Exerted Enterprise Computing: From Protocol-Oriented Networking to Exertion-Oriented Networking

Michael Sobolewski

SORCER Research Group
Polish-Japanese Institute of Information Technology,
Warsaw, Poland
sobol@sorcersoft.org

Abstract. Most enterprise computing programs are still *not* written in metaprogramming languages but rather composed line by line in software programming languages as they were decades ago. These programming languages are poorly suited to expressing enterprise processes targeted at complex, domain-specific and transdisciplinary problems. The current state of the art is that legacy programs and scripts can be used as programming instructions provided by *dynamic service objects*. New metaprograms (programs of programs) require *relevant operating systems* managing service objects as a virtual service *metaprocessor*. However, there are presently no acceptable metaprogramming methodologies to program, deploy, and dynamically federate these relevant service objects into a virtual processor securely and efficiently with fault detection and recovery. In this paper the emerging metacomputing SORCER platform with its federated method invocation and exertion-oriented programming model is contrasted with *service protocol-oriented architectures* (e.g., OGSA, CORBA, RMI) which limit us to one fixed wire protocol, static network configurations, and often restricts us to heavyweight containers (e.g., application servers) for hosting service objects.

Keywords: process expression, metacomputing, service-oriented computing, SOA, dynamic service objects.

1 Introduction

"Computing's core challenge is how not to make a mess of it." Edsger Dijkstra

Computing has evolved over centuries. It is and always has been about processes and process expressions. The creation or activation of process expressions has changed over time as it reflects the continuous change in problems being solved by humans and the languages used. As a current example, UML behavior diagrams allow us to define multiple process expressions that generalize flowchart diagrams which were introduced by Markov in 1954 to represent "algorithms" [15, 5].

As we reach adolescence in the Internet era we are facing the dawn of the metacomputing era, an era that will be marked not by PCs, workstations, and servers, but by computational capability that is embedded in all things around us—virtual

R. Meersman et al. (Eds.): OTM 2010 Workshops, LNCS 6428, pp. 182–201, 2010.

computing services as programming instructions of a virtual metacomputer. The term "metacomputing" was coined around 1987 by NCSA Director, Larry Smarr: "The metacomputer is, simply put, a collection of computers held together by state-of-the-art technology and balanced so that, to the individual user, it looks and acts like a single computer. The constituent parts of the resulting metacomputer could be housed locally, or distributed between buildings, even continents." [16]

In computing science the common thread in all computing disciplines are *process expression* and *actualization of process expression* [5], 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 *program* is an expression of a computing process.
4. A *programming language* is an environment within which to create symbolic process expressions.
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 its programming language, operating system, and processor.
10. A *computer* is an actualization of a computing platform.
11. A *metaprogram* is an expression of a metaprocess, as the *process of processes*.
12. A *metaprogramming 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 metaprogramming language, metaoperating system, and metaprocessor.
16. A *metacomputer* is an actualization of a metacomputing platform.
17. *Enterprise computing* is an expression of transdisciplinary enterprise processes.

Obviously, there is an essential overlap between the domains of mathematics and computer science, but the core concerns with the nature of process expression itself are usually ignored in mathematics since mathematicians are concerned with the nature of the behavior of a process independent of how that process is expressed. Computing science is concerned with computing processes and computer science is mainly concerned with the nature of the expression of processes independent of its process. In Fig. 1, the difference between programming and metaprogramming is illustrated where a metaprogram on its metaprocessor is the program of programs on multiple processors.

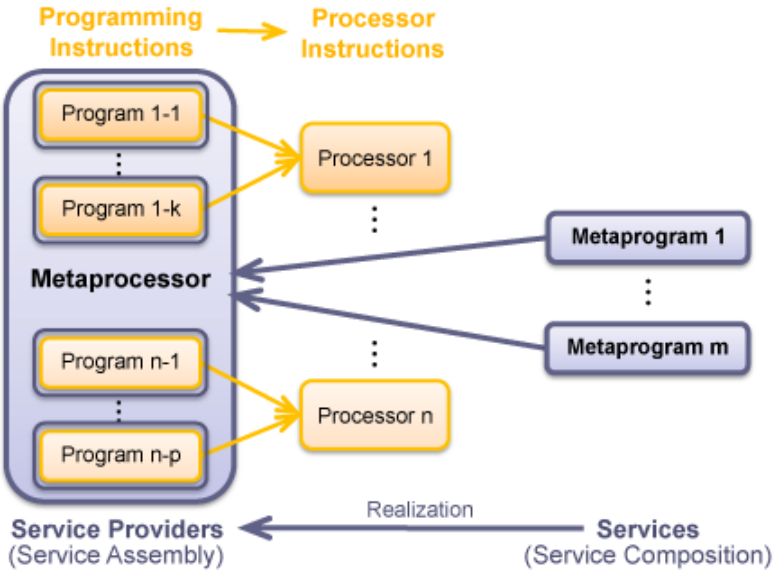

Fig. 1. The programming structure is indicated by yellow colors and the metaprogramming structure by bluish colors. Programming instructions are realized by native processor instructions, but metainstructions by services (with blue outlines) invoking legacy programs (with yellow outlines). Metaprogramming is focused on service compositions and metacomputer engineering on construction of metaprocessors—service assemblies from other services and modules.

Service providers expose existing programs that execute on a network of processors (see Fig. 1) as service types. These service types, e.g. Java interfaces, are implemented by service objects hosted by a provider. The service objects just consume services and provide services from and to each other respectively. Applications are increasingly moving to the network—self aware, autonomic networks that are always fully functional. A service provider exposes multiple interfaces implemented by its service objects that in turn provide instructions for the service-oriented processor (metaprocessor). Most current efforts in service systems are focused on *service-oriented engineering*—constructing metaprocessors by assembling service objects from other services and modules (e.g., OSGi, SCA, BPEL).

Thus, the metaprocessor via its operating system carries access to applications, tools, and utilities, i.e., programs exposed by service objects. Service providers can *federate* with each other dynamically to provide *service collaborations*—to realize a metaprogram—the service-oriented expression of the metaprocess.

The SORCER [20-24] service-oriented system is the enterprise platform based the service-oriented philosophy outlined above. Its architecture is derived from the metaprogramming model with three languages that allow for model-driven programming with service collaborations (Section 3.2). It supports three core neutralities [22]: requestor/provider wire protocol [28], provider implementation, and provider location in the network.

Let's consider the "Hello Service Arithmetic" example. Assume we have three services on the network:

`f3 = x1 - x2; f4 = x1 * x2; and f5 = x1 + x2`

which implement three interfaces: `Subtractor`, `Multiplier`, and `Adder`, respectively. We want to program a distributed service that mimics a function composition:

`f3(f4, f5)` and calculate: `f3(f4(10.0, 50.0), f5(20.0, 80.0))`

to get `400.0` as the result of collaboration of three services: `f4`, `f5`, and `f3`. Consider the equivalent service-oriented program (workflow) that can run in SORCER:

```
String arg = "arg", result = "result";
String x1 = "x1", x2 = "x2", y = "y";
Task f3 = task("f3", op("subtract", Subtractor.class),
  context("subtract", in(path(arg,x1), null),
    in(path(arg,x2),null), out(path(result,y),null)));
Task f4 = task("f4", op("multiply", Multiplier.class),
  context("multiply", in(path(arg, x1), 10.0),
    in(path(arg,x2), 50.0), out(path(result,y),null)));
Task f5 = task("f5", op("add", Adder.class),
  context("add", in(path(arg,x1), 20.0),
    in(path(arg,x2), 80.0),out(path(result,y),null)));
Job f1= job("f1",
  job("f2",f4,f5,strategy(Flow.PARALLEL, Access.PULL)),
  f3,
  pipe(out(f4, path(result,y)), in(f3, path(arg,x1))),
  pipe(out(f5, path(result,y)), in(f3, path(arg,x2))));
return value(exert(f1), path("f3", result));
```

The first two lines define the names of the arguments used in this program. Next, three tasks `f3`, `f4`, and `f5` are declared from which two composite services are declared: `f1` and `f2`. A few metalanguage operators are used in the program to define services: `op` (short for *operation*) defines the *service operation* by its name in the requested service type, e.g., the operation `"subtract"` in the Java interface `Subtractor.class` in `f3`; operators `in`, `out`, and `inout` specify service input and output parameters by `paths` in the associative array called `context`. The expressions that start with the operator `task` or `job` are called *exertions.* Exertions specify service compositions and define the *process* by its control *strategy* expressed by the `strategy` operator in jobs. Service compositions (exertions) define virtual services created from other services. Tasks are elementary services and jobs are compound services in exertion-oriented programming.

The program above defines a function composition `f3`:

`f3(f4(x1, x2), f5(x1, x2)),`

as a SORCER service composition `f1`:

`f1(f2(f4(x1, x2), f5(x1, x2)), f3).`

Task `f4` requests operation `"multiply"` of its arguments `"arg/x1"` and `"arg/x2"` by the service `Multiplier.class`. Task `f5` requests operation `"add"` of its arguments `"arg/x1"` and `"arg/x2"` by service `Adder.class`. Task `f3` requests to `"subtract" "arg/x2"` from `"arg/x1"` by `Subtractor.class` where input parameter values are not defined yet. Job `f2` requests execution of both `f4` and `f5` with its process strategy:

```
strategy (Flow.PARALLEL, Access.PULL))
```

This means that the component services `f4` and `f5` of `f2` are executed in parallel and the corresponding service objects will not be accessed directly (`PUSH`) by the SORCER OS. In this case the corresponding service objects will process their tasks via the SORCER shared exertion space (`PULL`) when they are available to do so [21]. The default control strategy is sequential (`Flow.SEQUENTIAL`) execution with `PUSH` access, which is applied to job `f2`.

Finally the job `f1`, executes first job `f2` and then via data pipes (defined with the `pipe` operator in `f1`) passes the results of tasks `f4` and `f5` on to task `f3` for `"arg/x1"` and `"arg/x2"` correspondingly. The last statement in the above program exerts the collaboration `exert(f1)`. Exerting means executing the service collaboration and returning the exertion with the processed contexts of all component exertions along with operational details like execution states, errors, exceptions, etc. Then it returns the value of the service collaboration `f1` with the path `path("f3", result)`, which selects the value `400.0` from the context of executed task `f3` at the path `"result"`. The single service activation, `exert(f1)`, creates at runtime a dynamic federation of required collaborating services with no network configuration. This type of process is referred to as "federated".

The rest of the paper is organized as follows: Section 2 differentiates metacomputing from computing and defines metacomputing concepts used in SORCER. Section 3 presents the SORCER platform with its metaprogramming languages and metaoperating system and Subsection 3.5 illustrates how to implement service-objects to execute the service-oriented program presented above. This is followed by concluding remarks and plans for future work.

2 From Computing to Metacomputing

From the very beginning of networked computing, the desire has existed to develop protocols and methods that facilitate the ability of people and automatic processes to share resources and information across different computing nodes in an optimized way. As ARPANET [14] began through the involvement of the NSF to evolve into the Internet for general use, the steady stream of ideas became a flood of techniques to submit, control, and schedule jobs across distributed systems. The latest in these ideas are the grid [6, 25, 26] and cloud [13], intended for use by a wide variety of different applications in a non-hierarchical manner to provide access to powerful aggregates of resources. Grids and clouds, in the ideal, are intended to be accessed for computation, data storage and distribution, visualization, and display, among other applications, without undue regard for the specific nature of the hardware and

underlying operating systems on the resources on which these jobs are carried out. While a grid is focused on computing *resource* utilization, clouds are focused on virtualization. In general, grid and cloud computing are client-server architectures that abstract away the details of the server—one requests a *resource* (service), not a specific *server* (machine). However, both terms are vague from the point of view of computing process expression and relevant programming models and referring to "everything that we already do" by providing various middleware architectures that are not only difficult to use but difficult for the end users to understand.

The concept of "middleware" has remained largely unchanged since client-server computing emerged in the late 1980s. It's software that provides a link between separate software applications or services. Middleware sits "in the middle" between application software that may be executing on different operating systems. Middleware consists of a set of services that allow multiple processes running on one or more machines to interact. The distinction between operating system and middleware functionality is, to some extent, arbitrary. Additional services provided by separately-developed middleware can be integrated into operating systems when needed.

Either middleware or an operating system (OS) is the expression of a process that manages the interpretation of *other process expressions*. Thus, to express a service-oriented (SO) process we need a service-oriented OS, but also we need an *expression of an SO process*. For the latter we need an *SO program* and the corresponding SO processor to *activate it* according to the OS *interpretation*. Thus, the SO process is expressed by three complementing each other process expressions:

1. *expression of an SO process*—the SO program;
2. *management of the service collaboration* representing the SO program—the SO operating system; and
3. activation of the SO collaboration—the SO processor.

Service architectures can be distinguished by the type of application metaprogramming language and related metaoperating system. Most existing service architectures are focused mainly on service provider assemblies at the middleware level (OSGi [OSGi Alliance], BPEL [11], Globus/Condor [26]), but not the metaprogramming by end users. It is reminiscent of the 60s when job schedulers were used while operating systems with high level programming environments were still in the development phases and only low-level application programming for job schedulers was available.

Lack of application metaprogramming languages is the main source of confusion regarding what SO programming is all about. It is still very difficult for most users to create user-defined SO programs. Instead of domain-specific SO programs, 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 enterprise techniques, and a multiplicity of specialized “enterprise-aware” tools that are not, in fact, aware of each other which defeats the basic purpose of the grid or cloud.

Let's consider, for example, Web Services (WS) [4], OSGi, and Jini [10, 1] architectures. Each is a service architecture but *built for different service semantics.* WS is a service architecture for distributed systems that are built on a static middleware

fixed on the XML/WSDL/SOAP/BPEL and running on Application Servers. OSGi is a service architecture (at least by name) for services that are in the same process address space. Jini is a service architecture for distributed systems that is built out of dynamic service objects that are separated by an unreliable network [4]. Each allows allow you to build programs out of collaborating services with detailed programming required. Each has a completely different concept of service that the user has to be familiar with. The major difference is in the type of collaboration you can create and how you can create service collaborations. Also, the unreliable network (Jini) is a very different environment [4] from the single virtual machine (OSGi), or an Application Server used for WS deployment.

Creating a collaboration of services in any of the three environments is easy for neither end users nor developers. Creating collaborations of services coming from all three environments in a uniform way is not possible and no metaprogramming is available that would differ from middleware programming. These environments are mainly focused on metaprocessor but not on three intrinsic layers of SO computing: *SO programming* (metalanguage), *SO management* (middleware), and *SO execution* (dynamic federations of service providers).

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 (physically expressed). 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 programs to run require a *processor*, *operating system*, and *programming environment* with related tools to create symbolic process expressions—*programs*. A computation is physically expressed by a processor and symbolically expressed by a program.
- A *distributed computation* allows for sharing computing resources usually llocated on several remote computers (compute nodes) to collaboratively run a single complex computation in a transparent and coherent way. In distributed computing, computations are decomposed into programs, processes, and compute nodes.
- A *metacomputer* is an interconnected and balanced set of compute nodes that operate as a single unit, which is accessible by its computing platform (*metaprocessor*, *metaoperating system*, and *metaprogramming environment*).
- A *metacomputation* is a form of distributed computation (a computation of computations) determined by *collaborating computations* that a metacomputer can interpret and execute. A *service object* selected at runtime by a metaoperating system implements metainstructions that invoke what are usually legacy programs.
- A collection of service providers selected and managed for a metacomputation is called a *virtual metaprocessor*.
- A *metaprogram* is an expression of metacomputation, represented in a *programming language*, which a *metacomputer* follows in processing shared data for a *service collaboration* managed by its *metaoperating system* on its virtual *metaprocessor*.

- A *service object* is a remote object that provides services to other service objects. *Service objects* are identified primarily by service 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* that they offer to service requestors. A service object that implements multiple interfaces provides *multiple services*. A *service provider* makes interfaces of multiple service objects available on the network.
- 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 programming. Service objects 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 end user or application) to dynamically find service objects on the network.
- If the application (wire) protocol between requestors and all service providers is predefined and constant then this type of SOA is called a *service-protocol oriented architecture* (SPOA). In contrast, if the communication is based on message passing and the wire protocol 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 [10].

The platforms and related programming models have evolved as process expression has evolved from the sequential process expression activated on a single computer to the concurrent process expression activated 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 [22]:

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 programming (e.g., DICEtalk [19])
6. Service-protocol oriented programming (e.g., Web and Grid Services)
7. Service-object oriented programming (e.g. SORCER)

SORCER introduces an exertion-oriented (EO) programming model with federated method invocation (FMI) in its SOOA. FMI defines the communication framework between three SORCER architectural layers: SO programming, 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 dynamic 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). It is based on Jini semantics of services [10] in the network and the Jini programming model [3, 1] with explicit leases, distributed events, transactions, and discovery/join protocols. Jini focuses on service management in a networked environment, SORCER is focused on exertion-oriented (EO) programming and the execution environment for exertions (see Fig. 2).

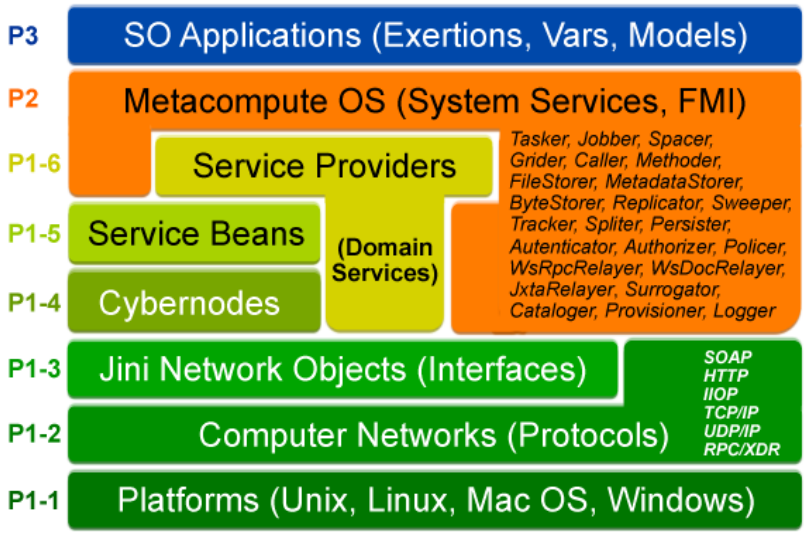

Fig. 2. SORCER layered architecture, where P1 metaprocessor, P1-6 application services, P2 operating system services, P3 programming environment

3.1 Exertion-Oriented Programming

The programming example presented in the Introduction implies that an expression of service (task or job) in the EO declarative language can be written as one line—the feature of functional programming. For example `f1` can be rewritten in one line by substituting references to the component exertions by their corresponding expressions.

The operators in the EO language, described in the Introduction, correspond to Java interfaces and classes [20]. The SORCER framework almost entirely designed in terms of Java interfaces. To explain how SORCER works we will refer to a few Java interfaces and classes in the remainder of this paper. For example, operators `task` and `job` return objects that are defined by the `Exertion` interface with corresponding reference implementations: `ServiceTask` and `ServiceJob` respectively. Thus for each EO operator there is a corresponding Java object. To avoid potential confusions of concepts with the dual representation in declarative language and implementation language we will refer to a declarative language concept as defined so far and appending "object" when referring to the corresponding Java type. For example, an "exertion" is an expression in the EO language and an "exertion object" as one implementing the `Exertion` interface.

An *exertion* is an expression of a *service collaboration* realized by both metaoperating system providers (in short *mos-providers*) and application providers (in short *app-providers*). For each exertion the mos-federation is formed *dynamically* to reflect the exertion's recursive service composition and control strategy [23]. The mos-federation manages for the exertion late bindings to the required app-providers in the *dynamically* formed app-federation (exertion's metaprocessor). The app-federation represents *service objects* that implement all exertion operations. Thus, the mos-federation provides the functionality of the SORCER OS (SOS) and the app-federation provides the functionality of the SORCER metaprocessor (SMP).

Please note that exertion objects are entities that encapsulate explicitly *data*, *operations*, and *control strategy*. SOS uses service compositions, interfaces, and control strategies, but data contexts and corresponding methods are used by SMP. The interfaces are dynamically bound to corresponding service-objects at runtime even to those that have to be provisioned on-demand. The service objects in the app-federation execute the exertion's operations transparently according to the exertion's *control strategy* managed by SOS. The SORCER *Triple Command Pattern* [9] defines federated method invocation (FMI) that integrates SOS with SMP. FMI is presented in more detail in Section. 5.4 [22]

From the SORCER platform point of view, exertions are entities at the *EO programming level*, sos-federations at the *SOS level*, and app-federations at the *SMP level*. Thus, an exertion represents the process of the cooperating SOS and SMP service providers (see Fig. 3).

The primary difference between the sos-federation and the app-federation is *management* and *execution*. The sos-federation and the app-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 the company—binding the top-level exertion to SOS) of the behavior of a *management system* (the Chairman's staff—sos-federation), while the app-federation refers to *the execution system* (the company employees—the service objects) that operates according to execution rules (SORCER FMI), but without centralized control.

The SORCER SOOA consists of three major types of remote objects: service providers, registries, and proxy objects. 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

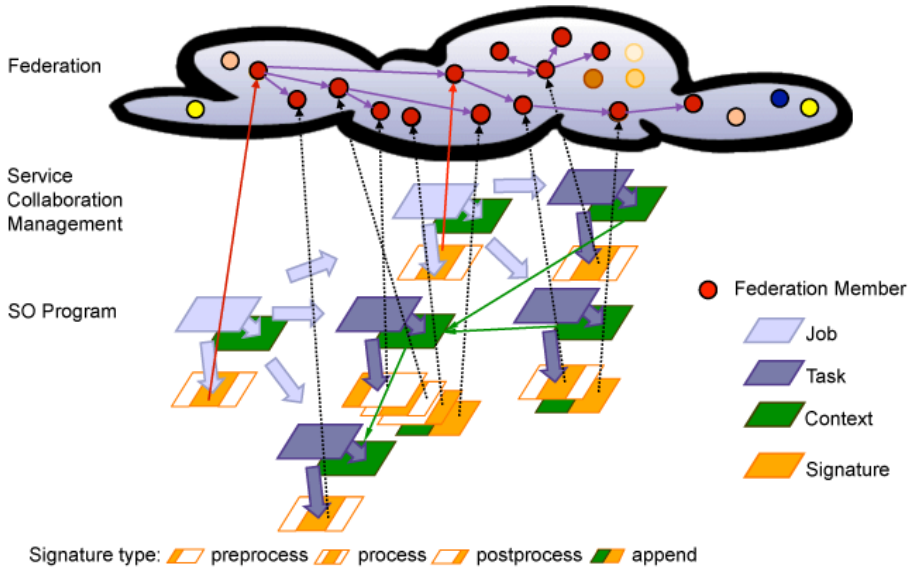

Fig. 3. Exertions and federations. The top-level exertion with component exertions is depicted below the service cloud. Green arrows between data contexts show data flow (context pipes). The solid red lines indicate late bindings to operating system services. Late bindings to all application services defined by the exertion signatures are indicated by dashed lines. The providers in the cloud, in red color, form the service federation—metaprocessor.

network only while present; registries intercept these announcements and cache proxy objects to the provider services. The requestor (e.g., exertion) discovers registries and then looks up proxies by sending queries to registries and making selections from the available service types. Queries contain search criteria (defined by the `op` operator) 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; requestors use discovery/join protocols to obtain service proxies on the network. SORCER uses Jini discovery/join protocols to implement dynamic service management for its SOS and SMP. Exertion objects are requestors capable of dynamically finding sos-providers, for example dynamically looking up or provisioning on-demnd `Taskers` and `Jobbers` that in turn manage corresponding app-federations.

A task object is an elementary command managed by a SOS provider of the `Tasker` type. A `Tasker` can provide a single service by itself or can manage a small-scale federation for the same data context used by all providers in its federation. A job object is defined hierarchically in terms of tasks and other jobs, including control flow exertions [22]. A job object is a composite command managed by rendezvous providers of `Jobber`, `Spacer`, or `Cataloger` type managing hierarchical large-scale collaborations.

The exertion's data, called a *data context* [20], describes the data that `Taskers` work on. A data context, or simply a *context*, is an associative array that describes

service provider ontology along with related data. A provider‘s ontology is controlled by the provider vocabulary that describes data structures in a provider's namespace within a specified service domain of interest. A requestor defining an exertion has to comply with that ontology as it specifies how the context data is interpreted and used by the provider. The notion of context is derived from the knowledge representation scheme called percept calculus [19]. Thus, data context can be used as a knowledge base the same way it is used in the DICEtalk platform [19] or as a var-oriented model presented in Section 3.2.

3.2 Var-Oriented Programming

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 SORCER var-oriented (VO) framework 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, or any data (object).

VO programming is a programming paradigm that treats any computation as the triplet: *value*, *evaluator*, and *filter* (VEF). Evaluators and filters can be executed locally or remotely, sequentially or concurrently. In particular, evaluators and filters can be considered as exertions, service providers, or conventional programs as indicated by green arrows in Fig. 4. The paradigm emphasizes the usage of *evaluators* and a *pipeline of filters* to define the variable value. Semantics of a *var*, whether it's a mathematical function, subroutine, coroutine, or just a value (object) depends on the evaluator type and pipeline of filters used with the variable. VO programming allows for exertions to use vars in data contexts. Alternatively, data contexts (implementing `Context` interface) with specialized structures of vars, called *VO models,* can be used for enterprise-wide metacomputing. Three VO analysis models: *response*, *parametric*, and *optimization* have been studied already.

The variable evaluation strategy is defined as follows: the associated current evaluator determines the variable’s raw value, and the current pipeline of filters returns the output value. 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 vars in turn can depend on other argument vars and so on. This var dependency chaining is called VO composition and provides in SORCER the integration framework for all possible types of computations represented by various types of evaluators including exertion evaluators.

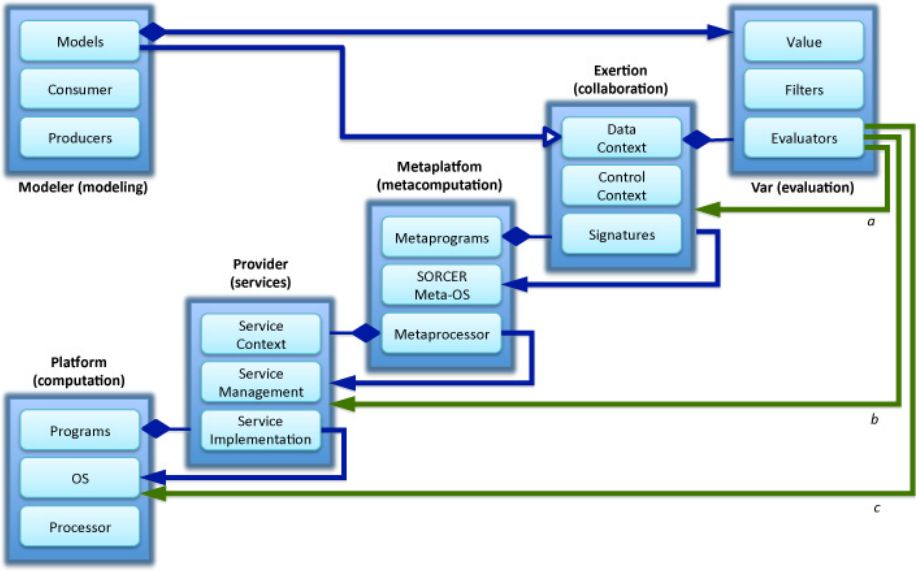

Fig. 4. SORCER computing abstractions: model, evaluation, collaboration, and computation. Arrows indicate associations, diamonds indicate compositions, the hollow arrow generalization, and arrows in green color indicate various ways of var evaluation.

The same evaluator with different filters can be associated with many vars. The modular VFE triplet structure of vars and reuse of evaluators and filters, including exertion evaluators with context filters, in defining VO-oriented models is the key feature of VO programming that complements SO programming with local computations.

VO models support *multidisciplinary* (vars from other models), and *multifidelity* (multiple evaluators per var) computing and are called *amorphous* models. For the same VO model an alternative set of evaluators (another fidelity) can be selected at runtime to evaluate a new particular version ("shape") of the model and quickly update the related process in the right evolving direction.

3.3 SORCER Operating System

The SORCER OS (SOS) allows executing service-oriented program and by itself is the service-oriented system. The overlay network of the services defining the functionality of SOS is called the *sos-federation* and the overlay network of application-specific services is called the *app-federation* (see Fig. 2). The *metainstruction set* of the SORCER metaprocessor consists of all operations offered by all services in the service federation—the union of the sos-federation and the app-federation. Thus, an EO program is composed of metainstructions with its own control strategy per service composition and data context representing the shared metaprogram data. Service signatures (instances of `Signature` type) correspond to `op` operators that specify operations of collaboration participants in the app-federation. Each signature primarily is defined by a service type, operation in that interface, and a set of optional 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 data context [20] describes the data that tasks and jobs work on. An APPEND signature defines the context received from the provider specified by this signature. The received context is then appended in runtime to the existing data context. The resulting context is then processed by PREPROCESS, PROCESS, and POSTPROCESS operations of the exertion. Appending a data context allows a requestor to use network shared data in runtime not available to the requestor when the exertion is declared. SOS allows for an exertion to create and manage a service collaboration and transparently coordinate the execution of all component exertions within the assembled federation. Please note that these meta-computing 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—the executable becomes the single service itself.

An exertion can be activated, it means its *collaboration exerted*, by invoking the `exert` operation on the exertion object:

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

where a parameter of the `Transaction` type is required when a transactional semantics is needed for all participating nested exertions within the parent one. Thus, EO programming allows us to execute an exertion and invoke exertion's signatures on collaborating service objects 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 all SORCER providers are required to provide:

```
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—`Exertion#exert()`—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 or 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 objects. The exertion caller has no direct dependency to service objects since the exertion uses only service types (interfaces) they implement.

Despite the fact that any Servicer can accept any exertion, SOS services have well defined roles in the S2S platform (see Fig. 2):

a) Taskers – accept service tasks; they are used to create application services by dependency injection (service assembly) or by inheritance (subclassing `ServiceTasker` and implementing required service interfaces);
b) Jobbers –manage service collaboration for `PUSH` signatures;
c) Spacers – manage service collaboration for `PULL` signatures using space-based computing [7];
d) Contexters – provide data contexts for `APPEND` signatures;
e) FileStorers – provide access to federated file system providers [2, 27];
f) Catalogers – Servicer registries, provide management for QoS-based federations;
g) SlaMonitors - provide monitoring of SLAs [18];
h) Provisioners - provide on-demand provisioning of services by SERVME [17, 18];
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;
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.

Domain specific servicers within the federation, or *task peers* (*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 service management providers; see Fig. 2. In the view of the P2P architecture [21] defined by the `Servicer` interface, a job can be sent to any servicer. A peer that is not a `Jobber` type is responsible for forwarding the job to one of 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.

3.4 Federated Method Invocation

An exertion is executed by invoking its exert operation. The SORCER Federated Method Invocation (FMI) defines the following three related operations:

1. `Exertion#exert(Transaction):Exertion`
   join the sos-federation; the invoked exertion is bound to the available provider specified by the exertion's `PROCESS` signature (a rendezvous provider if a job, otherwise a matching tasker);
2. `Servicer#service(Exertion, Transaction):Exertion`
   SOS request for a service by the bound provider in 1); and if the argument exertion object accepted by the bound provider, then the provider calls 3)
3. `Exerter#exert(Exertion, Transaction):Exertion`
   execute the argument exertion object by the service object of the provider accepting the service request in 2). Any component exertion of the parent exertion is then processed recursively by 1).

This above *triple command design pattern* [22, 9] defines various implementations of these three interfaces: `Exertion` (metaprogram), `Servicer` (service provider—peer), and `Exerter` (service object processing the data context of exertion). This approach allows for the P2P environment [21] via the `Servicer` interface, extensive modularization of `Exertions` and `Exerters`, and extensibility from the triple command design pattern so requestors can submit onto the network any EO program they want with or without transactional semantics. The triple command pattern is used by SOS as follows:

1. An exertion is activated by calling `Exertion#exert()`. The `exert` operation implemented in `ServiceExertion` uses `ServiceAccessor` to locate in runtime the provider matching the exertion's `PROCESS` signature.
2. If the matching provider is found, then on its access proxy the `Service#service()` method is invoked.
3. When the requestor is authenticated and authorized by the provider to invoke the method defined by the exertion's `PROCESS` signature, then the provider calls its own `exert` operation: `Exerter#exert()`.
4. `Exerter#exert()` operation is implemented accordingly by `ServiceTasker`, `ServiceJobber`, and `ServiceSpacer`. A `ServiceTasker` peer calls by reflection the operation specified in the `PROCESS` signature of the task object. All application-specific methods of an application interface have the same signature: a single `Context` type parameter and a `Context` type return value.

The exertion activated by a requestor can be submitted by SOS directly or indirectly to the corresponding service provider. In the direct approach, when signature's access type is `PUSH`, SOS finds the matching service provider against the service type and attributes of the `PROCESS` signature and submits the exertion object to the matching provider. The execution order of multiple signatures is defined by signature priorities, if the exertion's flow type is `SEQUENTIAL`; otherwise they are dispatched in parallel. EO

programming has a branch exertion (`IfExertion`) and loop exertions (`WhileExertion, ForExertion`) as well as two mechanisms for nonlinear flow control (`BreakExertion, ContinueExertion`). An exertion can reflect a process with branching and looping by applying control flow exertions [20].

Alternatively, when signature's access type is `PULL`, SOS uses a `Spacer` provider and simply drops the exertion into the shared exertion space to be pulled from by a matching provider. Spacers provide efficient load balancing for processing exertions from the shared space and are efficient for lengthy processes that might require services not present at all times during the process execution. The fastest available servicer gets an exertion from the space before other overloaded or slower servicers can do so. When an exertion consists of component jobs with different access and flow types, then we have the *hybrid* process execution when the collaboration potentially executes concurrently with multiple *pull* and *push* subcollaborations at the same time.

3.5 How to Create an Application Service?

To complete the example given in the Introduction, let's implement one of the arithmetic services, for example `Adder` that can be used by SOS.

A plain old Java object (POJO) becomes a SORCER service bean, injected into a `Tasker`, by implementing a Java interface (does not have to be `Remote`), which has the following characteristics:

1. Defines the service operations you'd like to call remotely
2. The single parameter and returned value of each operation is of the type `sorcer.service.Context`
3. Each method must declare `java.rmi.RemoteException` in its throws clause. The method can also declare application-specific exceptions
4. The class implementing the interface and local objects must be serializable

The interface for the `Adder` bean can be defined as follows:

```
interface Adder {
  Context add(Context context) throwsRemoteException;
}
```

The interface implementation:

```
public class AdderImpl implements Adder {
  public Context add(Context context) throws
    RemoteException {
      double result = 0;
      List<Double> inputs = context.getInValues();
        for (Object value : inputs)
          result += value;
    context.putValue(context.getOutPath(), result);
    return context;
  }
}
```

Finally start the `Tasker` with the following configuration file:

```
sorcer.core.provider.ServiceProvider {
  name = "SORCER Adder";
  beans = new String[]{"sorcer.arithmetic.AdderImpl"};
}
```

The same way you can implement and deploy `Multiplier` and `Subtractor` and you are ready to run the SO program given in the Introduction.

4 Conclusions

A distributed system is not just a collection of distributed objects—it is the unreliable network of objects that come and go. EO programming introduces the new abstractions of *service objects* and *exertions* for unreliable networks instead of *objects* and *messages* in object-oriented programming. Exertions encapsulate the triplet of *operations*, *data*, and *control strategy*. From the SORCER platform point of view, an *exertion* is the expression of service composition at the programming level, the *management federation* of service objects at the operating system level, and the *application federation* of service objects at the application service processor level. The exertions are programs that define reliable network collaborations in unreliable service networks. The SORCER operating system manages service collaborations on its virtual processor—the dynamically created federations that use FMI.

SORCER identifies a service with its service type. Applying the inversion principle, SOS looks up service objects by implemented interface types with optional search attributes, for example a provider name. 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-objects that do not require heavyweight containers like application servers.

The presented FMI framework allows P2P computing via the `Servicer` interface, extensive modularization of `Exertions` and `Exerters`, and extensibility from the triple command design pattern [20]. The SORCER platform uses a dynamic service discovery mechanism allowing new services to enter the network and disabled services to leave the network gracefully with no need for reconfiguration. This allows the exertion collaboration 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 SORCER platform with EO programming has been successfully deployed and tested in multiple concurrent engineering and large-scale distributed applications [8, 12, 29]. It is believed that incremental improvements of SPOA will not suffice, so we plan to continue the development of *Service-Object Oriented Optimization Toolkit for Distributed High Fidelity Engineering Design Optimization* at the Multidisciplinary Science and Technology Center, AFRL with three layers of programming: *model-driven programming* (transdisciplinary complex processes), *var-oriented programming* (for var multifidelity evaluations and var compositions), and *exertion-oriented*

programming (for network collaborations). We will investigate how *model-driven programming* can be used to address several fundamental challenges posed by the new value-filter-evaluator paradigm for real world complex optimization problems

I began my Introduction with Edsger Dijkstra's credo:

"Computing's core challenge is how not to make a mess of it."

The presented confrontation of computing and metacomputing, and the SORCER platform described in this paper implies that reducing both programming and metaprogramming to the same level of middleware programming and within the same computing platform (currently common practice for building SOA), introduces intolerable complexity for building large-scale adaptive and dynamic enterprise systems. SORCER defines clearly its separate metacomputing architectural layers: SO programming, management, and execution layers integrated via FMI. That introduces simplicity to the expression of SO processes at the application level using VO model-driven with var-oriented and exertion-oriented programming. Flexible enterprise interoperability is achieved via SORCER three neutralities (service protocol, implementation, and location [22]) and architectural means (Fig. 4), not by neutral data exchange formats, e.g., XML, when overused introduce unintended complexity and degraded performance.

Acknowledgments

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

1. Apache River, `http://incubator.apache.org/river/RIVER/index.html` (accessed on: August 10, 2010)
2. Berger, M., Sobolewski, M.: Lessons Learned from the SILENUS Federated File System. In: Loureiro, G., Curran, R. (eds.) Complex Systems Concurrent Engineering, pp. 431–440. Springer, Heidelberg (2007a)
3. Edwards, W.K.: Core Jini, 2nd edn. Prentice Hall, Englewood Cliffs (2000)
4. Fallacies of Distributed Computing (accessed on: August 10, 2010) `http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing`
5. Fant, K.M.: A Critical Review of the Notion of Algorithm in Computer Science. In: Proceedings of the 21st Annual Computer Science Conference, pp. 1–6 (February 1993)
6. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International J. Supercomputer Applications 15(3) (2001)
7. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces^{TM} Principles, Patterns, and Practice. Addison-Wesley, Reading ISBN: 0-201-30955-6

8. Goel, S., Talya, S.S., Sobolewski, M.: Mapping Engineering Design Processes onto a Service-Grid: Turbine Design Optimization. International Journal of Concurrent Engineering: Research & Applications, Concurrent Engineering 16, 139–147 (2008)
9. Grand, M.: Patterns in Java, vol. 1. Wiley, Chichester (1999) ISBN: 0-471-25841-5
10. Jini Architecture Specification (accessed on: August 10, 2010), http://www.jini.org/wiki/Jini_Architecture_Specification
11. Juric, M., Benny Mathew, B., Sarang, P.: Business Process Execution Language for Web Services BPEL and BPEL4WS, 2nd edn. Packt Publishing (2006) ISBN: 978-1904811817
12. Kolonay, R.M., Thompson, E.D., Camberos, J.A., Eastep, F.: Active Control of Transpiration Boundary Conditions for Drag Minimization with an Euler CFD Solver. In: 48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference on AIAA 2007-1891, Honolulu, Hawaii (2007)
13. Linthicum, D.S.: Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide. Addison-Wesley Professional, Reading (2009) ISBN-10 0136009220
14. Lynch, D., Rose, M.T. (eds.): Internet System handbook. Addison-Wesley, Reading (1992)
15. Markov, A.A.: Theory of Algorithms, trans. by Schorr-Kon, J.J. Keter Press (1971)
16. Metacomputing: Past to Present (August 10, 2010), http://archive.ncsa.uiuc.edu/Cyberia/MetaComp/MetaHistory.html
17. Rio Project, http://www.rio-project.org/ (accessed on: August 10, 2010)
18. Rubach, P., Sobolewski, M.: Autonomic SLA Management in Federated Computing Environments. In: International Conference on Parallel Processing Workshops, Vienna, Austria, pp. 314–321 (2009)
19. Sobolewski, M.: Multi-Agent Knowledge-Based Environment for Concurrent Engineering Applications. Concurrent Engineering: Research and Applications (CERA), Technomic (1996), http://cer.sagepub.com/cgi/content/abstract/4/1/89
20. Sobolewski, M.: Exertion Oriented Programming. IADIS 3(1), 86–109 (2008) ISBN: ISSN: 1646-3692
21. Sobolewski, M.: Federated Collaborations with Exertions. In: 17th IEEE International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), pp. 127–132 (2008)
22. Sobolewski, M.: Metacomputing with Federated Method Invocation. In: Akbar Hussain, M. (ed.) Advances in Computer Science and IT, pp. s337–s363 (2009) In-Tech, intechweb.org, ISBN 978-953-7619-51-0, http://sciyo.com/articles/show/title/metacomputing-with-federated-method-invocation (accessed on: August 10, 2010)
23. Sobolewski, M.: Object-Oriented Metacomputing with Exertions. In: Gunasekaran, A., Sandhu, M. (eds.) Handbook On Business Information Systems. World Scientific, Singapore (2010) ISBN: 978-981-283-605-2
24. SORCERsoft, http://sorcersoft.org (accessed on: August 10, 2010)
25. Sotomayor, B., Childers, L.: Globus® Toolkit 4: Programming Java Services. Morgan Kaufmann, San Francisco (2005)
26. Thain, D., Tannenbaum, T., Livny, M.: Condor and the Grid. In: Berman, F., Hey, A.J.G., Fox, G. (eds.) Grid Computing: Making The Global Infrastructure a Reality. John Wiley, Chichester (2003)
27. Turner, A., Sobolewski, M.: FICUS—A Federated Service-Oriented File Transfer Framework. In: Loureiro, G., Curran, L.,, R. (eds.) Complex Systems Concurrent Engineering, pp. 421–430. Springer, Heidelberg (2007) ISBN: 978-1-84628-975-0
28. Waldo, J.: The End of Protocols (accessed on: August 10, 2010), http://java.sun.com/developer/technicalArticles/jini/protocols.html
29. Xu, W., Cha, J., Sobolewski, M.: A Service-Oriented Collaborative Design Platform for Concurrent Engineering. Advanced Materials Research 44-46, 717–724 (2008)