Transdisciplinary Lifecycle Analysis of Systems
R. Curran et al. (Eds.)
© 2015 The authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License.
doi:10.3233/978-1-61499-544-9-541

Service-oriented Life Cycles for Developing Transdisciplinary Engineering Systems

Michael Sobolewski [a,b,1] and Raymond Kolonay [a]
[a] *Air Force Research Laboratory, WPAFB, Ohio 45433*
[b] *Polish Japanese Institute of IT, 02-008 Warsaw, Poland*

Abstract. A transdisciplinary computational model requires extensive computational resources to study the behavior of complex engineering systems by computer simulation. The large system under study that consists of hundreds or thousands of variables is often a complex engineering design system for which simple, intuitive analytical solutions are not readily available. In this paper the basic concepts of *mogramming* (modeling and programming, or both) for N^3 (three-dimensional design structure matrix) diagramming in a Service-ORriented Computing enviRonment (SORCER) are presented. On the one hand, mogramming with service variables allows for computational fidelity with multiple services, evaluations, and sources of data. On the other hand, any combination of local and remote services in the system can be described as a collaborative service federation of engineering applications, tools, and utilities. A service-oriented lifecycle for all phases of mogram-based systems development reflecting N^3 diagraming is presented. In particular all basic phases from inception through analysis, design, construction, transition, and maintenance are outlined in a service-oriented framework for deploying transdisciplinary engineering design systems.

Keywords. MADO, SDLC, service-orientation, N^2 and N^3 diagrams, exertion-oriented programming, mogramming, transdisciplinary systems, SORCER

Introduction

Multidisciplinary Analysis and Design Optimization (MADO) is a domain of research that studies the application of numerical analysis and optimization techniques for the design of engineering systems involving multiple disciplines. The formulation of MADO problems has become increasingly complex as the number of engineering disciplines and design variables included in typical studies has grown from a few dozen to thousands when applying high-fidelity physics-based modeling early in the design process [1]. The Service-oriented computing environment (SORCER) is a true service-oriented MADO environment that has been developed and applied to solve multidisciplinary design-optimization problems [2][3][4][6].

A *service* is the work performed in which a *service provider* (one that serves) exerts acquired abilities to execute a computation.

A *Service-oriented Architecture* (SOA) is a software architecture using loosely coupled service providers that introduces a *service registry*, the third component to client-server architecture. The registry allows finding service providers in the network.

[1] Corresponding Author, E-Mail: sobol@sorcersoft.org.

A *Service-object-oriented Architecture* (SOOA) is SOA with the communication based on *remote message passing* with the ability to pass data using *any wire protocol* that can be chosen by a *remote object* (provider) to satisfy efficient communication with its requestors. In SOOA a proxy object used by the requestor *is created, registered, and owned by the provider*.

Service-oriented programming (SOP) is a programming model organized around *service activities* rather than *service provider actions* and *service collaborations* rather than *service provider subroutines*. The approach is about *specifying service collaborations* (activities) *by the end user* rather than the programmer *developing subroutines* (actions) of a single service provider.

Historically, a program has been viewed as a *subroutine* (callable unit) that takes input data, processes it, and produces output data. The programming challenge was seen as how to write subroutines, *not how to manage data*, and *not how to manage collaborations of services*. Object-oriented programing shifted the focus from subroutines to data management - objects with encapsulated data managed by subroutines (methods). The SOP challenge is refocused on the *collaboration of local/remote autonomous services*. Service-oriented programming takes the view that what we really care about are the *service collaborations we want to manage* rather than the *subroutines with data required to manage them*.

The first step in SOP is to identify all the services the end user needs to use and how they relate to each other in a *compound service request*, an exercise often known as service modeling. Once services have been identified, corresponding service providers define the kind of data they contain - *data contexts* - and any subroutines that can process the data. Each distinct subroutine is known as a *service action* (operation) defined by the provider's *service type* used as a reference to service providers. Service providers communicate with well-defined *declarative service requests* called *context models* or *imperative service requests* called *service exertions*. We call a *context model* and *service exertion* respectively as a *model* and *exertion* for short, unless otherwise stated. Compound requests are called *service mograms* that are aggregations of both models and exertions- *service models* - expressed in a relevant *service-oriented language*. Mograms express work to be done by collaborating service providers, so they are *front-end* (abstract) *services* with respect to actualized service collaborations as their *back-end* (concrete) *services*. The MADO engineers (end users) usually create front-end services collaborations while software engineers develop service providers – actions of individual service providers.

The N^2 (N-squared) diagram or design structure matrix [5] represents the functional or physical interfaces between system elements depicted as diagonal nodes with connectors showing data flow between nodes. It is used to systematically define and analyze functional and physical interfaces of the system. It is an engineering tool for creating front-end MADO services or applications, for example in combination with essential design factor matrix [6] or with the full-scale UML-based flavor using SysML [7] tools. The analysis process and data flow represented as the N^2 diagram for the design of the next generation efficient supersonic air vehicle (ESAV) is discussed in [8][9].

SORCER is based on SOOA using service signatures dependent on service types (provider-requestor contracts) that play the role of references to service providers and allow binding to local or remote services (tools, applications, and utilities) at runtime [2][3]. In this service-oriented (SO) representation, systems, subsystems and components are implemented as scalable, dynamic, and transdisciplinary collaborations of local/remote services. Both system and subsystem components represented by N^2 diagrams are expressed in SORCER as mograms [10][11]. Five types of context models and tree types of exertions are distinguished in SORCER. With high expressive power of mograms composed of models and exertions, the N^2 diagram can be expanded recursively in the third direction with nested mograms as subsystems being again N^2 diagrams (multiple layers of interconnected N^2 diagrams). Also, each node in the N^2 diagram can be defined with multiple fidelities that expend the N^2 diagram in the third dimension (a single node substituted with multiple nodes as a multi fidelity mograms). We call hierarchically organized mogram-based diagrams with multi-fidelity components and flow of control as N^3 (N-cubed) diagrams. Both, N^2 and N^3 diagrams are discussed in Section 2.

In most service systems the focus is on back-end aggregation of services into a single provider, thus having more services performed by the same provider or by the same provider node, e.g., an application server. In either case these new services are still elementary services to the end user. This type of back-end aggregation, done by software developers is called *service assembly* in contrast to the MADO aggregation corresponding to the N^3 diagram created by the end user. In contrast, the front-end aggregation is a *service composition* and requires service-oriented languages to express declaratively and imperatively compositions of hierarchically organized services with multiple fidelities. Two service-oriented languages, declarative Context Modeling Language (CML) and imperative Exertion-Oriented Language (EOL) are discussed in Section 3.

Two ways of defining service composition coexist within SORCER: declarative context models and imperative exertions with SO flow of control. A context model is a collection of interrelated service variables - functional compositions - called service entries. Imperative service compositions – object compositions (composite design pattern [13]) with flow of control - are called *exertions*. Both models and exertions use service signatures to bind at runtime to corresponding service providers. A dynamic collection of service providers requested for the actualization of model or exertion is called a *service federation*. Note that *service collaboration* is an activity while a *service federation* is just a collection of service providers needed for the collaboration. Values of dependent variables in context models can be evaluated by exertions and context models can be used as service components of exertions. Therefore in SORCER, an MADO system represented by the N^3 diagram is a hierarchically organized aggregation of models and exertions with multiple fidelities.

The remainder of this paper is organized as follows: Section 1 describes briefly problem solving with N^2 and N^3 diagramming; Section 2 describes SO mogramming for N^3 diagramming; Section 3 describes life cycles for developing transdisciplinary MADO systems; finally Section 4 concludes with final remarks and comments.

1. Problem Solving with N-squared (N^2) and N-cubed (N^3) diagramming

Sometimes it is just hard to get started with service-oriented MADO. Faced with a long problem or project description it's not clear what is the required order of activities and related actions to perform. Project descriptions usually just include an overview of the project, because there are actually many ways to solve the problem or achieve a required purpose.

How do we actually approach the problem? One way to think about a problem is to consider it as interactions between uniform services within a system. Two methods of this form of interpretation are the top-down approach and the bottom-up approach. The top-down approach is considered the “compound service” approach, because the general idea of the system is first formulated declaratively and without getting down to the lowest level entities related to implementation of individual services (actions) and data used. The compound service, or activity, is then broken down into slightly smaller services. Those services are then split again until we reach the very bottom level of elementary services (actions). The bottom-up approach considers the lowest level entities first and their interaction with one another which build subsystems usually representing imperative processes. These subsystems will interact with each to form greater subsystems and slowly build our way up to the highest system.

Top-down and bottom-up describes two different methods of thinking: working at the top is considered strategic and declarative, while working at the bottom is tactical and imperative. How a given situation is actually perceived and processed will vary with the person, experience, and runtime environment chosen. However, the approach is to do whatever is best for managing complexity of the solution by a combination of both declarative and imperative thinking.

In declarative programming a process is expressed by a functional composition while in imperative programming is expressed by an algorithm. An algorithm is a procedure for solving a problem in the form of a self-contained step-by-step set of services (operations) to be performed with explicit control flow defined. The emphasis on explicit control flow distinguishes an imperative programming language from a declarative programming language.

The N^2 diagram design structure represents the functional or physical interfaces between system elements depicted as diagonal services with connectors showing data flow between services. In Fig.1 nodes of the N^2 diagram represent services e_1, m_1, m_3, and e_6 and the diagram as a whole defines the functional composition:

$$e_6(e_1, m_1(e_1), m_3(e_1))$$

where the parameters of strict function e_6 are evaluated sequentially in the order specified.

In general, N^2 diagramming is a graphical representation of a functional composition. The composition is defined in a declarative modeling language with no explicit flow of control for branching and looping. An expanded N^2 diagram with multi-fidelity of diagonal services that can be hierarchically organized with component diagrams along with flow of control is called an N^3 (N-cubed) diagram. An example of an N^3 diagram that expands the N^2 diagram from Fig. 1 is depicted in Fig. 2. It represents the following functional composition:

$$[g: e_{1,2;}\ e_{6,3}]e_{6,*}(e_{1,2}(m_{x,1}), m_1(e_{1,1}), m_{3,1}(m_{x,1}, e_{z,2,} e_{1,1}))$$

where $[g: e_{1,2;}\ e_{6,3}]$ denotes a guard for e_6 defining the loop under condition g: if g is true then $e_{1,2}$ else $e_{6,3}$. A current fidelity $e_{6,*}$ of e_6 is determined by this self-aware service at runtime. The third dimension here is represented by multiplicity of service nodes e_1, m_3, and e_6 with fidelities: $e_{1,1}$ and $e_{1,2}$ for e_1; $m_{3,1}$ and $m_{3,2}$ for m_3; and $e_{6,1}$, $e_{6,2}$, and $e_{6,3}$ for e_6. Additionally, each service node can be hierarchically nested with its own N^3 diagrams, for example $m_{3,1}$ depends on two diagrams $N3_{2,1}$ and $N3_{2,2}$ and $e_{1,1}$ on $N3_1$.

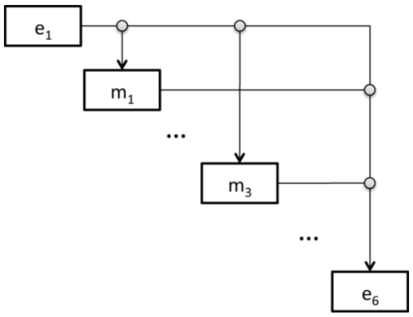

Figure 1. N^2 diagram for composition:
$e_6(e_1, m_1(e_1), m_3(e_1))$.

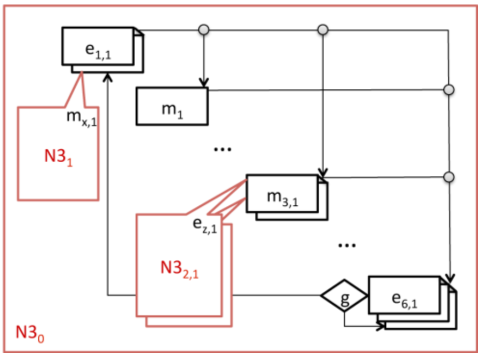

Figure 2. N^3 diagram for composition:
$[g: e_{1,2;}\ e_{6,3}]e_{6,*}(e_{1,2}(m_{x,1}), m_1(e_{1,1}), m_{3,1}(m_{x,1}, e_{z,2,}\ e_{1,1}))$.

The matrix crossing points (small circles) represent connectors that match outputs to relevant inputs between heterogeneous and autonomous services.

In SORCER a service-oriented process represented by an N^3 diagram can be defined declaratively with a Context Modeling Language (CML) as a *model* or algorithmically as an *exertion* with an Exertion-Oriented Language (EOL), or with both languages at the same time as a *mogram* – see the following Section for details. Within EOL a control flow exertion (conditional exertion) is a statement whose execution results in a choice being made as to which of two or more execution paths should be followed. Multi-fidelity diagonal services are represented by instances of multi-fidelity service of the context model type.

2. Service-oriented mogramming

A *service mogram* is a service model that is executed by a *dynamic federation* of services. In other words a mogram exerts the collaborating service providers in a *service federation created at runtime*. Mograms are specified in the Service Modeling Language (SML) that consists of two parts: Context Modeling Language (CML) and Exertion-Oriented Language (EOL). The former is used to specify *data models* (*data contexts*) for exertions and collections of interrelated functional compositions - *context models*. While CML is used for declarative service-oriented programming, EOL is focused on object-oriented composites of services - *exertions*.

A model is a declarative representation of something, especially a system, phenomenon, or service that accounts for its properties and is used to study its characteristics expressed in terms of service variables associated with functional compositions.

In every computing process *variables* represent data elements and the number of variables increases with the increased complexity of the problems being solved. The value of a *service variable* is not necessarily part of an equation or formula as in mathematics - its value is a result of service execution or the service itself. Handling large sets of interconnected variables for transdisciplinary computing requires adequate programming methodologies.

In SORCER interrelated *service variables* of a model are called entries. An *entry* used in a model refers by name (*path*) to one of the pieces of data - *value*. A value can be explicit or calculated by a subroutine. A *parameter* is a special kind of entry, named in a subroutine by its path (semantic name) and returning the value of the entry. These values, called *arguments*, that are used in subroutines are defined by input entries of the model. Most parameters are *functionals* – functions that take functions as their arguments. A selected subset of output entries defines a studied *response* of the model. Just as in standard mathematical usage, the *argument* is the actual input passed to a subroutine, whereas the *parameter* is the variable inside the implementation of the subroutine. Depending on the type of subroutine (*evaluation*, *invocation*, *service*, or *composite evaluation*) we distinguish four types of basic context models (EntModel, ParModel, SrvModel, VarModel) with ServiceContext as the data model for exertions, as depicted in the right part of in Fig. 3. Multi-fidelity diagonal services are represented by instances of multi-fidelity service of the context model type MultiFidelityService.

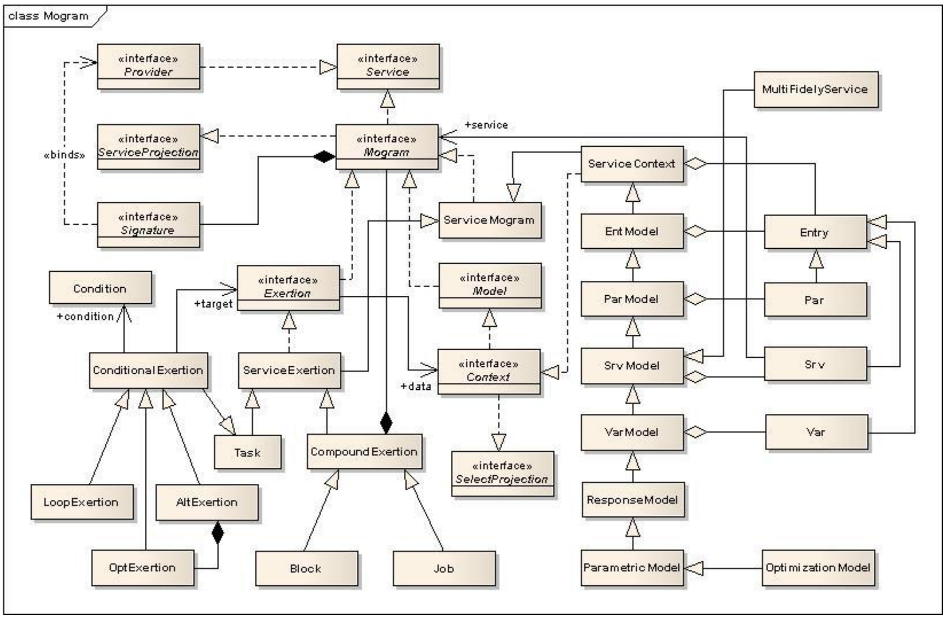

Figure 3. The UML diagram of SORCER top-level interfaces and classes.

A sketch of a context model is expressed in CML as follows:

Service e_1 = exertion(sig("doAnalysis", Optimization.class), context(…));
…
Service e_m = exertion (…);
Service m_1 = model(ent(…);
…
Service m_n = model(ent(…);
Condition g = condition(…);

Service mo = model(
    // order of entries does not matter
    ent("e_1", fi("$e_{1,1}$", e_1)),

```
        ent("m1", m1, args("e1"))),
        …
        ent("m3", fi("m3,1", m3, args("N32,1", "N32,2", "e1"))),
        …
        outEnt("e6", loop(g, e6, args("m3", "m1", "e1"))),
        response("e6"));

    Model out = exert(mo);                  // evaluate the model mo
    Context cxt = result(out);              // get the evaluation result
    cxt = response(mo);                     // get declared response e6
    Object obj = response(mo, "e1");        // get declared response e1
```

The first part of the mogram declares component services used in the model mo. All entries in the model define subroutines with arguments defined by other entries in the same model. At the end of mogram the basic CML operators are illustrated for model evaluation and obtaining results.

Exertions are structured by the composite design pattern [13] with elementary exertions called *tasks* and *compound exertions,* exertion of Blok and Job types, as seen in the left part in Fig. 3. A block represents concatenation of exertions with block-structured programming combined with flow of control exertions. Jobs represent object-oriented composites (workflows with pipes for data flow). Therefore, a mogram is either a model (declarative SO program) or an exertion (imperative SO program), or a hierarchical hybrid of both as defined by the UML sketch in Fig. 3.

Netlets are expressions in SML that are interpreted as SML scripts (text files) with the SORCER network shell (nsh). Technically netlets are both Groovy scripts and Java sources, therefore can be interpreted with a Groovy shell or compiled with a Java compiler. The former gives the agility of running MADO analyses and optimization and performing modifications to the netlets with no need for an IDE. The latter, with a Java IDE, allows for efficient development (compilation and debugging). Therefore, with their dual nature of netlets they can be developed much easier with Java IDEs and frequently updated as executable text files. In Java sources netlets can be used directly as services to provide implementation of service providers that can publish standard service types implemented by mograms.

A sketch of an exertion-oriented program is expressed in EOL as follows:

```
    Service e1 = exertion(sig(…), context(result("out/par"), …);
    …
    Service em = exertion (…);
    Service m1 = model(ent(…);
    …
    Service mn = model(ent(…);
    Condition g = condition(…);

    Service xrt = loop(g, block(
        // services are ordered for execution
        fi("e1,1", e1),
        m1,
        fi("m3,1", m3),
        e6,
        context(…, result("opti/value"));

    Exertion out = exert(xrt);                  // execute xrt
    Context cxt = context(out);                 // get result
    cxt = value(exertion);                      // get declared value at opti/value
    Object obj = value(xrt, "result/value");    // get value at the path result/value
```

The first part of the above exertion-oriented program declares component mograms used in the main exertion xrt. The execution of the exertion xrt is defined by the concatenation of component mograms (service-oriented statements) to be executed with the semantics of block-structured programing. At the end of the above mogram the usage of the basic EOL operators is illustrated for executing exertions and obtaining results.

Note that both the main model mo and the main exertion xrt implement the same N^3 diagram $N3_0$ in Fig. 2. It demonstrates that the main mograms representing N^3 diagrams can be implemented either way, with declarative (CML) or imperative language (EOL) for the top-level mograms.

3. Life cycles for developing service-oriented MADO systems

A systems development life cycle (SDLC) is composed of a number of distinct work phases that are used by engineers and system developers to plan for, design, build, test and deliver systems represented by N^3 diagrams. An N^3-based SDLC aims to produce high quality systems that meet or exceed customer expectations, based on requirements represented by hierarchically organized N^3 diagrams. A well-defined SDLC process enables the delivery of transdisciplinary systems, which move through each clearly defined phase of the generic template of planning, creating, testing, and deploying an information system.

In systems engineering, with the increasing level of service-orientation (everything as a service) and increasing number of legacy and new network services supplied by different development groups and organizations, also increases systems distribution and heterogeneity. To reliably manage the increasing level of distribution and heterogeneity, the SORCER environment has been expended to support N^3-based mogramming combined with its unique SDLC phases: inception, analysis, design, construction, transition, and maintenance.

1. Inception

a) Determine which processes better represent the problem being solved - top-down or bottom-up problem solving, or the hybrid approach
b) For top-down solutions use CML modeling, for bottom-up use EOL programming, for hybrid solutions use SO mogramming with CML/EOL or EOL/CML
c) Identify relevant existing services and those not available yet
d) Identify service UIs required for the end users

2. Analysis

a) Define N^3 diagrams representing the MADO process with N^3 nodes and N^3 components for the hierarchically organized MADO system identified in 1b

b) If multi-fidelities are required, define corresponding high fidelity alternatives for corresponding nodes in the N^3 diagrams defined in 2a
c) Define service signatures for all local/remote services used in the N^3 diagrams
d) For all service types used in signatures, define the Java interfaces that define the behavior of the service providers
e) Define entries in the data contexts and context models along with needed connectors to support seamless data flow across N^3 diagrams
f) Decide what codes to acquire/buy/develop in support of service providers that implement service types defined in 2d
g) Define service UIs required for the end users identified in 1d

3. Design

a) Design detailed service types (Java interfaces) for all N^3 analysis interfaces defined in 2d
b) Design all service providers in support of service types designed in 3a
c) Design component mograms, with API or SML/EOL, for N^3 diagrams defined in 2a
d) Design front-end netlets in SML/EOL N^3 diagrams for end-users defined in 2a
e) Design required service UIs for corresponding service providers as defined in 2g

4. Construction

a) Use SORCER project templates for developing and testing a service provider/requestor and service UIs
b) Implement all service types designed in 3a
c) Implement all service provider designed in3b
d) Implement component mograms designed in 3c
e) Implement service UIs designed in 3e
f) Implement netlets designed in 3d as standalone files executable with the nsh shell
g) Deploy SORCER operating system for development
h) Deploy all required services for development
i) Test services provider classes with local signatures
j) Test remote service providers with service types implemented in 4c
k) Test all service UIs implemented in 4e
l) Test all component mograms implemented in 4d
m) Test all netlets representing N^3 diagrams designed in 4f

5. Transition

a) Deploy SORCER operating system for production
b) Deploy all required services for production
c) Transition netlets to end users
d) Provide support for updating and executing netlets by the end users
e) Demonstrate all service UIs' functionality to the end users.
f) Demonstrate/run/modify netlets with the nsh shell
g) Capture MADO inputs, results with related netlets along with required codes and sources as a part of corporate design history
h) If updates are needed to N^3 diagrams go to 2

6. Maintenance

a) Maintain the repository of all codes and sources for developed services with the unique service IDs used in N^3 mograms for later reuse of persisted solution
b) Maintain continuous integration testing of all N^3 mograms/netlets/service UIs
c) If minor updates of mograms/netlets/service UIs are required go to 2 or 3
d) If essential updates of mograms/netlets/service UIs are required go to 1

4. Conclusions

Using presented higher-level SO abstractions for mogramming allows reducing the complexity of creating and using transdisciplinary MADO systems. These network-centric collaborative systems are created at runtime by teams of engineers working together and using many shared services that can be provisioned autonomically on demand.

Domain specific SO languages are for humans, unlike software languages that are for computers, intended to express domain specific complex MADO processes and related solutions. Two programming languages (CML and EOL) for SO computing are introduced in this paper in the context of N^2 and N^3 diagramming. The SORCER network shell (nsh) manages the corresponding service federations at runtime for the N^2 and N^3 diagrams expressed by mograms.

SDLC phases of mogram-based systems development are presented with the semantics of N^3 diagramming. These continue to evolve with the focus on the development of interactive tools that facilitate easy creation and testing of graphical N^3diagrams. In particular, all the basic phases from inception through analysis, design, transition and maintenance are tested and continuously improved for developing aerospace transdisciplinary engineering systems.

The SORCER mogramming environment supports the two-way convergence of modeling and programming. It allows for flexible problem solving solutions as presented in Section 1 and 2. On one hand, EOL is uniformly converged with CML to express front-end service exertions. On the other hand, CML is uniformly converged with EOL to express a front-end declarative context models. Both front-end exertions and models can be used as service providers directly within SORCER.

The evolving SORCER platform (the GitHub open source project [14]) introduces front-end mogramming languages [11] and an API with a modular service-oriented operating system [2]. It adds two entirely new layers of abstraction to the practice of SO computing. The presented SO MADO approach has been verified and validated in research projects at the Multidisciplinary Science and Technology Center, AFRL/WPAFB [8][15][16].

Acknowledgement

This work was supported by Air Force Research Lab, Aerospace Systems Directorate, Multidisciplinary Science and Technology Center, the contract number FA8650-10-D-3037, Service-Oriented Optimization Environment for Distributed High Fidelity Engineering Design Optimization.

References

[1] R. M. Kolonay, A physics-based distributed collaborative design process for military aerospace vehicle development and technology assessment, *International Journal on Agile Systems and Management*, Vol. 7 (2014) Nos. 3/4, pp. 242 – 260.

[2] M. Sobolewski, Service oriented computing platform: an architectural case study. In: R. Ramanathan, K. Raja (eds.) *Handbook of research on architectural trends in service-driven computing*, IGI Global, Hershey, 2014, pp. 220-255.

[3] M. Sobolewski, Unifying Front-end and Back-end Federated Services for Integrated Product Development, In: *J. Cha et al. (eds.) Moving Integated Product Development to Service Clods in Global Economy*, IOS Press, Amsterdam, pp. 3-16, 2014, Retrieved May 25, 2015, http://ebooks.iospress.nl/publication/37838.

[4] M. Sobolewski, Technology Foundations. In: J. Stjepandić et al. (eds.) *Concurrent Engineering in the 21st Century*, Springer International Publishing Switzerland, pp. 67-99, 2015.

[5] T. R. Browning, Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions, *IEEE Transactions on Engineering Management*, Vol. 48 (2001), No. 3, pp. 292-306.

[6] L. Nan, W. Xu and J. Cha, A Hierarchical Method for Coupling Analysis of Design Services in Distributed Collaborative Design Environment, *International Journal on Agile Systems and Management*, Vol. 8, 2015, Nos. 3/4, in press.

[7] L. Delligatti, *SysML Distilled: A Brief Guide to the Systems Modeling Language*, Addison-Wesley Professional, Upper Saddle River, 2013.

[8] S. A. Burton, E. J. Alyanak, and R. M. Kolonay, Efficient Supersonic Air Vehicle Analysis and Optimization Implementation using SORCER, *12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSM AIAA* 2012-5520.

[9] M. Sobolewski, S. Burton, and R. Kolonay, Parametric Mogramming with Var-oriented Modeling and Exertion-Oriented Programming Languages. *Proceedings of the 20th ISPE International Conference on Concurrent Engineering*, C. Bil et al. (Eds.), ISBN: 978-1-61499-301-8 (print), 978-1-61499-302-5 (online), IOS Press, 2013, pp. 381-390. Retrieved May 25, 2015, http://ebooks.iospress.nl/publication/34826.

[10] A. Kleppe, *Software Language Engineering*, Addison-Wesley Professional, Upper Saddle River, 2009.

[11] M. Sobolewski, and R. Kolonay, Unified Mogramming with Var-Oriented Modeling and Exertion-Oriented Programming Languages, *Int. J. Communications, Network and System Sciences, 2012*, 5, 9. http://www.scirp.org/journal/PaperInformation.aspx?paperID=22393, Accessed: 25 May 2015.

[12] M. Sobolewski, Object-Oriented Service Clouds for Transdisciplinary Computing, in: I. Ivanov et al. (eds.), *Cloud Computing and Services Science*, DOI 10.1007/978-1-4614-2326-3_1, Springer Science + Business Media New York, 2012.

[13] T. Bevis, *Java Design Pattern Essentials*, Ability First Limited, Leigh-on-Sea, 2012.

[14] *SORCER Project*. http://sorcersoft.org/project/site/, Accessed: 25 May 2015.

[15] R. M. Kolonay, and M. Sobolewski, Service ORiented Computing EnviRonment (SORCER) for Large Scale, Distributed, Dynamic Fidelity Aeroelastic Analysis & Optimization, *International Forum on Aeroelasticity and Structural Dynamics, IFASD2011*, 26–30 June, 2011, Paris.

[16] R. M. Kolonay, E. D. Thompson, J. A. Camberos and F. Eastep, 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, 2007.