Object-Oriented Service Clouds for Transdisciplinary Computing

Michael Sobolewski

Abstract A computing platform includes a programming environment to create applications with a coherent operating system and processor. The Service ORiented Computing EnviRonment (SORCER) is an object-oriented cloud platform that targets service abstractions for transdisciplinary complexity with support for high performance computing. SORCER service-commands are expressed in Exertion-Oriented Language (EOL) in convergence with two other languages: variable-oriented language (VOL) and variable-oriented modeling language (VML). The SORCER operating system (SOS) supports the two-way convergence of three programming models for transdisciplinary computing in service clouds. On one hand, EOP is uniformly converged with VOP and VOM to express an explicit network-centric service-oriented (SO) computation process in terms of other implicit (inter/intra) process expressions. On the other hand, VOM and VOP are uniformly converged with EOP to express an explicit declarative service model with multifidelity and multidisciplinary features in terms of other implicit (intra/inter) process expressions including network-centric service clouds.

Keywords Process expression • Metacomputing • Concurrent engineering • Cloud computing • Service object-oriented architectures • Service provisioning • Var-oriented modeling • Var-oriented programming • Exertion-oriented programming

M. Sobolewski (✉)
Air Force Research Laboratory, Wright-Patterson Air Force Base, Dayton, Ohio 45433, USA

Polish-Japanese Institute of Information Technology, Warsaw, Poland
e-mail: sobol@sorcersoft.org

I. Ivanov et al. (eds.), *Cloud Computing and Services Science*, Service Science: Research and Innovations in the Service Economy, DOI 10.1007/978-1-4614-2326-3_1,

1 Introduction

In transdisciplinary computing systems each service provider in the collaborative federation performs its services in an orchestrated workflow. Once the collaboration is complete, the federation dissolves and the providers disperse and seek other federations to join. The approach is network centric in which a service is defined as an independent self-sustaining entity—remote service provider—performing a specific network activity. These service providers have to be managed by a relevant operating system with commands for expressing interactions of providers in the network.

The reality at present, however, is that transdisciplinary computing environments are still very difficult for most users to access, and that detailed and low-level programming must be carried out by the user through command line and script execution to carefully tailor jobs on each end to the resources on which they will run, or for the data structure that they will access. This produces frustration on the part of the user, delays in the adoption of service-oriented (SO) techniques, and a multiplicity of specialized "cluster/grid/cloud-aware" tools that are not, in fact, aware of each other which defeats the basic purpose of the cluster/grid/cloud.

A computer as a programmable device that performs symbolic processing, especially one that can process, store and retrieve large amounts of data very quickly, requires a computing platform (runtime) to operate. Computing platforms that allow software to run on the computer require a processor, operating system, and programming environment with related runtime libraries and user agents. Therefore, the metacomputer requires a platform that describes a kind of networking framework to allow software to run utilizing virtual distributed resources. Different platforms of metacomputers can be distinguished along with corresponding types of virtual network processors.

We consider a metaprogram as the process expression of hierarchically organized collaboration of remote component programs. Its SO operating system makes decisions about where, when, and how to run these components. The specification of the service collaboration is a metaprogram—aprogram that manipulates other programs remotely as its data. Nowadays the similar computing abstraction is usually applied to the program executing on a single computer as to the program executing in the network of computers, even though the executing environments (platforms) are structurally completely different. Most, so called, SO programs are still written using software languages such as FORTRAN, C, C++ (compiled into native processor code), Java, Smalltalk (compiled into intermediate code), and interpreted languages such as Perl and Python, the way it usually works on a single host. The current trend is to have these programs and scripts define remote computational modules as service providers.

Instead of moving executable files around the computer networks we can autonomically provision the corresponding computational components (executable codes) as uniform metainstructions of the service metaprocessor. Now we can submit a metaprogram (service command) in terms of metainstructions (services)

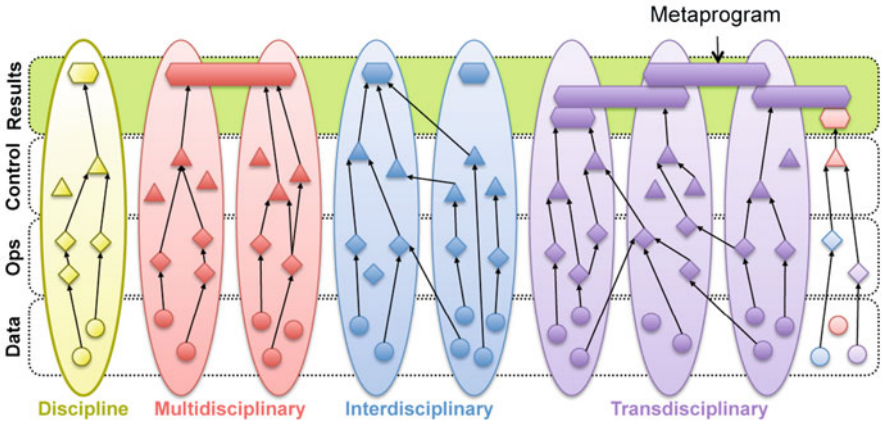

Fig. 1 By providing easy-to-use, self-discovering pervasive services representing domain knowledge (data), tools (operations), and management (control) with relevant metaprogramming methodology, the SORCER environment reduces integration and deployment costs, facilitate productivity, increases research collaboration, and advances the development and acceptance of secure and fault tolerant transdisciplinary concurrent engineering solutions

to the metacompute OS that manages dynamic federations of service providers and related resources, and enables the collaboration of the required service providers according to the metaprogram definition with its own control strategy. We treat services as service types (e.g. a form of the Java interface) and service providers as service instances implementing that service types. A provider can implement multiple service types so can provide multiple services.

One of the first metacompute platforms was developed under the sponsorship of the National Institute for Standards and Technology (NIST)—the Federated Intelligent Product Environment (FIPER) [15]. The goal of FIPER is to form a federation of distributed service objects that provide engineering data, applications, and tools on a network. A highly flexible software architecture had been developed for transdisciplinary computing (1999–2003), in which engineering tools like computer-aided design (CAD), computer-aided engineering (CAE), product data management (PDM), optimization, cost modeling, etc., act as both service providers and service requestors.

The Service-ORiented Computing EnviRonment (SORCER) [17–22, 24] builds on the top of FIPER to introduce a metacomputing operating system with all system services necessary, including service management (rendezvous services), a federated file system, and autonomic resource management, to support service-object oriented metaprogramming. It provides an integrated solution for complex transdisciplinary applications (see Fig. 1) that require multiple complex solutions across multiple disciplines combined at runtime into a transdisciplinary one. The SORCER metacomputing environment adds an entirely new layer of abstraction to the practice of metacomputing—exertion-oriented (EO) programming with a federated method invocation (FMI). The EO programming makes a positive difference

in SO programming primarily through a new metaprogramming abstraction as experienced in many SO computing projects including systems deployed at GE Global Research Center, GE Aviation, Air Force Research Lab, SORCER Lab, and SORCER partners in China.

As we look forward to metacomputing [12] globally distributed physical and virtual machines should play a major role in architecting very large SO systems. While there is a significant interest in high performance, grid [5], and cloud computing [11], 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 SO 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.

This chapter is organized as follows: Sect. 2 introduces used metacomputing and SO terminology; Sect. 3 briefly describes the SORCER metacomputing platform; Sect. 4 describes exertion-oriented programming; Sect. 5 describes var-oriented programming and var-oriented modeling; Sect. 6 presents the SORCER operating system (SOS) with its cloud processor; Sect. 7 describes the service cloud provisioning for exertion-oriented programming, and Sect. 7 provides concluding remarks.

2 Process Expressions and Metacomputing

In computing science the common thread in all computing disciplines are process expression and actualization of process expression [4], 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* [8] (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 metaprocesses consolidated on a meta-platform 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 SO programs run on virtual object-oriented and SO platforms respectively.

In Fig. 2 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, $k = 1, 2, \ldots, n$, 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

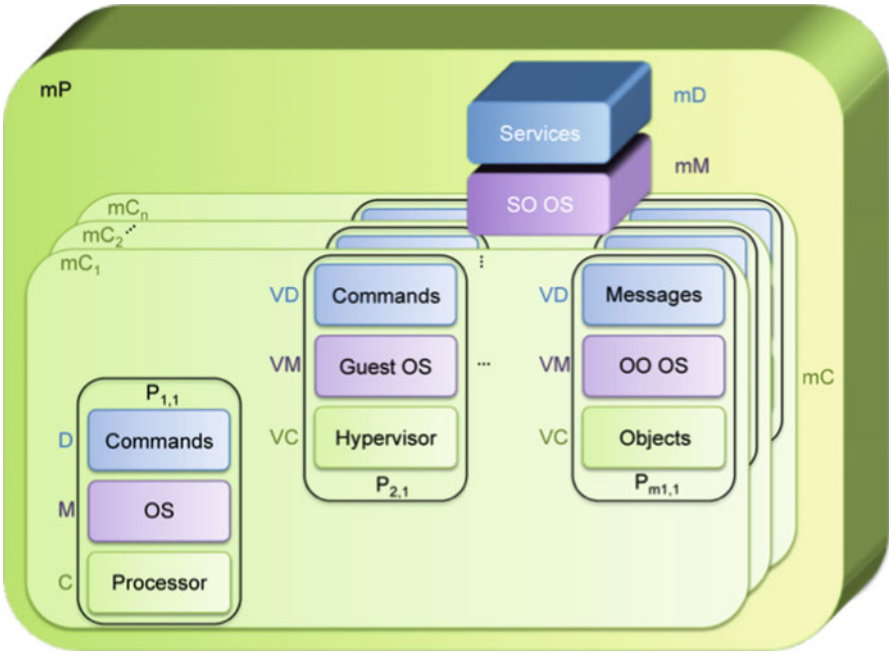

Fig. 2 Software cannot operate without a platform or be platform independent. The cloud platform (mP: mD/mM/mC) with its meta-domain (mD), meta-management (mD), and meta-carrier (mC) is composed of n distributed platform clouds mC_i, $i = 1, 2, \ldots, n$

$\mathrm{Pm}_{1,1}$. All distributed metacarriers mC_i, $i = 1, 2, \ldots, n$, 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 SO computing.

SORCER service commands (services) at the mD level are called exertions [15]. The SOS with its exertion command-line interpreter (exertion shell or SOS shell at nM) allows us to execute SO 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-$\mathrm{P}_{m1,k}$) or commands (at VD-$\mathrm{P}_{1,1}$ and D-$\mathrm{P}_{2,1}$). Each distributed metaprocessor (mC_k) runs multiple virtual platforms ($\mathrm{P}_{i,k}$, $i = 1, 2, \ldots, n$ and $k = 2, \ldots, mi$) on the same native platform (P1,i), while each virtual platform runs services implemented by 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 node* is a remote object that deploys/undeploys service providers. A service node, 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 service object that provides services via its service proxy 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 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 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 provider proxies and all corresponding service objects is predefined and final then this type of SOA is called a *service-protocol oriented architecture* (SPOA). In contrast, when the service proxy-service object communication is based on remote message passing using the wire protocol that can be chosen by a provider's service object to satisfy efficient communication with its requestors, then the architecture is called a

service-object oriented architecture (SOOA). A service in SOOA, the work performed by a service provider, is identified by a service type (interface type).

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 [1, 7].

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 [19]:

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 [23])
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 (Sect. 3) with var-oriented (VO) programming, var-oriented modeling (VM), and exertion-oriented (EO) programming model with FMI in its SOOA. FMI [17] defines the communication framework between three SORCER architectural layers: metamogramming, management, and execution.

3 Service-Object Oriented Platform: SORCER

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 SOOA [20]. The SORCER platform has evolved from the service-object abstractions introduced in the FIPER project (1999–2003 [15]), the SO operating system at the SORCER Lab, Texas Tech University (2002–2009 [24]), finally with metaprogramming languages for programming

convergence at the Multidisciplinary Science and Technology Center, AFRL/WPAFB (2006–2010) [22]. It is based on Jini semantics of services [7] in the network and the Jini programming model [3] 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 SO 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 SO 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 Fig. 3, 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 [17]. The collaboration specifies a view of cooperating providers identified by service types (interfaces)—a projection of service federation. It describes the associations between service types 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 SOS shell. The exertion participants in the federation collaborate transparently according to the exertion's control strategy managed by the SOS based on the Triple Command Pattern described in [19]. The functional SORCER architecture is depicted in Fig. 4 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

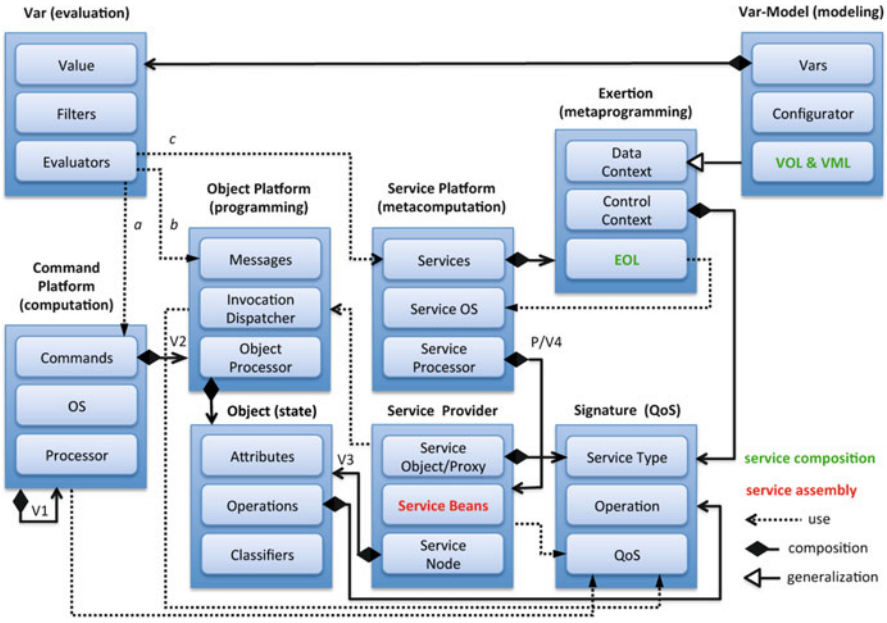

Fig. 3 The SORCER metamodeling architecture: evaluation, modeling, metaprogramming, metacomputation, programming, and computation. Each platform is shown as the DMC triplet (domain, management, carrier). The service platform manages the service providers (service cloud) that are autonomically provisioned by service nodes on virtualized object/command platforms

strategy and control flow exertions [17]. From the computing platform point of view, exertions are service commands at the programming level D in Fig. 4, 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 Fig. 4.

SOOA [20] consists of four major types of network objects: requestors, registries, and service objects with their proxies for remote communication. The provider is responsible for deploying the service object 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 node hosts multiple providers. A service-node container is actualized on the virtual object platform (e.g., Java Platform). Two containers called Tasker and Rio cybernodes [13] are used predominantly to

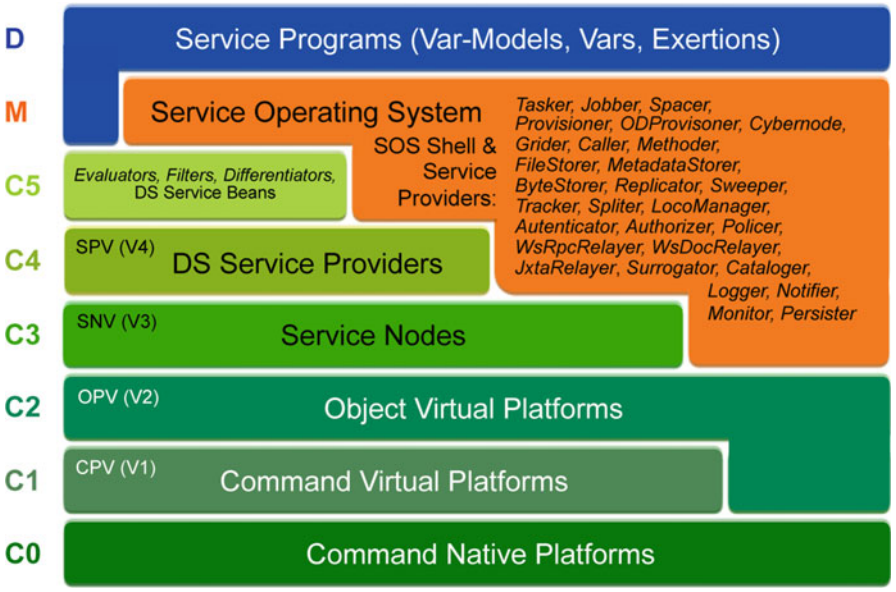

Fig. 4 The SORCER layered architecture, where C0–C5 (carrier)—the metaprocessor with its service cloud at C4 and C3, platform cloud at C2 and C1, M (management)—SORCER operating system, D (domain)—service requestors; where PV and OV stands for provider and object virtualization respectively with the prefix S for service, O for object, and C for command

host service providers. Thus, both service providers/nodes/command platforms can be provisioned by the SOS—service virtualization indicated by SPV/SNV at C4/C3 and platform virtualization indicated by OPV/CPV at C2/C1 in Fig. 4. The SOS uses Jini discovery/join protocols to implement its federated SOOA with its SOS shell.

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 the 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 *SO 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 SO program is composed of metainstructions with its own SO control strategy and data context representing the metaprogram data. Service signatures used in metaprograms, at the D level in Fig. 4, are bound by the SOS to methods of virtual providers at the C4 level that are hosted by service nodes at the C3 level. In turn, the required service nodes 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 execution of its exertion. The SOS 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.

4 Exertion-Oriented Programming Model

In language engineering—the art of creating languages—a *metamodel* is a model to specify a language. An exertion is a metamodel to model connectionist process expression that models behavioral phenomena as the emergent processes of interconnected networks of service providers. The central exertion principle is that a process can be described by the interconnected federation of simple and often uniform and efficient service providers that compete with one another to be *exerted* for a provided service in the dynamically created federation.

Exertion-oriented programming (EOP) is a SO programming paradigm using *service providers* and *service commands*. A service command—*exertion*—is interpreted by the SORCER Operating System (SOS) and represented by the data structure that consist of a *data context*, multiple *service signatures*, and a *control context* together with their interactions—to design distributed applications as service collaborations. In EOP a service signature determines a service invocation on a provider. 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 (*federation*), the control context defines for the SOS how and when the signature operations are applied to the data context. Please note that the service type is the classifier of service providers with respect to its behavior (interface), but the signature is the classifier of service providers with respect to the invocation (operation in the interface) and service deployment defined by its QoS.

An *exertion* is an *expression of a distributed process* that specifies for the SOS how a *service collaboration* is actualized by a collection of providers playing specific roles used in a specific way [18]. The *collaboration* specifies a collection of cooperating providers—the *exertion federation*—identified by the exertion's signatures. 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*.

The exerted members in the federation collaborate transparently according to their *control strategy* managed by the SOS. The SOS invocation model is based on the *Triple Command Pattern* [19] that defines the FMI.

A *task exertion* (or simply a *task*) is an *elementary service command* executed by a single service provider or its small-scale federation. The task federation is managed by the receiving provider for the same service context used by all providers in the federation. A *job exertion* is a *composite service command* defined hierarchically in terms of tasks and other jobs, including control flow exertions [17]. A job exertion is a kind of command script, that is similar conceptually to UNIX script, but with service commands, to execute a large-scale federation. The *job federation* is managed by one of two SOS rendezvous providers, (Jobber or Spacer) but the *task federation* by the receiving provider. Either a task or job is a SO program that is dynamically bound by the SOS to all required and currently available or provisioned on-demand service providers.

The exertion's data called *data context* describes the data that tasks and jobs work on. A data context, or simply a *context*, is a data structure that describes service provider ontology along with related data [19]. Conceptually a data context is similar in structure to a files system, where paths refer to objects instead to files. A provider's ontology (object paths) is controlled by the provider vocabulary 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 ontology as it specifies how the context data is interpreted and used by the provider.

The exertion collaboration defines its *interaction*. The *exertion interaction* specifies how context data flows between invocations of signature operations that are sent between service providers in a collaboration to perform a specific behavior. The interaction is defined by control contexts of all component exertions. From the computing platform point of view, exertions are entities considered at the *programming level*, interactions at the *operating system level*, and federations at the *processor level*. Thus, exertions are programs that define distributed collaborations on the service processor The SOS manages collaborations as interactions on its virtual service processor—the dynamically formed service federations.

The primary difference between *exertion* and *federation* is *management* and *implementation*. The exertion and the federation distinctions are based on the analogies between the *company management* and *employees*: the top-level exertion refers to the *central control* (the Chairman of company) of the behavior of a *management system* (the Chairman's staff vs. component exertions), while federation refers to an *implementation system* (the company employees vs. the service providers) which operates according to management rules (FMI), but without centralized control.

In SORCER the provider is responsible for deploying the service on the network, publishing its proxy 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 SOS 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. 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; the SOS uses discovery/join protocols to obtain service proxies on the network. While the exertion defines the *orchestration* of its service federation, the SOS implements the service *choreography* in the federation defined by its FMI.

Three forms of EOP have been developed: Exertion-oriented Java API, interactive graphical, and textual programming. Exertion-oriented Java API is presented in [17]. Graphical interactive exertion-oriented programming is presented in [16]. Details regarding textual EOP and two examples of simple EO programs can be found in [21, 22].

5 Var-Oriented Programming and Var-Oriented Modeling

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 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.

Var-oriented programming (VOP) is a programming paradigm using service variables called "vars"—data structures defined by the triplet <value, evaluator, filter> together with a var composition of evaluator's dependent variables—to design var-oriented multifidelity compositions. It is based on dataflow principles that changing the value of a var should automatically force recalculation of the values of vars, which depend on its value. VOP promotes values defined by evaluators/filters to become the main concept behind any processing.

Var-oriented modeling (VOM) is a modeling paradigm using vars in a specific way to define heterogeneous multidisciplinary var-oriented models, in particular large-scale multidisciplinary analysis models including response, parametric, and optimization component models. The programming style of VOM is declarative; models describe the desired results of the program, without explicitly listing command or steps that need to be carried out to achieve the results. VOM focuses on how vars connect, 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.

The SORCER metamodeling architecture [22] is the unifying representation for three concrete programming syntaxes: the SORCER Java API described in [17], the functional composition form [21, 22], and the graphical form described in [16]. The functional composition notation has been used for var-oriented language (VOL) and var-oriented modeling language (VML) that are usually complemented with the Java object-oriented syntax.

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. In SORCER the special type of variable called var allows one to use functions, subroutines, or coroutines in the same way. A value of var can be associated with mathematical function, subroutine, coroutine, object, or any local or distributed data. The concept of var links the three languages VOL, VML, and Exertion-Oriented Language (EOL) into a uniform SO programming model that combines federating services (EOP) with other type of process execution.

The semantics of a variable depends on the process expression formalism:

1. A variable in mathematics is a symbol that represents a quantity in a mathematical expression.
2. A variable in programming is a symbolic name associated with a value.
3. A variable in object-oriented programming is a set of object's attributes accessible via operations called getters.
4. A var in SO programming is a triplet <value, evaluator, filter>, where:

   (a) A *value* is a valid quantity in an expression; a value is invalid when the current evaluator or filter is changed, evaluator's arguments change, or the value is undefined;
   (b) An *evaluator* is a service with the argument vars that define the variable dependency composition; and
   (c) A *filter* is a getter operation.

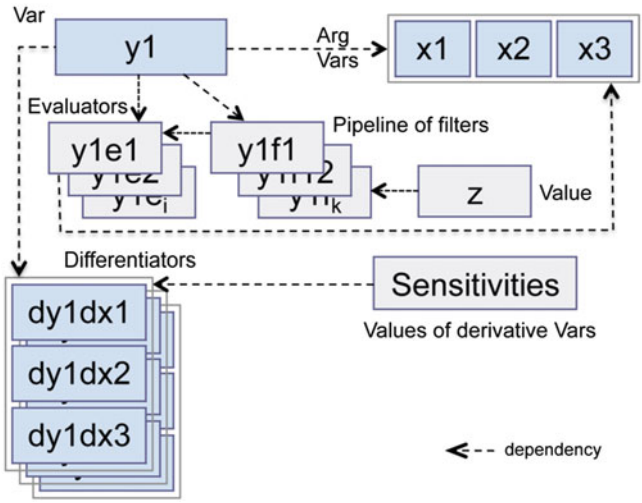

Fig. 5 The var structure: value/evaluator/filter. Vars are indicated in blue color. The basic var $y1$, $z = y1(x1, x2, x3)$, depends on its argument vars and derivative vars

Var-oriented programming is the programming paradigm that treats any computation as the VFE triplet: value, filter (pipeline of filters), and evaluator (see Fig. 5). Evaluators and filters can be executed locally or remotely on command, object, or service platform (see dependencies a, b, and c in Fig. 3). An evaluator may use a differentiator to calculate the rates at which the var quantities change. Multiple associations of evaluator-filter can be used with the same var (multifidelity). The VFE paradigm emphasizes the usage of multiple pairs of evaluator-filter (called var evaluations) to define the value of var. The semantics of the value, whether the var represents a mathematical function, subroutine, coroutine, or just data, depends on the evaluator and filter currently used by the var.

A service in VOL is the work performed by a variable's evaluator-filter pair. Multiple evaluators for a dependent var, that depend on their argument vars, define:

1. A var composition via the var arguments of its evaluator
2. Multiple processing services (mutifidelity)
3. Multiple differentiation services (mutifidelity)
4. Evaluators can execute commands (executable codes), object-oriented services (method invocations), and exertions (exerting service federations).

Thus, in the same process various forms of services (intra and interprocess) can be mixed within the same process expression in VOL. Also, the fidelity of var values can change as it depends on a currently used evaluator. Please note that vars used in data contexts of exertions extend EOP for the flexible service semantics defined by VOP.

The variable evaluation strategy is defined as follows: the var value is returned if is valid, otherwise the 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 and makes that value valid. 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 the var composition and provides the integration framework for all possible kinds of computations represented by various types of evaluators including exertions via exertion evaluators.

In general, it is perceived that the languages used for either modeling or programming are different. However, both are complementary views of process expression and after transformation and/or compilation both need to be executable. An initial model, for example an initial design of aircraft engine, can be imprecise, not executable, at high level with informal semantics. However its detailed model (detailed design) has to be precise, executable, low level, with execution semantics. Differences between modeling and programming that traditionally seemed very important are becoming less and less distinctive. For example models created with Executable UML [8] are precise and executable.

Data contexts (objects implementing SORCER's Context interface) with specialized aggregations of vars are called var-models. Three types of analysis models: response, parametric, and optimization have been studied already [21]. These models are expressed in VML using functional composition and/or Java API for var-oriented modeling.

The modularity of the VFE framework, reuse of evaluators and filters, including exertion evaluators, in defining var-models is the key feature of var-oriented modeling (VOM). The same evaluator with different filters can be associated with many vars in the same var-model. VOM integrates var-oriented modeling with other types of computing via various types of evaluators. In particular, evaluators in var-models can be associated with commands (executables), messages (objects), and services (exertions) as shown in Fig. 3 by dependencies a, b, and c.

Var-models support multidisciplinary and multifidelity traits of transdisciplinary 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 process ("shape") of the model and quickly update the related computations 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. $f = x1^2 - 5.0 \times x1 + x2^2 - 5.0 \times x2 + 2.0 \times x3^2 - 21.0 \times x3 + x4^2 + 7.0 \times x4 + 50.0$

4. $g1 = x1^2 + x1 + x2^2 - x2 + x3^2 + x3 + x4^2 - x4 - 8.0$
5. $g2 = x1^2 - x1 + 2.0 \times x2^2 + x3^2 + 2.0 \times x4^2 - x4 - 10.0$
6. $g3 = 2.0 \times x1^2 + 2.0 \times x1 + x2^2 - x2 + x3^2 - x4 - 5.0$

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 configureAnalysis-Model 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 service provider and used by the space exploration provider of the Exploration type that also uses the CONMIN optimization [2] 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

loger.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 [14], for example, the optimizer's signature can be expressed as follows:

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

wherethe `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", processor(
         entry("Available", ""))),
       var("cpu_util", sla(
         entry("CPU", null))))))),
   authorization(
     entry("estimatedDuration", 30000),
     entry("priority", range(5,10)),
   execDate("2012-01-10 00:00:00", "2012-01-11
            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 the required QoS: for the provider by operator `serviceProvider`, for the object platform by operator `objectPlatform`, for the command platform by operator `commandPlatform`, for the expected SLA by operator `sla` and authorization by `authorization` for the service requestor.

6 SORCER Operating System and Service Clouds

The SOS allows execution of a SO program and by itself is the SO system. The overlay network of the service providers defining the functionality of the SOS is called the *sos-cloud* (M layer in Fig. 4) and the overlay network of domain specific (application) providers is called the *app-cloud*—metaprocessor (C4 layer in Fig. 4). 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. 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. When services of a processor are provisioned (see Fig. 6) we call this type of processor a service cloud.

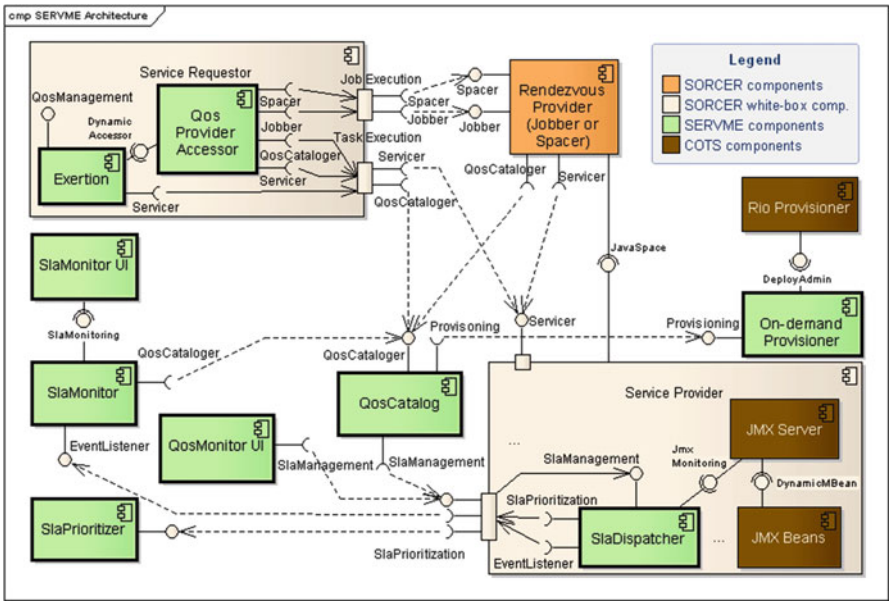

Fig. 6 SERVME resource management architecture

As explained in Sect. 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. The 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 the 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 using the exertion command-line interpreter (exertion shell), or programmatically 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 S2S 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, Transaction)`—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, the 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 [21] (see Fig. 4):

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–service provider registries, provide management for QoS-based federations;
g) SlaMonitors–provide monitoring of SLAs;
h) Provisioners–manage 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 providers (see at C4 in Fig. 4) 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 [18] 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.

7 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. User defined QoS attributes or QoS context in signatures are optional. However, if used, the required command platform (V1 in Fig. 3), object platform (V2 in Fig. 3), and provider (V3 in Fig. 3), 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) [14] 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 Sect. 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 Fig. 6. We distinguish two forms of autonomic provisioning: monitored and on-demand. In monitored provisioning the Rio Provisioner [13] 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.
- `OnDemandProvisioner` enables on-demand provisioning of services in cooperation with the Rio Provisioner [13]. 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, 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 Sect. 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 `QosProvider-Accessor` 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.

8 Conclusions

Cloud computing it is not just computing on a collection of virtualized platforms. A SO 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 (Fig. 3). 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 *SOS shell*. The exertions are service commands that define reliable network collaborations in unreliable networks.

Applying the inversion principle, the SOS looks up service providers by implemented interface types with optional QoS attributes. The 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. The 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 (expiration of leases for service proxies) without need for reconfiguration. This allows the exertion federation to be distributed without sacrificing the robustness of the SO 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 presented on-demand provisioning reduces the number of compute resources to those presently required for collaborations

Table 1 UNIX OS vs. SORCER OS. Pipes in UNIX are between processes, in SORCER they are between data contexts; instead of UNIX pipeline SORCER defines a workflow by composition of exertions; the UNIX shell is local but the SOS shell is a network shell

Features	UNIX	SOS
Data	File/file system	Object/data context
Data flow	Pipes	Data context pipes
Interpreter	e.g., C Shell	SOS shell
Programming language	Shell scripting—procedural	1. VOL, VML, and EOL scripting—declarative 2. Interactive visual programming 3. Java API
System (SW) language	C	Java/Jini/Rio

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 of 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".

As we move from the problems of the *information era* to more complex problems of the *molecular era*, it is becoming evident that new programming languages for transdisciplinary computing are required. These languages should reflect the complexity of metacomputing problems we are facing in SO computing, for example, concurrent engineering processes of the collaborative design by hundreds of people working together and using thousands of programs written already in software languages (languages for computers) that are dislocated around the globe. The transdisciplinary design of an aircraft engine or even a whole air vehicle requires large-scale high performance metacomputing systems handling anywhere-anytime executable codes represented by software languages.

DSLs are for humans, intended to express specific complex problems and related solutions. Three programming languages for transdisciplinary computing are described in this paper: VOL, VML, and EOL. The SOS shell interprets these languages. The essential differences between the UNIX operating system, and the SOS are illustrated in Table 1.

As complexity of problems being solved increases continuously, we have to recognize the fact that in transdisciplinary computing the only constant is change. The concept of the evaluator–filter pair in the VFE framework provides the uniform service-orientation for all computing and metacomputing needs with various applications, tools, utilities, and exertions as cloud services.

The SORCER operating system supports the two-way convergence of three programming models for transdisciplinary computing. On one hand, EOP is uniformly converged with VOP and VOM to express an explicit network-centric SO computation process in terms of other implicit (inter/intra) process expressions (the network is the computer). On the other hand, VOM and VOP are uniformly converged with EOP to express an explicit declarative transdisciplinary process in terms of other implicit (intra/inter) process expressions including exertions (the computer is the network).

SORCER defines clearly its separate metacomputing architectural layers: metamogramming, management, and cloud processor layers integrated via the SOS. That introduces simplicity to SORCER mogramming with flexible enterprise interoperability achieved via three neutralities (protocol, implementation, and location [19]) and architectural means, not by neutral data exchange formats, e.g., XML. Neutral data exchange formats when overused introduce unintended complexity and degraded performance.

The SO 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 [6, 9, 10, 25].

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 Fe-derated 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

1. Apache River, Available at: http://river.apache.org/. Accessed 5 September 2011
2. CONMIN User's Manual, Available at: http://www.eng.buffalo.edu/Research/MODEL/mdo.test.orig/CONMIN/manual.html. Accessed 5 September 2011
3. Edwards, W.K. 2000 *Core Jini*, 2nd ed., Prentice Hall Ptr, ISBN-13: 978-0130144690
4. 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
5. Foster I.; Kesselman C. and Tuecke S., 2001. The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International J. Supercomputer Applications*, 15(3)

6. Goel, S.; Talya, S. S. and 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
7. Jini Network Technology Specifications v2.1. Available at: http://www.jiniworld.com/doc/spec-index.html. Accessed 5 September 2011
8. Kleppe A., 2009. *Software Language Engineering*, Pearson Education, ISBN: 978–0–321–55345–4
9. Kolonay, R. M., Thompson, E. D., Camberos, J. A. and 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
10. Kolonay, R. M. and Sobolewski M., 2011. Service ORiented Computing EnviRonment (SORCER) for Large Scale, Distributed, Dynamic Fidelity Aeroelastic Analysis & Optimization, *International Forum on Aeroelasticity and Structural Dynamics,*IFASD2011, 26–30 June, Paris, France
11. Linthicum, D. S., 2009. *Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide*, Addison-Wesley Professional, ISBN-10 0136009220
12. Metacomputing: Past to Present, February 30, 2011. Available at: http://archive.ncsa.uiuc.edu/Cyberia/MetaComp/MetaHistory.html. Accessed 5 September 2011
13. Rio Project. Available at: http://www.rio-project.org/. Accessed 5 September 2011
14. Rubach, P. and Sobolewski, M., 2009. Autonomic SLA Management in Federated Computing Environments. *International Conference on Parallel Processing Workshops*, Vienna, Austria: 2009, pp. 314–321
15. Sobolewski, M., 2002. Federated P2P Services in CE Environments, *Advances in Concurrent Engineering, A.A. Balkema Publishers, Taylor and Francis, 2002*, ISBN 90 5809502 9, pp. 13–22
16. Sobolewski, M. and Kolonay, R., 2006 Federated Grid Computing with InteractiveService-oriented Programming, *International Journal of Concurrent Engineering: Research & Applications*, Vol. 14, No 1., pp. 55–66
17. Sobolewski, M., 2008. Exertion Oriented Programming, IADIS, vol. 3 no. 1, pp. 86–109, ISBN: ISSN: 1646–3692
18. Sobolewski, M., 2008. Federated Collaborations with Exertions, 17h IEEE International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), pp.127–132
19. 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 5 September 2011. Available at: http://sciyo.com/articles/show/title/metacomputing-with-federated-method-invocation
20. Sobolewski, M., 2010. Object-Oriented Metacomputing with Exertions, *Handbook On Business Information Systems*, A. Gunasekaran, M. Sandhu (Eds.), World Scientific Publishing Co. Pte. Ltd, ISBN: 978–981–283–605–2
21. Sobolewski, M., 2010. 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
22. Sobolewski, M., 2011. Provisioning Object-Oriented Service Clouds for Exertion-Oriented Programming, Proceedings of CLOSER 2011 - International Conference on Cloud Computing and Services Science, pp. IS-11–IS-25
23. Sobolewski, M., 1996. Multi-Agent Knowledge-Based Environment for Concurrent Engineering Applications. *Concurrent Engineering: Research and Applications(CERA)*, Technomic. Available at: http://cer.sagepub.com/cgi/content/abstract/4/1/89
24. SORCERsoft. Available at: http://sorcersoft.org. Accessed 5 September 2011
25. 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