PROVISIONING OBJECT-ORIENTED SERVICE CLOUDS FOR EXERTION-ORIENTED PROGRAMMING

Michael Sobolewski
SORCERsoft.org
sobol@sorcersoft.org

Keywords: Process expression, Metacomputing, cloud computing, Service object-oriented architectures, Service provisioning, var-modeling, var-oriented programming, Exertion-oriented programming.

Abstract: Each computing system requires a platform that allows software to run. The computing platform includes a programming environment to create application software with a coherent operating system and processor. Each platform's programming environment reflects a relevant abstraction, and usually the type and quality of the abstraction implies the complexity of problems we are able to solve. The *Service ORiented Computing EnviRonment* (SORCER) targets service abstractions for metacomputing complexity in globally distributed systems. SORCER metaprograms—*var-models*, *var-oriented,* and *exertion-oriented* programs—with the abstraction of a cloud processor, are executed on the network by the SORCER operating system (SOS). SOS manages the cloud processor using autonomic provisioning of service providers in the service cloud (service virtualization) and allocates compute resources in the platform cloud (platform virtualization) to satisfy QoS (quality of service) required by provisioned providers. Thus, the cloud processor and its both clouds are maintained by SOS and can also be updated on-the-fly to run metaprograms as collaborating net services. The QoS and SLA (service level agreement) required by service requestors is expressed directly in the metaprogramming language. While the service cloud consists of service providers that are implemented using conventional platforms and languages, for example the Java platform, the cloud requestors—metaprograms—are written in the new metaprogramming languages. The SORCER service platform is described in this paper as the coherent metacomputing architecture for a fusion of *service requestors* and a *virtualized cloud processor.*

1 INTRODUCTION

As we look forward to metacomputing (Metacomputing, n.d.), globally distributed physical and virtual machines should play a major role in architecting very large service-oriented systems. While there is a significant interest in high performance, grid (Foster, et al., 2001), and cloud computing (Linthicum, 2009), much of this interest is the artifact of the hype associated with them as most of this computing is predominantly reduced to executable files. Without understanding the behavior of metacomputing architectures and related applications, it is unclear how future service-oriented systems will be designed based on these architectures. Thus, it is greatly important to develop a complete picture of metacomputing applications and learn the architectures they require.

In computing science the common thread in all computing disciplines are *process expression* and *actualization of process expression* (Fant, 1993), for example:

1. An *architecture* is an expression of a continuously acting process to interpret symbolically expressed processes.
2. A *user interface* is an expression of an interactive human-machine process.
3. A *mogram* (Kleppe, 2009) (which can be *program* or *model*) is an expression of a computing process.
4. A *mogramming* (*programming or modeling*) *language* is an environment within which to create symbolic process expressions (mograms).
5. A *compiler* is an expression of a process that translates between symbolic process expressions in different languages.
6. An *operating system* is an expression of a process that manages the interpretation of other process expressions.
7. A *processor* is an actualization of a process.

8. An *application* is an expression of the application process.
9. A *computing platform* is an expression of a runtime process defined by the triplet: *domain*—mogramming language, *management*—operating system, and *carrier*—processor.
10. A *computer* is an actualization of a computing platform.
11. A *metamogram* (*metaprogram* or *metamodel*) is an expression of a metaprocess, as the *process of processes*.
12. A *metamogramming language* is an environment within which to create symbolic metaprocess expressions.
13. A *metaoperating system* is an expression of a process that manages the interpretation of other metaprocess expressions.
14. A *metaprocessor* is an actualization of the metaprocess on the aggregation of distinct computers working together so that to the user it looks and operates like a single processor.
15. A *metacomputing platform* is an expression of a runtime process defined by its *metamogramming language*, *metaoperating system*, and *metaprocessor*.
16. A *metacomputer* is an actualization of a metacomputing platform.
17. *Cloud computing* is an expression of meta-processes consolidated on a metaplatform with *virtualization of its services and required computing platforms*.

Obviously, there is an essential overlap between the domains of computer science and information technology (IT), but the core concerns with the nature of process expression itself are usually ignored in IT since the IT community is mainly concerned with the *efficiency of process actualization* independent of how that process is expressed. Computer science is mainly concerned with the *nature of the expression of processes* independent of its platform actualization.

The way of instructing a platform to perform a given task is referred to as entering a *command* for the underlying operating system. On many UNIX and derivative systems, a *shell* (command-line interpreter) then receives, analyzes, and executes the requested command.

A *shell script* is a list of commands which all work as part of a larger process, a sequence (*program*) that you would normally have issued yourself on the command line. UNIX shells have the capability of command flow logic (foreach, while, if/then/else), retrieving the command line parameters, defining and using (shell) variables etc. "Scripts" are distinct from the *executable code* of the application, as they are usually written in a different language with distinctive semantics. Scripts are often *interpreted* from source code, whereas application (command) source code is typically first compiled to a *native machine code* or to an *intermediate code* (e.g. Java bytecode). Machine code is the form of instructions that the native processor executes as a command, while the object-oriented method (intermediate code) is executed by an object-oriented virtual platform (e.g., a Java runtime environment) by sending a *message* from a sender to the recipient object. The message may create additional objects that can send and receive messages as well.

Consequently, a shell script can be treated as a *compound command*, while an executable file as an *elementary command* on a command platform. In contrast, object-oriented and service-oriented programs run on virtual object-oriented and service-oriented platforms respectively.

In Figure 1 each computing platform (P) is depicted as three layers: *domain* (D), *management* (M), and *carrier* (C) with the prefix V standing for *virtual*, and m for *meta*. Each distributed metacarrier mC_k, $k = 1, 2, ..., n$, consists of various platforms. For example, the mC_1 metaprocessor consists of the native command platform $P_{1,1}$, virtual command platform $P_{2,1}$, and object-oriented virtual platform $P_{m1,1}$. All distributed metacarriers mC_i, $i = 1, 2, ..., n$, constitute the *metacarrier* or simply the *cloud processor* (mC)—the processor of a metaplatform (mP). The metaplatform mP is treated as a cloud platform for service-oriented computing.

SORCER service commands (*services*) at the mD level are called exertions (Sobolewski, 2002). The SORCER Operating System (SOS) with its exertion shell (at mM) allows us to execute service-oriented programs—exertion scripts, by analogy to command scripts executed by a UNIX shell. Exertions as commands of the cloud processor (mC) invoke messages on service objects (at VD-$P_{m1,k}$) or commands (at VD-$P_{1,1}$ and D-$P_{2,1}$). Each distributed metaprocessor (mC_k) runs multiple virtual platforms ($P_{i,k}$, $i = 1, 2, ..., n$ and $k = 2, \dots, m_i$) on the same native platform ($P_{1,i}$), while each virtual platform runs services implemented by service objects and commands.

Before we delve into the SORCER metacomputing and metaprogramming concepts, the introduction of some terminology used throughout the paper is required:

- A *computation* is a process following a well-defined model that is understood and can be symbolically expressed and physically accomplished (actualized). A computation can be seen as a purely physical phenomenon occurring inside a system called a *computer*.

- Computing requires a *computing platform* (runtime) to operate. Computing platforms that allow mograms to run require a *processor*, *operating system*, and *mogramming* (*programming or modeling*) *environment* with related tools to create symbolic process expressions—mograms. A computation is physically expressed by a processor and symbolically expressed by a mogram.
- A *distributed computation* allows for sharing computing resources usually located on several remote computers to collaboratively run a single complex computation in a transparent and coherent way. In distributed computing, computations are decomposed into mograms, processes, and computers.
- A *metacomputer* is an interconnected and balanced set of computers that operate as a single unit, which is an actualization of its metacomputing platform (*metaprocessor*, *metaoperating system*, and *metamogramming environment*).
- A *service object* is a *remote* object that deploys/undeploys service providers. A service object, that *manages* multiple service providers, shares the same virtual platform for all its service providers in a *common container*.
- A *service provider* is a *remote* object that provides services to service requestors. Service providers are identified primarily by *service* (*interface*) *types* and typically do not have a lifecycle of their own; any state they do contain tends to be an aggregate of the states of the local entity objects (*service beans*) that they offer to service requestors. A service provider that implements multiple interface provides multiple services.
- A *metacomputation* is a form of distributed computation determined by *collaborating service providers* that a metacomputer can interpret and execute. A *service provider* selected at runtime by a metaoperating system implements services that invoke what are usually commands and messages.
- A collection of service providers selected and managed for a metacomputation is called a *service federation*.
- A *metamogram* is an expression of metacomputation, represented in a *mogramming language*, which a *metacomputer* follows in processing shared data for a *service collaboration*

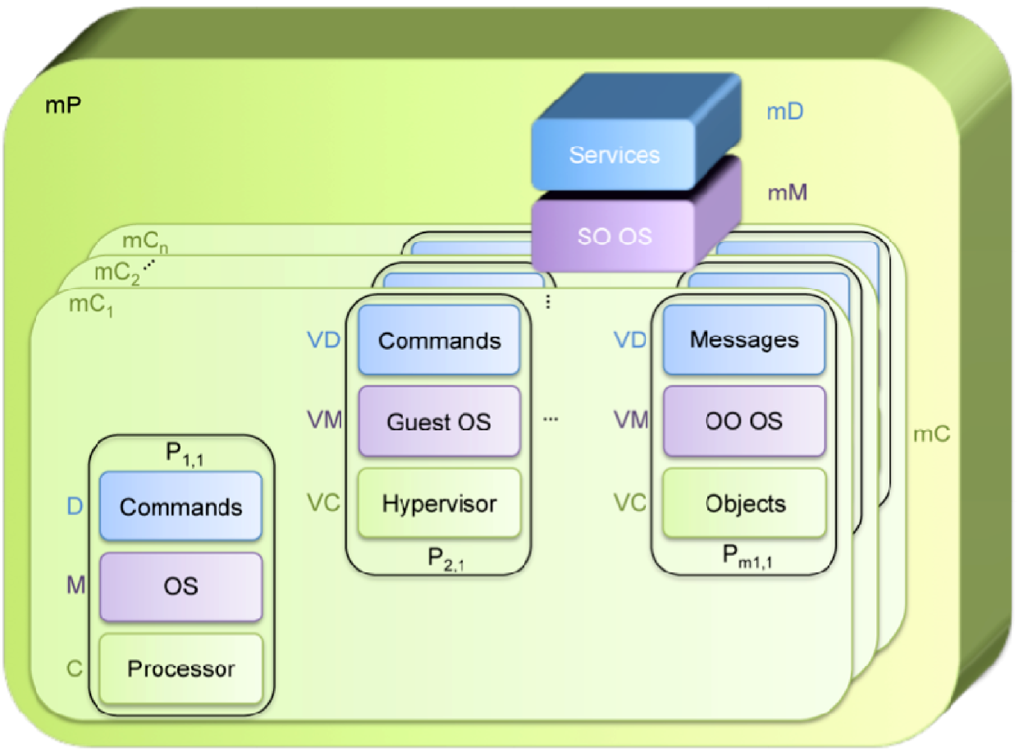

Figure 1: Software cannot operate without a platform or be platform independent. The cloud platform (mP: mD/mM/mC) with its meta-domain (mD), meta-management (mD), and meta-carrier (mC) is composed of n distributed platform clouds mC_i, i = 1, 2, ..., n.

managed by its *metaoperating system* on its *virtual metaprocessor*.

- A *service-oriented architecture* (*SOA*) is a software architecture using loosely coupled service providers. The SOA integrates them into a distributed computing system by means of service-oriented (SO) mogramming. Service providers are made available as independent components that can be accessed without a priori knowledge of their underlying platform, implementation, and location. The client-server architecture separates a client from a server, SOA introduces a third component, a service registry. The registry allows the metaoperating system (not the requestor) to dynamically find service providers on the network.
- If the application (wire) protocol between requestors and all service providers is predefined and final then this type of SOA is called a *service-protocol oriented architecture* (SPOA). In contrast, when the communication is based on remote message passing with the ability to pass objects using the wire protocol that can be chosen by a provider to satisfy efficient communication with its requestors, then the architecture is called a *service-object oriented architecture* (SOOA).

Let's emphasize the major distinction between SOOA and SPOA: in SOOA, a proxy object is created and always owned by the service provider, but in SPOA, the requestor creates and owns a proxy which has to meet the requirements of the protocol that the provider and requestor agreed upon a priori. Thus, in SPOA the protocol is always fixed, generic, and reduced to a common denominator—one size fits all—that leads to inefficient network communication with heterogeneous large datasets. In SOOA, each provider can decide on the most efficient protocol(s) needed for a particular distributed application. For example, SPOA wire protocols are: SOAP in Web and Grid Services, IIOP in CORBA, JRMP in Java RMI. SORCER implements its SOOA with the Jini service architecture (Jini Architecture, n.d.)

The computing platforms and related programming models have evolved as process expression has evolved from the sequential process expression actualized on a single computer to the concurrent process expression actualized on multiple computers. The evolution in process expression introduces new platform benefits but at the same time introduces additional programming complexity that operating systems have to deal with. We can distinguish seven quantum jumps in process expression and related programming complexity (Sobolewski, 2009):

1. Sequential programming (e.g., von Neumann architecture)
2. Multi-threaded programming (e.g., Java platform)
3. Multi-process programming (e.g., Unix platform)
4. Multi-machine-process programming (e.g., CORBA)
5. Knowledge-based distributed programming (e.g., DICEtalk (Sobolewski, 1996))
6. Service-protocol oriented programming (e.g., Web and Grid Services)
7. Service-object oriented programming (e.g. SORCER)

SORCER introduces a special type of *variable* called *var* (Section 3) with var-oriented (VO) programming, var-modeling (VM), and exertion-oriented (EO) programming model with federated method invocation (FMI) in its SOOA. FMI (Sobolewski, 2009) defines the communication framework between three SORCER architectural layers: metamogramming, management, and execution.

This paper is organized as follows: Section 2 briefly describes the SORCER metacomputing architecture; Section 3 introduces the programming model and related three programming languages; Section 4 presents the SORCER operating system with its cloud processor; Section 5 describes the service cloud provisioning for exertion-oriented programming, and Section 6 provides concluding remarks.

2 ARCHITECTURE

The term "federated" means that a single service invocation with no network configuration creates at runtime a federation of required collaborating services. SORCER (Service-ORiented Computing EnviRonment) is a federated service-to-service (S2S) metacomputing environment that treats service providers as network peers with well-defined semantics of a service-object oriented architecture (SOOA) (Sobolewski, 2010a). The SORCER platform has evolved from the service-object abstractions introduced in the FIPER project (1999–2003 (Sobolewski, 2002)), the service-oriented operating system at the SORCER Lab, Texas Tech University (2002-2009 (SORCERsoft, n.d.)), finally with metaprogramming languages for programming convergence at the Multidisciplinary Science and Technology Center, AFRL/WPAFB (2006-2010) (Sobolewski, 2010b). It is based on Jini semantics of services (Jini Architecture, n.d.) in the network and the Jini programming model (Edwards, 2000) with explicit leases, distributed events, transactions, and discovery/join protocols. The Jini network technology focuses on service management in a networked environment, while SORCER is focused on metamogramming and the environment for executing metamograms.

The languages in which languages are expressed are often called *metalanguages*. A language specification can be expressed by a *grammar* or by a *metamodel*. A *metamodel* is a model to define a language. An obvious consequence of multiple syntaxes of service-oriented programming languages in SORCER is that a *concrete syntax* cannot be the form of the language design. This makes the need for a unifying representation apparent. Consequently, the *abstract (conceptual) form* for a service-oriented programming has been developed in SORCER with multiple *concrete models* of the same *metamodel* (abstract model).

The SORCER metamodeling architecture is based on the notion of the *metamodel* called the *DMC-triplet*: *Domain*/*Management*/*Carrier*. For example, a *computing platform* is the DMC-triplet of a *mogramming language* (domain), an *operating system* (management), and a *processor* (carrier). A *language* is the DMC-triplet of *language expressions* (domain), a *grammar* or *metamodel* (management), and a *language alphabet* (carrier). Therefore, a platform is a composition of a domain-specific language (DSL), management with relevant constraints applied to the domain, and the carrier that allows for actualization of both the domain and its management.

The SORCER abstract metamodel is a kind of UML class diagram depicted in Figure 2, where each "class" is an instance of the DMC-triplet (meta-metamodel). This metamodeling architecture distinguishes three computing platforms (command, object, and service platforms) and three new programming platforms (var-modeling, var programming, and exertion programming platforms).

An *exertion* is a service command that specifies how a collaboration is realized by a federation of service providers playing specific roles used in a specific way (Sobolewski, 2008b). The collaboration specifies a view of cooperating providers identified by service types (interfaces)—a projection of service

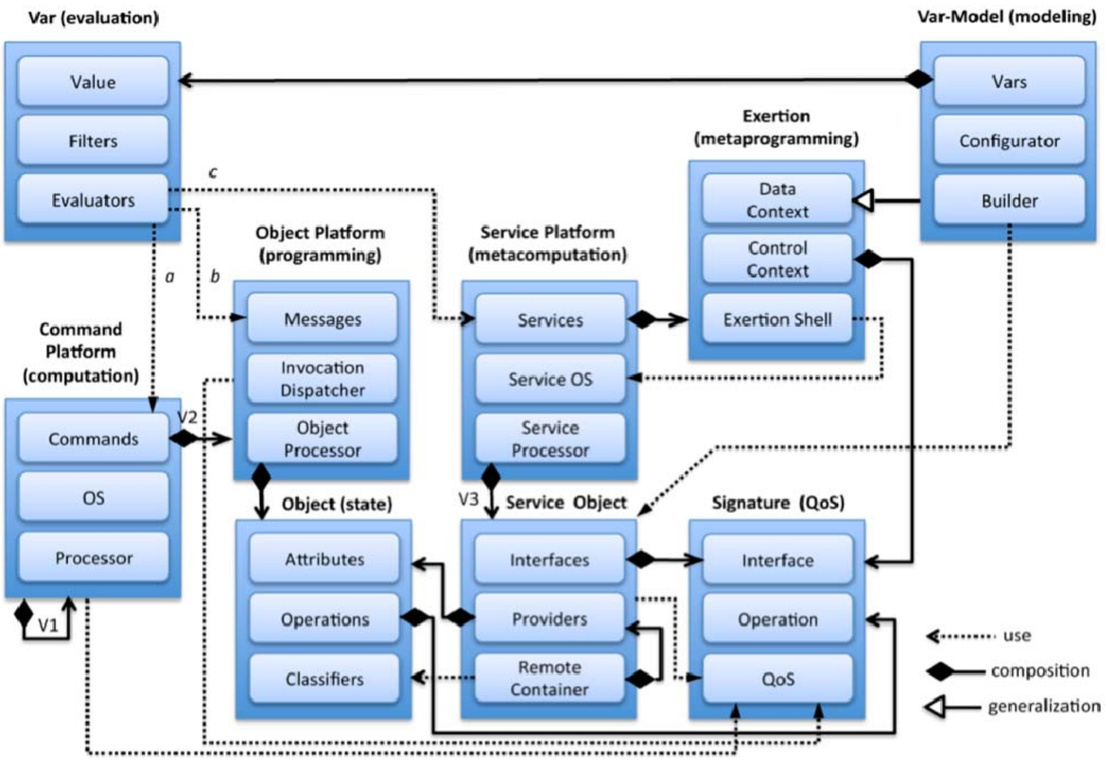

Figure 2: The SORCER metamodeling architecture: evaluation, modeling, metaprogramming, metacomputation, programming, and computation. Each platform is shown as the DMC triplet (domain, management, carrier). The service platform manages the service providers (service cloud) that are autonomically provisioned by service objects on virtualized object/command platforms.

federation. It describes the associations between service types (interfaces) that play the required roles in collaboration, as well as the quality of service (QoS) attributes that describe quality of the participating providers. Several exertions may describe different projections of the same federation since different operations can be used in the same set of federated interfaces. Exertions specify for collaborations explicitly: data (*data context*), operations with related QoS (*signatures*), and control strategy actualized by the *exertion shell*. The exertion participants in the federation collaborate transparently according to its control strategy managed by the SOS based on the Triple Command Pattern described in (Sobolewski, 2009). The functional SORCER architecture is depicted in Figure 3 as the DMC triplet as well.

The exertion's collaboration defines a *service interaction*. The service interaction describes how invocations of operations are managed between service providers to perform a collaborative computation. The interaction is defined by exertion's control strategy and control flow exertions (Sobolewski, 2008a). From the computing platform point of view, exertions are service commands at the programming level D, interactions at the SOS level M, and service federations are groups of domain-specific (DS) providers at the processor level C4. The SOS manages *federations* dynamically on its virtual metaprocessor (cloud processor)—the layers C0-C5 in Figure 3.

SOOA (Sobolewski, 2010a) consists of four major types of network objects: providers, requestors, registries, and proxies for remote communication. The provider is responsible for deploying the service on the network, publishing its proxy object to one or more registries, and allowing requestors to access its proxy. Providers advertise their availability on the network; registries intercept these announcements and cache proxy objects to the provider services. The requestor looks up proxies by sending queries to registries and making selections from the available service types. Queries generally contain search criteria related to the type and quality of service (QoS). Registries facilitate searching by storing proxy objects of services and making them available to requestors. Providers use discovery/join protocols to publish services on the network; requestors use discovery/join protocols to obtain service proxies on the network. Each provider implements multiple interfaces (services) and a single service object hosts multiple providers. A service-object's container is actualized on the virtual object platform (e.g., Java Platform). Two containers called Tasker and Rio cybernodes (Rio Project, n.d) are used predominantly to host service providers. Thus, both service providers/objects and object/command platforms can be provisioned by SOS—service virtualization indicated by SPV/SOV at C4/C3 and platform virtualization indicated by OPV/CPV at C2/C1 in Figure 3. SOS uses Jini discovery/join protocols to implement its federated SOOA with its exertion shell.

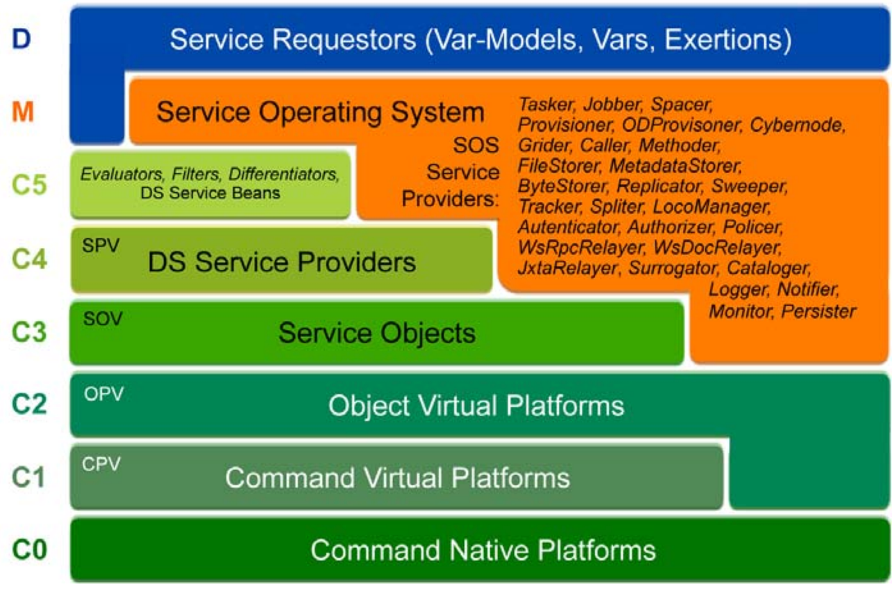
D
Service Requestors (Var-Models, Vars, Exertions)
M
Service Operating System
SOS Service Providers:
Tasker, Jobber, Spacer, Provisioner, ODProvisoner, Cybernode, Grider, Caller, Methoder, FileStorer, MetadataStorer, ByteStorer, Replicator, Sweeper, Tracker, Spliter, LocoManager, Autenticator, Authorizer, Policer, WsRpcRelayer, WsDocRelayer, JxtaRelayer, Surrogator, Cataloger, Logger, Notifier, Monitor, Persister
C5
Evaluators, Filters, Differentiators, DS Service Beans
C4
SPV
DS Service Providers
C3
SOV
Service Objects
C2
OPV
Object Virtual Platforms
C1
CPV
Command Virtual Platforms
C0
Command Native Platforms

Figure 3: The SORCER layered architecture, where C0-C5 (carrier)—the metaprocessor with its service cloud at C4 and C3, platform cloud at C2 and C1, M (management)—SORCER operating system, D (domain)—service requestors; where PV and OV stands for provider and object virtualization respectively with the prefix S for service, O for object, and C for command.

In SORCER, a *service bean* is a plain Java object (POJO) that implements domain-specific interfaces. A *service provider* exposes on the network interfaces of embedded service beans that implement own domain-specific interfaces, if any. The provider creates and registers its proxy object with service registries. Registered providers then execute operations in its published interfaces that are requested by service requestors and invoked by SOS on proxy objects.

A *task exertion* is an *elementary* service command executed by a single service provider or a small-scale federation managed by the provider executing the task. A *compound* service command called a *job exertion* is defined hierarchically in terms of tasks and other jobs, including control flow exertions. A job exertion is a kind of distributed metaprogram executed by a large-scale federation. The executing exertion, interpreted by the SORCER shell, is a *service-oriented program* that is dynamically bound to all required and currently available service providers on the network. This collection of service providers identified at runtime is called an *exertion federation*.

The overlay network of all service providers is called the *service metaprocessor* or the *service cloud*. The metainstruction set of the metaprocessor consists of all operations offered by all service providers in the cloud. Thus, a service-oriented program is composed of metainstructions with its own service-oriented control strategy and data context representing the metaprogram data. Service signatures used in metaprograms, at the D level in Figure 3, specify runtime SOS selected methods for virtual providers at the C4 level that are hosted by service objects at the C3 level. In turn, the required service objects are deployed on virtual object and command platforms at C2 and C1 correspondingly.

Each signature is defined by an interface type, operation in that interface, and a set of optional QoS attributes. Four types of signatures are distinguished: *PROCESS*, *PREPROCESS*, *POSTPROCESS*, and *APPEND*. A *PROCESS* signature—of which there is only one allowed per exertion—defines the dynamic late binding to a provider that implements the signature's interface. The service context describes the data on which tasks and jobs work. An *APPEND* signature defines the context received from the provider specified by the signature. The received context is then appended at runtime to the service context later processed by *PREPROCESS*, *PROCESS*, and *POSTPROCESS* operations of the exertion. Appending a service context allows a requestor to use actual network data produced at runtime not available to the requestor when it initiates its execution. A metacompute OS allows for an exertion to create and manage dynamic federation and transparently coordinate the execution of all component exertions within the federation. Please note that these metacomputing concepts are defined differently in traditional grid computing where a job is just an executing process for a submitted executable code with no federation being formed for the executable.

3 PROGRAMMING MODEL

The abstract model depicted in Figure 2 is the unifying representation for three concrete programming syntaxes: the functional composition form, the SORCER Java API described in (Sobolewski, 2008a), and the graphical form described in (Sobolewski and Kolonay, 2006). The functional composition notation has been developed for three related languages: Var-Modeling Language (VML), Var-Oriented Language (VOL), and Exertion-Oriented Language (EOL) that are usually complemented with the Java object-oriented syntax. In the reminder of this Section we will describe the basic syntax of these three languages. More details on EOL can be found in (Sobolewski, 2010b).

The fundamental principle of functional programming is that a computation can be realized by composing functions. Functional programming languages consider functions to be data, avoid states, and mutable values in the evaluation process in contrast to the imperative programming style, which emphasizes changes in state values. Thus, one can write a function that takes other functions as parameters, returning yet another function. Experience suggests that functional programs are more robust and easier to test than imperative ones.

Not all operations are mathematical functions. In nonfunctional programming languages, "functions" are subroutines that return values while in a mathematical sense a function is a unique mapping from input values to output values. The *var* entity allows one to use functions, subroutines, or coroutines in the same way. Here the term *var* is used to denote a mathematical function, subroutine, coroutine, any data or object.

Var-oriented programming is the programming paradigm that treats any computation as the triplet: *value*, *filter* (*pipeline of filters*), and *evaluator* (VFE, see Figure 4). Evaluators and filters can be executed locally or remotely, sequentially or concurrently. The evaluator may use a differentiator to calculate the rates at which the var quantities change. The

VFE paradigm emphasizes the usage of *evaluators* and *filters* to define the value of var. The semantics of the *var value*, whether the var represents a mathematical function, subroutine, coroutine, or just data, depends on the evaluator and filter currently used by the var. VO programming allows for exertions to use vars in data contexts. Alternatively, data contexts (implementing `Context` interface) with specialized aggregations of vars, called *var-models*, can be used for enterprise-wide metacomputing. Three types of models: *response*, *parametric*, and *optimization* have been studied already (Sobolewski, 2010b).

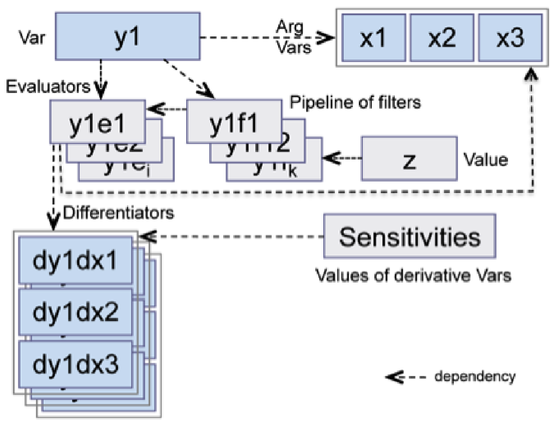

Figure 4: The var structure: value/filter/evaluator. Vars are indicated in blue color. The basic var y1, z=y1(x1, x2, x3), depends on its argument vars and derivative vars.

The variable evaluation strategy is defined as follows: the associated current evaluator determines the variable's *raw value* (not processed or subjected to analysis), and the current pipeline of filters returns the *output value* from the evaluator result. Multiple associations of evaluator-filter can be used with the same var (multifidelity). Evaluator's raw value may depend on other var arguments and those arguments in turn can depend on other argument vars and so on. This var dependency chaining is called *var composition* and provides the integration framework for all possible kinds of computations represented by various types of evaluators including exertions via exertion evaluators.

The modularity of the VFE metamodel, reuse of evaluators and filters, including exertion evaluators, in defining var-models is the key feature of variable-oriented programming (VOP). The same evaluator with different filters can be associated with many vars. The VOP model complements distributed service-oriented computing with other types of computing. In particular, evaluators can be associated with commands (executables), messages (objects), and services (exertions) as indicated by a, b, and c dependences in Figure 2.

Var-models support *multidisciplinary* and *multifidelity* computing. Var compositions across multiple models define multidisciplinary problems; multiple evaluators per var and multiple differentiators per evaluator define their multifidelity. They are called *amorphous* models. For the same var-model an alternative set of evaluators/filters (another fidelity) can be selected at runtime to evaluate a new particular version ("shape") of the model and quickly update the related computation in the right evolving or new direction.

Let's consider the Rosen-Suzuki optimization problem to illustrate the basic VML, VOL, and EOL concepts, where:

1. design vars: *x1*, *x2*, *x3*, *x4*
2. response vars: *f*, *g1*, *g2*, *g3*,
   and
3. *f = x1^2-5.0*x1+x2^2-5.0*x2+2.0*x3^2-21.0*x3+x4^2+7.0*x4+50.0*
4. *g1 = x1^2+x1+x2^2-x2+x3^2+x3+x4^2-x4-8.0*
5. *g2 = x1^2-x1+2.0*x2^2+x3^2+2.0*x4^2-x4-10.0*
6. *g3 = 2.0*x1^2+2.0*x1+x2^2-x2+x3^2-x4-5.0*

The goal is then to minimize *f* subject to
g1 <= 0, *g2 <= 0*, and *g3 <= 0.*

In VML this case is expressed by the following mogram:

```
int designVarCount = 4;
int responseVarCount = 4;
OptimizationModel model =  optimizationModel("Rosen-
Suzuki Model",
  designVars(vars(loop(designVarCount),
    "x", 20.0, -100.0, 100.0)),
  responseVars("f"),
  responseVars(loop(responseVarCount-1),
  "g"),
  objectiveVars(var("fo", "f", Target.min)),
  constraintVars(
    var("g1c", "g1", Relation.lte, 0.0),
    var("g2c", "g2", Relation.lte, 0.0),
    var("g3c","g3", Relation.lte, 0.0)));
configureAnalysisModel(model);
```

Response vars `f`, `g1`, `g2`, `g3` are configured by the function `configureAnalysisModel` defined in VOL, for example var `f` is configured as follows:

```
var(model, "f",
  evaluator("fe1",
    "x1^2- 5.0*x1+x2^2-5.0*x2+2.0
      *x3^2-21.0*x3+x4^2+7.0*x4+50.0"),
  args("x1", "x2", "x3", "x4"));
```

The `model` above can be provisioned directly in SORCER as a servicer provider and used by the space exploration provider of the *Exploration* type that also uses the CONMIN optimization (CONMIN, n.d) service provider of the *Optimization* type.

The requestor creates the exertion *opti* as follows:

```
// Create an optimization data context
Context exploreContext = exploreContext("Rosen-Suzuki
context",
  varsInfo(varInfo("x1", 1.0),
    varInfo("x2", 1.0),
    varInfo("x3", 1.0),
    varInfo("x4", 1.0)),
  strategy(new ConminStrategy(
    new File(System.getProperty(
      "conmin.strategy.file")))),
  dispatcher(
    sig(null, RosenSuzukiDispatcher.class,
  Process.INTRA)),
  model(sig("register",
    OptimizationModeling.class,
      "Rosen-Suzuki Model")),
  optimizer(sig("register",
    Optimization.class,
      "Rosen-Suzuki Optimizer")));

// Create a task exertion
Task opti= task("opti",
  sig("explore", Exploration.class,
    "Rosen-Suzuki Explorer"),
  exploreContext);
```

then executes the *opti* exertion:

```
// Execute the exertion and log results
logger.info(">>>>>>>>>>>>> results: "
  + context(exert(opti));
```

with the exertion's output data context logged as follows:

```
[java] Objective Function fo =     6.002607805900986
[java] Design Variable Values
[java] x1 = 2.5802964087086235E-4
  x2 = 0.9995594642481355
  x3 = 2.000313835134211
  x4 = -0.9986692050113675
[java] Constraint Values
[java] g1c = -0.002603585246998996
  g2c =-1.0074147118087602
  g3c = 4.948009193483927E-7
[java] ITERATIONS
[java] Number of Objective Evaluations = 88
[java] Number of Constraint Evaluations = 88
[java] Number of Objective
  Gradient Evaluations = 29
[java] Number of Constraint Gradient
  Evaluations = 29
```

The *exploreContext* defines initialization of design vars (*varsInfo*), the optimization strategy, and the exploration dispatcher with two required services—two signatures for: optimizer and model. The context then is used to define the exertion task *opti* with the signature for exploration service named *Rosen-Suzuki Explorer* of the *Exploration* type. For simplicity, signatures above do not specify QoS for the specified providers. To illustrate the provider QoS concept (Rubach and Sobolewski, 2009), for example, the optimizer's signature can be expressed as follows:

```
 sig("register", Optimization.class, qosCtx))
```

where the *qosCtx* context may be defined as follows:

```
QosContext qosCtx = qos(
  serviceProvider(entry(
    "Name","Rosen-Suzuki Optimizer"),
    libs(entry("Name","Conmin"),
      entry("Class",
        NativeLibrarySupport.class),
  entry("FileName",
        "conmin.so"))),
  objectPlatform(entry("Name","Java"),
    entry("Class", J2SESupport.class),
    entry("Version", "1.5.*")),
  commandPlatform(
    processor(entry("Available", "2"),
  entry("Architecture", "x86")),
    memory(entry("Capacity", "4G"),
      entry("Available", "2G")),
    disk(entry("Capacity", "20G"),
      entry("Available", "4G"))),
  sla(
    entry("cost", 200),
    entry("time", 5000),
    entry("CPU", range(0.0, 0.9)),
    entry("Memory", range(0.2, 0.5)),
    entry("ProcAvail_CPU_Util", range(1.5,
  2.0),
    metric("ProcAvail_CPU_Util",
      impl("result
        = Double.parseDouble(proc_avail)
          * cpu_util",
        args(
          var("proc_avail", ...)))),
          var("cpu_util", )))))))),
  authorization(
    entry("estimatedDuration", 30000l),
    entry("priority", range(5,10)),
    execDate("2011-01-10 00:00:00",
      "2011-01-11 12:00:00"),
    project(
      entry("name","RS"),
      entry("manager","Smith"),
      entry("description",
        "RS optimization")),
    organization(
        entry("name", "TTU"),
        entry("department", "CS"),
        entry("description",
          "SORCER Testbed"))));
```

The *qosCtx* context specifies for the optimizer required QoS: for the provider by operator *serviceProvider*, for the object platform by operator *objectPlatform*, for the command platform by operator *commandPlatform*, expected SLA by operator *sla* and authorization by *authorization* for the service requestor.

4 SORCER OPERATING SYSTEM AND SERVICE CLOUDS

SOS allows execution of a service-oriented program and by itself is the service-oriented system. The overlay network of the service providers defining the functionality of SOS is called the *sos-cloud* (M layer in Figure 3) and the overlay network of application providers is called the *app-cloud*—metaprocessor (C4 layer in Fgure 3). Both the sos-cloud and app-cloud constitute the *service cloud*. The *metainstruction set* of the SORCER metaprocessor consists of all operations offered by all service providers in the app-cloud. Thus, an exertion is composed of metainstructions with its own control strategy per service composition and data context representing the shared data for the underlying federation. Service signatures (instances of *Signature* type) returned by *sig* operators specify operations of collaboration participants in the app-cloud. Each signature is defined primarily by an interface type, operation in that interface, and by a QoS context.

As explained in Section 2, four types of signatures are distinguished. A *PROCESS* signature—of which there is only one allowed per exertion—defines the dynamic late binding to a provider that implements the signature's interface. The data context describes the data that tasks and jobs operate on and create. SOS allows for an exertion to create and manage a service collaboration and transparently coordinate the execution of all exertion's nested signatures within the assembled federation. These exertion-based computing concepts are defined differently in traditional grid/cloud computing where a job is just an executing process for a submitted executable code—the executable becomes the single service itself that can be parallelized on multiple processors, if needed. Here, a job is the federation of collaborating executable codes (command providers) and related other providers that is formed by SOS for a single exertion as specified by its all nested signatures.

An exertion object of the *Exertion* type is returned by either *task* or *job* operators of EOL. Then, the exertion can be actualized by the exertion shell, by invoking the *exert* operation on the shell as follows:

```
ExertionShell#exert(Exertion ,Transaction)
   :Exertion,
```

where a parameter of the *Transaction* type is required when a transactional semantics is needed for required participating service providers within the collaboration defined by the exertion. Thus, EO programming allows us to execute an exertion and invoke exertion's signatures on collaborating service providers indirectly, but where does the service-to-service communication come into play? How do these services communicate with one another if they are all different? Top-level communication between services, or the sending of service requests, is done through the use of the generic *Servicer* interface and the operation *service* that SORCER providers are required to implement:

```
Servicer#service(Exertion,
   Transaction):Exertion.
```

This top-level service operation takes an exertion object as an argument and gives back an exertion object as the return value.

So why are exertion objects used rather than directly calling on a provider's method and passing data contexts? There are two basic answers to this. First, passing exertion objects helps to aid with the network-centric messaging. A service requestor can send an exertion object implicitly out onto the network—*SorcerShell#exert(Exertion)*—and any service provider can pick it up. The receiving provider can then look at the signature's interface and operation requested within the exertion object, and if it doesn't implement the desired interface and provide the desired method, it can continue forwarding it to another service provider who can service it. Second, passing exertion objects helps with fault detection and recovery. Each exertion object has its own completion state associated with it to specify if it has yet to run, has already completed, or has failed. Since full exertion objects are both passed and returned, the user can view the failed exertion to see what method was being called as well as what was used in the data context input that may have caused the problem. Since exertion objects provide all the information needed to execute the exertion including its control strategy, the user would be able to pause a job between component exertions, analyze it and make needed updates. To figure out where to resume an exertion, the executing provider would simply have to look at the exertion's completion states and resume the first one that wasn't completed yet. In other words, EO programming allows the user, not programmer to update the metaprogram on-the-fly, what practically translates into creating new interactive collaborative applications at runtime.

Applying the inversion principle, SOS executes the exertion's collaboration with dynamically found, if present, or provisioned on-demand service providers. The exertion caller has no direct dependency to service provider since the exertion uses only service types they implement.

Despite the fact that any Servicer can accept any exertion, SOS services have well defined roles in the S2S platform (Sobolewski, 2010b) (see Figure 3):

a) Taskers–accept exertion tasks; they are used to create application services by dependency injection (service assembly from service beans) or by inheritance (subclassing *ServiceTasker* and implementing required service interfaces);
b) Jobbers–manage service collaboration for *PUSH* service access;
c) Spacers–manage service collaboration for *PULL* service access using space-based computing;
d) Contexters–provide data contexts for *APPEND* signatures;
e) FileStorers–provide access to federated file system providers;
f) Catalogers–Servicer registries, provide management for QoS-based federations;
g) SlaMonitors–provide monitoring of SLAs;
h) Provisionersvprovide on-demand provisioning;
i) Persisters–persist data contexts, tasks, and jobs to be reused for interactive EO programming;
j) Relayers–gateway providers; transform exertions to native representation, for example integration with Web services and JXTA;
k) Authenticators, Authorizers, Policers, KeyStorers –provide support for service security;
l) Auditors, Reporters, Loggers–support for accountability, reporting, and logging
m) Griders, Callers, Methoders–support for a conventional compute grid (allocating executables codes on the network);
n) Notifiers–use third party services for collecting provider notifications for time consuming programs and disconnected requestors.

Both sos-providers and app-providers do not have mutual associations prior to the execution of an exertion; they come together dynamically (federate) for all nested tasks and jobs in the exertion.

DS servicers within the app-cloud—taskers—execute task exertions. Rendezvous peers (jobbers, spacers, and catalogers) manage service collaborations. Providers of the *Tasker*, *Jobber*, *Spacer*, and *Cataloger* type are basic SOS exertion management providers. In the view of the P2P architecture (Sobolewski, 2008b) defined by the *Servicer* interface, a job can be sent to any servicer. A peer that is not a *Jobber* or *Spacer* type is responsible for forwarding the job to one of the available rendezvous peers in the SORCER environment and returning results to the requestor. Thus implicitly, any peer can handle any exertion type. Once the exertion execution is complete, the federation dissolves and the providers in the federation disperse to seek other exertions to join.

5 CLOUD PROVISIONING

The SOS manages service collaborations with dynamically found, if present, or provisioned on-demand service providers. The exertion caller has no direct dependency to service providers since the exertion uses only interface types they implement. The SOS shell offers requestors the ability to dynamically invoke SOS services in the sos-cloud and then required collaborating providers in the app-cloud selected by SOS provisioning. Selected QoS attributes or QoS context in signatures are optional. However, if used, the required command platform (V1 in Figure 2), object platform (V2 in Figure 2), and provider (V2 in Figure 2), can be provisioned if no service provider is available in the app-cloud.

Without an efficient resource management the assignment of providers to the requestor's signatures cannot be optimized and cannot offer high reliability without relevant SLA guarantees. A SLA-based SERViceable Metacomputing Environment (SERVME) (Rubach and Sobolewski, 2009) is capable of matching providers based on QoS requirements and performing autonomic provisioning and deprovisioning of providers according to dynamic requestor needs. In SERVME an exertion signature includes a QoS context with SLA requirements as illustrated in Section 4. SERVME is a generic resource management framework in terms of common QoS/SLA data structures and extensible communication interfaces which hide all implementation details.

Along with the QoS/SLA object model SERVME defines basic components and communication interfaces as depicted in the UML component diagram illustrated in Figure 5. We distinguish two forms of autonomic provisioning: monitored and on-demand. In monitored provisioning the Rio Provisioner (Rio Project, n.d.) deploys a requested collection of providers, then monitors them for presence and makes sure that the required number of providers is always on the network as defined by the collection's deployment descriptor. On-demand provisioning refers to a type of provisioning (On-demand Provisioner) where the actual provider is presented to the requestor, once an SLA subscription to the requested service is successfully processed. In both cases, if services become unavailable, or fail to meet processing requirements, the recovery of those service providers to available compute resources is enabled by the Rio provisioning mechanisms.

The basic SERVME components are defined as follows:

- QosProviderAccessor is a component used by the service requestor that is responsible for processing the exertion request containing QosContext in its signature. If the exertion type is Task then QosCatalog is used, otherwise a relevant rendezvous peer: Jobber, Spacer is used.
- QosCatalog is an independent service that acts as a QoS-based Lookup Service. The QosCatalog uses the functional requirements as well as related non-functional QoS requirements to find a service provider from currently available in the network. If a matching provider does not exist, the QosCatalog may provision the needed one.
- SlaDispatcher is a component built into each service provider. It performs two roles. On one hand, it is responsible for retrieving the actual QoS parameter values from the operating system in which it is running, and on the other hand, it exposes the interface used by QosCatalog to negotiate, sign and manage the SLA with its provider.
- SlaPrioritizer is a component that allows controlling the prioritization of the execution of exertions according to the organizational requirements of SlaContext.
- QosMonitor UI provides an embedded GUI that allows the monitoring of provider's QoS parameters at runtime.
- SlaMonitor is an independent service that acts as a registry for negotiated SLA contracts and exposes the GUI for administrators to allow them to monitor, update or cancel active SLAs.
- On-demandProvisioner enables on-demand provisioning of services in cooperation with the Rio Provisioner ("Rio Project", n.d.) The QosCatalog uses it when no matching service provider can be found that meets requestor's QoS requirements.

Two forms of provisioning are considered: monitored and on-demand. In monitored provisioning the provisioner deploys a requested collection of providers, then monitors them for presence and in the case of any failure in the deployed collection, the provisioner makes sure that the collection is always on the network as defined by the collection's deployment descriptor. On-demand provisioning refers to a type of provisioning when the actual provider is presented to the requestor,

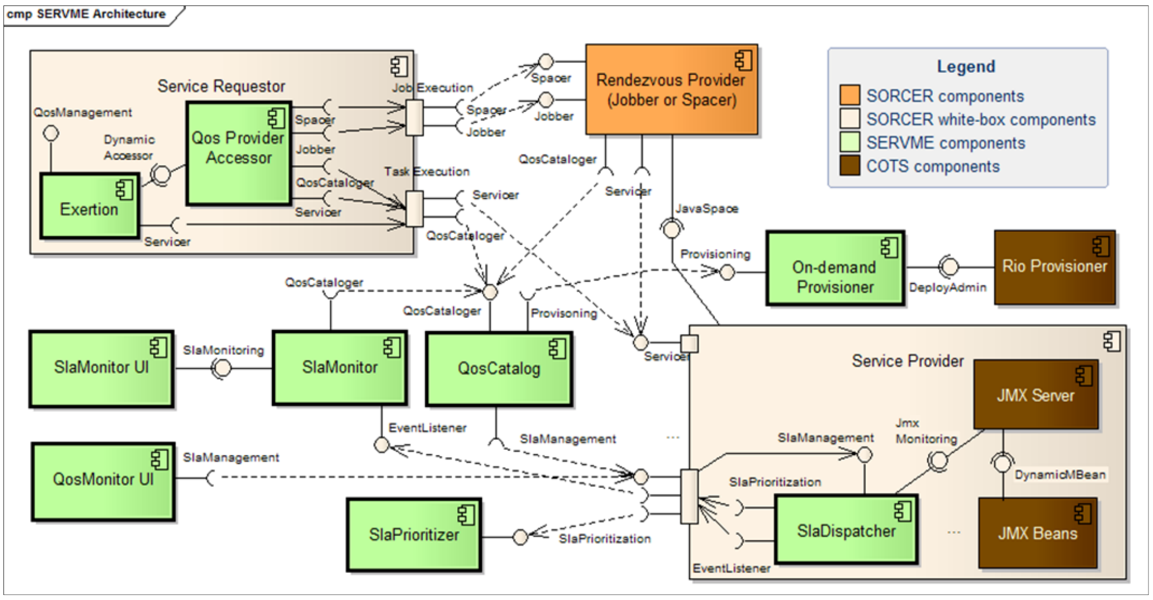

Figure 5: SERVME resource management architecture.

once a subscription to the requested service is successfully processed. In both cases, if services become unavailable, or fail to meet processing requirements, the recovery of those service providers to available compute resources is enabled by Rio provisioning mechanisms.

As described in Section 4, the service requestor submits the exertion with QoS contexts into the network by invoking *evaluate(Exertion)* in EOL. If the exertion is of Task type, then QosProviderAccessor via QosCalatog finds in runtime a matching service provider with a corresponding SLA.

If the SLA can be directly provided then the contracting provider approached by the QosCalatog returns it in the form of a SlaContext, otherwise a negotiation can take place for the agreeable SlaContext between the requestor and provider. The provider's SlaDispatcher drives this negotiation in cooperation with SlaPrioritizer and the exertion's requestor.

If the task contains multiple signatures then the provider is responsible for contracting SLAs for all other signatures of the task before the SLA for its *PROCESS* signature is guaranteed.

However, if the submitted exertion is of Job type, then QosProviderAccessor via QosCalatog finds at runtime a matching rendezvous provider with a guaranteed SLA. Before the guaranteed SLA is returned, the rendezvous provider recursively acquires SLAs for all component exertions as described above depending on the type (Task or Job) of component exertion.

6 CONCLUSIONS

Cloud computing it is not just computing on a collection of virtualized platforms. A service-oriented system is not just a collection of distributed objects—it is the unreliable network of service providers that may come and go on virtualized platforms. SORCER introduces the new metacomputing abstractions for evaluation, modeling, metaprogramming, metacomputation, programming, and computation (Figure 2). EO programming introduces the new abstractions of *service providers* and *exertions* for metaprogramming instead of *objects* and *messages* in object-oriented programming. Exertions encapsulate *service data*, *signatures* with QoS, and a *control strategy* interpreted by the *exertion shell*. The exertions are service commands that define reliable network collaborations in unreliable networks.

Applying the inversion principle, SOS looks up service providers by implemented interface types with optional QoS attributes. SOS utilizes Jini-based service management that provides for dynamic services, mobile code shared over the network, and network security. Federations are aggregated from independent service-providers in the service cloud that do not require heavyweight containers like application servers. SOS defines the coherent framework between three SORCER architectural layers: programming, management, and cloud processor.

The SORCER platform uses a dynamic service discovery mechanism allowing new services to enter the network and disabled services to leave the network gracefully without need for reconfiguration. This allows the exertion federation to be distributed without sacrificing the robustness of the service-oriented process. This architecture also improves the utilization of the network resources by distributing the execution load over multiple nodes of the network. The exertion's federation shows resilience to service failures on the network as it can search for alternate services and maintain continuity of operations even during periods when there is no service available.

The presented approach to the autonomic provisioning framework with QoS in exertions and the negotiation for the acceptable SLA addresses the challenges of spontaneous federations and allows for better resource allocation. Also, SERVME provides for better hardware utilization due to Rio monitored provisioning and SORCER on-demand provisioning. The SOS on-demand provisioning reduces the number of compute resources to those presently required for collaborations defined by corresponding exertions. Once a service is provisioned, the SOS provisioners ensure that services are maintained to the expected QoS. Provisioning thus refers to bootstrapping of the service provider, and monitoring to the ongoing service responsibility. In the case of any provider's failure the service provider is re-provisioned in the network with the same required QoS. When diverse and specialized hardware is used, SERVME provides a means to manage the prioritization of tasks according to the organization’s strategy that defines "who is computing what and where".

SORCER defines clearly its separate meta-computing architectural layers: metamogramming, management, and cloud processor layers integrated via SOS. That introduces simplicity to SORCER mogramming with flexible enterprise interoperability achieved via three neutralities (protocol, implementation, and location (Sobolewski, 2009)) and architectural means (Figure 1, 2, 3, and 4), not by neutral data exchange formats, e.g., XML. Neutral data exchange formats when overused introduce unintended complexity and degraded performance.

The service-oriented cloud computing philosophy described in this paper implies that separating explicitly programming from metaprogramming with corresponding platforms and metaplatforms gives us the understanding of what is and is not important for building large-scale adaptive and dynamic enterprise systems. Otherwise, reducing programming to the level of middleware programming only and within the command/object platform, introduces intolerable complexity for building such distributed systems.

SORCER with its hierarchical mogramming model has been successfully deployed and tested in multiple concurrent engineering and large-scale distributed applications (Kolonay et al., 2007; Goel, et al., 2008; Xu, et al., 2009).

ACKNOWLEDGEMENTS

This work was partially supported by Air Force Research Lab, Air Vehicles Directorate, Multidisciplinary Science and Technology Center, the contract number F33615-03-D-3307, Algorithms for Federated High Fidelity Engineering Design Optimization. I would like to express my gratitude to all those who helped me in my SORCER research at AFRL, GE Global Research Center, and my students at the SORCER Lab, TTU. Especially I would like to express my gratitude to Dr. Ray Kolonay, my technical advisor at AFRL/RBSD for his support, encouragement, and advice.

REFERENCES

Apache River, Available at: http://incubator.apache.org/river/RIVER/index.html. Accessed on: August 10, 2010.

CONMIN User's Manual, Available at: http://www.eng.buffalo.edu/Research/MODEL/mdo.test.orig/CONMIN/manual.html. Accessed on: August 10, 2010.

Edwards, W.K. (2000) *Core Jini*, 2nd ed., Prentice Hall

Fant, K.M., 1993. A Critical Review of the Notion of Algorithm in Computer Science, *Proceedings of the 21st Annual Computer Science Conference*, February 1993, pp. 1-6.

Foster I.; Kesselman C. & Tuecke S., 2001. The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International J. Supercomputer Applications*, 15(3).

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

Jini Architecture Specification. Accessed on: February 30, 2011. Available at: http://www.jini.org/wiki/Jini_Architecture_Specification.

Kleppe A. 2009. *Software Language Engineering*, Pearson Education, ISBN: 978-0-321-55345-4.

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

Linthicum, D. S., 2009) *Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide*, Addison-Wesley Professional, ISBN-10 0136009220.

Metacomputing: Past to Present, February 30, 2011. Avail able at: http://archive.ncsa.uiuc.edu/Cyberia/MetaComp/MetaHistory.html.

Rio Project. Available at: http://www.rio-project.org/. Accessed on: February 30, 2011.

Rubach, P. & Sobolewski, M., 2009. Autonomic SLA Management in Federated Computing Environments. *International Conference on Parallel Processing Workshops*, Vienna, Austria: 2009, pp. 314-321.

Sobolewski, M., 2002. Federated P2P Services in CE Environments, *Advances in Concurrent Engineering, A.A. Balkema Publishers, 2002,* ISBN 90 5809502 9, pp. 13-22.

Sobolewski, M. & Kolonay, R., 2006. Federated Grid Computing with InteractiveService-oriented Programming, *International Journal of Concurrent Engineering: Research & Applications*, Vol.14, No.1, pp. 55-66.

Sobolewski, M., 2008a. Exertion Oriented Programming, IADIS, vol. 3 no. 1, pp. 86-109, ISBN: ISSN: 1646-3692.

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

Sobolewski, M., 2009. Metacomputing with Federated Method Invocation, *Advances in Computer Science and IT*, edited by M. Akbar Hussain, In-Tech, intechweb.org, ISBN 978-953-7619-51-0, s. 337-363. Accessed on: February 30, 2011. Available at: http: // sciyo.com/articles/show/title/metacomputing-with-federated-method-invocation.

Sobolewski, M., 2010a. Object-Oriented Metacomputing with Exertions, *Handbook On Business Information Systems*, A. Gunasekaran, M. Sandhu (Eds.), World Scientific, ISBN: 978-981-283-605-2.

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

SORCERsoft. Available at: http://sorcersoft.org. Accessed on: August 10, 2010.

Xu, W., Cha, J., Sobolewski, M., 2008. A Service-Oriented Collaborative Design Platform for Concurrent Engineering, *Advanced Materials Research,* Vols. 44-46 (2008) pp. 717-724.

BRIEF BIOGRAPHY

Mike Sobolewski received his Ph.D. from the Institute of Computer Science, Polish Academy of Sciences. He is the Principal Investigator of the SORCER Lab (http://SORCERsoft.org) focused on research in distributed service-centric metacomputing. Currently he is a World Class Scientist at the Air Force Research Lab (AFRL), WPAFB/USA. Before, he was a Professor of Computer Science, Texas Tech University and Director of SORCER Lab from 2002 till 2009. Now he is engaged in development of Algorithms for Federated High Fidelity Engineering Design Optimization applying his innovative SORCER solutions at the Multidisciplinary Science and Technology Center, AFRL/WPAFB.

While at the GE Global Research Center (GE GRC), 1994-2002, he was a senior computer scientist and the chief architect of large-scale projects funded by the United States Federal Government including the Federated Intelligent Product EnviRonment (FIPER) project and Computer Aided Manufacturing Network (CAMnet). Also, based on his web-based generic application framework he developed seventeen successful distributed systems for various GE business components. Before his work at GE GRC, he was a Research Associate at the Concurrent Engineering Center (CERC) and and Visiting Professor at Computer Science Department, West Virginia University (1998-1994). At CERC/WVU he was a project leader for knowledge-based integration for the DARPA Initiative in Concurrent Engineering (DICE).

Prior to coming to the USA, during 18-year career with the Polish Academy of Sciences, Warsaw, Poland, he was the Head of the Pattern Recognition and Image Processing Department, the Head of the Expert Systems Laboratory, and was engaged in research in the area of knowledge representation, knowledge-based systems, pattern recognition, image processing, neural networks, object-oriented programming, and graphical interfaces. He has served as a visiting professor, lecturer, or consultant in Sweden, Finland, Italy, Switzerland, Germany, Hungary, Slovakia, Poland, Russia, China, and USA.