","t4"));
      }
   } else if (fi.getSelectName().equals("t4")) {
      // t4 is a multiply task
      Double val = ((Double) value(context(value), "result/y"));
      putValue(context(value), "result/y", 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");
      }
   }
};

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

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

```
// 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));

// selecting 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));
```

The above example can be found in the *multiFi* branch of the SORCER project (http://sorcersoft.org/project/site/) in the module *examples* at *sml/src/test/main/java/mograms/ModelMultiFidelities*.

V. Object-oriented Model of the SORCER Platform

The relationship of the main SORCER types required to implement multifidelity services is depicted in the UML class diagram in Fig. 2. Services of the *Request* type are instances of two elementary subtypes: *Entry* and *Task,* and the federated request *Mogram* type. 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 are shown Fig. 2 in order to illustrate the architectural OO view of key SO concepts (*Fi<T>*, *Request*, *Mogram*, *Signature*, and *Provider*) all of the common *Service* type.

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

We distinguish three main categories of services: operation, elementary, and federated services. From the SO point of view creation of user-centric request services is the primary mogramming objective assuming that service providers implement multitypes with preferred programming paradigms and can be incorporated into service federations as subroutines to be bound to 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 indirectly to executable subroutines of evaluators and service providers, correspondingly.

Each service provider implements a multitype of service types. Each service types 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 any implementation available in the network. Service providers to differentiae from each other may implement complementary service types, for example, tag interfaces associated with implementation details. Complementary types can be registered with primary service types, then all to be used in signatures when looking up a service provider. Multityping of signatures is the concept of a service classifier of redundant provider instances in the network or instances with multiple implementations of the same multitype.

Fidelity is defined usually as "the quality or state given with strong assurance; accuracy in details". For a computing process, accuracy and fidelity have the same meaning and are used interchangeably. Similarly, "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. When selecting the fidelity level for collaborating subroutines in a service federation, it is important to appropriately balance the fundamental trade between cost and accuracy.

Multifidelities can be observable and observed. Therefore, the positive or negative feedback received regarding applied service fidelities from observable multifidelities 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 morphers, updates fidelities. 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 created 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

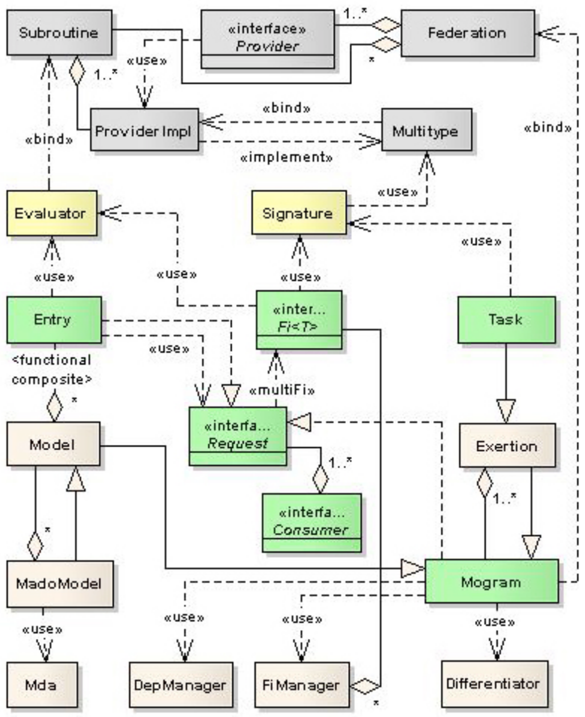

Fig. 2. The core SORCER types in support of SML.

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 [9]. 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.

VI. Conclusions

Markov tried to consolidate all work of others on effective computability. He has introduced the term of algorithm in his 1954 book Theory of Algorithms [1]. 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. In this paper a generic term, subroutine corresponds to an algorithm executed by a corresponding evaluator. 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. In this paper, service-orientation is proposed as a class of distributed emergent processes with multifidelity federated services.

The "everything is a service" semantics is introduced with multifidelity federated services – mograms – as SO process expressions, to be actualized by dynamic federations of service providers 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.

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 [9].

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 federated governance of net-centric and multifidelity service consumers that federate mograms representing SO applications created by the end users. It elevates net-centric, dynamic service providers (applications, tools, and utilities) into 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 (e.g. [4]).

Acknowledgment

This work was partially supported by the Air Force Research Lab, Aerospace Systems Directorate, Multidisciplinary Science and Technology Center, the contract number FA8650-16-C-2641, Collaborative Research and Development for Innovative Aerospace Leadership; and by the Polish-Japanese Academy of Information Technology.

References

[1] Markov, A.A. (1971) Theory of Algorithms, trans. by Schorr-Kon, J.J., Keter Press

[2] O'Hearn P.W. & Tennent, R.D. (eds), Algol-like Languages (Progress in Theoretical Computer Science), ISBN-10: 0817638806 Vol. 1, Birkhäuser; 1997

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

[4] 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

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

[6] 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. ¾

[7] 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

[8] 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

[9] 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

[10] SORCER Project (open source version). Available at http://sorcersoft.org/project/site/ (Accessed: August 10, 2018).