Josip Stjepandić · Nel Wognum
Wim J.C. Verhagen
Editors

# Concurrent Engineering in the 21st Century

## Foundations, Developments and Challenges

Springer

# Chapter 4 Technology Foundations

**Michael Sobolewski**

**Abstract** The chapter focuses on the underlying concepts of concurrent engineering technology from the point of view of concurrent process expression and its actualization. It lays out the evolution of computing platforms and networking complexity that constitute the foundation of every distributed information system required for concurrent engineering. Network integration is the working foundation for computer-based approaches to concurrent engineering. Therefore, an architecture of a concurrent engineering system is presented from the point of view of evolving methodologies of remote method invocation. Within the architecture the integration of concurrent distributed processes, humans, tools and methods to form a transdisciplinary concurrent engineering environment for the development of products is presented in the context of cooperating processes and their actualization. To work effectively in large, distributed environments, concurrent engineering teams need a service-oriented programming methodology along with a common design process, domain-independent representations of designs, and general criteria for decision making. Evolving domain-specific languages (DSLs) and service-oriented platforms reflect the complexity of computing problems we are facing in transdisciplinary concurrent engineering processes. An architecture of a service-oriented computing environment (SORCER) is described with a service-oriented programing and a coherent operating system for transdisciplinary large-scale computing.

**Keywords** Concurrent engineering · Metacomputing · Transdisciplinary computing · Service-oriented architectures · Var-modeling · Var-oriented programming · Exertion-oriented programming

M. Sobolewski (✉)
United States Air Force Research Laboratory, WPAFB, Dayton, OH 45433, USA
e-mail: sobol@sorcersoft.org

M. Sobolewski
Polish-Japanese Institute of Information Technology, 02-008 Warsaw, Poland

© Springer International Publishing Switzerland 2015
J. Stjepandić et al. (eds.), *Concurrent Engineering in the 21st Century*,
DOI 10.1007/978-3-319-13776-6_4

## 4.1 Introduction

Concurrent engineering, as the name suggests, is the approach of doing all necessary activities at the same time [1]. It is the unison of all facets of the product life cycle to minimize modifications in a prototype, i.e., to decrease design iterations performed during product development.

Concurrent Engineering (henceforth CE) is characterized by a focus on customer requirements. Moreover, it embodies the belief that quality is built into the product, and that it (quality) is a result of continuous improvement of a process. This concept is not new; in fact, the approach is quite similar to the "tiger team" approach characteristic of small organizations. The "tiger team" essentially is a small group of people working closely for a common endeavor, which might be product development. The magnitude of the problem is usually small with few conflicting constraints. The approach works well for small organizations; however, in large organizations the technique needs to be modified and restructured. It is here that CE comes into picture. CE envisages translating the "tiger team" concept to big organizations and such "tiger teams" will work with a unified product concept. Because team members can be at geographically different, networked locations, this requires far-reaching changes in the work culture, ethical values and information technology (IT), and a distributed infrastructure of the organization.

Commonplace design activities involve sequential information transfer from "concept designers" to "design finishers". When design activities are finished, the people involved in them get detached from the design chain. Thus, the people involved in earlier design phases do not interact with people in the later stages. A natural consequence of this procedure is that errors go on propagating themselves down the chain and are usually detected at a stage where rectifications/modifications become both costly and undesirable. The philosophy of continuous improvement implies changes at the initial stages with the aim of minimizing changes at later stages. To achieve this, it is imperative that strong communication exists between product developers of all stages and end-users.

Integrated, parallel, product and process design is the key to concurrent design. The CE approach as opposed to the sequential approach advocates such a parallel design effort. The objective is to ensure that serious errors don't go undetected and that the design intent is fully captured. The above-mentioned integrated design process should have the following features:

1. There must be strong information sharing system, thus enabling design teams to have access to all corporate facilities as well as work done by individual teams.
2. Any design process is necessarily an iterative process requiring successive redesigns and modifications. The CE process should ensure that the effects of a change incorporated by one team on other design aspects are automatically analyzed. Moreover, the affected functional units should be notified of the changes.

3. The CE process must facilitate an appropriate trade-off analysis leading to product-process design optimization. Conflicting requirements and constraint violations must be identified and concurrently resolved.
4. All relevant aspects of the design process must be recorded and documented for future reference.

The integration process discussed here is *transdisciplinary* and *strategic* integration in an organization. This binds the various functional discipline areas as engineering, support, manufacturing, logistics, etc., in the organization for greater efficiency of the whole venture. Strategic integration focuses on a company's business strategy. This strategy should tie decision making and other organizational policies together with the objective of realizing total quality management. Logistic integration is basically close coordination of the manufacturer with its customers and suppliers for cutting down logistic problems.

The realization of functional integration requires a versatile and a flexible *information management system* within the organization. The system must be domain independent and adaptable to both large and small industrial enterprises. Some essential and desirable characteristics of this system are listed below:

1. The system should be adaptable to the needs of the specific organization. It should be generic and at the same time modifiable to the requirements of the enterprise.
2. It should have a diverse repository of organized knowledge which is easily accessible across the spectrum of product life-cycle disciplines.
3. There must be an intelligent information distribution system which could provide information on a "need to know" and/or user-specified basis.
4. It should have facility of interfacing with software tools and application databases existing in user's organization.
5. The system should be capable of making the whole design team cognizant of the modifications done by a sub-group. In addition, it should have the ability to appraise the impacts of the modifications in a global manner, i.e., on all other design activities.
6. Even though most of the activities should be automated, there must be provision for human intervention at every stage. Furthermore, a manual bypass alternative for autonomous activities should be provided.
7. The system must support progressive refinement of product and process development from "design initiation" to the "design finalization" stage.
8. There must be tools permitting rapid prototyping and testing, therefore paving the way for commercial production.

The information management system can be visualized as the environment that allows expression of concurrent processes and their actualization. While process expression requires relevant domain-specific languages (DSLs) their actualization requires three basic blocks: data architecture (domain), distributed management framework (operating system), and software services (domain-specific processor).

Understanding the principles that run across process expressions and appreciating which language features and computing platforms are best suited for which type of process, bring these process expressions to useful life. No matter how complex and polished the individual process operations (software services) are, it is often the quality of the distributed operating system that determines the power of the concurrent engineering environments.

This chapter gives insight in the technological foundations of CE. In the Sect. 4.2 the basic terms are described from their historical perspective. Section 4.3 gives overview of rising complexity of computing. In the following Sect. 4.4 service platforms are classified in three categories and compared. Typical use case is described in Sect. 4.5, followed by conclusions and outlook (Sect. 4.6).

## 4.2 Background

Markov tried to consolidate all work of others on effective computability. He has introduced the term of algorithm in his 1954 book Teoriya Algorifmov [2]. The term was not used by any mathematician before him and reflects a limiting definition of what constitutes an acceptable solution to a mathematical problem:

> In mathematics, "algorithm" is commonly understood to be an exact prescription, defining a computational process, leading from various initial data to the desired result [2].

The important keyword in the Markov definition is "computational process" and the "algorithm" can be seen as a way to mathematically express the process. The mathematical view of process expression has limited computing science to the class of processes expressed by algorithms. The following definition of an algorithm (consistent with Hilbert's proposal of 1920) is typical:

1. An algorithm must be a step-by-step sequence of operations.
2. Each operation must be precisely defined.
3. An algorithm must terminate in a finite number of steps.
4. An algorithm must efficiently yield a correct solution.
5. An algorithm must be deterministic in that, given the same input, it will always yield the same solution.

From experience in concurrent engineering since 1989 it becomes obvious that in computing science the common thread in all computing disciplines is process expression; that is not limited to algorithm or actualization of process expression by a single computer. Several process expressions have been defined. Below a list of known process expression and actualization solutions is presented:

1. An architecture is an expression of a continuously acting process to interpret symbolically expressed processes.
2. A user interface is an expression of an interactive human-machine process.

3. A mogram (which can be program or model, or both) is an expression of a computing process [3].
4. A mogramming (programming or modeling, or both) language is an environment within which to create symbolic process expressions (mograms).
5. A compiler is an expression of a process that translates between symbolic process expressions in different languages.
6. An operating system is an expression of a process that manages the interpretation of other process expressions.
7. A logic circuit is an actualization of a logical process.
8. A processor is an actualization of a process.
9. An application is an expression of the application process.
10. A computing platform is an expression of a runtime process defined by the triplet: domain—mogramming language, management—operating system, and carrier—processor.
11. A computer is an actualization of a computing platform.
12. A metamogram (metaprogram or metamodel, or both) is an expression of a metaprocess, as the process of processes.
13. A metamogramming language is an environment within which to create symbolic metaprocess expressions.
14. A metaoperating system is an expression of a process that manages the interpretation of other metaprocess expressions.
15. 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.
16. A metacomputing platform is an expression of a runtime process defined by its metamogramming language, metaoperating system, and metaprocessor.
17. A metacomputer is an actualization of a metacomputing platform.
18. Computer science is the science of process expression.
19. Computer engineering is the science of process actualization.
20. Software engineering is an expression of a reliable development process within which to create program design, its implementation, and all related documents.
21. An information system is an expression of an efficient process to retrieve, store, and transmit information.
22. IT is an expression of a reliable (24/7) process to maintain and manage computing and data assets to meet current and expected user needs.
23. Artificial intelligence is an expression of integrated processes to reason, learn, plan, communicate knowledge, and perceive the world to move and manipulate objects.
24. Concurrent engineering is an expression of concurrent integrated processes to develop complex products.
25. Cloud computing is an expression of a consolidated process using virtualized (guest) platforms on native (host) platforms with related software as services running on these host and guest platforms.

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 is usually ignored in mathematics as mathematicians are concerned with the nature of behavior of a process independent of how that process is expressed. By contrast, computer science is mainly concerned with the nature of the expression of processes independent of its behavior. That became obvious in 1990s when computer science was redefined as one of many disciplines of computing science (see Fig. 4.1):

I. Computer Engineering (CE)
II. Computer Science (CS)
III. Software Engineering (SE)
IV. Information Technology (IT)
V. Information Systems (IS)
VI. Concurrent Engineering (CCE).

A comprehensive definition of concurrent engineering is given in the IDA (Institute for Defense Analysis) report on concurrent engineering (see Chap. 2) [4]:

> Concurrent engineering is the systematic approach to the integrated, concurrent design of products and related processes including manufacture and support. This approach is to cause the developers, from the outset, to consider all the elements of product life-cycle from conception through disposal including quality, cost, schedule and user requirement.

The concurrent engineering method is still a relatively new design management system, but has had the opportunity to mature in recent years and to become a well-defined systems approach towards optimizing engineering design cycles. One of the most important reasons for the huge success of concurrent engineering is that by definition it redefines the basic design process structure that was commonplace for

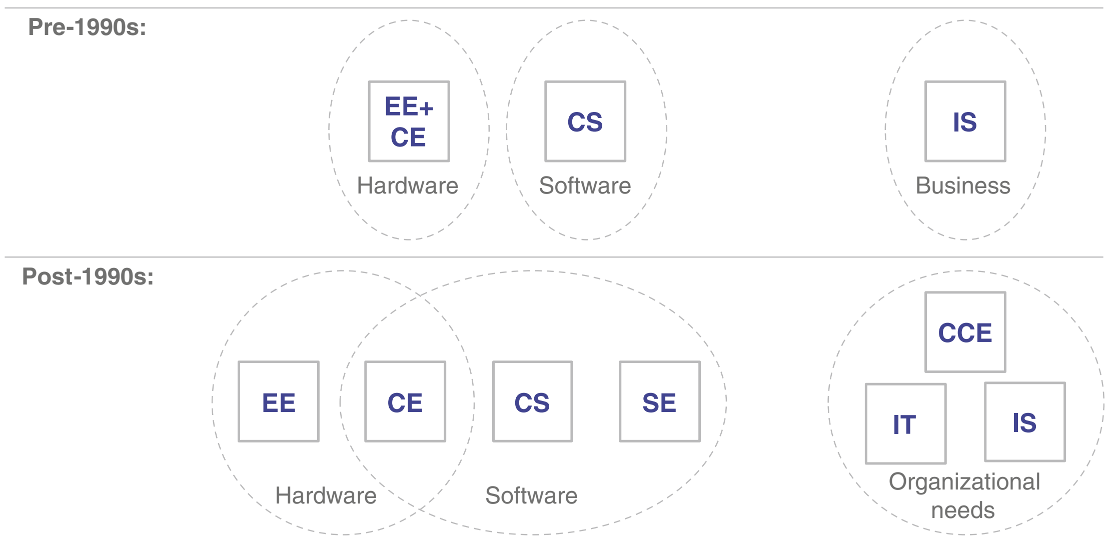

**Fig. 4.1** More computing disciplines post 1990s; computer science becoming the science of process expression. Concurrent engineering (*CCE*) initially funded by DICE/DARPA in 1989 has become a new discipline of computing science and engineering to address organizational needs

decades. This was a structure based on a sequential design flow. Concurrent engineering significantly modifies this outdated method and instead opts to use what has been termed an iterative or integrated development method based on concurrent (non-algorithmic) process expression in socio-technical systems (see Chap. 8).

The notion of algorithm separates all process expressions into algorithm and non-algorithm, but what purpose does it serve to know that one program is an acceptable mathematical solution and another is not? If the same process can be expressed by an algorithm and by a non-algorithm then which expression is better? Does determining whether or not a given expression is an acceptable mathematical solution implies a better computer system or helps in writing a better program? In fact, important process expressions do not qualify as Markov algorithms. A process defined by a neural network or every logic circuit is not a sequence of operations—it is not a process expression of the mathematical type. An operating system does not to have to terminate or yield a singular solution. It is not deterministic as it receives uncoordinated inputs from the outside world. Any simulation process with random inputs is not an algorithm. No program with a bug can be an algorithm as well as a concurrent program that does not satisfy the concept of sequential behavior. These and other facts have forced computer scientists to patch the concept of algorithm with multiple revisions and redefinitions of algorithm (nondeterministic algorithm, semi-algorithm, probabilistically good algorithm, random algorithm, infinite algorithm, etc.). Thus, the notion of mathematical algorithm simply does not provide a conceptual framework for questions that mostly computer scientists are concerned with nowadays.

Research in knowledge-based process expression at the Concurrent Engineering Research Center, West Virginia University (CERC/WVU, 1989–1994) has resulted in the DICEtalk platform [5]. Later, further work at the GE Global Research Center (GE GRC 1994–2002), Texas Tech University (TTU, 2002–2009), and Air Force Research Lab (AFRLWPAFB, 2006–present) has been focused on large-scale distributed process expression actualized by service-oriented platforms. Computing concepts and related results have been verified practically in large-scale real world systems for concurrent engineering. For the description of programming and distributed computing relevant to concurrent engineering platforms presented in this chapter we have taken into account most of the relevant papers published before in this area.

## 4.3 Complexity of Computing Systems

The concurrent engineering challenge is not to get lost in the continuously increasing complexities of products, their development processes and computing platforms. In this section we consider evolving complexity of programming languages and platforms with the complexity of remote procedure calls (RPC) directly related to network programming. Then service-oriented concepts are described with a meta-modeling architecture with convergence of three computing platforms for service-oriented mogramming.

Thinking more explicitly about programming languages (languages for humans) instead of software languages (languages for computers) may be our best tool for dealing with real-world complexity. Understanding the principles that run across process expressions and appreciating which language features and related computing platforms are best suited for which type of process, bring these process expressions to useful life. No matter how complex and polished individual process operations are (tools, applications, and utilities), it is often the abstraction and quality of the operating system and underlying network processor that determine the power of the computing system.

### *4.3.1 Meta-computing*

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."

From the very beginning of networked computing, the desire existed to develop protocols and methods that facilitate the ability of people and automated processes across different computers to share resources and information across different computing nodes in an optimized way. As ARPANET [6] began through the involvement of the NSF [7, 8] 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 [9, 10]. The latest of these ideas are the grid [11] and cloud [12], intended to be used by a wide variety of different users 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, and visualization and display, among other applications without regard for the specific nature of the hardware and underlying operating systems on the resources on which these jobs (executable files) are carried out. While a grid is focused on computing resource utilization, clouds are focused on platform virtualization in computer networks. In general, grid and cloud computing is client-server computing that abstract the details of the server away—one requests a service (resource), not a specific server (machine). However, both terms are vague from the point of view of computing architectures and programming models and refer to "everything that we already do" with executable files and client-server architectures.

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 containing service providers. These service providers 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. Service providers hosted by service objects (service containers) implement instructions of the virtual service processor (meta-processor). The meta-processor, with the help of its operating system, carries access to applications, tools, and utilities, i.e., programs as the instructions (services) of the meta-processor (while a processor is executing native machine instructions of executable codes). Services can collaborate with each other dynamically to provide aggregated services that execute a program collaborating with other component programs remotely and/or locally. Thus, a metacomputer is a collection of computers and devices connected by communication channels that facilitates distributed inter-process communications between users and allows users to share resources with other users.

Therefore, every meta-computer requires a computing platform that allows software to run utilizing multiple component computing platforms that communicate through a computer network. Different distributed platforms can be distinguished along with corresponding meta-processors—virtual organizations of computing nodes.

### *4.3.2 Programming and Platform Complexity*

The functionality of a computing platform depends on its operating system and its programming environment. Not every computing environment supports a complete platform. Each of them has a kind of programming environment but not each has an adequate operating system. For example, the first computer ENIAC did not have an operating system and it was programmed with switches and cables. In order to run a new program, ENIAC needed not only a new program to be entered manually using switches but also to be rewired. It took us the past half-century to move from programming environments with cables and switches, via perforated tapes, punch cards, to executable files and scripts to be easily created within integrated development environments (IDE) by software developers. The platforms and related programming models have evolved as process expression from the sequential process expression actualized on a single computer to the concurrent process expression activated on multiple computers. An 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 eight quantum leaps in process expression and their related architectures:

1. Sequential programming (e.g. stored-program on digital computer)
2. Multi-threaded programming (e.g. Java platform)
3. Multi-process programming (e.g. Unix platform)
4. Multi-machine-process programming (e.g. computer network)
5. Knowledge-based programming (e.g. DICEtalk)
6. Service-protocol oriented programming (e.g. Web and Grid services)
7. Service-object oriented programming (e.g. Jini)
8. Federated service-object oriented programming (e.g. SORCER)

One of the key elements of each distributed platform is communication between computer nodes and management of reliable network connections. All service-driven platforms are usually focused on communication protocols and service execution using a form of network middleware. However, most do not have a service-oriented operating system that deals with efficient management of services as platform commands, front-end programming (service scripting), and reliable runtime networking.

### 4.3.3 Generations of Remote Procedure Call

Socket-based communication forces us to design distributed applications using a read/write (input/output communication) interface, which is not how we generally design non-distributed applications based on procedure call (request/response). In 1983, Birrell and Nelson devised the RPC [13], a mechanism to allow programs to call procedures on other hosts. So far, six RPC generations can be distinguished:

1. First generation RPC—Sun RPC (ONC RPC) and DCE RPC, which are language, architecture, and OS independent;
2. Second generation RPC—CORBA [14] and Microsoft DCOM/OPC, which add distributed object support;
3. Third generation RPC—Java RMI is conceptually similar to the second generation but supports the semantics of object invocation in different address spaces that are built for Java only [15]. Java RMI fits cleanly into the language with no need for standardized data representation, external interface definition language, and with behavioural transfer that allows remote objects to perform operations that are determined at runtime;
4. Fourth generation RPC—next generation of Java RMI, Jini Extensible Remote Invocation—JERI—[16] with dynamic proxies, smart proxies, network security, and with dependency injection by defining exporters, end points, and security properties in configuration files;
5. Fifth generation RPC—Web/OGSA Services [17, 18] and the XML movement including Microsoft WCF/.NET;
6. Sixth generation RPC—Federated Method Invocation (FMI) [19] allows for concurrent invocations on multiple federating compute nodes in the Service-Oriented Computing Environment (SORCER) [20–25].

All RPC generations listed above are based on a form of service-oriented architecture (SOA). CORBA, RMI, and Web/OGSA service providers are object-oriented wrappers of network interfaces that hide object distribution and ignore the real nature of a network using classical object abstractions that encapsulate network connectivity by using existing network technologies. The fact that object-oriented languages are used to create corresponding object wrappers does not mean that distributed objects created this way have a great deal to do with object-oriented distributed programming.

Each platform and its programming language reflect a relevant abstraction, and usually the type and quality of the abstraction implies the complexity of problems we are able to solve. For example, a procedural language provides an abstraction of an underlying machine language. In the SORCER environment developed at Texas Tech University [19], a service provider is a remote object that accepts network requests to participate in a collaboration—a process by which service providers work together to seek solutions that reach beyond what any one of them could accomplish on their own. SORCER messaging is based on exertions, the service commands that encapsulate explicitly data, operations, and control strategy. An exertion can federate multiple service providers concurrently according to its control strategy by managing transparently all low-level Jini/JERI networking details.

The SORCER meta-computing environment adds an entirely new layer of abstraction to the practice of service-oriented computing: exertion-oriented (EO) programming. The EO programming makes a positive difference in service-oriented programming primarily through a new meta-computing platform as experienced in many large-scale projects including applications deployed at GE Global Research Center, GE Aviation, Air Force Research Lab (AFRL), and SORCER Lab. The new abstraction is about managing object-oriented distributed system complexity laid upon the complexity of the unreliable network of computers—the meta-computer.

An exertion submitted to the network dynamically binds to all relevant and currently available service providers in the network. The providers that dynamically participate in this invocation are collectively called the exertion federation. This federation is also called the exertion meta-processor since federating services are located on multiple computer nodes held together by the SORCER operating system (SOS) so that, to the requestor submitting the exertion, it looks and acts like a single processor.

The SORCER environment provides the means to create service-oriented mograms [25] and execute them using the SORCER runtime infrastructure. Exertions can be created using interactive user agents uploaded/downloaded on-the-fly to/from service providers. Using these interfaces, the user can create, execute, and monitor the execution of exertions within the SORCER platform. Exertions can be kept for later reuse, allowing the user to quickly create new scripts or EO programs on-the-fly in terms of existing exertions, usually kept for reuse.

SORCER is based on the evolution of concepts and lessons learned in the federated intelligent product environment (FIPER) project [26], a $21.5 million program founded by National Institute of Standards and Technology (NIST). Academic research on FMI, SOS, and EO programming was established at the SORCER Laboratory, TTU (2002–2009), where twenty-eight SORCER related research studies were performed. Currently, the SORCER Lab (http://sorcersoft.org) as the independent open source organization is focused on maturing the SORCER platform in collaboration with AFRL/WPAFB, SORCERsoft.com, and collaborating partners in China, and Russia.

### *4.3.4 SOA and Metamodeling Architecture*

The SOA is a software architecture using loosely coupled software services that integrates them 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, as illustrated in Fig. 4.2 (the left chart). In SOA, the client is referred to as a service requestor and the server as a service provider which is responsible for deploying a service in the network, publishing its service to one or more registries, and allowing requestors to bind and execute the service. Providers advertise their availability in the network; registries intercept these announcements and collect published services. The requestor looks for a service by sending queries to registries and making selections from the available services. Requestors and providers can use discovery and join protocols [16] to locate registries and then publish or acquire services in the network.

We can distinguish the service object-oriented architecture (SOOA), where providers are network objects accepting remote invocations (call/response), from the service protocol-oriented architecture (SPOA), where a communication (read/write) protocol is fixed and known beforehand by the provider and requestor. Based on that protocol and a service description obtained from the service registry, the requestor can bind to the service provider by creating a proxy used 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 correct name of the service beforehand.

In SOOA, a proxy—an object implementing the same service interfaces (service types) as its service provider—is registered with the registries and it is always ready for use by requestors. Thus, in SOOA, the service provider publishes the proxy as the active surrogate object with a codebase annotation, e.g., URLs to the code defining proxy behavior (RMI and Jini ERI). In SPOA, by contrast, a passive

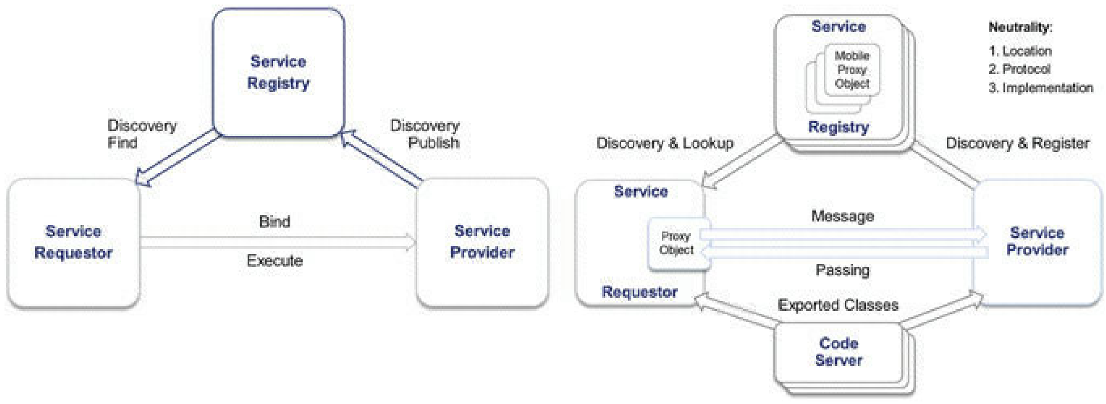

**Fig. 4.2** SOA versus SOOA

service description is registered (e.g., an XML document in WSDL for Web/OGSA services, or an interface description in IDL for CORBA); the requestor then has to generate the proxy (a stub forwarding calls to a provider) based on a service description and the fixed communication protocol (e.g., SOAP in Web/OGSA services, IIOP in CORBA).

SPOA and SOOA differ in their method of discovering the service registry (see Fig. 4.2, the right chart). SORCER uses dynamic discovery protocols to locate available registries (lookup services) as defined in the Jini architecture [16].

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, 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 distributed application.

The first challenge of SORCER based on SOOA is to allow the end user not only to use the existing individual services as-is in the network, but also to create new compound services at runtime that are both globally and locally distributed federations of services. In other words, instead of invoking a single standard service in the network as-is, the computing environment should allow end users to create front-end complex service collaborations that become innovative new tools, applications, or utilities composed from the existing and new services at will.

The second challenge of the SORCER is to bring to front-end SO programming all existing programming styles seamlessly unified. Therefore, firstly the mogramming environment should be designed to express service collaborations functionally and procedurally as workflows. Secondly it should enable service interactions under control of the proper operating system to actualize the front-end services (exertions) using the domain-specific back-end heterogeneous services (service providers).

No matter how complex and polished the individual operations are, it is often the quality of the glue that determines the power of the distributed computing system. The SOS based on SOOA serves as the service-oriented glue for exertion-oriented mogramming [27]. It uses federated remote method invocation with location of service provider not explicitly specified in exertions [19]. A specialized infrastructure of distributed services supports discovery/join protocols for the SOS shell, federated file system, autonomic resource management, and the rendez-vous providers responsible for coordination of exertion federations. The infrastructure defines SORCER's service 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. Object proxying with discovery/join protocols provides for comprehensive neutrality of provider protocol, location, and implementation that is missing in SPOA programming environments as well.

A meta-model is usually defined as a model of a model. What is meant is that where a model defines a system (instance), a meta-model (classifier) defines the model. In particular, the process expression in which other expressions are modeled is often called a meta-model. Note, that key modeling concepts are Classifier and Instance, and the ability to navigate from an instance to its classifier. This fundamental concept can be used to handle any number of layers (sometimes referred to as meta-levels).

The DMC meta-modeling architecture is based on the notion of the meta-model, also called the DMC platform or DMC-triplet, in the form: <Domain, Management, Carrier>. For example, a computing platform: <mograms, operating system, processor> is the model of the DMC-triplet. A language platform: <language expressions, grammar, language alphabet> is the model of the DMC triplet as well.

Therefore, a computing platform is a composition of a DSL, management of mogram execution, and the processor that provides the actualization of both the language and its management.

The SORCER meta-model is a kind of UML class diagram depicted in Fig. 4.3, where each "class" is a DMC-triplet. This meta-modeling architecture distinguishes three computing platforms (instruction set (IS), object-oriented, and service-oriented platforms) and three mogramming platforms [exertion-oriented programming (EOP), par-oriented modeling (POM), and var-oriented modeling (VOM)].

When dealing with meta-levels to define computing systems (process expression/actualization) there are at least two layers that always have to be taken into account: the process expression/actualization or the meta-model (abstract model); and the specification of the concrete process expression/actualization or the model (concrete model).

The domain-specific problems expressed with the higher expressive power of DSLs and adequate service-oriented syntax and semantics allow end users to be focused on their domain problems and solution. Unfortunately, the common service-based practice forces end users to learn back-end software-programming languages, development processes, and creating software for computers instead of writing easily understandable and modifiable front-end domain-specific mograms for and by others and themselves.

The SORCER reference architecture presented below in Fig. 4.5 is a model of the DMC meta-model depicted in Fig. 4.3. SORCER is the SOOA platform that introduces an innovative programming model and operating system to create easily service-oriented mograms that express complex domain-specific solutions. SORCER mograms—var-models, par-oriented, var-oriented, and EOP—with the abstraction of a virtual service-oriented processor, are executed in the network by the SOS. SORCER introduces a FMI [21] used in its SOOA [28] by the SOS. The SOS implements all features required for the SOO platform listed in Table 4.1.

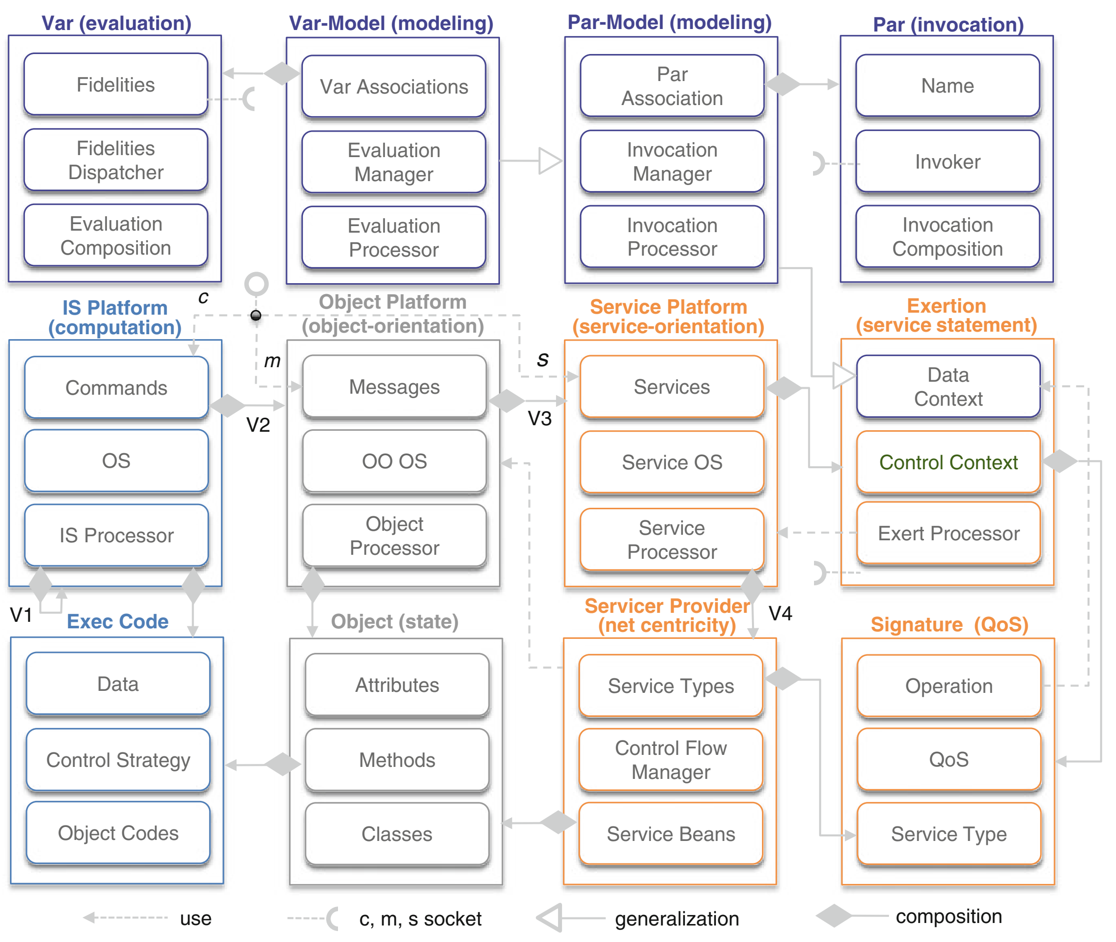

**Fig. 4.3** The DMC meta-modeling architecture with platforms for: computation (instruction set), object-orientation, and service-orientation. Each platform is shown as the instance of DMC triplet with corresponding executable item (exec code, object, service provider). The top layer of the architecture shows elements of service-oriented modeling. That complements exertion-oriented programming. The service platform manages the service providers (virtualization of service cloud —*V*3/*V*4 that are autonomically provisioned by the SOS on virtualized computation/object platforms (*V*1/*V*2)

## 4.4 Service Platforms

In this section three basic categories of service platforms are distinguished: *multiple machine* (*MM*), *service protocol-oriented*, and *service-object-oriented platforms*. Then three types of SORCER-based platforms are described that are SOOA based specialization for *true service-oriented computing* (SORCER), *grid computing*, (SGrid) and at the *integration framework for intergrid computations* (iGrid).

**Table 4.1** Quantum leaps in programming and platform complexity

| Programming | Benefit | Lost benefit | Platform support |
|---|---|---|---|
| Sequential | Order | | Batch processing before 1960s; from 1970s OS, e.g., UNIX with shell programming |
| Multi-threaded | Parallelism | Order | Thread management, e.g., Smalltalk, Java platform |
| Multi-process | SW isolation, safety | Execution context | OSs with pipes and sockets, e.g., UNIX with interprocess communication |
| Multi-machine | HW isolation, scalability | Global state, security | OSs with RPC and network tile systems, e.g., UNIX/NFS |
| Knowledge-base oriented | Ill-structured problem representation, logic (declarative) programming | Procedural execution | Knowledge representation handling, inference engine with procedural attachment, e.g., DICEtalk with percept knowledge representation |
| Service-protocol oriented | Service registry, code/resource location neutrality, implementation neutrality, platform neutrality | Trust, protocol neutrality as proxy is owned by the requestor, code/resource security due to dislocation | Service registry, protocol security, job scheduler, and virtual file system, e.g., CAMnet, CORBA, RMI, FIPER, Web Services, OSGA/Globus |
| Service-object oriented | Service-object spontaneity, code mobility, dynamic federations, registry location neutrality, protocol neutrality as proxy is owned by the provider, autonomic service-object/provider provisioning | Static service location, code security due to code mobility | Service interface types, object proxying, object registry, distributed events, transactions, leases, mobile code security, and disconnected operations, e.g., Jini/JERI, FMI/SORCER |

### *4.4.1 Three Categories of Service Platforms*

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

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

Different platforms of meta-computers can be distinguished along with corresponding types of virtual service processors. For a meta-program, the control strategy is a plan for achieving the desired results by applying the platform operations (services) to the data in the required service collaboration and by leveraging the dynamically federating resources. We can distinguish three generic meta-computing platforms, which are described below. Meta-computing requires a relevant computing abstraction as well.

Procedural languages provide an abstraction of an underlying machine language. An executable file represents a computing program whose content is interpreted as a program on the underlying native processor. A command can be submitted to a *job* (*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 network of compute resources. That can be done, for example, with the Nimrod-G grid resource broker scheduler or the Condor-G high-throughput scheduler [29]. Both rely on Globus/GRAM (Grid Resource Allocation and Management) protocol [18]. In this type of platform, called a *MM Platform* (see Fig. 4.4), executable files are moved around the network of compute nodes—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 fully developed. A series of programs ("jobs") was executed on a computer without human interaction or the possibility to

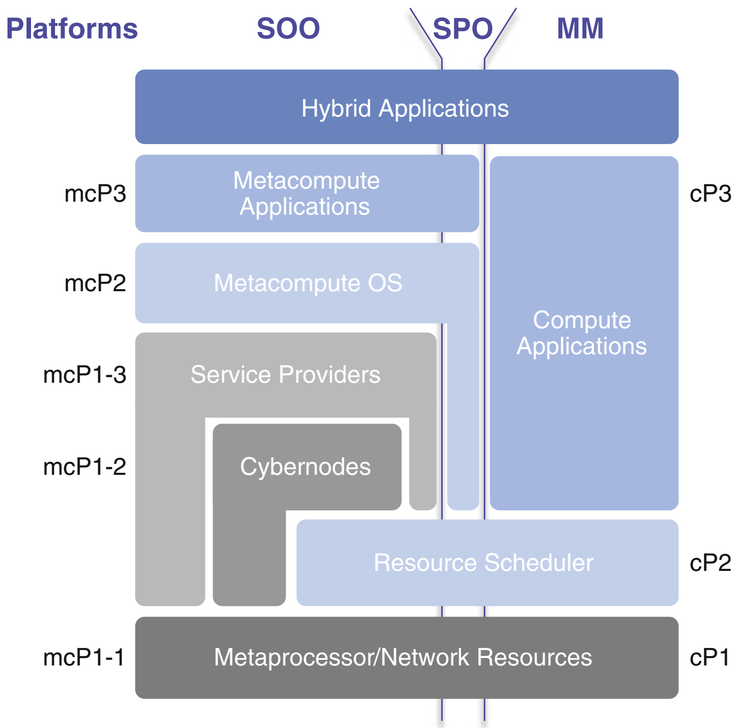

**Fig. 4.4** Three types of platforms: MM-Platform (*cP1*, *cP2*, *cP3*), SPO-Platform in the middle between two *vertical lines* (*cP1*, *cP2*, *mcP2*, *mcP3*), and SOO-Platform (*mcP1-1*, *mcP1-2*, *mcP1-3*, *mcP2*, *mcP3*). A cybernode provides a lightweight dynamic service container (service object), turning heterogeneous compute resources into homogeneous services available to the meta-computing OS

view any results before the execution is complete. The UNIX operating system (OS) of the 1970s matured with the well understood architectures used still successfully nowadays.

We consider a true meta-program as the process expression of hierarchically organized collaboration of remote component programs. A meta-program is a program of programs such that its instructions correspond to dynamically bound service providers corresponding to applications, tools, and utilities in the network. Its service-oriented operating system makes decisions about where, when, and how to run these service providers. In other words, the meta-program manipulates other programs remotely and dynamically as its data. Nowadays the similar computing abstraction is usually applied to the program executing on a single computer as to the meta-program executing in the network of computers, even though the executing environments (platforms) are structurally completely different. Most so called service-oriented programs are still written using software languages such as FORTRAN, C, C++ (compiled into native processor code), Java, Smalltalk (compiled into intermediate code), and interpreted languages such as Perl and Python the way it usually works on a single host. The current trend is to have these programs and scripts define remote computational modules as *service providers*. However, most grid programs are developed using the same abstractions and, in principle, run the same way on the MM-Computer as on a single computer, for example using executable codes moved to the available computing nodes in the network.

The meta-computer based on Web Services or Grid Services can be considered as the SPO-Computer with SPO-Platform shown in Fig. 4.4. The SPO-Platform uses a SPO-Manager running usually on a Web Application Server. This type of meta-computing in concept is reduced practically to the client-server model with all drawbacks related to static network connections as discussed in Sect. 4.3.3. The Web Services model with a SPO-Manager that supports for example Business Process Execution Language (BPEL) [30] allows for deployment of service assembly on the application server that looks to the end user as a single server command—not a program for the service collaboration created by the end user. In this case the WSBPEL compliant WS engine is required on the application server, for example Apache ODE. The engine organizes web services' calls following a process description written in the BPEL XML grammar. BPEL's messaging facilities depend on the use of the Web Services Description Language (WSDL) to describe outgoing and incoming messages from web services as specified during the service deployment on the application server, that is not available to end users.

We can distinguish three types of computing platforms depending on the nature of network operating system or middleware (see Fig. 4.4): the MM-Platform with computing layers cP1, cP2 and cP3; the SPO-Platform with computing/met-computing layers cP1, cP2, mcP2, and the SOO-Platform with meta-computing layers mcP-1, mcP1-2, mcP1-3, mcP2, and mcP3; and the hybrid of the previous three—intergrids (iGrids). Note that the MM-Platform is a virtual federation of processors (roughly CPUs) that execute submitted executable codes with the help of a resource broker. Either an SPO-Platform or SOO-Platform federation of services is managed by the form of middleware (operating system), however in the SPO case the application servers are used, but in the SOO case the federations of dynamic autonomous service objects are managed by the relevant operating system. Thus, the latter approach requires a service object-oriented methodology while in the former case the conventional client/server programming is sufficient. The hybrid of three platform abstractions allows for an iGrid to execute both programs and meta-programs as depicted in Fig. 4.4, where platform layers *P*1, *P*2, and *P*3 correspond to resources, resource management, and programming environment correspondingly.

One of the first SOO-Platform was developed under the sponsorship of the National Institute for Standards and Technology (NIST)—the FIPER [26, 31]. The goal of FIPER is to form a federation of distributed service objects that provide engineering data, applications, and tools in 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 both federating service providers and service requestors [32–37].

SORCER builds on top of FIPER to introduce a meta-computing operating system with all system services necessary, including service management (rendezvous services), a federated file system, and autonomic resource management, to support service-object oriented meta-programming. It provides an integrated solution for complex meta-computing applications. The SORCER meta-computing environment adds an entirely new layer of abstraction to the practice of meta-computing—exertion-oriented (EO) mogramming with a FMI. The EO mogramming makes a positive difference in service-oriented programming primarily through a new federated programming and modeling abstractions as experienced in many service-oriented computing projects including systems deployed at GE Global Research Center, GE Aviation, AFRL, SORCER Lab, and SORCER partners in China and Russia.

### *4.4.2 Service-Object Oriented Platform: SORCER*

The SORCER is a federated service-to-service (S2S) meta-computing environment that treats service providers as network peers with well-defined semantics of a federated SOOA that is based on the FMI. It incorporates Jini semantics of services [16] in the network and the Jini programming model with explicit leases, distributed events, transactions, and discovery/join protocols [16]. While Jini focuses on service management in a networked environment, SORCER is focused on EO mogramming and the execution environment for exertions (see the service platform in Fig. 4.3 as the meta-model of the architecture presented in Fig. 4.5). The abstract model depicted in Fig. 4.3 is the unifying representation for three concrete programming models: imperative languages for IS platforms (executable codes), the SORCER Java API for object platform [27], Exertion-Oriented Language (EOL)

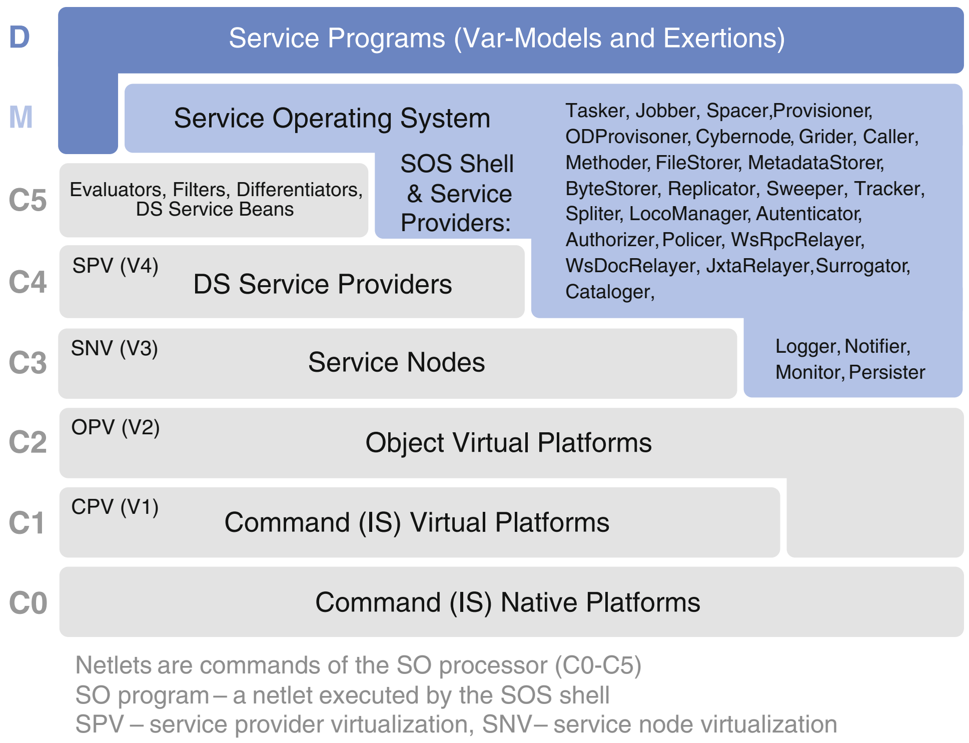
D
Service Programs (Var-Models and Exertions)
M
Service Operating System
SOS Shell & Service Providers:
Tasker, Jobber, Spacer,Provisioner, ODProvisoner, Cybernode, Grider, Caller, Methoder, FileStorer, MetadataStorer, ByteStorer, Replicator, Sweeper, Tracker, Spliter, LocoManager, Autenticator, Authorizer, Policer, WsRpcRelayer, WsDocRelayer, JxtaRelayer,Surrogator, Cataloger,
C5
Evaluators, Filters, Differentiators, DS Service Beans
C4
SPV (V4)
DS Service Providers
C3
SNV (V3)
Service Nodes
Logger, Notifier, Monitor, Persister
C2
OPV (V2)
Object Virtual Platforms
C1
CPV (V1)
Command (IS) Virtual Platforms
C0
Command (IS) Native Platforms
Netlets are commands of the SO processor (C0-C5)
SO program – a netlet executed by the SOS shell
SPV – service provider virtualization, SNV – service node virtualization

**Fig. 4.5** The SORCER layered architecture, where *C*0–*C*5 (carrier)—the metaprocessor with its service cloud at *C*4 and *C*3, platform cloud at *C*2 and *C*1, *M* (management)—SOS, *D* (domain)—service requestors; where *PV* and *OV* stands for provider and object virtualization respectively with the prefix *S* for service, *O* for object, and *C* for command

and Var-oriented Modeling Language (VML) for the SOS [25]. The notation of functional composition has been developed for both EOL and VML, which are usually complemented with the Java object-oriented syntax. Modeling in SORCER is emphasized in Fig. 4.3 by the top layer of DMC triplets: par, par-model, var, and var-model [24]. More details on EOL and VML can be found in source [25].

EOP is a service-oriented programming paradigm using service providers and service commands. Service commands—exertions—are interpreted by the SOS (M layer in Fig. 4.5) and represented by hierarchical data structures that consist of a data context, multiple service signatures, and a control context—to design distributed applications and computer programs. In EOP a service invocation on a provider is determined by a *service signature*. The signature usually includes the service type, operation of the service type, and expected quality of service (QoS). While exertion's signatures identify (match) the required collaborating providers, the control context defines for the SOS how and when the signature operations are applied to the data context.

An exertion is an expression of a distributed process that specifies for the SOS how service collaboration is actualized by a collection of providers playing specific roles used in a specific way [38]. The collaboration specifies a collection of cooperating providers—the exertion federation—identified by the exertion's signatures. Exertions encapsulate explicitly data, operations, and control strategy for the collaboration. The signatures are dynamically bound to corresponding service providers—members of the exerted collaboration.

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

Var-Oriented Programming (VOP) is a programming paradigm using service variables called *vars*—data structures defined by the triplet (fidelity) <evaluator, getter, setter> together with a var composition of evaluator's dependent variables—to design service-oriented programs and models. It is based on dataflow principles that changing the value of a var should automatically force recalculation of the values of vars that depend on its value. VOP promotes values defined by evaluators/getters/setters to become the main concept behind any processing. Getters play the role of filters and setters of vars persist values of their vars. Each var might have multiple fidelities selected dynamically during modeling/simulation analyses.

VOM is a modeling paradigm using vars in a specific way to define heterogeneous service federations of var-oriented models, in particular large-scale multidisciplinary analysis (MDA) models including response, parametric, and optimization models [39, 40]. The programming style of VOM is declarative. Models describe the desired results of the program without explicitly listing command or steps that need to be carried out to achieve the results. VOM focuses on how vars connect, unlike imperative programming, which focuses on how evaluators calculate. VOM represents models as a series of interdependent var connections, with the evaluators/getters/setters between these connections being of secondary importance.

The SORCER service requestors (D layer in Fig. 4.5) are expressed in three concrete programming syntaxes: the SORCER Java API [27], the functional composition form (EOL and VML) [25], and the graphical form [41]. The convergence of VML, and EOL into a service-oriented process expression is described in details in [25].

An Exertion is actualized by calling its exert operation. The SORCER FMI defines the following three related operations:

1. `Exertion#exert(Transaction):Exertion`—join the federation; the activated exertion binds to the available provider specified by the exertion's PROCESS signature;
2. `Servicer#service(Exertion, Transaction):Exertion`—requesting the service federation initiated at runtime by the bounding provider from (1) above; and
3. `Exerter#exert(Exertion, Transaction):Exertion`—invoked by all providers in the federation from (2) for their own component exertions. Each component exertion of the parent exertion from (1) is processed as in (2) above.

The above Triple Command Pattern [19] defines three key SORCER interfaces: `Exertion` (metaprogram), `Service` (S2S provider), and `Exerter` (domain-specific service provider specified by the exertion signature).

This approach allows for the S2S environment [38] via the `Service` interface, extensive modularization of Exertions and Exerters, and extensibility from the triple 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 as follows:

1. An exertion is actualized 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(Exertion)` 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(Exertion)`.
4. `Exerter#exert(Exertion)` operation is implemented by `ServiceTasker`, `ServiceJobber`, `ServiceSpacer`, and `ServiceConcatenator`. The `ServiceTasker` calls by the reflection the domain-specific operation given in the PROCESS signature of its argument exertion. All operations of provider's service type have a single argument: a `Context` type parameter and a `Context` type return value. Each service type is a Java interface implemented by the domain-specific service provider [15]. Tasker, Concatenator, Jobber, and Spacer are rendezvous system services of the SOS. They manage dynamic service federations (batch, block, workflow) of service providers in the network [24].

### *4.4.3 SORCER Grid Platform: SGrid*

To use legacy applications, SORCER supports a traditional approach to grid computing similar to those found in Condor [29] and Globus [18]. The SORCER-based incarnation is called SORCER grid or in short SGrid. Here, instead of exertions being executed by services providing business logic for collaborating exertions, the business logic comes from the service requestor's executable codes that seek compute resources in the network.

The SGrid services in the SORCER environment include Griders accepting exertions and collaborating with Jobbers, Spacers, and Concatenators as SGrid schedulers. Caller and Methoder service providers are used for task execution received from Concatenators, Jobbers, or pulled up from exertion space via Spacers. Callers execute provided codes via a system call as described by the standardized Caller's service context of the submitted task. Methoders download required Java code (task method) from requestors to process a submitted data context with the downloadable code specified in the requestor's exertion signature. In either case, the business logic comes from requestors; it is executable code specified directly or indirectly in the service context used by Callers, or mobile Java code executed by Methoders that is annotated by exertion signatures.

The SGrid with Methoders was used to deploy an algorithm called Basic Local Alignment Search Tool (BLAST) [42] to compare newly discovered, unknown DNA and protein sequences against a large database with more than three gigabytes of known sequences [43]. BLAST (C++ code) searches the database for sequences that are identical or similar to the unknown sequence. This process enables scientists to make inferences about the function of the unknown sequence based on what is understood about the similar sequences found in the database. Many projects at the USDA–ARS Research Unit, for example, involve as many as 10,000 unknown sequences, each of which must be analyzed via the BLAST algorithm. A project involving 10,000 unknown sequences requires about three weeks to complete on a single desktop computer. The S-BLAST implemented in SORCER [43], a federated form of the BLAST algorithm, reduces the amount of time required to perform searches for large sets of unknown sequences to less than one day. S-BLAST is comprised of BlastProvider (with the attached BLAST Service UI), Jobbers, Spacers, and Methoders. Methoders in S-BLAST download Java code (to execute a task operation) that initializes a required database before making a system call on the BLAST code. Armed with the S-BLAST's SGrid and seventeen commodity computers, projects that previously took three weeks to complete can now be finished in less than one day.

## *4.4.4 SORCER Intergrid Platform: IGrid*

In Sects. 4.4.2 and 4.4.3 two complementary platforms: SORCER and SGrid are described respectively. As shown in Fig. 4.6 a hybrid of three types of platforms is feasible to create intergrid (iGrid) applications that take advantage of both SORCER and SGrid platforms, and SPO-Platform synergistically. Legacy applications can be reused directly in SGrid or any MM-Platform and new complex, for example concurrent engineering, applications [44, 45] can be defined in SORCER.

Relayers are SORCER gateway providers that transform exertions to native representations and vice versa. The following exertion gateways have been developed: JxtaRelayer for JXTA and WsRpcRelayer and WsDocRelayer for for RPC and document style Web services, respectively. Relayers exhibit native and SORCER Grid behavior by implementing dual protocols. For example a JxtaRelayer (1) in Fig. 4.6 is at the same time a `Service` (1-) and a JXTA peer (1-) implementing JXTA interfaces. Therefore it shows up also in SORCER Grid and in the JXTA Grid as well. Native Grid providers can play the SORCER role (as SORCER wrappers), thus are available in the iGrid along with SORCER providers. For example, a JXTA peer 4- implements the `Service` interface, so shows up in the JXTA iGrid as provider 4. Also, native Grid providers via corresponding relayers can access iGrid services (bottom-up) Thus, the iGrid is a projection of Services onto meta-compute and compute Grids.

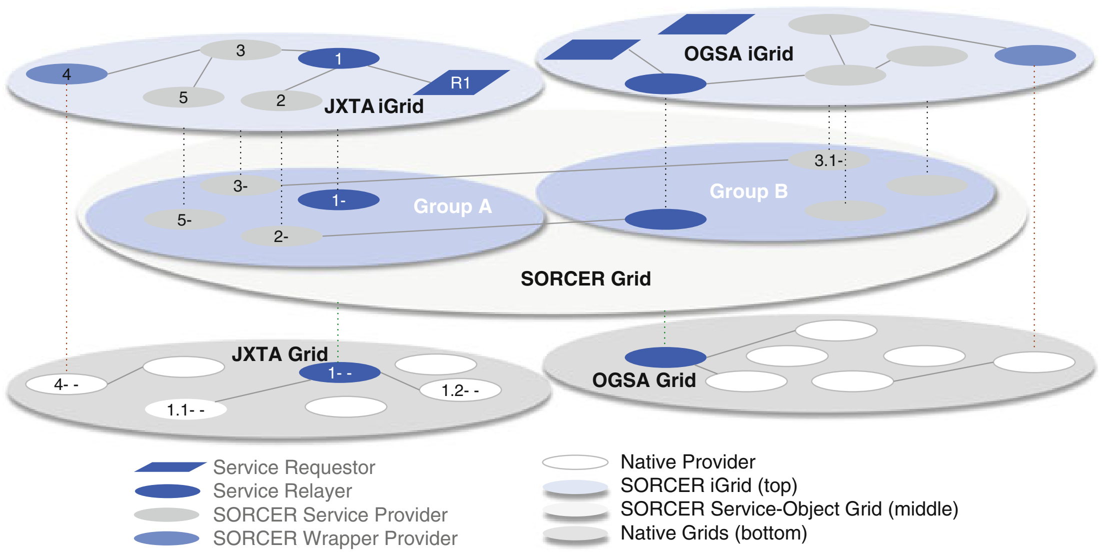

**Fig. 4.6** Integrating and wrapping native Grids with SORCER Grids (Group A and Group B). Two requestors, one in JXTA iGrid, one in OGSA iGrid submits exertion to a corresponding relayer. Two federations are formed that include providers from the two horizontal layers below the iGrid layer (as indicated by *continuous* and *dashed links*)

The iGrid-integrating model is illustrated in Fig. 4.6, where horizontal native technology grids (bottom) are seamlessly integrated with horizontal SORCER metacopute Grids (top) via the SOS services in Sorcer Grid (middle). Through the use of open standards-based communication—Jini, Web Services, Globus/OGSA, and Java interoperability—iGrid leverages the federated SOOA with its inherent neutrality of provider's protocol, location, and implementation along with the flexibility of EO mogramming and meta-compute federated OS.

## 4.5 A Case Study of an Efficient Supersonic Air Vehicle

The AFRL's Multidisciplinary Science and Technology Center (MSTC) is investigating conceptual design processes and computing frameworks that could significantly impact the design of the next generation of efficient supersonic air vehicle (ESAV). To make the technological advancements required of a new ESAV, the conceptual design process must accommodate both low- and high-fidelity trans-disciplinary engineering analyses. These analyses may be coupled and computationally expensive, which poses a challenge since a large number of configurations must be analyzed. In light of these observations, the ESAV design process was implemented using the SOS to combine propulsion, structures, aerodynamics, performance, and aero-elasticity in a MDA [40, 46]. The SORCER platform provides MDA automation and flexible service-oriented integration to distributed computing resources necessary to achieve the volume of analyses required for conceptual design.

While most service systems are based on the SPOA the SORCER platform utilizes the SOOA architecture. That makes SORCER exhibiting the following unique features with respect to existing grid and cloud systems:

1. Smart proxying for balancing business logic execution between a service requestor and provider with fat proxying for running the provider's code completely at the requestor side;
2. A self-healing runtime environment using network discover/join protocols for dynamic lookup services;
3. Location/implementation neutrality, but most importantly wire protocol neutrality and transport protocol selection at service deployment (transport endpoints);
4. The front-end mogramming with the capability of both back and frontend service provider development;
5. Unification of SO procedural (EOP) with SO functional composition (par-oriented and var-oriented-modeling);
6. Ease of parallelization with self-balancing exertion space computing and transactional semantics;
7. Front-end choice of PUSH or PULL execution of nested exertions;
8. Front-end (on-demand) autonomic service provider provisioning/unprovisioning;
9. Context awareness of the service-oriented computing with interoperability across service federations; and
10. Code mobility across service federations—dynamic behavioral transfer between requestors and providers.

The MDA is a blend of conceptual and preliminary design methods from propulsion, structures, aerodynamics, performance, and aero-elasticity disciplines. The process begins by parametrically generating discretized geometry suitable for several different analyses at varying fidelities. The geometry is used as input to compute several figures of merit of the aircraft, which include the aircraft drag polars, design mass, range, and aero-elastic performance. The different responses are evaluated for several flight conditions and maneuvers. These responses are then used to construct the objective and constraints of the multidisciplinary optimization (MDO) problem.

MDO generally requires a large number of MDAs be performed. This significant computational burden is addressed by using the SORCER platform. The SO and network-centric approach of SORCER enables the use of heterogeneous computing resources, including a variety of operating systems, hardware, and software. Specifically, the ESAV studies performed herein use SORCER in conjunction with a mix of Linux-based cluster computers, desktop Linux-based PCs, Windows PCs, and Macintosh PCs. The ability of SORCER to leverage these resources is significant to MDO applications in two ways: (1) it supports platform-specific executable codes that may be required by an MDA; and (2) it enables a variety of computing resources to be used as one entity (including stand-alone PCs, computing clusters, and high-performance computing facilities). The main requirements for using a computational resource in SORCER are network connectivity and Java platform compatibility. SORCER also supports load balancing across computational resources using space computing, making the evaluation of MDO objective and constraint functions in parallel a simple and a dynamically scalable process.

SOS Spacer providers enable different processes on different computers to communicate asynchronously with transactional semantics in a reliable manner [38]. Using Spacer services, SOS implements a self-load balancing app cloud (see Fig. 4.4) that can dynamically grow and shrink during the course of an optimization study, see an ESAV example Fig. 4.6-left. Various service providers or multiple instances of the same multifunctional (implementing multiple services types) service provider can be configured for parallelization in deployment/provisioning accordingly to compute power of their hosting environment (laptop, workstation, cluster, and supercomputer).

An exertion space or "space" is exertion storage in the network that is managed by the SOS. The space provides a type of shared memory where requestors can put exertions they wish to be processed by service providers. Service providers, in turn, continuously lookup up exertions in their space (also SOS service) to be processed. If a service provider sees a task it can operate on in its space, and the task has not been processed yet, the provider takes the task (removes it) from the space. The provider then performs the requested service and returns the task to the space as completed. Once the task has been returned to the space, SOS Spacer that initially wrote the task to the space detects the completed task then takes the task from the space and returns it to the submitting service requestor. Pars and vars are frequent SORCER requestors with their invokers and evaluators as space exertions. This way par-oriented and var-oriented models access various applications, tools, and utilities as ubiquities dynamic services in the network as illustrated in Fig. 4.7.

The ESAV service providers are used with an external optimization program as the SORCER requestor (Matlab Client, Fig. 4.7) to optimize an ESAV for range. The optimized design has a higher aspect ratio than the baseline design. The received results provide a degree of validation of the optimization code implementation, the SORCER ESAV parametric model, the SORCER providers, and the SOS [46].

The use of the space computing proved reliable and efficient. It was a straightforward process to add computers to the SORCER service Cloud as needed during the course of the two optimization studies. This flexibility proved valuable as the number of computers available varied from day-to-day. Parallelization of var-responses in parametric models with exertion space coordination resulted in significant savings. In this case it reduced the computational time to perform the optimization from 24 h to proximately 2 h [47]. The reduction is achieved mainly by the parallelization of SORCER parametric models for each parametric response (a vector of var values) and parallelization of var exertion evaluators (for each vector) executed using exertion space as illustrated in Fig. 4.7.

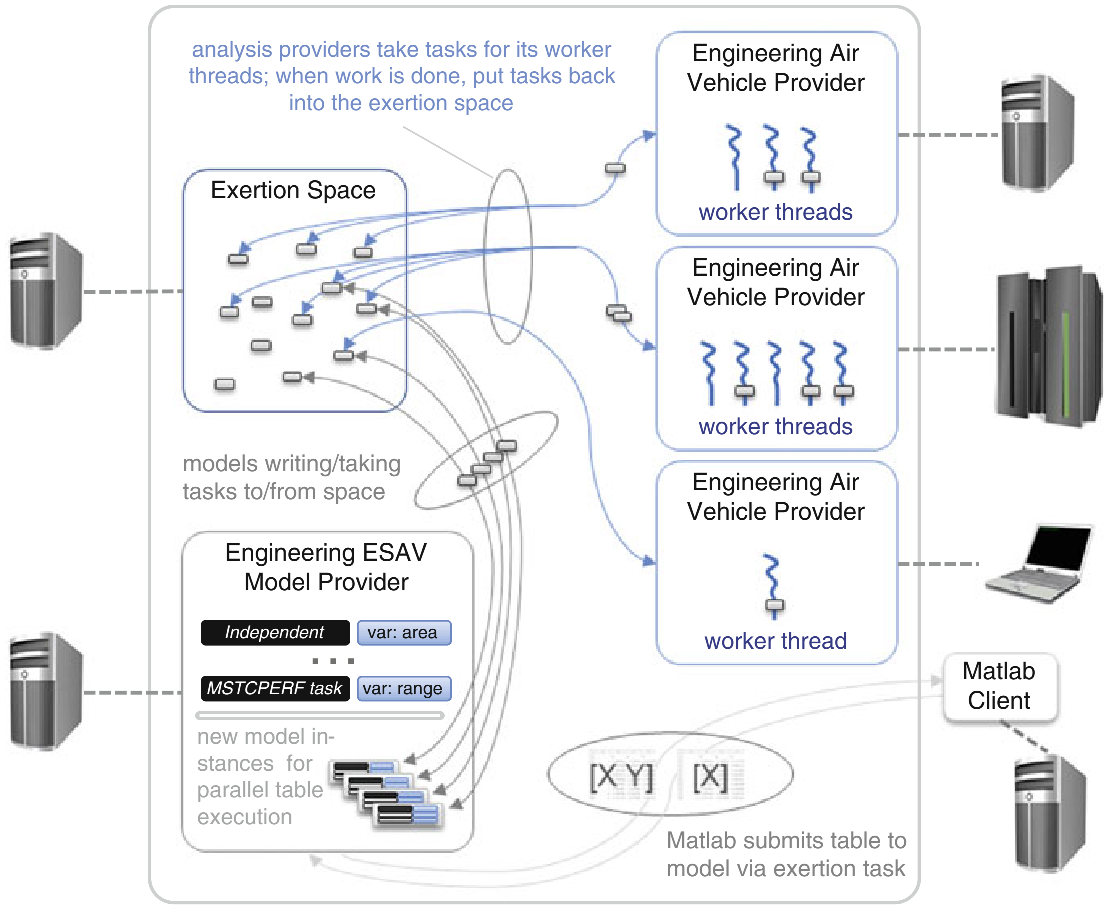
analysis providers take tasks for its worker threads; when work is done, put tasks back into the exertion space
Exertion Space
Engineering Air Vehicle Provider
worker threads
Engineering Air Vehicle Provider
worker threads
Engineering Air Vehicle Provider
worker thread
models writing/taking tasks to/from space
Engineering ESAV Model Provider
Independent
var: area
MSTCPERF task
var: range
new model instances for parallel table execution
Matlab Client
[X Y] [X]
Matlab submits table to model via exertion task

**Fig. 4.7** SORCER provides exertion space for a flexible, dynamic space computing for ESAV optimization studies. Exertions of variables in the parametric model are written into the space when variable values are needed. The exertions from the space are read and processed by engineering air vehicle service providers that return the results into the apace to be collected by the model for the requestor's optimization program

## 4.6 Conclusions and Outlook

To work effectively in large, distributed environments, concurrent engineering teams need a service-oriented programming methodology along with the common design process, domain-independent representations of designs, and general criteria for decision making. Distributed MDA and optimization are essential for decision making in engineering design that provide a foundation for service oriented concurrent engineering [44, 45].

As we move from the problems of the information era to more complex problems of the molecular era, it is becoming evident that new programming languages are required. These languages should reflect the complexity of meta-computing problems we are facing already in the molecular era, for example, concurrent engineering processes of collaborative design by hundreds or thousands of people working together and using thousands of programs written already in software

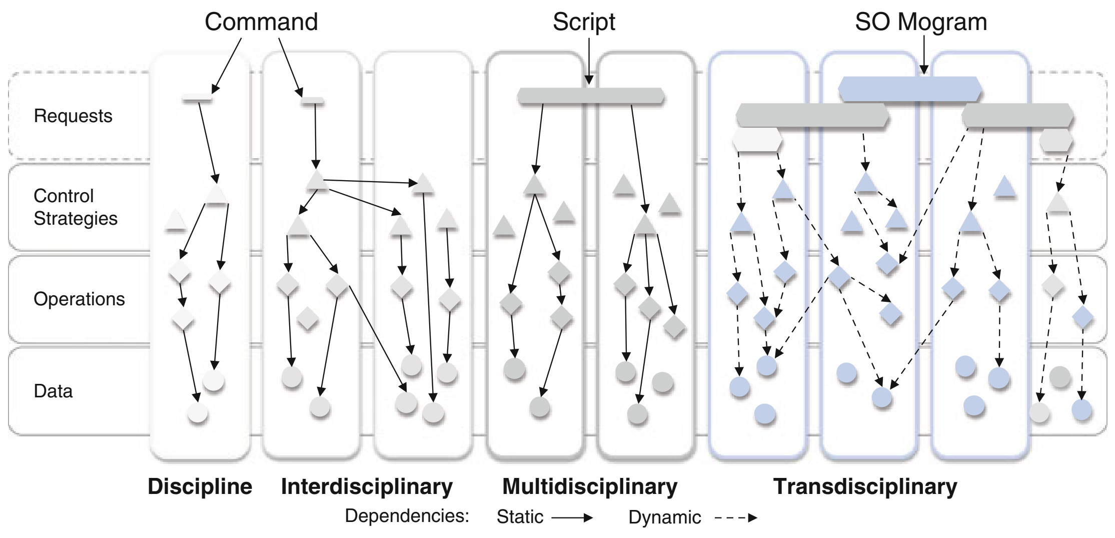

**Fig. 4.8** By providing easy-to-use, self-discovering services representing domain knowledge (data), tools (operations), and related technologies (control) with exertion-oriented and var-modeling (mogramming) methodology, the SORCER environment reduces integration and deployment costs, facilitate productivity, increases research collaboration, and advances the development and acceptance of secure and fault tolerant cross-disciplinary concurrent engineering solutions

languages that are located around the globe. The cross-disciplinary design of an aircraft engine or even a whole air vehicle requires dynamic large-scale cross-disciplinary meta-computing systems (see Fig. 4.8).

The EOP introduces the new abstraction of service-oriented programming with service providers and exertions instead of object-oriented conventional objects and messages. An exertion not only encapsulates signatures, data context, and control strategy, it encapsulates the matching federation of service providers as well. From the meta-computing platform point of view, exertions are entities considered at the programming level, service interactions at the operating system level, and federations at the processor level. Thus, exertions are process expressions that define service collaborations. The SOS manages the collaborations as FMI interactions on its virtual processor—the dynamically formed service federations.

Service providers can be easily deployed in SORCER by injecting implementation (executable code) of domain-specific interfaces into the FMI framework. The providers register proxies, including smart proxies, via dependency injection defined during their deployment. Executing the exertion, by sending it onto the network, means forming a required federation from currently available or provisioned service providers at runtime. Service providers in the federation work on service contexts of all component exertions collaboratively as specified by the control strategies of the corresponding component exertions.

The FMI framework defines federated SOA and allows for the P2P computing via the `Service` interface, extensive modularization of Exertions and Exerters, and extensibility from the Triple Command design pattern. The presented EOP methodology with the SOS, its federated file system (SILENUS/FICUS), and resource management framework (SERVME) has been successfully deployed and tested in multiple concurrent engineering and large-scale distributed applications [31, 32, 34–37, 46–51].

The presented description of the exertion-oriented methodology can be finalized as follows:

1. Thinking more explicitly about programming languages (DSL languages for humans as VML and EOL) instead of software languages (languages for computers) may be our best tool for dealing with real world complexity.
2. Understanding the principles that run across process expressions and appreciating which language features and related computing platforms are best suited for which type of process, bring these process expressions to useful life, e.g., seamless federations of tools, applications, and utilities in concurrent engineering processes.
3. No matter how complex and polished the individual process operations are, it is often the quality of the operating system that determines the power of the computing system. Note that the SOS is the service-oriented operating systems for exertion-oriented mogramming or generic middleware for SGrid and iGrid.
4. It provides unified programming and modeling environment. Procedural service-oriented programming enables bottom up problem solving and VOM enables top-down problem solving [39, 40] depicted in Fig. 4.9.

The presented description suggests that mixing both a process expression and implementation components (service providers) within a single computing platform and with the same programming language for both introduces inefficiencies and complexity of large transdisciplinary computing systems beyond human

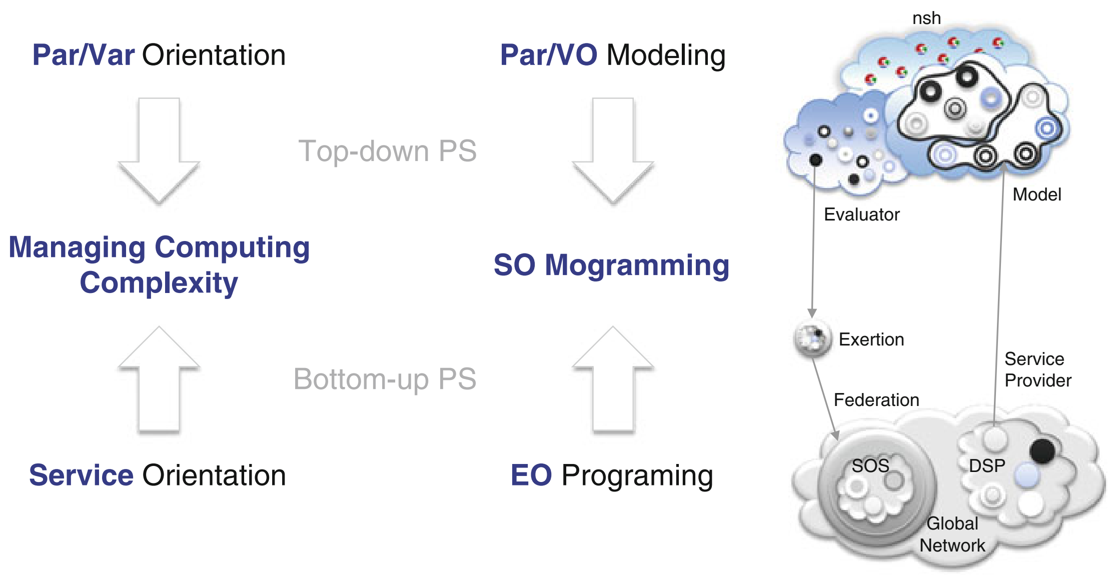

**Fig. 4.9** Managing transdisciplinary complexity with convergence of *top-down* service-oriented modeling and *bottom-up* service-oriented programming (*right* exertions in models as exertion evaluators and models as service providers in exertions)

comprehension. The proposed solution is to use the DCM and MCM architectures for implementation of transdisciplinary systems with the service-object oriented platform for coherent management of various heterogeneous component computing platforms.

Complex adaptive designs involve hundreds of low-level designs and simulations with thousands of programs written already in software languages (languages to create executable codes that are dislocated around the globe and have to be integrated into meta-applications written in DSL expressing problem to be solved by humans for human beings.

DSLs are for humans, intended to express specific complex problems, related processes, and corresponding solutions. In SORCER two basic programming languages for transdisciplinary computing are EOL and VML. These languages are interpreted by the SOS shell as service-oriented commands. The concept of the evaluator/getter/setter triplet in modeling provides the uniform service-orientation for all computing and meta-computing needs with various engineering applications, tools, and utilities.

The SORCER platform with three layers of converged programming: exertion-oriented (for service collaborations), and VOM (multidisciplinary var-oriented models with multi-fidelity compositions) has been successfully deployed and tested for the SO space exploration and parametric and optimization mogramming in recent application at AFRL/WPAFB [46, 47, 49, 51–53].

**Acknowledgments** This work was partially supported by Air Force Research Lab, Aerospace Systems Directorate, Multidisciplinary Science and Technology Center, the contract number F33615-03-D-3307, Algorithms for Federated High Fidelity Engineering Design Optimization, Polish-Japanese Institute of IT (the contract number POIG.01.04.00-14-062/12 the National Center for Research and Development), and the National Natural Science Foundation of China (Project No. 51175033).

## References

1. Dwivedi SN, Sobolewski M (1990) Concurrent engineering—an introduction. In: Proceedings of the fifth international conference on CAD/CAM robotics and factories of the future '90. Concurrent engineering, vol 1. Springer, New York, pp 3–16
2. Markov AA (1971) Theory of algorithms (trans by Schorr-Kon JJ). Keter Press, Israel
3. Kleppe A (2009) Software language engineering: creating domain-specific languages using metamodels. Pearson Education, Boston
4. N. N (1988) The role of concurrent engineering in weapon systems acquisition. IDA Report R-338, Dec 1988
5. Sobolewski M (1990) In: Percept knowledge and concurrency. Proceedings of the second national symposium on concurrent engineering, West Virginia University, Morgantown, pp 111–137
6. Postel J, Sunshine C, Cohen D (1981) The ARPA internet protocol. Comput Netw 5:261–271
7. Lynch D, Rose MT (eds) (1992) Internet system handbook. Addison-Wesley, Reading
8. Postel J, Reynolds J (1987) Request for comments reference guide (RFC1000). Internet Engineering Task Force

9. Lee J (ed.) (1992) Time-sharing and interactive computing at MIT. IEEE Ann Hist Comput 14:1
10. Hafner K, Lyon M (1996) Where wizards stay up late. Simon and Schuster, New York
11. Foster I, Kesselman C, Tuecke S (2001) The anatomy of the grid: enabling scalable virtual organizations. Int J Supercomputer Appl 15(3):200–222
12. Linthicum DS (2009) Cloud computing and SOA convergence in your enterprise: a step-by-step guide. Addison-Wesley Professional, Boston
13. Birrell AD, Nelson BJ (1983) Implementing remote procedure calls. XEROX CSL-83-7
14. Ruh WA, Herron T, Klinker P (1999) IIOP complete: understanding CORBA and middleware interoperability. Addison-Wesley, Boston
15. Pitt E, McNiff K (2001) java.rmi: the remote method invocation guide. Addison-Wesley Professional, Boston
16. Newmarch J (2006) Foundations of Jini 2 programming. Apress, Berkeley, ISBN-13: 978-1590597163
17. McGovern J, Tyagi S, Stevens ME, Mathew S (2003) Java web services architecture. Morgan Kaufmann, San Francisco
18. Sotomayor B, Childers L (2005) Globus® toolkit 4: programming java services. Morgan Kaufmann, San Francisco
19. Sobolewski M (2007) In: Federated method invocation with exertions. Proceedings of the IMCSIT conference. PTI Press, ISSN 1896-7094, pp 765–778
20. Sobolewski M (2008) SORCER: computing and metacomputing intergrid. In: 10th international conference on enterprise information systems, Barcelona, Spain. Available at: http://sorcer.cs.ttu.edu/publications/papers/2008/C3_344_Sobolewski.pdf
21. Sobolewski M (2009) Metacomputing with federated method invocation. In: Hussain MA (ed) Advances in computer science and IT. In-Tech, Rijeka, pp 337–363. http://www.intechopen.com/books/advances-in-computer-science-and-it/metacomputing-with-federated-method-invocation. Accessed 15 Feb 2014
22. Sobolewski M (2010) Exerted enterprise computing: from protocol-oriented networking to exertion-oriented networking. In: Meersman R et al (eds) OTM 2010 workshops, LNCS 6428, 2010. Springer, Berlin, pp 182–201
23. Sobolewski M (2011) Provisioning object-oriented service clouds for exertion-oriented programming. In: The 1st international conference on cloud computing and services science, CLOSER 2011, Noordwijkerhout, The Netherlands, 7–9 May 2011, SciTePress Digital Library. http://sorcersoft.org/publications/papers/2011/CLOSER_2011_KS.pdf. Accessed 15 Feb 2014
24. Sobolewski M (2014) Service oriented computing platform: an architectural case study. In: Ramanathan R, Raja K (eds) Handbook of research on architectural trends in service-driven computing, IGI Global, Hershey
25. Sobolewski M, Kolonay RM (2012) Unified mogramming with var-oriented modeling and exertion-oriented programming languages. Int J Commun Netw Syst Sci 5:579–592. Published Online Sep 2012 (http://www.SciRP.org/journal/ijcns)
26. Sobolewski M (2002) Federated P2P services in CE environments, advances in concurrent engineering. A.A. Balkema Publishers, Boca Raton, pp 13–22
27. Sobolewski M (2008) Exertion oriented programming. Int J Comput Sci Inf Syst 3(1):86–109
28. Sobolewski M (2010) Object-oriented metacomputing with exertions. In: Gunasekaran A, Sandhu M (eds) Handbook on business information systems. World Scientific, Singapore
29. Thain D, Tannenbaum T, Livny M (2003) Condor and the grid. In: Berman F, Hey AJG, Fox G (eds) Grid computing: making the global infrastructure a reality. Wiley, Chichester
30. Juric MB, Benny M, Sarang P (2006) Business process execution language for web services BPEL and BPEL4WS, 2nd edn. Packt Publishing, Birmingham
31. Röhl PJ, Kolonay RM, Irani RK, Sobolewski M, Kao K (2000) A federated intelligent product environment. In: AIAA-2000-4902, 8th AIAA/USAF/NASA/ISSMO symposium on multidisciplinary analysis and optimization, Long Beach

32. Kolonay RM, Sobolewski M, Tappeta R, Paradis M, Burton S (2002) Network-centric MAO environment. In: The society for modeling and simulation international, western multiconference, San Antonio
33. Sampath R, Kolonay RM, Kuhne CM (2002) 2D/3D CFD design optimization using the federated intelligent product environment (FIPER) technology. In: AIAA-2002-5479, 9th AIAA/ISSMO symposium on multidisciplinary analysis and optimization, Atlanta, GA
34. Kao KJ, Seeley CE, Yin S, Kolonay RM, Rus T, Paradis MJ (2003) Business-to-business virtual collaboration of aircraft engine combustor design. In: Proceedings of DETC'03 ASME 2003 design engineering technical conferences and computers and information in engineering conference, Chicago
35. Goel S, Talya S, Sobolewski M (2005) Preliminary design using distributed service-based computing. In: Sobolewski M, Ghodous P (eds) Next generation concurrent engineering. Proceeding of the 12th conference on concurrent engineering: research and applications. ISPE Inc./Omnipress, New York, pp 113–120
36. Goel S, Shashishekara S, Talya S, Sobolewski M (2007) Service-based P2P overlay network for collaborative problem solving. Decis Support Syst 43(2):547–568
37. Goel S, Talya S, Sobolewski M (2008) Mapping engineering design processes onto a service-grid: turbine design optimization. Int J Concurrent Eng Res Appl Concurrent Eng 16:139–147
38. Sobolewski M (2008) Federated collaborations with exertions. In: 17th IEEE international workshop on enabling technologies: infrastructures for collaborative enterprises (WETICE), pp 127–132
39. Sobolewski M, Kolonay R (2013) Service-oriented programming for design space exploration. In: Stjepandić J, Rock G, Bil C (eds) Concurrent engineering approaches for sustainable product development in a multi-disciplinary environment, proceedings of the 19th ISPE international conference on concurrent engineering, Springer-Verlag, London, pp 995–1007
40. Sobolewski M, Burton S, Kolonay R (2013) Parametric mogramming with var-oriented modeling and exertion-oriented programming languages. In: Bil C et al (eds) Proceedings of the 20th ISPE international conference on concurrent engineering, IOS Press, pp 381–390, http://ebooks.iospress.nl/publication/34826. Accessed on 9 March 2014
41. Sobolewski M, Kolonay R (2006) Federated grid computing with interactive service-oriented programming. Int J Concurrent Eng Res Appl 14(1):55–66
42. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. J Mol Biol 215:403–410
43. Khurana V, Berger M, Sobolewski M (2005) A federated grid environment with replication services. Next Gener Concurrent Eng. In: Sobolewski M, Ghodus P (eds) Next generation concurrent engineering: smart and concurrent integration of product data, services, and control strategies. ISPE/Omnipress, pp 93–103
44. Sobolewski M, Cha J (eds) (2004) Concurrent engineering: the worldwide engineering grid. Tsinghua Press and Springer-Verlag, London
45. Sobolewski M, Ghodous P (eds) (2005) Next generation concurrent engineering: smart and concurrent integration of product data, services, and control strategies. ISPE/Omnipress
46. Burton SA, Alyanak EJ, Kolonay RM (2012) Efficient supersonic air vehicle analysis and optimization implementation using SORCER. In: 12th AIAA aviation technology, integration, and operations (ATIO) conference and 14th AIAA/ISSM AIAA 2012-5520, Indianapolis, Indiana (AIAA 2012-5520), 17–19 Sep 2012
47. Kolonay RM (2013) Physics-based distributed collaborative design for aerospace vehicle development and technology assessment. In: Bil C et al (eds) Proceedings of the 20th ISPE international conference on concurrent engineering. IOS Press, pp 198–215. http://ebooks.iospress.nl/publication/34808. Accessed 15 March 2014
48. Burton SA, Tappeta R, Kolonay RM, Padmanabhan D (2002) Turbine blade reliability-based optimization using variable-complexity method. In: 43rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics, and materials conference, Denver, Colorado. AIAA 2002-1710

49. Kolonay RM, Thompson ED, Camberos JA, Eastep F (2007) Active control of transpiration boundary conditions for drag minimization with an euler CFD solver. In: AIAA-2007-1891, 48th AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics, and materials conference, Honolulu, Hawaii
50. Xu W, Cha J, Sobolewski M (2008) A service-oriented collaborative design platform for concurrent engineering. Adv Mater Res 44–46(2008):717–724
51. Kolonay RM, Sobolewski M (2011) Service ORiented Computing EnviRonment (SORCER) for large scale, distributed, dynamic fidelity aeroelastic analysis & optimization, international forum on aeroelasticity and structural dynamics. In: IFASD 2011, Paris, 26–30 June 2011
52. Kolonay RM (2014) A physics-based distributed collaborative design process for military aerospace vehicle development and technology assessment. Int J Agile Syst Manage 7(3/4): 242–260
53. Sobolewski M (2014) Unifying front-end and back-end federated services for integrated product development. In: Cha J et al (eds) Moving integrated product development to service clouds in global economy. Proceedings of the 21st ISPE Inc. international conference on concurrent engineering, IOS Press, Amsterdam, pp 3–16