Moving Integrated Product Development to Service Clouds in the Global Economy
J. Cha et al. (Eds.)

doi:10.3233/978-1-61499-440-4-3

Unifying Front-end and Back-end Federated Services for Integrated Product Development

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

Abstract. Improvements in the design and manufacturing processes, and the related technologies that enable them, have led to significant improvements in product functionality and quality. However, the need for further improvements in these areas is needed due to increasing complexity of integrated product process development (IPD). Introduction of a new IPD project is more complex than most people realize and getting more complex all the time. Some of the complexity is due to rapidly changing and advancing technologies in underlying hardware and software, and the interplay of individual complex methods in system configurations. A strong IPD methodology, with intrinsically higher fidelity models to actualize the agile service-oriented design/manufacturing processes, is needed which can be continuously upgraded and modified. This paper describes a true service-oriented architecture that describes everything, anywhere, anytime as a service with the innovative service-oriented process expression (front-end services called exertions) and its dynamic and on-demand actualization (back-end service providers). Domain-specific languages (DSLs) for modeling or programming or both (mogramming) are introduced and their unifying role of front/back-end services is presented. Moving to the back-end of IPD systems front-end process expressions, that are easily created and updated by the end users, is the key strategy in reducing complexity of large-scale IPD systems. It allows for process expressions in DSLs to become directly available as back-end service providers that normally are developed by experts and software developers that cope constantly with the compatibility, software, and system integration issues that become more complex all the time.

Keywords. SORCER, SOA, SOOA, exertions, var-models, service-oriented mogramming, IPD, concurrent engineering

Introduction

The increase in complexity of integrated product development (IPD) systems is directly related to sweeping changes in the structure and dynamics of human creativity, increasing competiveness, and interdependence of the global economic and social system. Complexity of existence has increased and is increasing, therefore the development of robust and optimal products and processes in today's environment of step-by-step reductions in cycle time, cost take-out, and improved performance, diminishes the capabilities of today's design systems, which directly impacts life cycle costs. Since complex products are designed, manufactured, and serviced at

geographically disparate locations, the need to improve IPD of always moving, changing, and adopting product data and business logic incorporating has to be constantly reevaluated [1]. Therefore, the requirement for a federated service-oriented architecture, which exploits the concept of front/back-end services and permits context-aware views of composite processes is required to seamlessly integrate relevant technologies to enable rapid instantiation and simulation-based evaluations of products and processes, with the best-in-class applications, tools, and utilities as services.

As a result, the methodology of product development needs to be changed. A strong, dependable IPD methodology with higher fidelity models to perform the conceptual design and compute the information required for the modeling and simulation analysis has to be considered which can be continuously upgraded and modified. Such a methodology should lead to a significant reduction in cost and development time without scarifying any of the desired product specifications. Moreover, it should be simple to comprehend, easy to implement and easily adaptable to a diverse nature of product development activities. Transdisciplinary concurrent engineering (TCE) is the approach, which provides all the above capabilities, and it can prove to be the agile service-oriented solution unifying front/back-end services. Moreover, it embodies the belief that quality is inbuilt in the product, and that it (quality) is a result of continuous improvement of a federated service-oriented process.

The TCE system envisages providing a whole range of software tools and services that will support an economical and an optimum product design. In addition to a multitude of CAD/CAE/CAM tools, there will be a host of other front-end tools for programming, modeling, project management, process planning etc.

Networked product developers may use different platforms appropriate for their tasks. In a general case, one developer can use a collaborative federation of services, and there is a need to use the best-in-class engineering applications, tools, and utilities running under different operating systems in the network. On the other hand, the coordination of complex tasks involving many humans and a long series of interactions requires a homogeneous operating system—a kind of service-oriented metaoperating system [2]. The metaoperating system enables distributed collaborative analysis and hierarchical design space explorations. Creative at runtime front-end integration of resources used by a product developer directly is a key enabler for performing higher fidelity designs.

In the Service-ORiented Computing EnviRonment (SORCER), such a metaoperating system is called the SORCER Operating System (SOS). The SOS consists of the collection of distributed service providers as network modules for interpreting and executing front-end services, called exertions, by creating, provisioning, and managing federations of back-end service providers at runtime. Roughly speaking, the SOS, through its system services, provides connectivity, location transparency and network-wide access in the SORCER heterogeneous service environment [3].

The service-oriented process expression (front-end) and its actualization (back-end) of the SORCER computing platform enables collaborative design across organizational boundaries and full usage of all compute resource in the network ranging from desktops to high performance computing machines. This is the key to executing the process within the same amount of time and resources as a traditional conceptual design process. The SORCER service-oriented architecture describes everything, anywhere, anytime as a service.

This paper introduces the SORCER platform that provides a service-oriented modeling and/or programming (mogramming) environment with its operating system that runs front-end services (process expressions) and dynamically manages corresponding back-end federations of local and remote service providers [3]. A layered view of SORCER services is depicted in Fig. 1. Three types of front/back-end unification that allows for moving to back-end of the IPD system front-end process expressions created and updated easily by the end users are presented in the following three Sections.

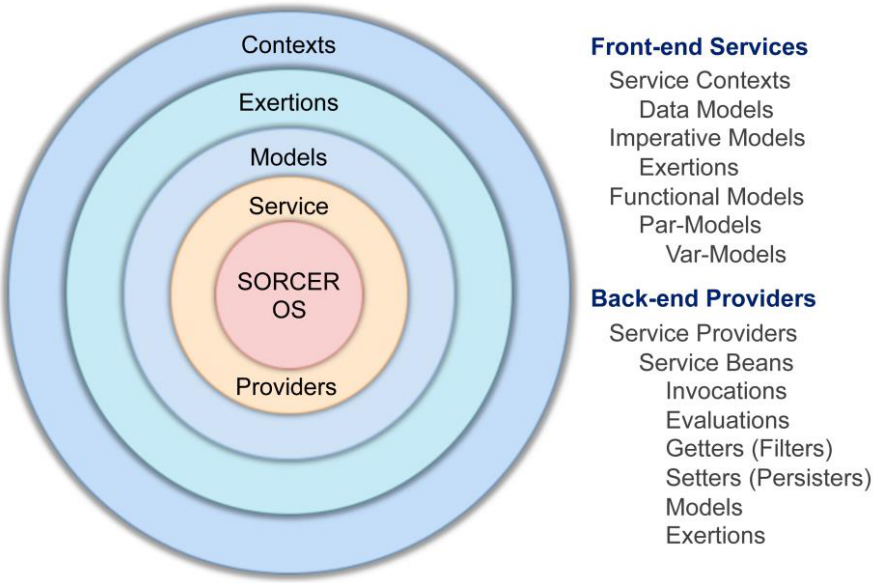

Figure 1. The layered view of basic SORCER front-end services: contexts, exertions, and models and back-end service providers with its operating system (SOS).

1. Unification of Service Data and Control: Service Contexts

In SORCER, *data as service* (DaaS) and *control as a service* (CaaS), are based on the concept that data and control strategy can be provided on demand to the service requestor or service provider regardless of geographic or organizational separation of provider and requestor. Additionally, the emergence of SORCER operating system (SOS) has rendered the actual platform on which the data resides irrelevant. This approach has enabled the service-oriented programming and modeling with a concept of *service context* as a form of interoperable dynamic associative memory as a service.

Traditionally, most enterprises have used data stored in a self-contained repository, for which software was specifically developed to access and present the data in a human-readable form. One result of this paradigm is the bundling of both the data and the software needed to interpret it into a single package. As the number of bundled software/data packages proliferated and required interaction among one another, next layer of interface was required. These interfaces, collectively known as enterprise application integration (EAI), often tended to encourage vendor lock-in, as it is generally easy to integrate applications that are built upon the same foundation

technology. The result of the combined software/data consumer package and required EAI middleware has been an increased amount of software for organizations to manage and maintain, simply for the use of particular data.

An exertion is a service-oriented process expression in exertion-oriented language (EOL) that specifies a service federation created at runtime by the corresponding operating system [4]. A *task exertion* (or simply a *task*) is an elementary service provided by a single service provider. A *batch task* (or simply a *batch*) is a concatenation of elementary tasks with a shared service context. A *job exertion* (or simply a *job*) is a service composition that represents a hierarchically organized collaborative service federation (workflow). A *block exertion* (or simply a *block*) is a concatenation of exertions having common block scope for its control flow.

The exertion's data called *service context* describes the data that tasks, batches, jobs, and blocks work on and create. A *data context*, or simply a *context*, is a data structure that describes a service provider's namespace along with related data. Conceptually a data context is similar in structure to a file system, where paths refer to objects instead files. A provider's namespace (object paths) is controlled by the provider vocabulary (attributes) that describes data structures in a provider's namespace within a specified service domain of interest. A requestor submitting an exertion to a provider has to comply with that namespace as it specifies how the context data is interpreted and used by the provider independently where the data is coming from. A *control context* is a specialization of service context for defining a control strategy for executing exertions by the SOS.

A *service parameter* (for short a *par*) is a special kind of variable, used in service contexts to refer to one of the named pieces of data to a service used as either the *passive* value or the *active* value. The active value is the value calculated by a par's procedural attachment called an *invoker*.

A service variable (*var*) is a collection of triplets: { <*evaluator*, *getter*, *setter*> }, called var *fidelities*, where:

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

Collections of pars and vars within a service context constitute *par-models* and *var-models* that can be used in exertions as data or as standalone modeling service providers. Var-models are instances of the VarModel class which subclasses from the ParModel class (see Fig. 2). Therefore all functionality of service contexts and par-models is inherited by var-models. Invokers of par-models are used as procedural attachments for both par-models and var-models. In particular var-models can be reconfigured at runtime as needed by their related pars, for example to update fidelities of vars at runtime.

In EOL a *service signature* is a handle to a service provider that determines a service invocation on the provider [5]. The signature usually includes the *service type*, *operation* of the service type, and expected *quality of service* (QoS). While exertion's signatures identify (match) the required collaborating providers in service federations, the *control context* defines for the SOS a strategy how and when the signature operations are applied to the data context. The collaboration specifies a collection of cooperating providers—the *exertion federation*—identified by all nested signatures of

the exertion. Exertions encapsulate explicitly data, operations, and control strategy for the collaboration. The signatures are dynamically bound to corresponding service providers—members of the exerted collaboration.

A service context (either data or control) can be specified in exertions explicitly by the service requestor or can be referenced by the requestor (using append signatures) to any combination of context providers called *contexters* that append requested runtime data as specified by provided patterns in exertion's data contexts.

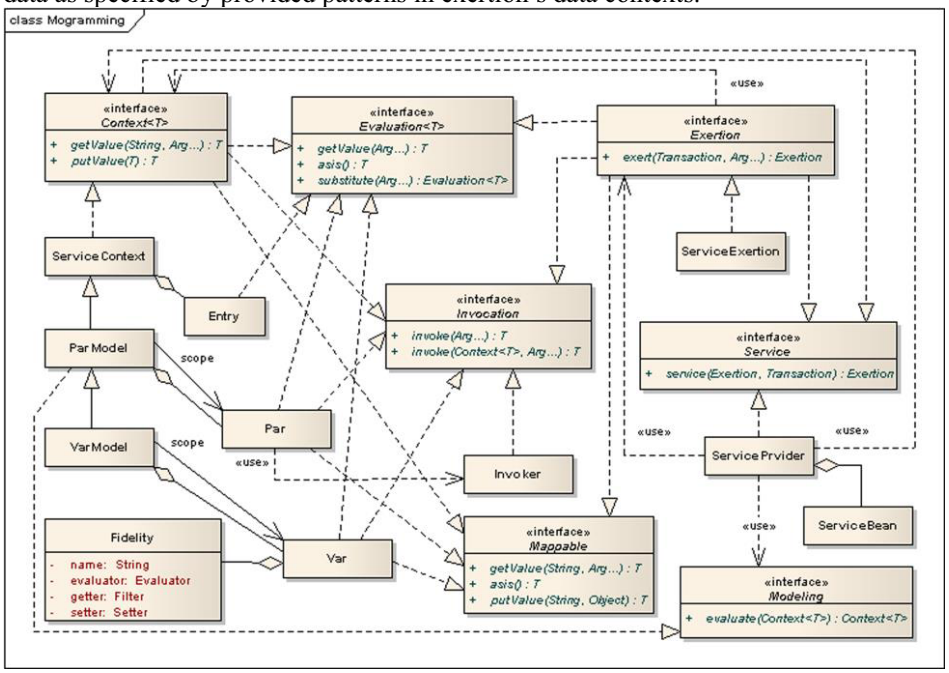

Figure 2. Top-level Java interfaces of the SORCER programming and modeling environment.

All SORCER service contexts: data context, control context, and modeling contexts (par-model and var-model) implement the `Context` interface as the common interoperability structure for system services, application services, and third party context-aware services (see Fig. 2). This commonality provides for context-awareness in service-oriented mogramming and wide-open standardized data transfer between service requestors, providers, the SOS, and third party services in the SORCER expanded environment. The same `Context` interface provides for data unification of front-end (process expression) and back-end (process actualization) of all services (exertions and service providers).

Context-aware communication and computing allows continuous adaptation of collaborative service federations to the constantly changing distributed service contexts specifying runtime data, control strategies, and service configurations. Hierarchically organized context data in exertions is the information characterizing the situation of a participating entity in the federation and providing information about the present status to federating members in the constantly changing environment. An entity is a person or service relevant to the collaboration between the users and service providers that depend on the current state of exertion contexts including those shared and persisted in the network. Context awareness enables customization or creation of the federated

applications that match the preferences of the individual user and participating services based on current hierarchically organized context for complex adaptive analyses or space exploration problems. Exertions with signatures of the append type (DATA_APD or CONTROL_APD) can update their current contexts from collaborating data/control-oriented services or accept relevant default values at runtime.

In particular, control context awareness in SORCER is related to control flow and asynchronous execution expressed by control context of exertions. Parallel (Flow.PAR) or sequential (Flow.SEQ) control flow of job exertions, synchronous (Access.PUSH) or asynchronous (Access.PULL) access to service providers, or provisioning new services (Provision.YES) can be updated by the requestors or collaborating providers at runtime depending on availability and state of the currently executing service federation. On the one hand, the modeling context awareness in par-oriented modeling allows for preferred use of procedural attachment to update data/control contexts and to reconfigure var models. On the other hand, the modeling context awareness allows for preferred choices of var fidelities in var-models adjusted at runtime to corresponding computation resources and strategies used by var evaluators.

Context awareness in SORCER can be used quite differently under different conditions, and layers, such as selecting preferred service providers and models in federations, proxy registration updates, currently used provider's wire protocols, leasing resources and transaction management, network garbage collection, and security preferences. With uniform interoperability of context-aware data and control strategies across the SORCER environment, the SOS manages complex structured and behavioral dependencies and makes its service federations self-aware of adaptivity to a changing computing environment by interpreting all contexts across every service federation as active distributed associative memory.

The managed structured (configuration) dependencies by the SOS refer to nested compositions of exertions. The SOS manages the behavioral (execution) dependencies as follows [3]:

1. Control contexts in exertions
2. Calling an executable code
3. Calling a method on an object
4. Calling a service.
    - invokers of a par-model (invocation processor).
    - evaluators, getters, and setters of var fidelities (evaluation processor).
    - service providers (subclasses of the ServiceProvider class).
    - service beans (components of service providers).

A service container is configured for deployment/provisioning [6] by dependency injection with a corresponding deployment context specified in a configuration file. This context configures basic properties of a provider including its service beans, object proxy, wire protocol, thread pools, exertion space connectivity, security properties, proxy verifier, etc. A number of deployment parameters can be updated at runtime or the whole context can be updated as needed for a provider to be re-provisioned dynamically for a new deployment configuration.

A service container (ServiceProvider in Fig. 2) allows for deploying service beans that implement service types as configurable service providers. In particular, service contexts, exertions, and par/var models are service beans so can be directly deployed as providers in the engineering/manufacturing application service cloud. Therefore front-end services specified in DSLs can be used to deploy back-end service providers. In

Fig. 3 the same exertion is used as a front-end service *E-fe* (*E-fe* is executed by the SOS shell) and a back-end exertion *E-be* (a copy of of *E-fe*) is executed by exerting the task exertion *T-fe*. In that case, the provider SP6 managing the bean *E-be* creates the same federation as the SOS shell for executing *F-fe*.

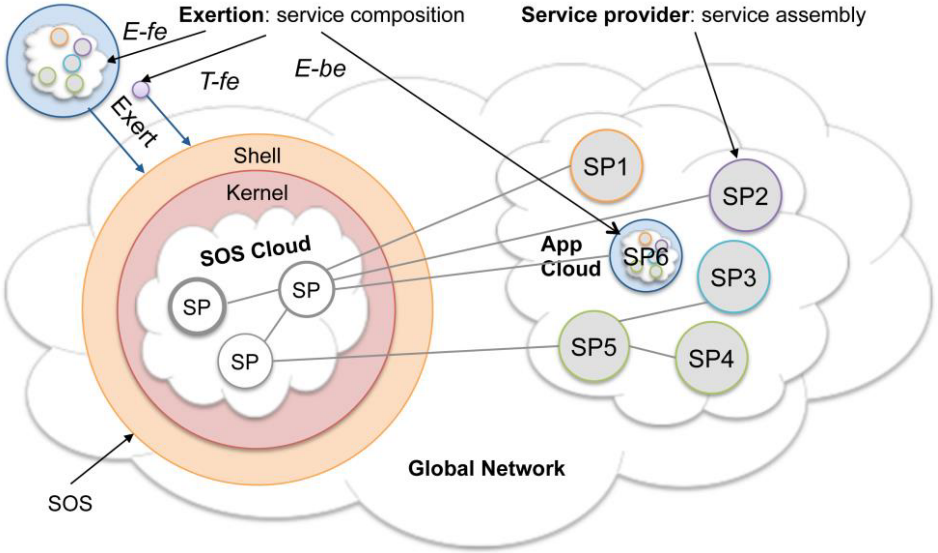

Figure 3. A front-end exertion *E-fe* is executed directly by the SOS shell and another its instance *E-be* is deployed as the service provider SP6. The provider SP6 can be exerted with a front-end task *T-fe* that executes the same way as direct execution of the front-end exertion *E-fe*.

2. Unification of Local and Remote Services: Service Signatures and Exertions

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

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

1. sig(<selector>, Class | <object>, <QoS>)
2. sig(<selector>, <service type>, <QoS>)

where Class is a Java class (for an object signature) and <service type> (for a net signature) is a Java interface. Object signatures define local providers and net signatures define remote providers by unifying local/remote services in the same exertion.

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

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

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

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

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

Exertions as services have hierarchically organized data contexts (properties that describe the service data), control contexts (properties that describe the service control strategy), and associated service providers known via service signatures. For convenience tasks, batches, jobs, and blocks are defined with the task, batch, job, and block operators as follows:

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

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

Exertions encapsulate explicitly *data*, *operations*, and a *control strategy* for the collaboration. The SOS dynamically binds the signatures to corresponding service providers—members of the exerted federation. The exerted members in the federation collaborate transparently according to the exertion's *control strategy* managed by the SOS. The SOS invocation model is based on the *Triple Command Pattern* that defines the federated method invocation (FMI) [7].

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

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

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

where, entries define a substitution for the exertion closure.

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

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

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

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

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

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

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

```
get(Context, path {, path}) :Object
```

or assigned with the put operator:

```
put(Context {, entry(path, Object) }):Context
```

Exertion-oriented programming (EOP [5]) is a service-oriented programming paradigm using *service providers* and *exertions*. Exertions can be created with textual language (*netlets*), API (*exertlets*), and user agents that behind visual interactions create exertlets. Netlets are interpreted scripts and executed by the network shell nsh of the SORCER Operating System (SOS). Invoking the *exert* operation on the exertlet (Java object) returns the collaborative result of the requested service federation. Netlets are executed with a SORCER network shell (nsh) the same way Unix scripts are executed with any Unix shell [8].

In EOL service providers are uniformly accessed through two types of references: *class* and *interface* signatures. *Class* and *interface* signatures are also called *object* and *net* signatures correspondingly. The former is used for specifying local service, the

latter for network services. Therefore, any combination of object and net signatures can unify both local and remote services within the same exertion that refers to the corresponding service federation managed by the SOS.

3. Unification of Procedural and Declarative Services: Exertions and Models

Usually computing and business processes are distinguished as semantically different ones. On the one hand a computing process is an instance of a computer program that is being executed. A computer program, or just a program, is a sequence of instructions, written to perform a specified computation with a computer. On the other hand a business process is a collection of related, structured activities or tasks that produce a specific service or product for a particular requestor or requestors.

A project can be broken into tasks then each task can be broken down into assignments that have a defined start and end time for completion. A collection of assignments on a project puts the task under execution. Project, task, and assignment dependency that specifies how they rely on each other to execute the project requires a control strategy. The ill-defined strategy can lead to the stagnation of a project when many tasks cannot get started unless others are finished correctly.

In service management, a service is an activity that needs to be accomplished within a defined period of time or by a deadline to work towards domain-specific goals. In service-oriented approach everything anytime anywhere is considered as a service. That means that either a computer program or business process can be uniformly organized hierarchically from services. In that approach all steps of the process expression and its actualization are uniform services.

Regular thinking is that a service requestor asks for a provider's service so services are always actions of providers (that exist at the back-end). Now, if everything is a service then the service request is a service as well. But services are usually created and composed (aggregated) at the back-end. That approach requires always programming new service providers by experts and software developers (low level programming—executable codes). In SORCER the back-end programming of composing services is usually shifted to the front-end programming by the end users—not professional programmers. Usually, a service written at the back-end and the front-end are quite different is style and semantics so the term *exertion* is referred to a front-end service program—requestor's service. SORCER introduces exertion-oriented language and par/var-oriented modeling languages (mogramming at the front-end, similarly to shell programming, for example, in Unix).

In exertion-oriented programming process expressions are called *exertions*. An exertion exerts the abilities of a service federation to perform a service (job and block exertions are business projects; batch exertions are business tasks; elementary task exertions are business assignments). In object-oriented programming everything is an object, so for example an instance of a class is an object and the class is an object as well. By analogy in service-oriented programming, an instance of exertion—a service federation—is a (back-end) service and the exertion itself is a (front-end) service. Therefore an exertion is a classifier of its service federations like in object-oriented programming a class is a classifier of its instances.

The exertion-oriented programming is drawn primarily from the procedural semantics of a routine but par/var-oriented programing from the semantics of a function composition of declarative service variables. In every computing, process

variables represent data elements and the number of variables increases with the increased complexity of problems being solved. The value of a computing variable is not necessarily a part of an equation or formula as in mathematics. In computing, a variable may be employed in a repetitive process: assigned a value in one place, then used elsewhere, then reassigned a new value and used again in the same way. Handling large sets of interconnected variables for transdisciplinary computing requires adequate programming methodologies.

A *service parameter* (for short a par) is a special kind of variable, used in service contexts to refer to one of the named pieces of data to a service used as either the passive value or the active value. The active value is the value calculated by a par's procedural attachment only when requested. Therefore, each par has an argument (value) associated with a name such that its name is a path in the associated service context and the value of the path in the context is the par itself. However, the value of par is to-be the result of evaluation:

Evaluation#getValue() or invocation Invocation#invoke(Context);

otherwise the par's value is as-is. The parameter Context in invoke(Context) refers to the context to be appended to the current context associated with the par, if any. The current context associated with a par defines the scope of its invoker's formal parameters. Therefore, invokers play a role of procedural attachment in service contexts and context-based models.

Note that par values are defined as above in all Context types, however values of other objects of Evaluation or Invocation types (not pars) are returned as-is in ServiceContexts, but in Modeling contexts both pars and all other objects implementing Evaluation or Invocation types are returned with to-be semantics. As-is and to-be context semantics are the major differentiators between ServiceContext type and Modeling types (par-models and var-models [3]).

A service variable (*var*) is a collection of triplets:{ <*evaluator*, *getter*, *setter*> }, where:

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

The var value is invalid when the current evaluator, getter, or setter is changed, current evaluator's arguments are changed, or the value is undefined. VOP is a programming paradigm that uses vars to design var-oriented multifidelity compositions. A triplet <*evaluator, getter, setter*> is called a var fidelity. It is based on dataflow principles where changing the value of any argument var should automatically force recalculation of the var's value. VOP promotes values defined by selectable var fidelities and their dependency chains of argument vars to become the main concept behind any processing.

Evaluators, getters, and setters can be executed locally or remotely. An evaluator may use a differentiator to calculate the rates at which the var quantities change with respect to the argument vars. Multiple associations of <evaluator, getter, setter> can be used with the same var allowing var's fidelity. The semantics of the *value*, whether the var represents a mathematical function, subroutine, coroutine, or data, depends on the evaluator, getter, and setter currently used by the var. The var dependency chaining

provides the integration framework for all possible kinds of computations represented by various types of evaluators including exertions.

Var-Oriented Modeling is a modeling paradigm using vars in a specific way to define heterogeneous var-oriented models, in particular large-scale multidisciplinary models including response, parametric, and optimization models. The programming style of VOM is declarative; models describe the desired results of the output vars, without explicitly listing instructions or steps that need to be carried out to achieve the results. VOM focuses on how vars connect (compose) in the scope of the model, unlike imperative programming, which focuses on how evaluators calculate. VOM represents models as a series of interdependent var connections, with the evaluators/filters between the connections being of secondary importance.

A *var-oriented model* or simply *var-model* is an aggregation of related vars. A var-model defines the lexical scope for var unique names in the model. Three types of models: response, parametric [11], and optimization [12] have been studied to date. These models are declared in VML using the function composition syntax and possibly with EOL and the Java API to configure the vars.

The *inputvar* is typically the variable representing the value being manipulated or changed and the *outputvar* is the observed result of the input vars being manipulated. If there is a relation specifying output in terms of given inputs, then output is known as an "output var" and the var's inputs are "argument vars". Argument vars can be either output or input vars. A function composition of a var is a way to combine simple argument vars to build more complicated ones. Like the composition of functions in mathematics, the result of each var is passed as the argument of the next, and the result of the last one is the result of the whole. The functions of the model correspond to fidelities of vars. A single var can define multiple functions—multiple fidelities.

The central exertion principle is that a computation can be expressed and actualized by the interconnected federation of simple, often uniform, and efficient service providers that compete with one another to be *exerted* for their services in the dynamically created federation. Each service provider implements multiple actions of a cohesive (well integrated) service type, usually defined by an interface type. A service provider implementing multiple service types provides multiple services. Its service type complemented by its QoS parameters can identify functionality of a provider. In an exertion-oriented language (EOL) a *service exertion* can be used as a closure over free variables in the exertion's data and control contexts. In exertion-oriented programming everything is a service. Exertions can be used directly as service providers as well (see Fig. 3).

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

In Fig. 4 three service clouds are depicted that collaborate for the execution of front-end exertion *E-fe*. The SOS shell by exerting *E-fe* with services of the SOS cloud unifies the front-end federation specified by *E-fe* with federations created by back-end exertions (as evaluators) in vars of models in the model cloud.

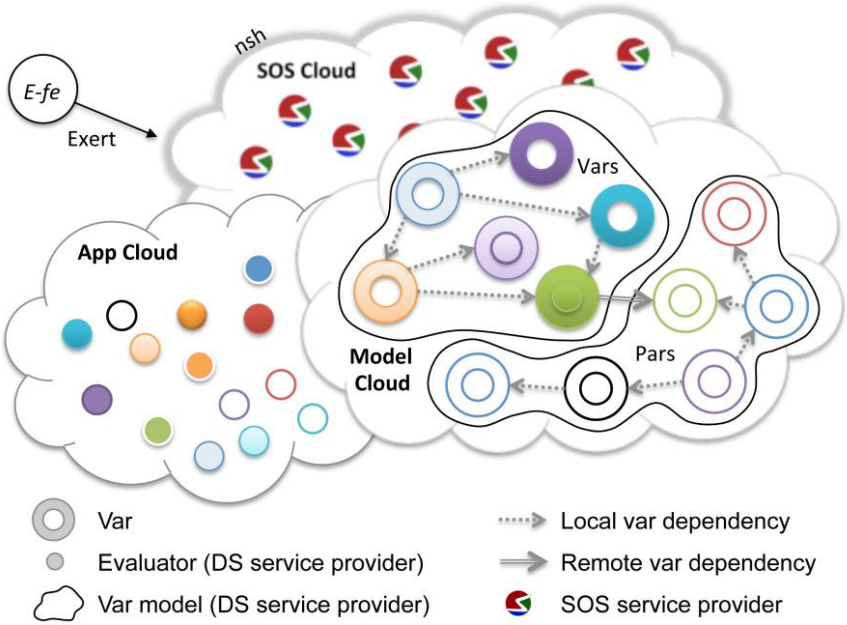

Figure 4. Managing transdisciplinary complexity with convergence of service-oriented modeling and programming (top: SOS service providers; bottom-left: service providers and exertion evaluators in the application cloud; bottom-right: models as service providers in exertions with local evaluators and remote evaluators in the application cloud).

4. Conclusions

Data and control interoperability is exemplified in SORCER via service contexts (DaaS and CaaS) as associative local/distributed memory defined explicitly by requestors in exertions (front-end services) or provided by contexters (back-end services). Data and control contexts return values directly but active service contexts in the form of par- and var models return results of invocations or evaluations respectively. The former provides values by procedural attachment, the latter by function compositions of var fidelities.

All front-end services: contexts, models, and exertions can be used as process expressions but also can be used as process actualizations (service providers). Actualization of front-end services is done by dependency injection of service beans (contexts, models, exertions, and business objects exposing SORCER service types) into a generic service provider container (ServiceProvider). Moving to back-end easily created and updated exertions by the end users is the key strategy in reducing complexity of IPD systems. It allows for exertions, contexts, and models to become directly available as back-end service providers that normally are developed by experts and software developers that cope constantly with the compatibility, software, and system integration issues that become more complex.

With object and net signatures, local or remote service can be mixed and unified by the same exertion. Just by replacing in an exertion signature a provider's class with its implemented interface the service is becoming remote and vice versa.

The SORCER platform integrates three programming styles: context-driven, exertion-oriented (procedural) programming, and par/var-oriented (declarative) modeling. The SORCER platform has been successfully deployed and tested for the engineering mogramming in multiple applications at AFRL/WPAFB [3, 9, 10, 11, 12].

Acknowledgment

This work was partially supported by Air Force Research Lab, Aerospace Systems Directorate, Multidisciplinary Science and Technology Center, the contract number F33615-03-D-3307, Algorithms for Federated High Fidelity Engineering Design Optimization and the National Natural Science Foundation of China (Project No. 51175033).

References

[1] R.M. Kolonay, Physics-Based Distributed Collaborative Design for Aerospace Vehicle Development and Technology Assessment. In: C. Bil et al. (eds.) *Proceedings of the 20th ISPE International Conference on Concurrent Engineering*, IOS Press, 2013, pp 198-215, http://ebooks.iospress.nl/publication/34808, Accessed 15 March 2014.

[2] M. Sobolewski, Object-Oriented Metacomputing with Exertions, In: Gunasekaran A, Sandhu M (eds.) *Handbook On Business Information Systems*, World Scientific, Singapore, 2010.

[3] M. Sobolewski (2014) Service Oriented Computing Platform: An Architectural Case Study. In R. Ramanathan and K. Raja, *Handbook of Research on Architectural Trends in Service-Driven Computing*, Vol. 1, Chapter 10. Hershey, PA: IGI Global, 2014. doi:10.4018/978-1-4666-6178-3.

[4] M. Sobolewski, Exerted Enterprise Computing: From Protocol-Oriented Networking to Exertion-Oriented Networking, In: Meersman R et al. (eds.) *OTM 2010 Workshops, LNCS 6428*, 2010, Springer-Verlag Berlin Heidelberg, pp 182– 201.

[5] M. Sobolewski Exertion Oriented Programming, *International Journal on Computer Science and Information Systems*, vol. 3, no. 1, (2008) pp 86-109.

[6] M. Sobolewski, Provisioning Object-oriented Service Clouds for Exertion-oriented Programming. *The 1st International Conference on Cloud Computing and Services Science, CLOSER 2011*, Noordwijkerhout, the Netherlands, 7-9 May 2011, SciTePress Digital Library.

[7] M. Sobolewski, Metacomputing with Federated Method Invocation, In: Hussain MA (ed.) *Advances in Computer Science and IT*, In-Tech, Rijeka, (2009) pp 337-363.

[8] M. Sobolewski, R.M. Kolonay Unified Mogramming with Var-oriented Modeling and Exertion-oriented Programming Languages, *Int. J. Communications, Network and System Sciences*, (2012) 5, 579-592. Published Online September 2012 (http://www.SciRP.org/journal/ijcns)

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

[10] S.A. Burton, E.J. Alyanak, R.M. Kolonay, 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*, 17-19 September 2012, Indianapolis, Indiana (AIAA 2012-5520).

[11] M. Sobolewski, S. Burton, R. Kolonay, Parametric Mogramming with Var-oriented Modeling and Exertion-Oriented Programming Languages. In: Bil C et al. (eds.) *Proceedings of the 20th ISPE International Conference on Concurrent Engineering*, IOS Press, 2013, pp 381-390, http://ebooks.iospress.nl/publication/34826, Accessed on: March 9, 2014

[12] M. Sobolewski, R. Kolonay, Service-oriented Programming for Design Space Exploration, In: Stjepandić J et al. (eds.) *Concurrent Engineering Approaches for Sustainable Product Development in a Multidisciplinary Environment*, Springer-Verlag London, 2013, pp 995-1007.