Handbook on Business Information Systems

CHAPTER 36

OBJECT-ORIENTED METACOMPTING WITH EXERTIONS

Michael Sobolewski

Department of Computer Science, Texas Tech University
P.O. Box 43104, Lubbock, TX 79409
E-mail: sobol@cs.ttu.edu

This chapter investigates service-oriented computing in the context of object-oriented distributed platforms. A platform consists of virtual compute resources, a programming environment allowing for the development of distributed applications, and an operating system to run user programs and to make solving complex user problems easier. Service protocol-oriented architectures are contrasted with service object-oriented architectures, and then a metacompute grid based on a service object-oriented architecture is described and analyzed. A new object-oriented network programming methodology is presented in this chapter. It uses the intuitive metacomputing semantics and the new Triple Command design pattern. The pattern defines how service objects communicate by sending one another a form of service messages called exertions that encapsulate the triplet: data, operations, and control strategy.

1. Introduction

The term "grid computing" originated in the early 1990s as a metaphor for accessing computer power as easy as an electric power grid. Today there are many definitions of grid computing with a varying focus on architectures, resource management and access, virtualization, provisioning, and sharing between heterogeneous compute domains. Thus, diverse compute resources across different administrative domains form a *grid* for the shared and coordinated use of resources in dynamic, distributed, and virtual computing organizations [9]. Therefore, the grid requires a *platform* that describes some sort of framework to allow software to run utilizing virtual organizations. These organizations are dynamic subsets of departmental grids, enterprise grids, and global grids, which allow programs to use shared resources—collaborative federations.

Different platforms of grids can be distinguished along with corresponding types of virtual federations. However, in order to make any grid-based computing possible, computational modules have to be defined in terms of platform data, operations, and relevant control strategies. For a grid program, the control strategy is a plan for achieving the desired results by applying the platform operations to the data in the required sequence and by leveraging the dynamically federating resources. We can distinguish three generic grid platforms, which are described below.

Programmers use abstractions all the time. The source code written in programming language is an abstraction of the machine language. From machine language to object-oriented programming, layers of abstractions have accumulated like geological strata. Every generation of programmers uses its era's programming languages and tools to build programs of next generation. Each programming language reflects a relevant abstraction, and usually the type and quality of the abstraction implies the complexity of problems we are able to solve.

Procedural languages provide an abstraction of an underlying machine language. An executable file represents a computing component whose content is meant to be interpreted as a program by the underlying native processor. A request can be submitted to a *grid resource broker* to execute a machine code in a particular way, e.g., by parallelizing and collocating it dynamically to the right processors in the grid. That can be done, for example, with the Nimrod-G grid resource broker scheduler [24] or the Condor-G high-throughput scheduler [43]. Both rely on Globus/GRAM (Grid Resource Allocation and Management) protocol [9]. In this type of grid, called a *compute grid*, executable files are moved around the grid to form virtual federations of required processors. This approach is reminiscent of batch processing in the era when operating systems were not yet developed. A series of programs ("jobs") is executed on a computer without human interaction or the possibility to view any results before the execution is complete.

A grid programming language is the abstraction of hierarchically organized networked processors running a grid computing program—*metaprogram*—that makes decisions about component programs such as when and how to run them. Nowadays the same computing abstraction is usually applied to the program executing on a single computer as to the metaprogram executing in the grid of computers, even though the executing environments are structurally completely different. Most grid programs are still written using compiled languages such as FORTRAN, C, C++, Java, and interpreted languages such as Perl and Python the way it usually works on a single host. The current trend is still to have these

programs and scripts define grid computational modules. Thus, most grid computing modules are developed using the same abstractions and, in principle, run the same way on the grid as on a single processor. There is presently no grid programming methodologies to deploy a metaprogram that will dynamically federate all needed resources in the grid according to a control strategy using a kind of *grid algorithmic logic*. Applying the same programming abstractions to the grid as to a single computer does not foster transitioning from the current phase of early grid adopters to public recognition and then to mass adoption phases.

The reality at present is that grid resources are still very difficult for most users to access, and that detailed 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 adoption of grid techniques, and a multiplicity of specialized "grid-aware" tools that are not, in fact, aware of each other that defeat the basic purpose of the compute grid.

Instead of moving executable files around the grid, we can autonomically provision the corresponding computational components as uniform services on the grid. All grid services can be interpreted as instructions (metainstructions) of the *metacompute grid*. Now we can submit a metaprogram in terms of metainstructions to the *grid platform* that manages a dynamic federation of service providers and related resources, and enables the metaprogram to interact with the service providers according to the metaprogram control strategy.

We can distinguish three types of grids depending on the nature of computational components: *compute grids* (*cGrids*), *metacompute grids* (*mcGrids*), and the hybrid of the previous two—*intergrids* (*iGrids*). Note that a cGrid is a virtual federation of processors (roughly CPUs) that execute submitted executable codes with the help of a grid resource broker. However, an mcGrid is a federation of service providers managed by the mcGrid operating system. Thus, the latter approach requires a metaprogramming methodology while in the former case the conventional procedural programming languages are used. The hybrid of both cGrid and mcGrid abstractions allows for an iGrid to execute both programs and metaprograms as depicted in Fig. 1, where platform layers P1, P2, and P3 correspond to resources, resource management, and programming environment correspondingly.

One of the first mcGrids was developed under the sponsorship of the National Institute for Standards and Technology (NIST)—the Federated Intelligent Product Environment (FIPER) [8, 28, 37]. The goal of FIPER is to form a federation of distributed services that provide engineering data, applications, and

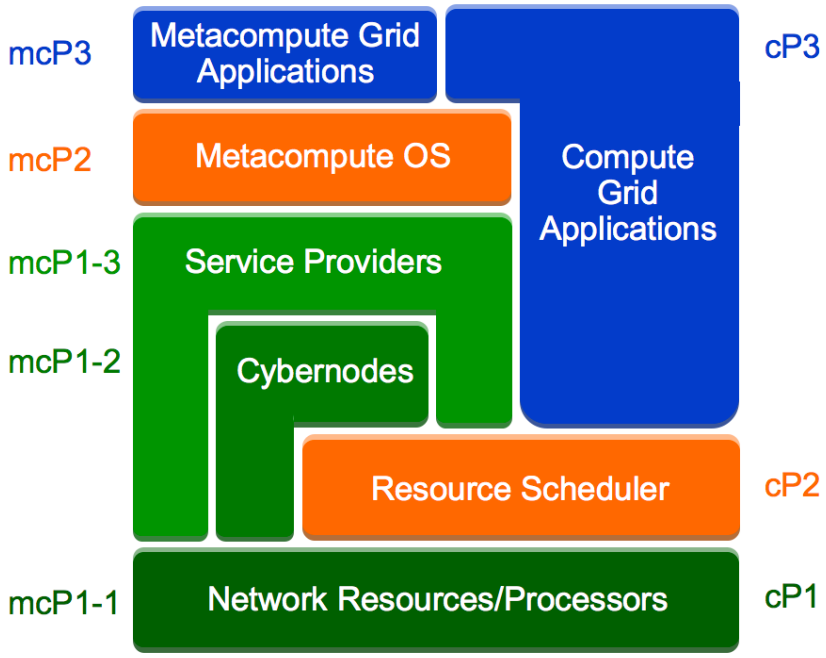

Fig. 1. Three types of grids: compute grid, metacompute grid, and intergrid. A cybernode provides a lightweight dynamic virtual processor, turning heterogeneous compute resources into homogeneous services available to the metacomputing OS [27].

tools on a network. A highly flexible software architecture had been developed (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 federating service providers and service requestors.

The Service-ORiented Computing EnviRonment (SORCER) [40, 41, 31, 32, 33, 34] builds on the top of FIPER to introduce a metacomputing operating system with all basic services necessary, including a federated file system, to support service-oriented metaprogramming. It provides an integrated solution for complex metacomputing applications. The SORCER metacomputing environment adds an entirely new layer of abstraction to the practice of grid computing—exertion-oriented (EO) programming. The EO programming makes a positive difference in service-oriented programming primarily through a new metaprogramming abstraction as experienced in many service-oriented computing projects including systems deployed at GE Global Research Center, GE Aviation, Air Force Research Lab, and SORCER Lab [5, 20, 30, 18, 22, 19, 35, 2, 44, 13, 12, 21, 11].

The paper is organized as follows. Section 2 provides a brief description of two service-oriented architectures used in grid computing with a related discussion of distribution transparency; Section 3 describes the SORCER metacomputing philosophy and its mcGrid; Section 4 describes the exertion-oriented programming, and Section 5 the federated method invocation; Section 7 provides concluding remarks.

2. SPOA versus SOOA

Various definitions of a Service-Oriented Architecture (SOA) leave a lot of room for interpretation. Nowadays SOA becomes the leading architectural approach to most grid developments. In general terms, SOA is a software architecture consisting of loosely coupled software services integrated into a distributed computing system by means of *service-oriented programming*. Service providers in the SOA environment are made available as independent service components that can be accessed without a priori knowledge of their underlying platform or implementation. While the client-server architecture separates a client from a server, SOA introduces a third component, a service registry. In SOA, the client is referred to as a *service requestor* and the server as a *service provider*. The provider is responsible for deploying a service on the network, publishing its service to one or more registries, and allowing requestors to bind and execute the requested service. Providers advertise their availability on the network; registries intercept these announcements and add published services. The requestor looks up a service by sending queries to registries and making selections from the available services. Queries generally contain search criteria related to the service name/type and quality of service. Registries facilitate searching by storing the service representation (description or proxies) and making it available to requestors. Providers and requestors can use discovery and join protocols to locate registries dynamically and then publish or acquire services on the network respectively. Thus service-oriented programming is focused on development and execution of distributed programs in terms of services that are available via network registries.

We can distinguish the *service object-oriented architecture* (SOOA), where providers, requestors, and proxies are network objects, from the *service protocol oriented architecture* (SPOA), where a communication protocol is fixed and known beforehand to the both provider and requestor. Using SPOA, a requestor can use this fixed protocol and a service description obtained from a service registry to create a proxy for binding to the service provider and for remote communication over the fixed protocol. In SPOA a service is usually identified

by a name. If a service provider registers its service description by name, the requestors have to know the name of the service beforehand.

In SOOA (see Fig. 2), a proxy—an object implementing the same service interfaces as its service provider—is registered with the registries and it is always ready for use by requestors. Thus, the service provider publishes the proxy as the active surrogate object with a codebase annotation, for example URLs in Jini ERI [25] to the code defining proxies behavior.

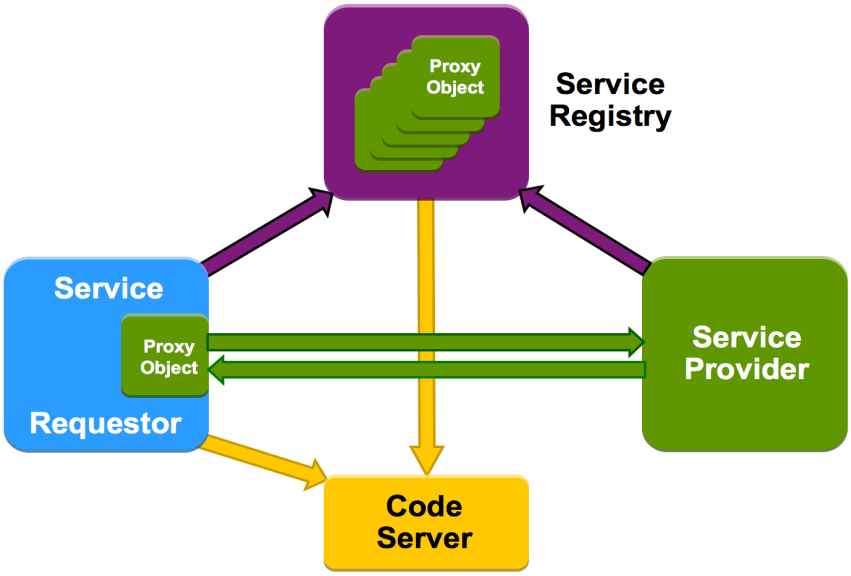

Fig. 2. Service object-oriented architecture.

In SPOA, by contrast, a passive service description is registered, for instance an XML document in WSDL for Web/OGSA services [23, 42] or an interface description in IDL for CORBA [29]. Then, the requestor has to generate the proxy (a stub forwarding calls to a provider) based on the service description and the fixed communication protocol, for example SOAP in Web/OGSA services, IIOP in CORBA. This is referred to as a bind operation. The binding operation is not needed in SOOA since the requestor holds the active surrogate object already created by the provider and obtained by the requestor from the registry.

Web services and OGSA services cannot change the communication protocol between requestors and providers while the SOOA approach is protocol neutral [46]. In SOOA, the way an object proxy communicates with a provider is established by the contract between the provider and its published proxy and defined accordingly by the provider implementation. The proxy's requestor does

not need to know who implements the interface, how it is implemented, or where the provider is located—three neutralities of SOOA. So-called smart proxies, for example provided by Jini ERI, can grant access to both local and remote resources. They can also communicate with multiple providers on the network regardless of who originally registered the proxy, thus separate providers on the network can implement different parts of the smart proxy interface(s). Communication protocols may also vary, and a single smart proxy can also talk over multiple protocols including application-specific protocols.

SPOA and SOOA differ in their method of discovering the service registry. For example, SORCER uses dynamic discovery protocols to locate available registries (lookup services) as defined in the Jini architecture [17]. Neither the requestor who is looking up a proxy by its interfaces nor the provider registering a proxy needs to know specific registry locations. In SPOA, however, the requestor and provider usually do need to know the explicit location of the service registry—e.g., a URL for RMI registry [26], a URL for UDDI registry [23], an IP address and port of a COS Name Server [29]—to open a static connection and find or register a service. In deployment of Web and OGSA services, a UDDI registry can be omitted just by using WSDL files available directly from service developers. In SOOA, lookup services are mandatory due to the dynamic nature of object proxies registered by bootstrapping providers and identified by service types. Interactions with registries in SPOA are more like static client-server connections while in SOOA they are dynamic (Jini discovery/join protocols) as proxy registrations are leased to registering providers.

Crucial to the success of SOOA is interface standardization. Services are identified by interface types (e.g., Java interfaces) and additional provider's specific properties if needed; the exact identity of the service provider is not crucial to the architecture. As long as services adhere to a given set of rules (common interfaces), they can collaborate to execute published operations, provided the requestor is authorized to do so.

Let us emphasize the major distinction between SOOA and SPOA: in SOOA, a proxy 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 a generic one (e.g., HTTP, SOAP, IIOP), reduced to a common denominator—one size fits all—that leads to inefficient network communication in many cases. In SOOA, each provider can decide on the most efficient protocol(s) needed for a particular service provider.

Service providers in SOOA can be considered as independent network objects finding each other via service registries. These objects are identified by service types and communicating through message passing. A collection of these objects sending and receiving messages—the only way these objects communicate with one another—looks very much like a service object-oriented distributed system. However, do you remember the eight fallacies [7] of network computing? We cannot just take an object-oriented program developed without distribution in mind and make it a distributed system, ignoring the unpredictable network behavior. Most RPC systems, with notable exception of Jini [6] and SORCER, hide the network behavior and try to transform local communication into remote communication by creating distribution transparency based on a local assumption of what the network might be. However, every single distributed object cannot do that in a uniform way as the network is a *dynamic, heterogeneous*, and *unreliable*. Thus a *distributed system* cannot be represented completely as a collection of independent objects, each of them incorporating a transparent and local view of the network.

The network is dynamic, cannot preserve constant topology, and introduces latency for remote invocations. Network latency also depends on potential failure handling and recovery mechanisms, so we cannot assume that a local invocation is similar to remote invocation. Thus, complete distribution transparency—by making calls on distributed objects as though they were local—is impossible to achieve in practice. The network distribution is simply not just an object-oriented implementation of a isolated distributed objects; it is a metasystemic issue in *object-oriented distributed programming*. In that context, Web/OGSA services define independent distributed "objects", but do not have anything common with dynamic object-oriented distributed systems that for example the Jini architecture emphasizes.

Object-oriented programming can be seen as an attempt to abstract both *data* and related *operations* in an entity called *object*. Thus, an object-oriented program may be seen as a collection of cooperating *objects* communicating via *message* passing, as opposed to a traditional view in which a program may be seen as a list of instructions to the computer. Instead of *objects* and *messages*, in exertion-oriented programming (EO) [33] *service providers* and *exertions* constitute a program. An *exertion* is a kind of meta-request sent onto the network. The exertion can be considered as the *specification of distributed collaboration* that encapsulates *data*, related *operations,* and *control strategy*. The operation signatures specify implicitly the required service providers on the network. The invoked (activated) exertion creates a federation of service providers at runtime to execute a collaboration according to the exertion's control

strategy. Thus, the exertion is the *metaprogram* and its *metashell* that submits the request onto the network to run the collaboration in which all federated providers pass to one another the component exertions only. This type of metashell that coordinates in runtime the execution of an exertion by federated providers was created for the SORCER metacompute operating system (see Fig. 3)—the exemplification of SOOA with autonomic management of system and domain-specific service providers to run EO programs.

The SORCER environment, described in the next Section, defines the object-oriented distribution for EO programming. It uses indirect federated remote method invocation [34] with no explicit location of service providers specified in exertions. A specialized infrastructure of distributed services provides support for management of exertions and services, the exertion shell, federated file system, service discovery/join, and the system services for coordination of executing runtime federations. That infrastructure defines SORCER's *object-oriented distributed* modularity, extensibility, and reuse of providers and exertions—key features of object-oriented distributed programming that are usually missing in SPOA programming environments.

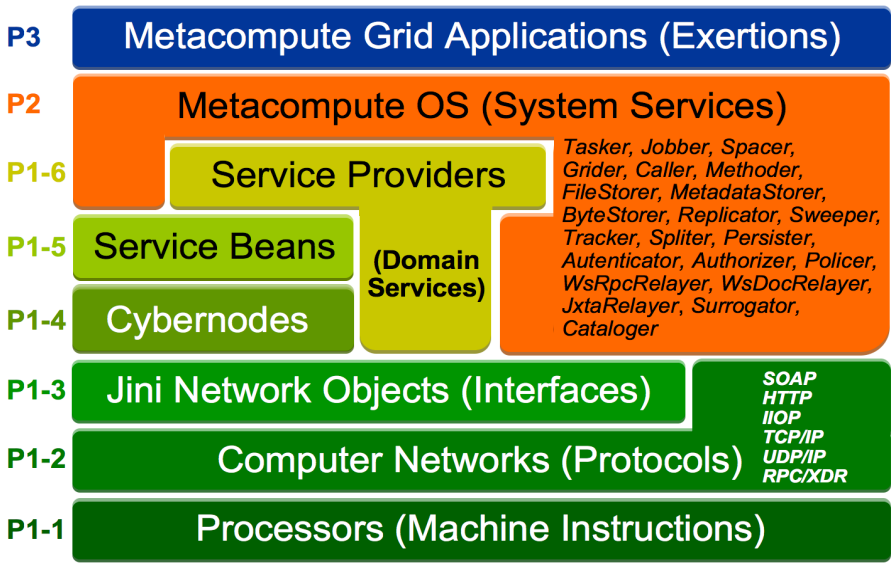

Fig 3. SORCER layered platform, where P1 resources, P2 resource management, P3 programming environment.

3. Metacompute Grid

SORCER is a federated service-to-service (S2S) metacomputing environment that treats service providers as network peers with well-defined semantics of a federated service object-oriented architecture (FSOOA). It is based on Jini semantics of services [17] in the network and the Jini programming model [6] with explicit leases, distributed events, transactions, and discovery/join protocols. While Jini focuses on service management in a networked environment, SORCER hides all low-level programming details of the Jini programming model and is focused on EO programming and the execution environment for exertions based on the Triple Command pattern [34] presented in Section 5.

As described in Section 2, SOOA consists of four major types of network objects: providers, requestors, registries, and proxies. 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. The requestor looks up proxies by sending queries to registries and making selections from the available services. Queries generally contain search criteria related to the type and quality of service. Registries facilitate searching by storing proxy objects with related attributes 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. The SORCER metacompute OS uses Jini discovery/join protocols to implement its FSOOA.

In FSOOA, a service provider is a remote object that receives exertions from service requestors to execute collaborations. An exertion encapsulates collaboration data, operations, and control strategy. A *task exertion* is an elementary service request, a kind of elementary remote instruction (elementary statement) executed by a single service provider or a small-scale federation. A composite exertion called a *job exertion* is defined hierarchically in terms of tasks and other jobs, including *control exertions* that manage the flow of control in a collaboration. A job exertion is a kind of network procedure executed by a large-scale federation. Thus, the executing exertion is a service-oriented program that is dynamically bound to all required and currently available service providers on the network. This collection of providers identified at runtime is called an *exertion federation*. While this sounds similar to the object-oriented paradigm, it really is not. In the object-oriented paradigm, the *object space* is a program itself; here the exertion federation is *not* the program, it is the *execution environment* for the exertion, and the exertion is the *object-oriented* program—the *specification* of service collaboration. This changes the programming paradigm completely. In the former case the object space is hosted by a single

computer, but in the latter case the top-level and its component exertions along with related service providers are hosted by the network of computers.

The overlay network of all service providers is called the *service grid* and an exertion federation is called a *virtual metacomputer*. The *metainstruction set* of the metacomputer consists of all operations offered by all providers in the service grid. Thus, a service-oriented program is composed of metainstructions with its own service-oriented control strategy and service context representing the metaprogram data. *Service signatures* specify metainstructions in SORCER. Each signature primarily is defined by a service type (interface name), 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 service context [47, 31] 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 service context later processed by `PREPROCESS`, `PROCESS`, and `POSTPROCESS` operations of the exertion. Appending a service context allows a requestor to use actual shared network data with other requestors at runtime. A job exertion allows for a dynamic federation to transparently coordinate the execution of all component exertions within the service grid.

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 that code managed by the local operation system on a selected processor by the grid scheduler.

An exertion can be activated by calling exertion's `exert` operation: `Exertion.exert(Transaction):Exertion`, where a parameter of the `Transaction` type is required when a transactional semantics is needed for all nested exertions participating within the parent exertion's collaboration. Thus, EO programming allows us to submit an exertion onto the network implicitly (no receiving provider identified a priori) and to perform execution of exertion's signatures on various service providers in runtime by the exertion metashell. Top-level S2S communication between collaborating services is managed by rendezvous services through the use of the generic `Servicer` interface and the operation `service` that all SORCER services are required to provide:

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

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

Despite the fact that every `Servicer` can accept any exertion, `Servicer`s have well defined roles in the S2S platform (see Fig. 3):

1. `Tasker`s—process service tasks;
2. `Jobber`s—rendezvous providers that process service jobs;
3. `Spacer`s—rendezvous providers that process tasks and jobs via shared exertion space for space-based computing [10];
4. `Contexter`s—provide service contexts for `APPEND` signatures;
5. `FileStorer`s—provide access to federated file system providers [36, 1, 2, 44];
6. `Cataloger`s—SORCER services registries;
7. `Persister`s—persist service contexts, tasks, and jobs to be reused for interactive EO programming;
8. `Relayer`s—gateway providers; transform exertions to native representation, for example integration with Web services and JXTA [16];
9. `Autenticator`s, `Authorizer`s, `Policer`s, `KeyStorer`s—provide support for service security;
10. `Auditor`s, `Reporter`s, `Logger`s—support for accountability, reporting, and logging;
11. `Grider`s, `Caller`s, `Methoder`s—support traditional compute grid;
12. `ServiceTasker`, `ServiceJobber`, and `ServiceSpacer` are basic three implementations of providers used to configure domain-specific providers via dependency injection—configuration files for smart proxying and embedding business objects, called service beans, into service providers. Also, domain-specific providers can subclass any of these three providers and implement required domain-specific interfaces with operations returning a service context and taking a service context as its single parameter. These domain-specific interfaces and operations are usually used in service task signatures; and
13. `ServiceProviderBean`s—to enable autonomic provisioning of service providers with the Rio framework [27].

Service 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 top-level exertion. Domain specific providers within the federation, or *task peers*, execute service tasks. Job collaborations are coordinated by *rendezvous peers*: a `Jobber` or `Spacer`, two of the SORCER platform system services. However, a job can be sent to any peer. A peer that is not a rendezvous peer is responsible for forwarding the job to an available rendezvous peer 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 disperse to seek other exertions to join.

Exertions can be created interactively [35] or programmatically (using SORCER API), and its execution can be monitored and debugged [39] in the overlay service network via service user interfaces [45] attached to providers and installed on-the-fly by generic service browsers [15].

3.1 *Federated File System*

The SILENUS federated file system [1, 2] was designed and developed to provide data access for metaprograms. It expands the file store developed for FIPER [36] with the true P2P services. The SILENUS system itself is a collection of service providers that use the SORCER framework for communication.

In classical client-server file systems, a heavy load may occur on a single file server. If multiple service requestors try to access large files at the same time, the server will be overloaded. In a P2P architecture, every provider is a client and a server at the same time. The load can be balanced between all peers if files are spread across all of them. The SORCER architecture splits up the functionality of the metacomputer into smaller service peers (`ServicerS`), and this approach was applied to the distributed file system as well.

The SILENUS federated file system is comprised of several network services that run within the SORCER environment. These services include a byte store service for holding file data, a metadata service for holding metadata information about the files, several optional optimizer services, and façade [14] services to assist in accessing federating services. SILENUS is designed so that many instances of these services can run on a network, and the required services will federate together to perform the necessary functions of a file system. In fact, the SILENUS system is completely decentralized, eliminating all potential single point failures. SILENUS services can be broadly categorized into gateway components, data services, and management services.

The SILENUS façade service provides a gateway service to the SILENUS grid for requestors that want to use the file system. Since the metadata and actual file contents are stored by different services, there is a need to coordinate communication between these two services. The façade service itself is a combination of a control component, called the coordinator, and a smart proxy component that contains needed inner proxies provided dynamically by the coordinator. These inner proxies facilitate direct P2P communications for file

upload and download between the requestor and SILENUS federating services like metadata and byte stores.

Core SILENUS services have been deployed as SORCER services along with WebDAV and NFS adapters. The SILENUS file system scales well with a virtual disk space adjusted as needed by the corresponding number of required byte store providers and the appropriate number of metadata stores required to satisfy the needs of current users and service requestors. The system handles several types of network and computer outages by utilizing disconnected operation and data synchronization mechanisms [3]. It provides a number of user agents including a zero-install file browser attached to the SILENUS façade. Also a simpler version of SILENUS file browser is available for smart MIDP phones.

SILENUS supports storing very large files [44] by providing two services: a splitter service and a tracker service. When a file is uploaded to the file system, the splitter service determines how that file should be stored. If a file is sufficiently large enough, the file will be split into multiple replicated parts, or chunks, and stored across many byte store services. Once the upload is complete, a tracker service keeps a record of where each chunk was stored. When a user requests to download the full file later on, the tracker service can be queried to determine the location of each chunk and the file can be reassembled to the original form.

4. Exertion-oriented Programming

Each programming language provides a specific computing abstraction. Procedural languages are abstractions of assembly languages. Object-oriented languages abstract entities in the problem domain that refer to "objects" communicating via message passing as their representation in the corresponding solution domain, e.g., Java and SORCER objects. EO programming is a form of distributed programming that allows us to describe the distributed problem explicitly in terms of the intrinsic unpredictable network domain instead of in terms of conventional distributed objects that hide the notion of the network domain.

What intrinsic distributed abstractions are defined in SORCER? Well, *service providers* are "objects", but they are specific objects—they are *network objects* with *leased network resources*, a *network state*, *network behavior*, and *network typs*. There is still a connection to conventional distributed objects: each service provider looks like a remote object (data or compute node). However, service providers act as *network peers* with leased network resources; they implement the same top-level interface; they are replicated and dynamically provisioned for

reliability to compensate for network failures [27]; they can be found dynamically at runtime by types they implement; they can federate for executing a specific network request called an *exertion* and process collaboratively nested (component) exertions. An exertion encapsulates in a modular way service *data*, *operations*, and requestor's *control strategy*. The component exertions may need to share context data of ancestor exertions, and the top-level exertion is complete only if all nested exertions are successful.

With that very concise introduction to the abstraction of EO programming, let's look into a simple analogy to Unix shell scripts execution and then in detail at how FSOOA is defined.

Let's first look at the EO approach to see how it works. EO programs consist of *exertion* objects called tasks and jobs. An exertion *task* corresponds to an individual network request to be executed on a service provider. An exertion *job* consists of a structured collection of tasks and other jobs. The data upon which to execute a task or job is called a s*ervice context*. Tasks are analogous to executing a single program or command on a computer, and the service context would be the input and output streams that the program or command uses. A job is analogous to a batch script that can contain various commands and calls to other scripts. Pipelining Unix commands allows us to perform complex tasks without writing complex programs. As an example, consider a script `sort.sh` connecting simple processes in a pipeline as follows:

```
cat hello.txt | sort | uniq > bye.txt
```

The script is similar to an exertion job in that it consists of individual tasks that are organized in a particular fashion. Also, other scripts can call the script `sort.sh`. An exertion job can consist of tasks and other jobs, much like a script can contain calls to commands and other scripts.

Each of the individual commands, such as `cat`, `sort`, and `uniq`, would be analogous to a task. Each task works with a particular service context. The input context for the `cat` "task" would be the file `hello.txt`, and the "task" would return an output context consisting of the contents of `hello.txt`. This output context can then be used as the input context for another task, namely the `sort` command. Again the output context for `sort` could be used as the input context for the `uniq` task, which would in turn give an output service context in the form of `bye.txt`.

To further clarify what an exertion is, an exertion consists mainly of three parts: a set of *service signatures*, which is a description of operations in a collaboration, the associated *service context* upon which to execute the exertion, and control strategy (default provided) that defines how signatures are applied in the collaboration. A *service signature* specifies at least the provider's interface

that the service requestor would like to use and a selected operation to run within that interface. There are four types of signatures that can be used for an exertion: PREPROCESS, PROCESS, POSTPROCESS, and APPEND. An exertion must have one and only one PROCESS signature that specifies what the exertion should do and who works on it. An exertion can optionally have multiple PREPROCESS, POSTPROCESS, and APPEND signatures that are primarily used for formatting the data within the associated service context. A *service context* consists of several data nodes used for either input, output, or both. A task may work with only a single service context, while a job may work with multiple service contexts since it can contain multiple tasks. The requestor can define a control strategy as needed for the underlying exertion by choosing relevant control exertion types and configuring attributes of service signatures accordingly (see Section 4.2, 4.4 and 4.5 for details).

Here is the basic structure of the EO program that is analogous to the `sort.sh` script.

```
1.  // Create service signatures
2.  Signature catSignature, sortSignature, uniqSiganture;
3.  catSignature = new ServiceSignature("Reader", "cat");
4.  sortSignature = new ServiceSignature("Sorter", "sort");
5.  uniqSiganture = new ServiceSignature("Filter", "uniq");
6.
7.  // Create component exertions
8.  Task catTask, sortTask, uniqTask;
9.  catTask = new Task("cat", catSignature);
10. sortTask = new Task("sort", sortSignature);
11. uniqTask = new Task("uniq", uniqSiganture);
12.
13. // Create top-level exertion
14. Job sortJob = new Job("main-sort");
15. sortJob.addExertion(catTask);
16. sortJob.addExertion(sortTask);
17. sortJob.addExertion(uniqTask);
18.
19. // Create service contexts
```

```
20. Context catContext, sortContext, uniqContext;
21. catContext = new ServiceContext("cat");
22. sortContext = new ServiceContext("sort");
23. uniqContext = new ServiceContext("uniq");
24.
25. catContext.putInValue("/text/in/URL",
26.        "http://host/hello.txt");
27. catContext.putOutValue("/text/out/contents", null);
28.
29. sortContext.putInValue("/text/in/contents", null);
30. sortContext.putOutValue("/text/out/sorted", null);
31.
32. uniqContext.putInValue("/text/in/sorted", null);
33. uniqContext.putOutValue("/text/out/URL",
34.        "http://host/bye.txt");
35.
36. //Map context outputs to inputs
37. catContext.map("/text/out/contents",
38.        "/text/in/contents", sortContext);
39. sortContext.map("/text/out/sorted",
40.        "/text/in/sorted", uniqContext);
41.
42. catTask.setContext(catContext);
43. sortTask.setContext(sortContext);
44. uniqTask.setContext(uniqContext);
45.
46. // exert collaboration
47. sortJob.exert(null);
```

In the above EO program we create three signatures (lines 2-5), each signature is defined by the interface name and the operation name that we want to run by any remote object implementing the interface. We use the three signatures to create three tasks (lines 8-11) and by line 12, we have three separate commands `cat`, `sort`, and `uniq` to be used in the `sort.sh` script. The three tasks

are combined into the job by analogy to piping Unix commands in the sort.sh script. Thus, by line 18, we have added these commands to `sort.sh` script, but have not provided input/output parameters nor piped them together:

as is: `cat sort uniq`

to be: `cat hello.txt | sort | uniq > bye.txt`

Lines 20-34 create and define three service contexts for our three tasks. By line 35, we have specified some input and output parameters, but still no piping:

as is: `cat hello.txt sort uniq bye.txt`

to be: `cat hello.txt | sort | uniq > bye.txt`

Lines 37-40 define mapping of context output to the related context input parameters. The parameters are context paths from a source context to a target context. The target context is the last parameter in the map operation. By line 45, we have piping setup and by the analogy our sort.sh script is complete now:

as is: `cat hello.txt | sort | uniq > bye.txt`

On line 47, we execute the script. If we use the Tenex C shell (`tcsh`), invoking the script is equivalent to: `tcsh sort.sh`, i.e., passing the script `sort.sh` on to `tcsh`. Similarly, to invoke the exertion `sortJob`, we call `sortJob.exert(null)`. Thus, the exertion is the program and the network shell at the same time, which might first come as a surprise, but close evaluation of this fact, shows it to be consistent with the meaning of object-oriented distributed programming. Here, the *virtual metacomputer* is a federation that does not exist when the exertion is created. Thus, the notion of the *virtual metacomputer* is encapsulated in the exertion that creates the required federation on-the-fly. The federation provides the implementation (metacomputer instructions) as specified in signatures of the EO program before the exertion runs on the network.

The `sortJob` program described above can be rewritten with just one exertion task only instead of exertion job as follow:

```
1.  // Create service signatures
2.  Signature catSignature, sortSignature, uniqSiganture;
3.  catSignature = new ServiceSignature("Reader",
4.         "cat", Type.PREPROCESS);
5.  sortSignature = newServiceSignature("Sorter",
6.         "sort", Type.PROCESS);
7.  uniqSiganture = new ServiceSignature("Filter",
8.         "uniq", Type.POSTPROCESS);
9.
10. // Create an exertion task
11. Task sortTask = new Task("task-sort");
```

```
12. sortTask.addSignature(catSignature);
13. sortTask.addSignature(sortSignature);
14. sortTask.addSignature(uniqSiganture);
15.
16. // Create a service context
17. Context taskContext = new ServiceContext("c-sort");
18. taskContext.putInValue("/text/in/URL",
19.        "http://host/hello.txt");
20. taskContext.putOutValue("/text/out/contents", null);
21. taskContext.putOutValue("/text/out/sorted", null);
22. taskContext.putOutValue("/text/out/URL",
23.        "http://host/bye.txt");
24.
25. sortTask.setContext(taskContext);
26.
27. // Activate the task exertion
28. sortTask.exert(null);
```

In this version of the `sort.sh` analogy—`taskSort`, we create three signatures (lines 2-8), but in this case three signature types are assigned, so we can batch them into a single task (lines 11-14). In the `jobSort` version all signatures are of the default `PROCESS` type and each task is created with its own context. Here we create one common `taskContext` (lines 17-23) that is shared by all signature operations. Finally, on line 28, we execute the exertion task `sortTask`.

The major difference between the two EO programs `jobSort` and `taskSort` is in the exertion execution. The execution of `jobSort` is in fact coordinated by a `Jobber`, but the execution of the `taskSort` is coordinated by the service provider implementing the `Sorter` interface that binds to the `PROCESS` signature `sortSignature`. If the provider implementing the `Sorter` interface, implements also two other interfaces `Reader` and `Filter`, then the execution of `taskSort` is more efficient as all three operations can be executed by the same provider with no need of network communication between a `Jobber` and three collaborating providers in the `jobSort` federation.

4.1. *Service Messaging and Exertions*

In object-oriented terminology, a message is the single means of passing control to an object. If the object responds to the message, it has an operation and its implementation (method) for that message. Because object data is

encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name (identifier) of the receiving object, the name (selector) of operation to be invoked, and its parameters. In the unreliable network of objects; the receiving object might not be present or can go away at any time. Thus, we should postpone receiving object identification as late as possible. Grouping related messages per one request for the same data set makes a lot of sense due to network invocation latency and common errors in handling. These observations lead us to service-oriented messages called *exertions*. An exertion encapsulates multiple *service signatures* that define operations, a s*ervice context* that defines data, and a *control strategy* that defines how operations flow during exertion execution. Different types of control exertions (Section 4.4) can be used to define collaboration control strategies that can also be configured with signature flow type and access type attributes (see Section 4.2). Two basic exertion categories are distinguished: elementary and composite exertion called *exertion task* and *exertion job*, respectively. Corresponding task and job control strategies are described in Section 4.5.

As explained in Section 3, an exertion can be activated by calling exertion's `exert` operation: `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 submit an exertion onto the network and to perform executions of exertion's signatures on various service providers, 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 (exertions), is done through the use of the generic `Servicer` interface and the operation `service` that all SORCER services are required to provide—`Servicer.service(Exertion, Transaction):Exertion`. This top-level service operation takes an exertion as an argument and gives back an exertion as the return value. In Section 5 we describe how this operation is realized in the federated method invocation framework.

So why are exertions used rather than directly calling on a provider's method and passing service contexts? There are three basic answers to this.

First, passing exertions helps to aid with the network-centric messaging. A service requestor can send an exertion out onto the network—`Exertion.exert()`—and any service provider can pick it up. The receiving provider can then look at the interface and operation requested within the exertion, and if it doesn't implement the desired `PROCESS` interface or provide its

desired method, it can continue forwarding it to another service provider who can service it.

Second, passing exertions helps with fault detection and recovery. Each exertion has its own completion state associated with it to specify if it has yet to run, has already completed, or has failed. Since full exertions 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 service context input nodes that may have caused the problem. Since exertions provide all the information needed to execute an exertion including its control strategy, a user would be able to pause a job between tasks, analyze it and make needed updates. To figure out where to resume an exertion, the service provider would simply have to look at the exertion's completion states and resume the first component 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 creation new collaborative applications at the exertion runtime [35].

Third, the provider can analyze the received exertion for compliance with security polices before any of its signatures can be executed. In particular the `Authenticator` provider is used to check for the requestor proper identity, and the `Authorizer` provider is consulted if all exertion's signatures are accessible to the requestor.

4.2. *Service Signatures*

An activated exertion—`Exertion.exert()`—initiates the dynamic federation of all needed service providers dynamically—as late as possible—as specified by signatures of top-level and nested exertions. An exertion signature is compared to the operations defined in the service provider's interface along with a set of signature attributes describing the provider, and if a match is found, the appropriate operation can be invoked on the service provider. In federated method invocation (FMI) [34] signatures specify indirect invocations of provider methods via the service operation of the top-level `Servicer` interface as described in Section 4.1.

A service `Signature` is defined by:

- signature name—a custom name
- service type name—a service name corresponding to the provider's type (Java interface)
- selector of the service operation—an operation name defined in the service type

- operation type—`Signature.Type`: `PROCESS` (default), `PREPROCESS`, `POSTPROCESS`, `APPEND`
- service access type—`Signature.Access`: `PUSH` (default) - *direct* binding to service providers, or `PULL` – *indirect* binding via a shared exertion space maintained by the `Spacer` service
- flow of control type—`Signature.FlowType`: `SEQUENTIAL` (default), `PARALLEL`, `CONCURRENT`
- priority—integer value used by exertion's control strategy
- execution time flag—if true, the execution time is returned in the service context
- notifyees—list of email addresses to notify upon exertion completion
- service attributes—required requestor's attributes matching provider's registration attributes

An exertion can comprise of a collection of `PREPRROCESS`, `POSTPROCESS`, and `APPEND` signatures, but having only one `PROCESS` signature. The `PROCESS` signature defines the binding provider for the exertion. An `APPEND` signature defines the service context received from the provider specified by this signature. The received context is then appended in runtime to the exiting context that is processed later by `PREPROCESS`, `PROCESS`, and `POSTPROCESS` operations of the exertion. Appending a service context allows a requestor to use actual network data in runtime not available to the requestor when the exertion is activated.

Different languages have different interpretations as to what constitutes and operation signature. For example, in C+ and Java the return type is ignored. In FMI the parameters and return type are all of the Context type. Using the UML advanced operation syntax, the exertion operation (prefixing it with the `<<service>>` stereotype and postfixing with tagged values), can be defined as follow:

```
Context { interface = service-type-name, type = operation-type,
access = access-type, flow = flow-type, priority = integer,
timing = boolean, notfiees = notfiees-list, attributes =
registration-attribute-list }.
```

4.3. *Service Contexts*

A *service context,* or simply a *context,* defined by the `Context` interface, is a data structure that describes service provider ontology along with related data. A provider‘s ontology is controlled by the provider vocabulary that describes data and the relations between them 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. In service context, *attributes* and their *values* are used as atomic conceptual primitives, and *complements* are used as composite ones. A *complement* is an attribute sequence (path) with a value at the last position. A *context* consists of a *subject complement* and a set of defining *context complements*. The context usually corresponds to a sentence of natural language (a subject with multiple complements) [38].

A service context is a tree-like structure described conceptually by the EBNF syntax specification as follows:

```
1. context = [ subject  ":" ] complement { complement }.
2. subject = element.
3. complement = element ";".
4. element= path [ "=" value ].
5. path = ["/"] attribute { "/" attribute } [ { "<"
   association ">" } ] [ { "/" attribute } ].
6. value = object.
7. attribute = identifier.
8. relation = domain "|" product.
9. association = domain "|" tuple.
10. product = attribute { "|" attribute }.
11. tuple = value { "|" value }.
12. attribute  = identifier.
13. domain = identifier.
14. association = identifier.
15. identifier = letter { letter | digit }.
```

A relation with a single attribute taking values, is called a *property* and is denoted as `attribute|attribute`. To illustrate the idea of service context, let's consider the following example (graphically depicted in Fig. 4 where the subject `/laboratory/name=SORCER` is indicated in green color and the association person in red):

```
/laboratory/name=SORCER: /university=TTU;
/university/department/name=CS;
/university/department/room;
number=20B;
phone/number=806-742-1194;
phone/ext=237;
```

```
/director<person|Mike|W|Sobolewski>/email=sobol@cs.ttu.edu;
```

where absolute and relative paths are used, and the relation `person` is defined as follows:

`person|firstname|initial|lastname` and the following properties are used: `firstname`, `initial`, `lastname`, `name`, `university`, `email`, `number`, `ext`.

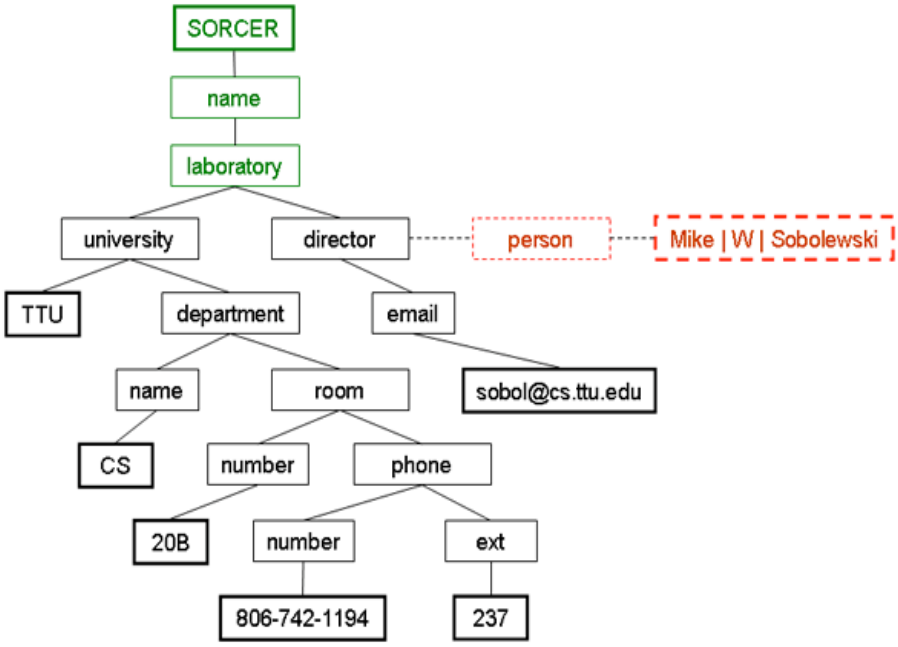

Fig. 4. An example of a service context

A context leaf node or *data node* is where the actual data resides. All absolute context paths define a service namespace. The context namespace with data nodes appended to its context paths is called a *context model*, or simply a *context*. A context path is a hierarchical name for a data item in a leaf node. Note that a limited service context can be represented as an XML document—what has been done in SORCER for interoperability—but the power of the `Context` type comes from the fact that any Java object can be naturally used as a data node. In particular, exertions themselves can be used as data nodes and then executed by providers as needed to run complex iterative programs, e.g., nonlinear multidisciplinary optimization [20].

4.4. *Exertion Types*

A `Task` instance specifies an elementary step in EO program. It is an analog of a statement in procedural programming languages (see the examples in Section 4). Thus, it is a minimal unit of structuring in EO programming. If the provider binds to a `Task`, it has a method for the task's `PROCESS` signature. Other signatures associated with the `Task` exertion provide for appending, preprocessing, and postprocessing service contexts by the same provider or its collaborating providers. An `APPEND` signature defines the context received from the provider specified by this signature. The received context is then appended at runtime to the task context later processed by `PREPROCESS`, `PROCESS`, and `POSTPROCESS` operations of the task. Appending a service context allows a requestor to use shared network data in runtime. A `Task` is the single means of passing control to an application service provider in FSOOA. Note that a task can specify a batch of operations that operate on the same service context—a `Task`'s shared execution state. All operations of the task, which are defined by its signatures, can be executed by the receiving provider or a group of federating providers coordinated by the provider receiving the task.

A `Job` instance specifies a "block" of task and other jobs. It is the analog of a procedure in imperative programming languages. In EO programming it is a composite of exertions that makeup the network collaboration. A `Job` can reflect a workflow with branching and looping by using control exertions (see Fig. 5).

The following control exertions define algorithmic logic in EO programming: `IfExertion`, `WhileExertion`, `ForExertion`, `DoExertionThrowExertion`, `TryExertion`, `BreakExertion`, `ContinueExertion`. Currently implemented in SORCER exertions including control types are depicted in Fig. 5.

4.5. *Exertion Control Strategies*

In Section 4.1 and 4.2 top-level exertion messaging and service signatures were described. This section will present how they are used, at the task level and job level, to manage flow of control in EO programs. Before we delve into a task and job execution strategy, let's look at three related infrastructure providers identified by the following interfaces: `Jobber`, `Spacer`, and `Cataloger`.

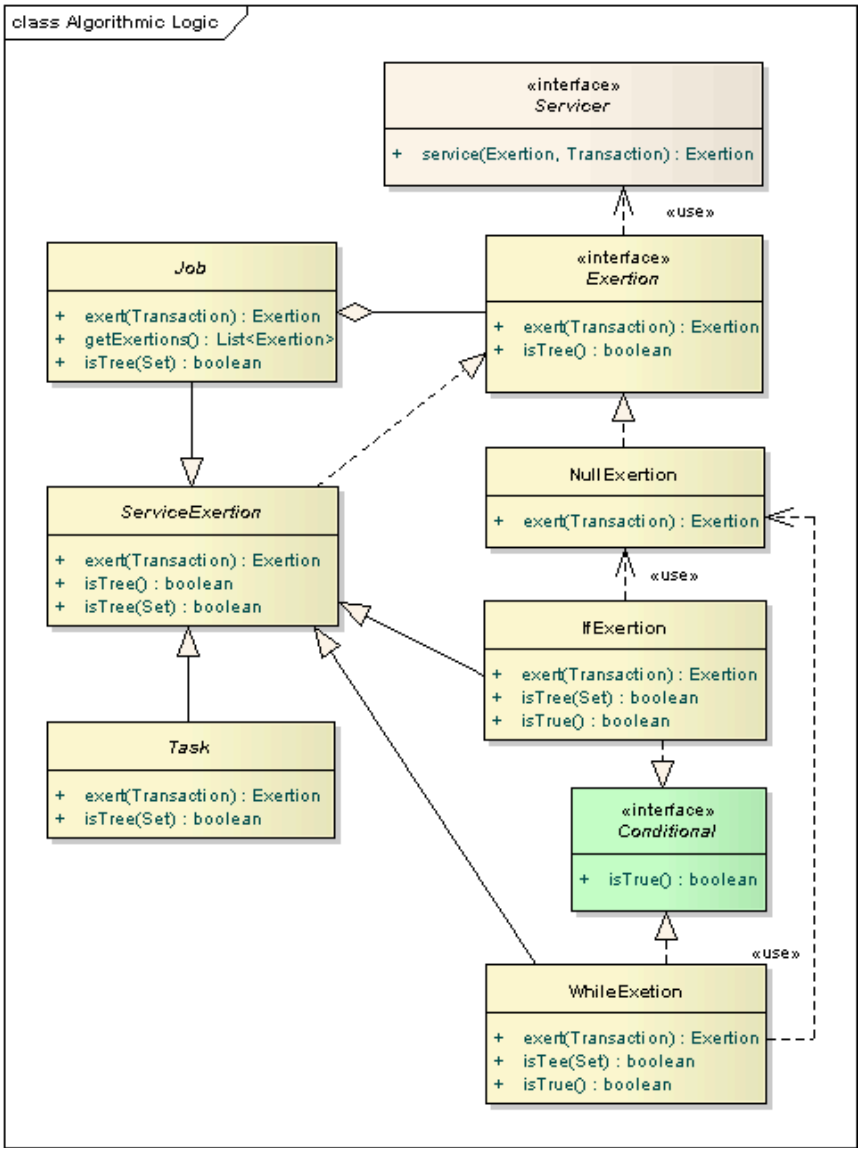

Fig. 5. Exertion types including control exertions that allow for algorithmic logic in EO programming.

To begin processing a job, a service requestor must exert the job that finds its way dynamically to either a `Jobber` or `Spacer` service using:

```
Exertion.exert(Transaction):Exertion.
```

The `Jobber` is responsible for coordinating the execution of the job, much like a command shell coordinates the execution of a batch script (see the

programming examples in Section 4). The `Jobber` acts as a service broker by calling upon the proper service providers to execute the component exertions within the given job. The `Jobber` can dispatch nested service requests either *directly*, when the jobber finds a proper provider by way of a `Cataloger` service or falling back to the Jini lookup service, or it can be dispatched *indirectly* via a shared exertion space through the use of a `Spacer` service.

SORCER extends the discovery and registration capabilities of the service-oriented architecture through the use of a service called the `Cataloger` service. A cataloger service looks through all the Jini lookup services that it is aware of and requests all the SORCER service registrations it can get. The cataloger organizes these registrations that include service proxies, into groups of the same type. Whenever a service requestor needs a certain service, it can go to a cataloger instead of a lookup service to find what it needs. The cataloger will distribute registrations for the same service in a round-robin fashion to help balance the load between service providers of the same service type.

SORCER also extends task/job execution abilities through the use of a `Spacer` service. The `Spacer` service can drop an exertion into a shared object space, provided by the Jini JavaSpaces service [10], in which several providers can retrieve relevant exertions from the object space, execute them, and return the results back to the object space.

As described before, an exertion is associated with a collection of signatures. There is only one `PROCESS` signature in this collection and multiple instances of `APPEND`, `PREPROCESS`, and `POSTPROCESS` signatures. The `PROCESS` signature is responsible for binding to the service provider that executes the exertion. The exertion activated by a service requestor can be submitted directly or indirectly to the matching service provider. In the direct approach, when signature's access type is `PUSH`, the exertion's `ServicerAccessor` (see Fig. 6) finds the matching service provider against the service type and attributes of the `PROCESS` signature and submits the exertion to the found provider. Alternatively, when signature's access type is `PULL`, a `ServiceAccessor` can use a `Spacer` provider that simply drops the exertion into the shared exertion space to be pulled by matching providers. Each SORCER service provider looks continuously into the space for exertions that match a provider's interfaces and attributes. Each service provider that picks up a matched exertion from the exertion space returns the exertion being executed back into the space, then the requestor (`Tasker`, `Jobber`, or `Spacer`) picks up the executed exertion from the space. The exertion space provides a kind of automatic load balancing—the fastest available service provider gets an exertion from the space and joins the exertion's federation.

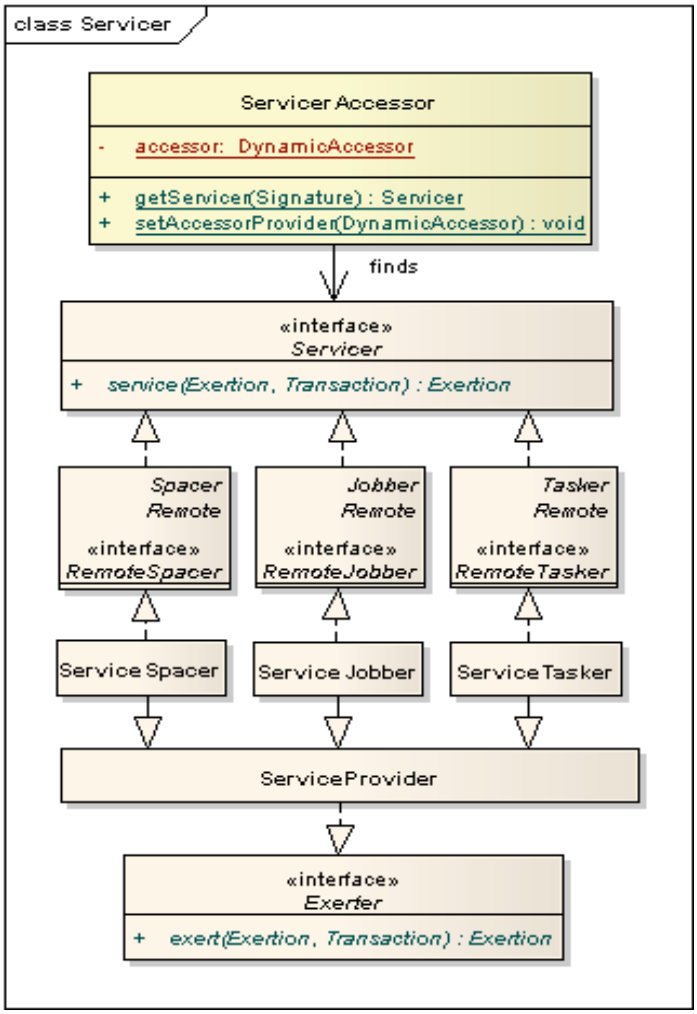

Fig. 6. The primary types of SORCER providers: `Tasker`,

When a receiving service provider gets a task (directly or indirectly) then the task signatures are executed as follows:

- First, all `APPEND` signatures are processed by the receiving provider in the order specified in the task. The order of signatures is defined by signature priorities, if the task's flow type is `SEQUENTIAL`; otherwise they are dispatch in parallel. In the result the task's service context is appended with dynamic data delivered from context providers specified by the append signatures. Appended complementary shared contexts are managed by the receiving provider according to the remote Observer/Observable design pattern [14].

- Second, all `PREPROCESS` signatures are executed in the order specified in the task. The order is defined as in 1) above. In the result the task context is ready for applying its `PROCESS` method.
- Third, the `PROCESS` signature is executed and results are captured in the task context including any exceptions and errors.
- Forth, all `POSTPROCESS` signatures are executed in the order specified in the task. The order is defined as in 1) above. Finally the resulting task with the processed context is returned to the requestor.

A domain-specific provider calls by reflection the method specified in the exertion `PROCESS` signature (interface and selector). All application domain methods that are used in exertion signatures have the same signature: a single `Context` type parameter and a `Context` type return vale. Thus a domain-specific interface looks like a common Java RMI interface with the above simplification on the common signature for all application-specific operations defined in the provider remote interfaces.

The default job's `PROCESS` signature defines a runtime binding to a `Jobber`. Alternatively the `Spacer` interface can be used in a job `PROCESS` signature. Two major parameters: job `PROCESS` signature's access type and its flow type determine the top-level control strategy. Additionally, job's service context, called a *control context*, defines job's execution preferences. When a `Jobber` or `Spacer` gets an exertion job then a relevant dispatcher is assigned by a dispatcher factory that takes into account job's access type, flow type, and its control context configuration. In the SORCER environment there are twelve types of dispatchers that implement different types of control strategies. The assigned dispatcher manages the execution of the job's component exertions either sequentially or in parallel (depending on the value of flow type), and accessing collaborating providers either directly or indirectly (depending on the value of access type). The default top-level control strategy implements a master/slave computing model with sequential or parallel execution of slave exertions with the master exertion executed as the last one, if any. In general, full algorithmic logic operations: concatenation, branching, and looping are supported. A job's workflow can be defined in terms of control exertions defined in Section 4.4. The access types of job signatures specify the way a jobber or spacer accesses collaborating service providers: directly or indirectly. Thus the `Spacer` provider is usually used for *asynchronous* access and the `Jobber` service is usually used to access needed service providers *synchronously*.

4.6. *Service-to-Service Infrastructure*

Exertion tasks are usually executed by service providers of the `Tasker` type and exertion jobs by rendezvous providers of `Jobber` or `Spacer` type. While a `Tasker` manages a single service context for the received task, a rendezvous provider manages a shared context (shared execution state) for the job federation and provides substitutions for input parameters that are mapped to output parameters (see the first programming example in Section 4) in service contexts of component exertions. Either one, a `Tasker` or rendezvous provider creates a federation of required service providers in runtime, but federations managed by rendezvous providers are usually larger in size than those managed by `Tasker`s. All SORCER service providers implement the top-level `Servicer` interface. A peer of the `Servicer` type that is unable to execute an `Exertion` for any reason forwards the `Exertion` to any available `Servicer` matching the exertion's `PROCESS` signature and returns the resulting exertion back to its requestor.

Thus, each `Servicer` can initiate a federation created in response to `Servicer.service(Exertion, Transaction)`. `Servicer`s come together to form a federation participating in collaboration for the activated exertion. When the exertion is complete, `Servicer`s leave the federation and seek a new exertion to join. Note that the same exertion can form a different federation for each execution due to the dynamic nature of looking up `Servicer`s by their required interfaces. Despite the fact that every `Servicer` can accept any exertion, many specialized `Servicer`s have well defined roles in FSOOA as described in Section 3.

5. The Triple Command Pattern

Polymorphism let us encapsulate a request then establish the signature of operation to call and vary the effect of calling the underlying operation by varying its implementation. The Command design pattern [14] establishes an operation signature in a generic interface and defines various implementations of the interface. In Federated Method Invocation (FMI), the three interfaces are defined with the following three commands:

1. `Exertion.exert(Transaction):Exertion`—join the federation;
2. `Servicer.service(Exertion, Transaction):Exertion`—
   request a service in the federation from the top-level `Servicer` obtained for the activated exertion;
3. `Exerter.exert(Exertion, Transaction):Exertion`—
   execute the argument exertion by the target provider in the federation.

These three commands define the *Triple Command* pattern that makes EO programming possible via various implementations of the three interfaces: `Exertion`, `Servicer`, and `Exerter`. The FMI approach allows for:

- the P2P environment via the `Servicer` interface,
- extensive modularization of programming P2P collaborations by the `Exertion` type,
- the customized execution of exertions by providers of the `Exerter` type, and
- common synergistic extensibility (exertions, servicers, exerters) from the triple design pattern.

Thus, requestors can exert simple (tasks) and structured metaprograms (jobs with control exertions) with or without transactional semantics as specified in 1) above. The Triple Command pattern in SORCER works as follows:

1. An exertion is invoked by calling `Exertion.exert(Transaction)`. The `exert` operation implemented in `ServiceExertion` uses `ServicerAccessor` to locate in runtime the provider matching the exertion's `PROCESS` signature. If a `Subject` in the exertion is not set, the requestor has to authenticate itself with the `Authenticator` service. A `Subject` represents a grouping of related security information (public and private credentials) for the requestor. After the successful requestor's authentication the `Subject` instance is created and the exertion can be passed onto the network.
2. If the matching provider is found, then on its access proxy (that can also be a smart proxy) the `Servicer.service(Exertion, Transaction)` method is invoked. The matching provider first verifies with the `Authorizer` service if the exertion's `Subject` is authorized to execute the operation defined by the exertion's `PROCESS` signature.
3. When the requestor is authenticated and authorized by the `Servicer` to invoke the method defined by the exertion's `PROCESS` signature, then the `Servicer` calls its `Exerter` operation: `Exerter.exert(Exertion, Transaction)`.
4. `Exerter.exert` method calls `exert` on `Tasker`, `Jobber`, or `Spacer` depending on the type of the exertion (`Task` or `Job`) and its control strategy. Permissions to execute the remaining signatures, if any, of `APPEND`, `PREPROCESS`, and `POSTPROCESS` types are checked with the `Authorizer` service. If all of them are authorized, then the provider calls the `APPEND`, `PREPROCESS`, `PROCESS`, and `POSTPROCESS` methods as described in Section 4.5.

In the FMI approach, a requestor can create an exertion, composed from any hierarchically nested exertions, with required service contexts. The provider's object proxy, service context template, and registration attributes are network-

centric; all of them are part of the provider's registration, so they can be accessed via `Cataloger` or lookup services by requestors on the network, for example service browsers [15], or custom service UI agents [45]. In SORCER, using these zero-install service UIs, the user can define data nodes in downloaded service context templates directly from providers and create related tasks/job interactively to be executed and monitored on the virtual metacomputer.

Individual service providers either `Tasker`s or rendezvous peers, implement their own `exert(Exertion, Transaction)` method according to their service semantics and related control strategy. SORCER taskers, jobbers, and spacers are implemented by `ServiceTasker`, `ServiceJobber`, and `ServiceSpacer` classes respectively (see Fig. 6). A SORCER domain-specific provider can be a subclass of `ServiceTasker`, `ServiceJobber`, or `ServiceSpacer`. Alternatively, any of these three providers can be set up as an application provider by dependency injection—using the Jini configuration methodology. Twelve proxying methods have been developed in SORCER to configure off-the-shelf `ServiceTasker`, `ServiceJobber`, or `ServiceSpacer`. In general, many different implementations of taskers, jobbers, and spacers can be used in the SORCER environment with different implementations of exertions. A service requestor via related attributes in its signatures will make appropriate runtime choices as to what implementations to run in exertion collaboration.

Invoking an exertion, let's say `program`, is similar to invoking an executable program `program.exe` at the command prompt. If we use the Tenex C shell (`tcsh`), invoking the program is equivalent to: "`tcsh program.exe`", i.e., passing the executable `program.exe` to `tcsh`. Similarly, to invoke a metaprogram using FMI, in this case the exertion `program`, we call "`program.exert(null)`", if no transactional semantics is required. Thus, the exertion is the metaprogram and the network shell of the SORCER metaoperating system. Here, the *virtual metacomputer* is a federation that does not exist when the exertion is created. Thus, the notion of the *virtual metacomputer* is encapsulated in the exertion that is managed by the FMI framework. In Fig. 7 a cloud represents a service grid while the metacomputer is a subset of providers that federate for the job shown below the cloud.

The fact that the exertion is the metaprogram and the network shell at the same time brings us back to the distribution transparency issue discussed in Section 2. It might appear that `Exertion` objects are network wrappers as they hide network intrinsic unpredictable behavior. However, `Exertion`s are not distributed objects, as they do not implement any remote interfaces; they are local objects representing network requests only. `Servicer`s are distributed objects, but `Servicer`s collaborate dynamically with other FMI infrastructure

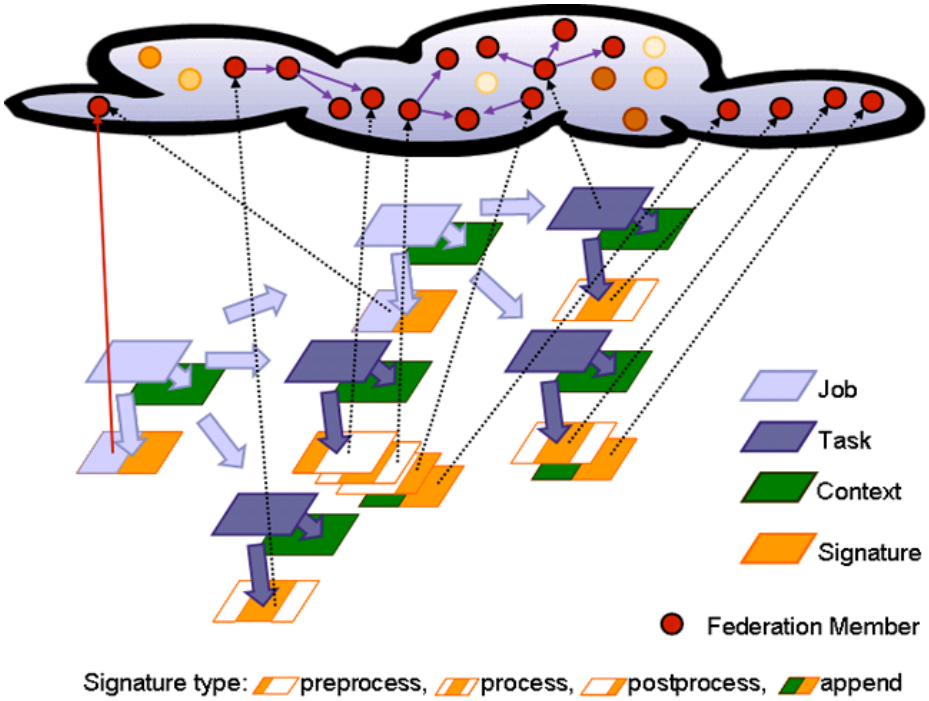

Fig. 7. A job federation. The solid line (the first from the left) indicates the originating FMI invocation: `Exertion.exert(Transaction)`. The top-level job with component exertions is depicted below the service grid (a cloud). Late bindings of all signatures are indicated by dashed lines that define the job's federation (metacomputer).

providers addressing real aspects of networking. The network intrinsic unpredictable behavior is addressed by the SORCER distributed objects: `TaskerS`, `JobberS`, `SpacerS`, `CatalogerS`, `FileStorerS`, `AuthenticatorS`, `AuthorizerS`, `KeyStorerS`, `PolicerS`, etc. (see Fig. 3) that define metacomputing operating system services. The FMI infrastructure facilitates EO programming and concurrent metaprogram execution using the presented framework and allows for constructing large-scale reliable object-oriented distributed systems from unreliable distributed components—`ServicerS`.

6. Conclusions

A distributed system is not just a collection of independent distributed objects—it is the network of dynamic objects that come and go. From the object-oriented point of view, the network of dynamic objects is the *problem domain* of object-

oriented distributed system that requires relevant abstractions in the *solution space*—for example the presented FMI framework. The exertion-based programming introduces the new abstraction of the solution space with *service providers* and *exertions* instead of object-oriented conventional *objects* and *messages*. Exertions not only encapsulate operations, data, and control strategies; they encapsulate relevant federations of dynamic service providers as well.

Service providers can be easily deployed in SORCER by injecting implementation of domain-specific interfaces into the FMI framework. These providers register proxies, including smart proxies, via dependency injection using twelve methods investigated in SORCER lab already. Executing a top-level exertion means sending it onto the network and forming a federation of currently available infrastructure (FMI) and domain-specific providers at runtime. The federation works on service contexts of all nested exertions collaboratively as specified by control strategies of the top-level and component exertions. The fact that control strategy is exposed directly to the user in a modular way allows him/her to create new distributed applications on-the-fly. For the updated exertion and its refined control strategy, the created federation becomes the new implementation of the applied exertion—a truly adaptable exertion–oriented application. When the federation is formed then each exertion operation has its corresponding method (code) on the network available. Services, as specified by exertion signatures, are invoked only indirectly by passing exertions on to providers via service object proxies that in fact are access proxies allowing for service providers to enforce security policies on access to required services. If the access to use the operation is granted, then the operation defined by an exertion's `PROCESS` signature is invoked by reflection.

The FMI framework allows for the P2P computing via the `Servicer` interface, extensive modularization of `Exertion`s and `Exerter`s, and extensibility from the Triple Command design pattern. The presented EO programming methodology has been successfully deployed and tested in multiple concurrent engineering and large-scale distributed applications [5, 20, 30, 18, 22, 19, 35, 2, 44, 13, 12, 21, 11].

Acknowledgments

This work was partially supported by Air Force Research Lab, Air Vehicles Directorate, Multidisciplinary 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. I would like to thank all my colleagues at

AFRL/RBSD and the SORCER Lab, TTU. They shared their views, ideas, and experience with me, and I am very thankful to them for that. 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. Berger, M., and Sobolewski, M., SILENUS – A Federated Service-oriented Approach to Distributed File Systems, In Next Generation Concurrent Engineering, ISPE/Omnipress, pp. 89-96 (2005)
2. Berger, M., Sobolewski, M. (2007) Lessons Learned from the SILENUS Federated File System, *Complex Systems Concurrent Engineering*, Loureiro, G. and L.Curran, R. (Eds.). Springer Verlag, ISBN: 978-1-84628-975-0, pp. 431-440.
3. Berger M., Sobolewski, M., A Dual-time Vector Clock Based Synchronization Mechanism for Key-value Data in the SILENUS File System, IEEE Third International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS '07), Hsinchu, Taiwan (2007)
4. Birrell, A. D. & Nelson, B. J., Implementing Remote Procedure Calls, XEROX CSL-83-7, October 1983.
5. Burton, S. A., Tappeta, R., Kolonay, R. M., Padmanabhan, D.,Turbine Blade Reliability-based Optimization Using Variable-Complexity Method, 43rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Denver, Colorado. AIAA 2002-1710 (2002).
6. Edwards W.K., Core Jini, 2nd ed., Prentice Hall (2000)
7. Fallacies of Distributed Computing. Accessed on: January 15, 2008. Available at: http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing
8. FIPER: Federated Intelligent Product EnviRonmet. Available at: http://sorcer.cs.ttu.edu/fiper/fiper.html. Accessed on: January 15, 2008.
9. Foster I., Kesselman C., Tuecke S., The Anatomy of the J. Supercomputer Applications, 15(3) (2001)
10. Freeman, E., Hupfer, S., & Arnold, K. JavaSpaces™ Principles, Patterns, and Practice, Addison-Wesley, ISBN: 0-201-30955-6 (1999)
11. 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 2008, Vol.16, pp 139-147.
12. Goel S., Shashishekara, Talya S.S., Sobolewski M., Service-based P2P overlay network for collaborative problem solving, Decision Support Systems, Volume 43, Issue 2, March 2007, pp. 547-568 (2007)
13. Goel, S, Talya S., and Sobolewski, M., Preliminary Design Using Distributed Service-based Computing, Proceeding of the 12th Conference on Concurrent Engineering: Research and Applications, ISPE, Inc., pp. 113-120 (2005)
14. Grand M., Patterns in Java, Volume 1, Wiley, ISBN: 0-471-25841-5 (1999)

15. Inca X™ Service Browser for Jini Technology. Available at: http://www.incax.com/index.htm?http://www.incax.com/service-browser.htm. Accessed on: January 15, 2008.
16. JXTA. Available at: https://jxta.dev.java.net/. Accessed on: January 15, 2008.
17. Jini architecture specification, Version 2.1. Available at: http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html. Accessed on: January 15, 2008(2001)
18. Kao, K. J., Seeley, C.E., Yin, S., Kolonay, R.M., Rus, T., Paradis, M.J., Business-to-Business Virtual Collaboartion of Aircraft Engine Combustor Design, Proceedings of DETC'03 ASME 2003 Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Chicago, Illinois (2003)
19. Khurana V., Berger M., Sobolewski M., A Federated Grid Env. with Replication Services. In Next Generation Concurrent Engineering, ISPE/Omnipress (2005)
20. Kolonay, R.M., Sobolewski, M., Tappeta, R., Paradis, M., Burton, S., Network-Centric MAO Environment. The Society for Modeling and Simulation International, Westrn Multiconference, San Antonio, TX (2002)
21. Kolonay, R. M., Thompson, E.D., Camberos, J.A., Franklin Eastep, F., 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 (2007)
22. Lapinski, M., Sobolewski, M., Managing Notifications in a Federated S2S Environment, International Journal of Concurrent Engineering: Research & Applications, Vol. 11, pp. 17-25 (2003)
23. McGovern J., Tyagi S., Stevens M.E., Mathew S., Java Web Services Architecture, Morgan Kaufmann (2003)
24. Nimrod: Tools for Distributed Parametric Modelling. Retrieved March 5, 2008, from: http://www.csse.monash.edu.au/~davida/nimrod/nimrodg.htm.
25. Package net.jini.jeri. Available at: http://java.sun.com/products/jini/2.1/doc/api/net/jini/jeri/package-summary.html. Accessed on: January 15, 2008.
26. Pitt E., McNiff K., java.rmi: The Remote Method Invocation Guide, Addison-Wesley Professional (2001)
27. Project Rio, A Dynamic Service Architecture for Distributed Applications. Available at: https://rio.dev.java.net/. Accessed on: January 15, 2008.
28. Röhl, P.J., Kolonay, R.M., Irani, R.K., Sobolewski, M., Kao, K. A Federated Intelligent Product Environment, AIAA-2000-4902, 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Long Beach, CA, September 6-8 (2000)
29. Ruh W.A., Herron T., Klinker P., IIOP Complete: Understanding CORBA and Middleware Interoperability, Addison-Wesley (1999)
30. Sampath, R., Kolonay, R. M, Kuhne, C. M., 2D/3D CFD Design Optimization Using the Federated Intelligent Product Environment (FIPER) Technology. AIAA-2002-5479, 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Atlanta, GA, (2002).
31. Sobolewski, M., Federated Collaborations with Exertions, 17h IEEE International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (2008).

32. Sobolewski, M, SORCER: Computing and Metacomputing Intergrid, 10th International Conference on Enterprise Information Systems, Barcelona, Spain (2008).
33. Sobolewski, M., Exertion Oriented Programming, IADIS, vol. 3 no. 1, pp. 86-109, ISBN: ISSN: 1646-3692. 2008.
34. Sobolewski M., Federated Method Invocation with Exertions, Proceedings of the 2007 IMCSIT Conference, PTI Press, ISSN 1896-7094, pp. 765-778, (2007)
35. Sobolewski M., Kolonay R., Federated Grid Computing with Interactive Service-oriented Programming, International Journal of Concurrent Engineering: Research & Applications, Vol. 14, No 1., pp. 55-66 (2006)
36. Sobolewski, M., Soorianarayanan, S., Malladi-Venkata, R-K., Service-Oriented File Sharing, Proceedings of the IASTED Intl., Conference on Communications, Internet, and Information technology, pp. 633-639, ACTA Press (2003)
37. Sobolewski M., Federated P2P services in CE Environments, Advances in Concurrent Engineering, A.A. Balkema Publishers, 2002, pp. 13-22 (2002)
38. Sobolewski, M., Percept Conceptualizations and Their Knowledge Representation Schemes, Ras Z.W. and Zemankova M. (Eds.) Methodologies for Intelligent Systems, Lecture Notes in AI 542, Berlin: Springe-Verlag, pp. 236-245 (1991)
39. Soorianarayanan, S., Sobolewski, M., Monitoring Federated Services in CE, Concurrent Engineering: The Worldwide Engineering Grid, Tsinghua Press and Springer Verlag, pp. 89-95 (2004)
40. SORCER Research Group. Available at: http://sorcer.cs.ttu.edu/. Accessed on: January 15, 2008.
41. SORCER Research Topics. Available at: http://sorcer.cs.ttu.edu/theses/. Accessed on: January 15, 2008
42. Sotomayor B., Childers L., Globus® Toolkit 4: Programming Java Services, Morgan Kaufmann (2005)
43. Thain D., Tannenbaum T., Livny M. Condor and the Grid. In Fran Berman, Anthony J.G. Hey, and Geoffrey Fox, editors, Grid Computing: Making The Global Infrastructure a Reality. John Wiley (2003)
44. Turner, A., Sobolewski, M., FICUS—A Federated Service-Oriented File Transfer Framework, *Complex Systems Concurrent Engineering*, Loureiro, G. and L.Curran, R. (Eds.). Springer Verlag, ISBN: 978-1-84628-975-0, pp. 421-430 (2007)
45. The Service UI Project. Available at: http://www.artima.com/jini/serviceui/index.html. Accessed on: January 15, 2008.
46. Waldo J., The End of Protocols, Available at: http://java.sun.com/developer/technicalArticles/jini/protocols.html. Accessed on: January 15, 2008.
47. Zhao, S., and Sobolewski, M., Context Model Sharing in the FIPER Environment, Proc. of the 8th Int. Conference on Concurrent Engineering: Research and Applications, Anaheim, CA (2001)