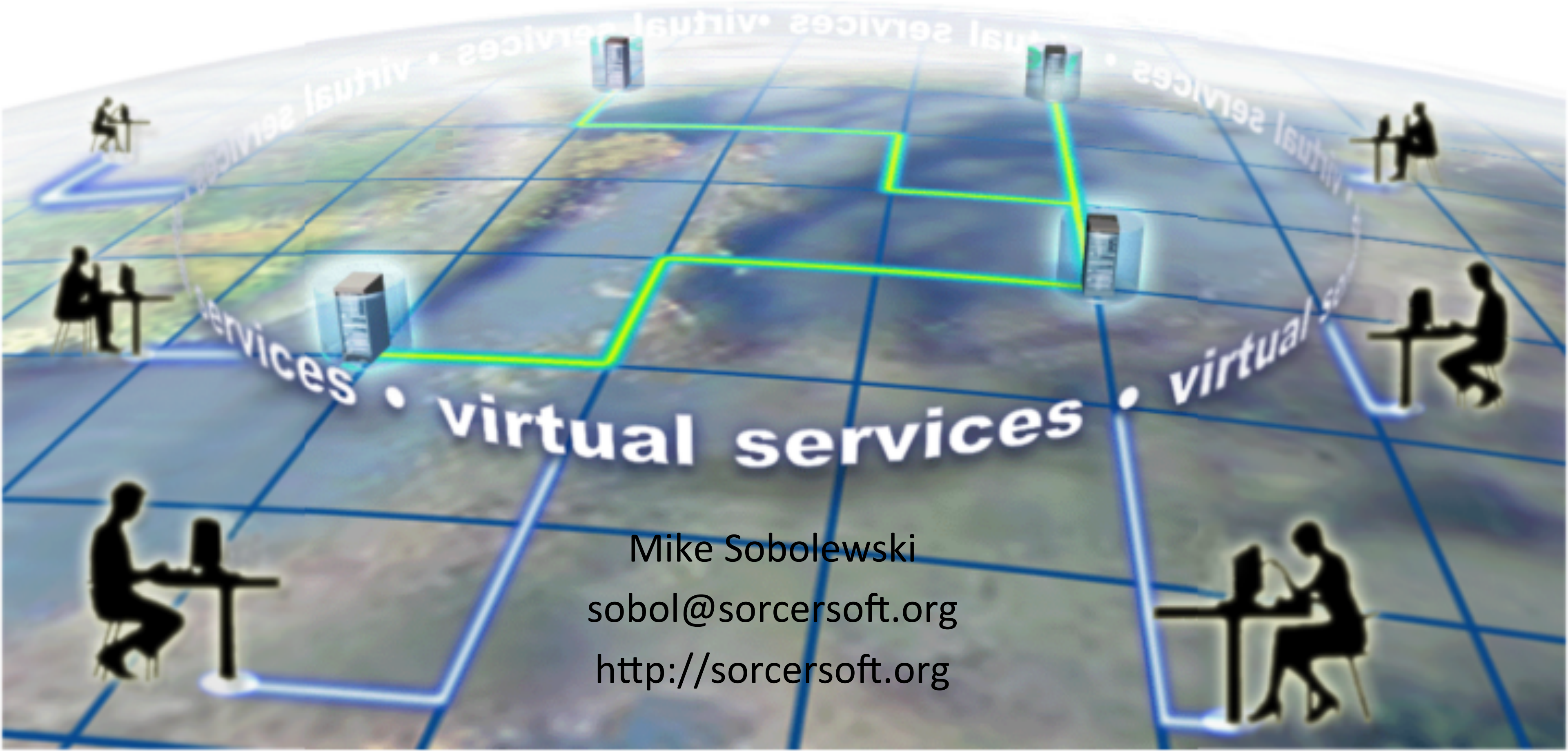
The Dual Nature of Service Orientation with Exertions

Service Computation 2012

July 26, Nice, France



Mike Sobolewski

sobol@sorcersoft.org

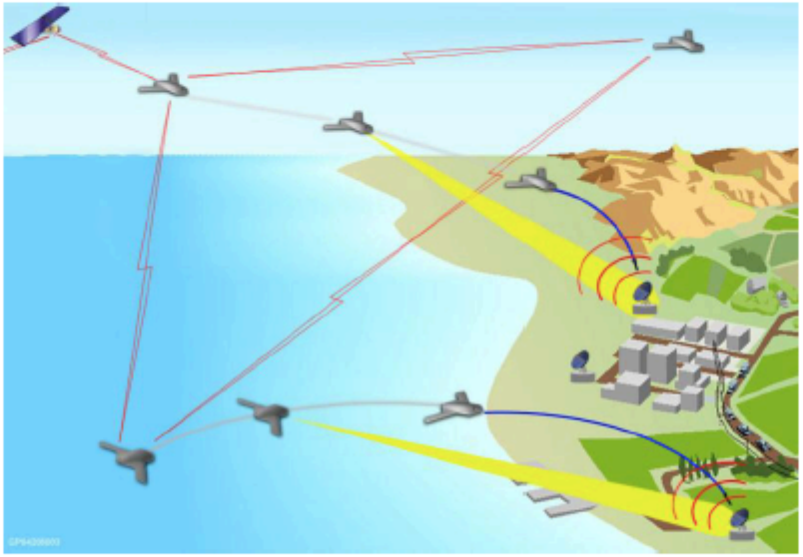
http://sorcersoft.org

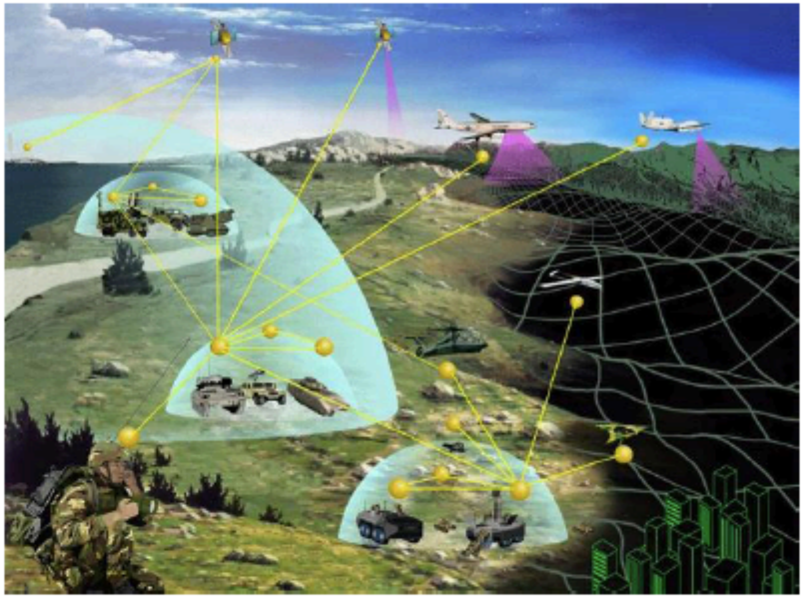
Agenda

- Why Do We Need Service-oriented Systems?
- Service vs. Service-oriented Systems
- Var-Oriented Modeling with VOL and VML
- Exertion-oriented Programming with EOL
- Mogramming Parametric/Optimization Problems
- SOS: SORCER Operating System
- Conclusions

Why Do We Need Service Systems?

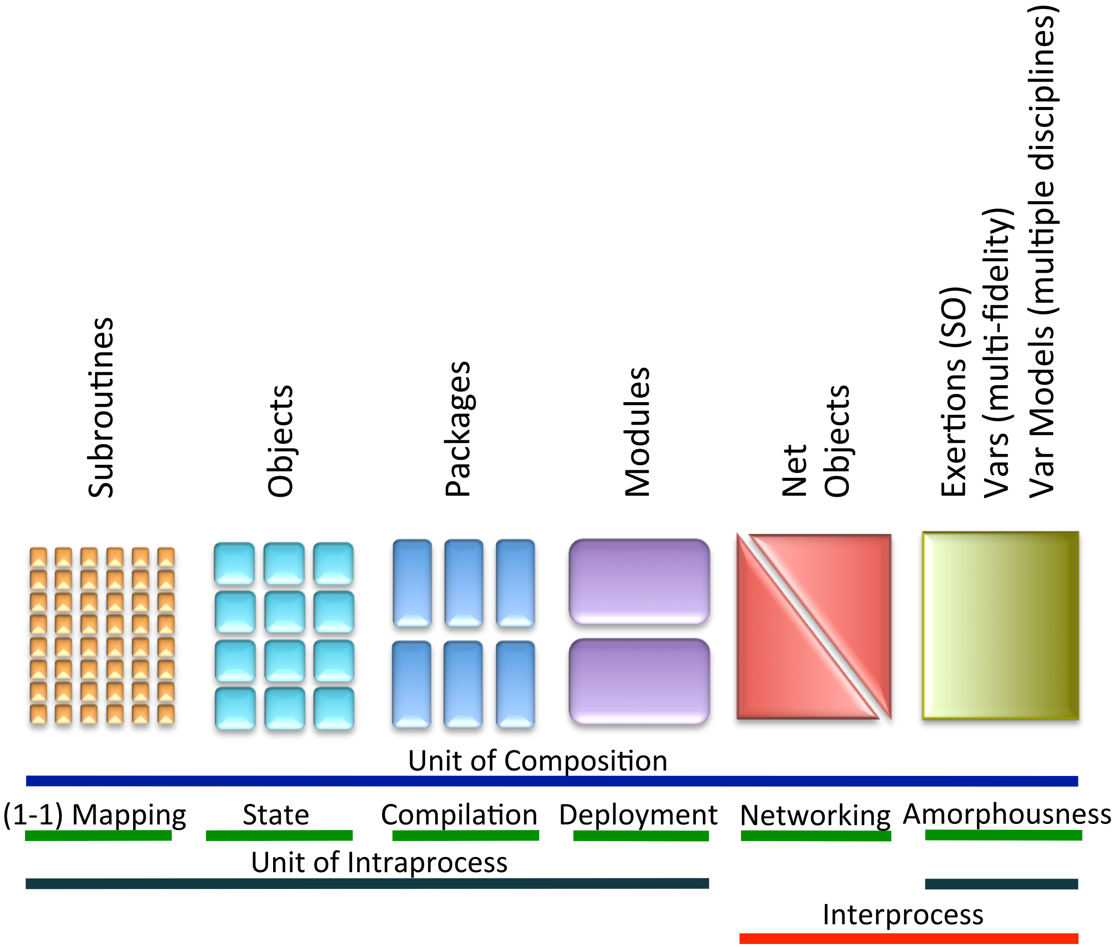
- What works with small units often does not scale well to larger sizes
- From automation to autonomy
- From remote to federated (dynamically integrated)
- From low to high fidelity
- From task oriented to goal oriented (collaboration)
- From deterministic to nondeterministic federations
- From established to continuously adaptable systems
- From code complexity to logic complexity due to adaptivity





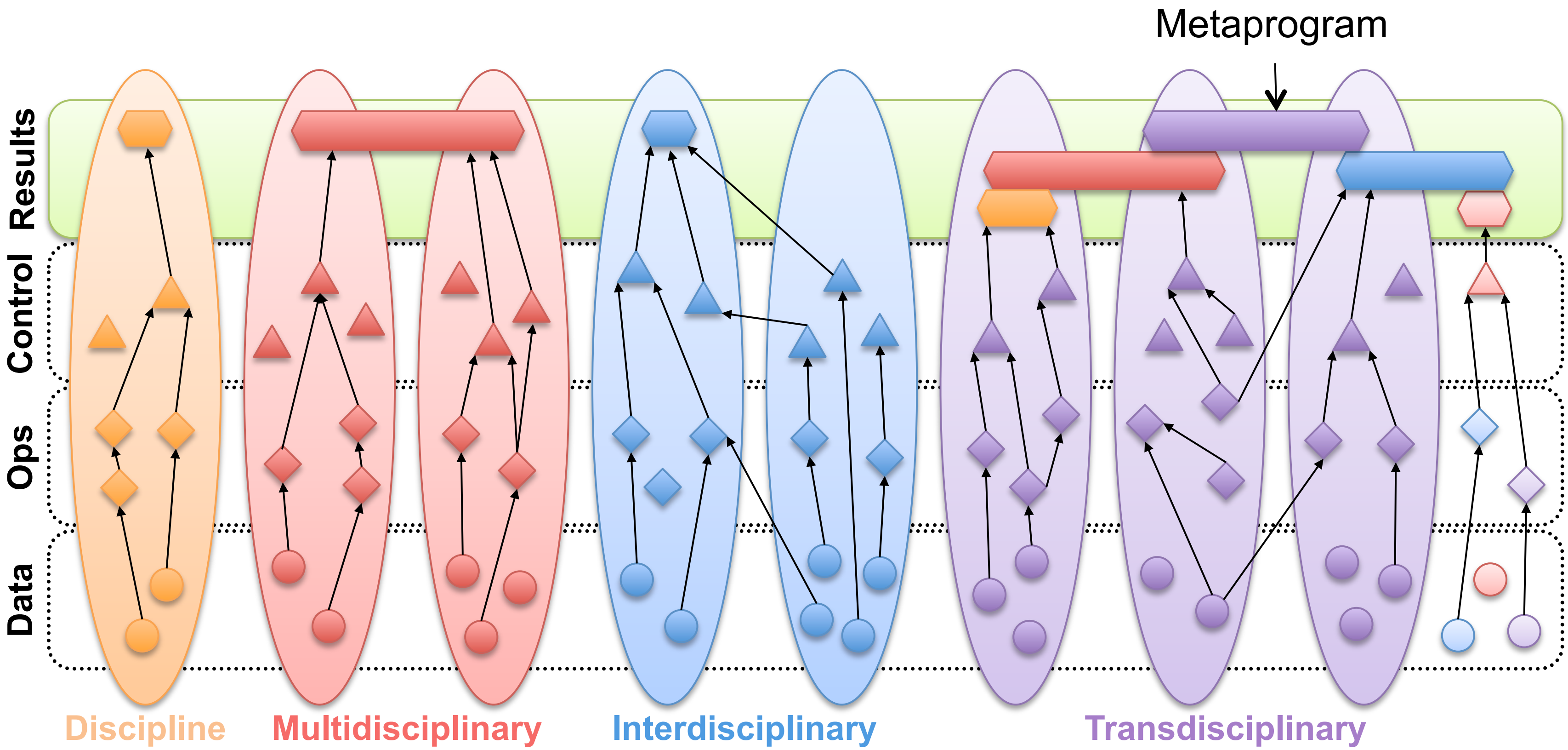
Service vs. Service-oriented Systems

- Requesting service (C/S) does not make service orientation
- Granularity of Units of Functionality is essential



f(args) -> receiver.selector(args)->server.message(args)->network.exert(federation)

Transdisciplinary Computing (CDIO)



Operations: apps, tools, utilities -> **programs**
Metaprogram: program of **programs**

Service Programming Terminology

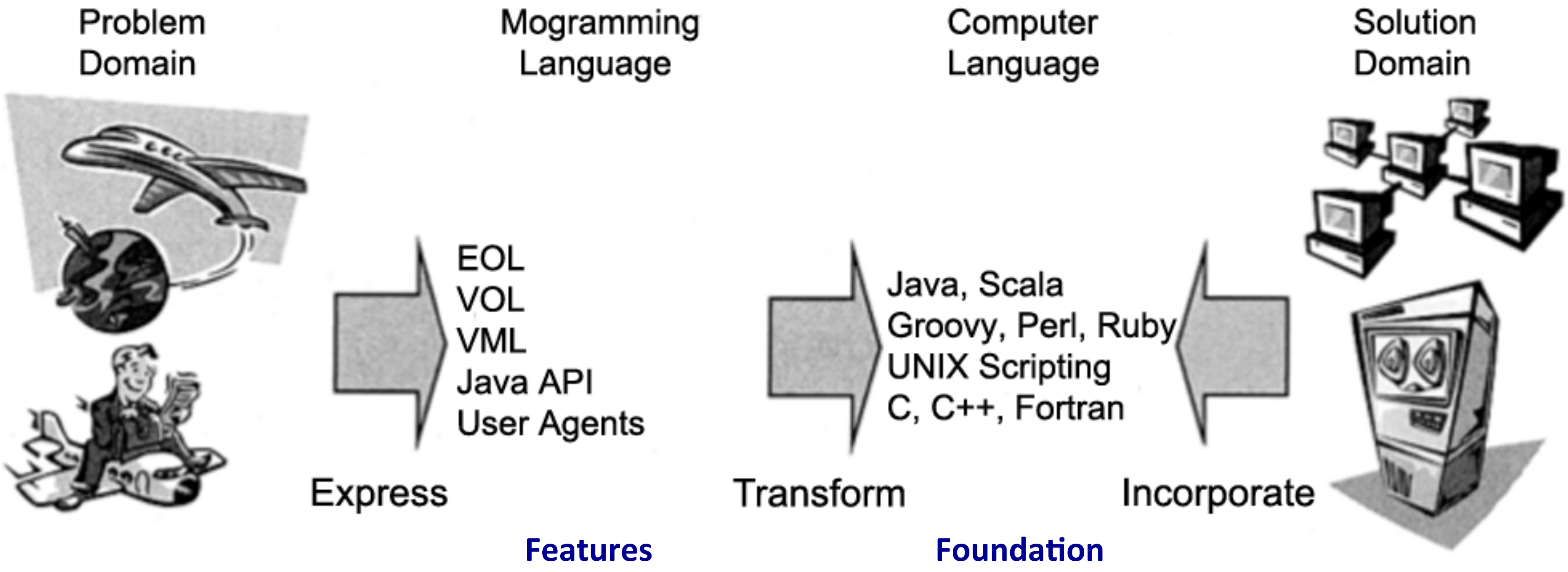
- A *service* is the work performed in which a service provider **exerts acquired abilities to execute a computation**.
- Service providers run computations
  - Executable codes
  - Local
    - Created
  - Distributed
    - Found
    - Deployed
    - Provisioned
  - Virtual
    - Dynamic federation
- Service requestors
  - Service (oriented) programs—expressions of service federations

Command vs. Service Platforms

- Processor
  - instructions
- Programming language (programs)
  - statements
  - messages
- OS
  - commands/scripts

- Network of providers
  - services
- Programming language(mograms)
  - exertions
  - vars/models
- OS
  - services/netlets

DS Languages vs. SW Languages
Language Eng. vs. SW Eng.



Problem Domain

Mogramming Language

Computer Language

Solution Domain

EOL
VOL
VML
Java API
User Agents

Java, Scala
Groovy, Perl, Ruby
UNIX Scripting
C, C++, Fortran

Express

Transform

Incorporate

Features

Foundation

Language engineering is the art of creating languages

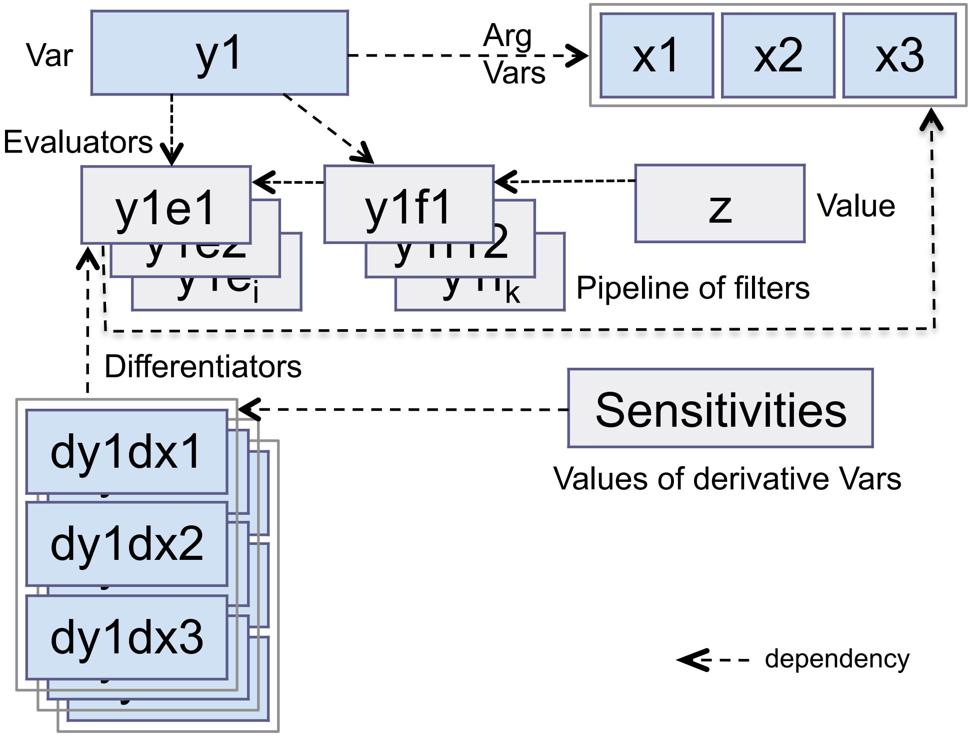
Types of Variables

- Variable (mathematics), a symbol that represents a quantity in a mathematical expression
- Variable (programming), a symbolic name associated with a value that may be changed
- Variable (OO programming), a set of object's attributes accessible via 'getters'
- Variable (SO programming), a triplet <value, evaluator, filter>
  - value: a valid quantity
  - evaluator: a service with dependent variables (composition)
  - filter: a selective getter

Service Variable Structure

(Value/Fidelity/Evaluation-VFE)

$z = y_1(x_1, x_2, x_3)$



<--- dependency

Evaluators can execute exertions

Evaluating var y with arg vars

```
Var x1 = var ("x1", 10.0); Var x2 = var("x2", 50.0),
Var x3 = var ("x3", 20.0); Var x4 = var ("x4", 80.0);

Var y = var("y",
    expr("(x1 * x2) - (x3 + x4)",
        args(x1, x2, x3, x4)));

assertEquals(value(y), 400.0);
```

Service Closures

- A var, context, exertion, evaluator, filter, and model with their referencing environments (substitutions) for their free variables are called **service closures**
- An **upvalue** is a free variable closed over with a closure
- **Service upvalues** are vars, context paths, signatures, fidelities, control strategies, subcomponents

Var Closure

- Var Closure – closing over x1, x2, x3, and x4

```
Var y = var("y",
 expr("(x1 * x2) - (x3 + x4)",
     args("x1", "x2", "x3", "x4")));

Object val = value(y,
     entry("x1", 10.0), entry("x2", 50.0),
     entry("x3", 20.0), entry("x4", 80.0));

assertEquals(val, 400.0);
```

- Var Closure – closing over x1, x2, and an evaluator

```
 Var y = var("y", evaluators(
     expr("e1", "x1 * x2", vars("x1", "x2")),
     expr("e2", "x1 * x2 + 0.1", vars("x1", "x2"))));

assertEquals(value(y,
     entry("x1", 10.0), entry("x2", 50.0), eFi("e1")), 500.0)
assertEquals(value(y,
     entry("x1", 10.0), entry("x2", 50.0), eFi("e2")), 500.1);
```

Var-Oriented Modeling (VOM)

- VOM is a modeling paradigm using vars in a specific way to define heterogeneous var-oriented models, in particular large-scale multidisciplinary models including response, parametric, and optimization models.
- The programming style of VOM is declarative; models describe the desired results of the output vars, without explicitly listing instructions or steps that need to be carried out to achieve the results.
- VOM focuses on how vars connect (compose) in the scope of the model, unlike imperative programming, which focuses on how evaluators calculate.
- VOM represents models as a series of interdependent var connections, with the evaluators/filters between the connections being of secondary importance.

“Hello Arithmetic” Model

- A *var-model* is an aggregation of related vars.
- A var-model defines the lexical scope for var unique names in the model.

```
VarModel vm = model("Hello Arithmetic",
        inputs(
                var("x1"), var("x2"),
                var("x3", 20.0), var("x4", 80.0)),
        outputs(
                var("f4", expression("x1 * x2",
                        args( "x1", "x2"))),
                var("f5", expression("x3 + x4",
                        args("x3", "x4"))),
                var("f3", expression("f4 - f5",
                        args("f4", "f5")))));

assertEquals(value(var(put(vm,
        entry("x1", 10.0), entry("x2", 50.0)), "f3"), 400.0);
```

“Hello Arithmetic” Closure

- Closing over x1 and x2

```
VarModel vm = varModel("Hello Arithmetic",
 independentVars(
      var("x1"), var("x2"),
      var("x3", 20.0), var("x4", 80.0)),
 dependentVars(
      var("t4", expression("x1 * x2",
            args(vars("x1", "x2")))),
      var("t5", expression("x3 + x4",
            args(vars("x3", "x4")))),
      var("j1", expression("t4 - t5",
            args(vars("t4", "t5"))))));

assertEquals(value(var(put(vm,
 entry("x1", 10.0), entry("x2", 50.0)), "j1")), 400.0);
```

Rosen-Suzuki Model

design vars: $x1, x2, x3, x4$

response vars: $f, g1, g2, g3,$

and

$f = x1^2-5.0*x1+x2^2-5.0*x2+2.0*x3^2-21.0*x3+x4^2+7.0*x4+50.0$

$g1 = x1^2+x1+x2^2-x2+x3^2+x3+x4^2-x4-8.0$

$g2 = x1^2-x1+2.0*x2^2+x3^2+2.0*x4^2-x4-10.0$

$g3 = 2.0*x1^2+2.0*x1+x2^2-x2+x3^2-x4-5.0$

The goal is then to minimize f subject to

$g1 <= 0$, $g2 <= 0$, and $g3 <= 0$.

Rosen-Suzuki Model in VML

```
int inputVarCount = 4;
int outputVarCount = 4;
OptimizationModel om = optimizationModel(”Rosen-Suzuki Model",
    inputVars(vars(loop(inputVarCount), "x", 20.0, -100.0, 100.0)),
    outputVars("f"),
    outputVars("g”,outputVarCount - 1),
    objectiveVars(var("fo", "f", Target.min )),
    constraintVars(var("g1c", "g1", Relation.lt, 0.0),
        var("g2c", "g2", Relation.lt, 0.0),
        var("g3c","g3", Relation.lt, 0.0)));

configureVarModel(om);
configureSensitivityModel(om);
```

A var-model can be a local object or remote service

Exertion-oriented Programming

- An ***exertion*** is the expression of a service structure that consists of a ***data context***, a ***control context,*** and ***component exertions*** to design hybrid (distributed/local) service collaborations.
- A control context comprises of a ***control strategy*** and multiple ***service signatures***, which define the service invocations on federated providers.
- The signature usually includes the ***service type***, ***operation*** within the *service type*, and expected ***quality of service***.
- An exertion's signatures identify the required providers.
- The control strategy for the SOS defines how and when the signature operations are applied to the data context in the federated collaboration.

Service Providers and Service Messages

- A service is the work in which a service provider exerts acquired abilities to perform something.
- provider(sig(…)):Object
  - Net Object
  - Object
  - Command
  - Evaluator
  - Filter
  - Var
  - VarModel
- A *service* message
  - provider(sig(...)). selector(sig(...))(Context):Context

red color indicates the SO operators and types

Service Signatures —> Service Providers

• sig(<selector>, <code>)	command sig
• sig(<selector>, Class | Object)	object sig
• sig(<selector>, <service type>)	net sig
• sig(Evaluation)	evaluator sig
• sig([<selector>,] Filter)	filter sig
• sig(Fidelity, Var)	var sig
• sig(<selector>, Modeling)	model sig

Return path can be specified: sig(…, result(<path> [, Direction])) : Signtaure

Direction: `IN, OUT, INOUT`

A signature can be tagged:

type(Singature, Type):Signature

with types: `SRV, PRE, POST, APD`

You don't understand anything until you ***learn it more than one way****.*

Exertion-Oriented Language (EOL)

- service(sig(...), context(...) {, exertion(...) } {, pipe (...) }\ [, strategy(...)] [, qos(...)])
- sig(<selector>, <service provider>)
  - sig("add", *Adder.class*)
- context({ (in | out | inout | entry | result | args | target) (<path> [, <value>]) })
- var(service | { <evaluator> {,<filter>] })
- service = task | job | srv
- cf-service = opt | alt | loop | break| seq | par | pull | push
- exertion = service | cf-service

sorcer.co.operator.*
sorcer.eo.operator.*
sorcer.vo.operator.*

Running Services and Getting Results

- **exert**(Exertion {, parameter }):Exertion
- **value**(Evaluation [, <component selector>] {, parameter }):Object
- close(Evaluation):Object
- asis(Evaluation):Object
- parameter = entry | in | out | inout (path, value
  [, fidelitySelector | fidelity(…)])
  | varInfo(…) | strategy(…) | fidelity(…)

- **get**(Evaluation [, <component selector>]):Object
- **put**(Evaluation [, <component selector>] {, parameter }): Evaluation
- context(Exertion [, <component selector>]):Context
- control(Exertion [, <component selector>]):ControlContext
- trace(Exertion):List<String>
- exceptions(Exertion):List<ExceptionTrace>
- Evaluation = Context, Exertion, Evaluator, Filter, Var, VarModel

Task: Elementary Service

$y = x1 * x2$

```
Task t = task(
    sig("multiply", Multiplier.class),
    context(
        in("arg/x1", 10.0d),
        in("arg/x2", 50.0d),
        result("result/y"));

assertEquals(value(t), 500.0);
```

- `Multiplier.class` **is a service type (Java interface)**
- **A task may have multiple service signatures (batch task)**

Setters and Getters

```
Task t = task(
    sig("multiply", Multiplier.class),
    context(
        input("arg/x1", 10.0d),
        input("arg/x2", 50.0d),
        result("result/y"));


put(t, entry("arg/x1", 1.0d), entry("arg/x2",5.0d));

double y = (Double)value(t);
t = exert(t);
y = (Double)get(t, "result/y");
y = (Double)get(context(exert(t)), "result/y");

print(exceptions(t));
print(trace(t));
```

“Hello Arithmetic” Batch Task

$y = (x1 * x2) – (x3 + x4)$

```
Task batch = task("batch",
    sig(expr("x1 * x2", vars("x1", "x2")),
        result("x5"))
    sig(expr("x3 + x4", vars("x3","x4")),
        result("x6")),
    sig(expr("x5 - x6", vars("x5", "x6")),
        result("result/y")),
    context(in("arg/x1", 10.0), in("arg/x2", 50.0),
        in("arg/x3", 20.0), in("arg/x4", 80.0)));

assertEquals(value(batch), 400.0);
```

Batch Task with Context Scoping

*y = (x1 * x2) – (x3 + x4) with selector/context scoping for arguments*

```
Task batch = task("batch",
    sig("multiply#op1", MultiplierImpl.class,
        result("op3/x1", Direction.IN)),
    sig("add#op2", AdderImpl.class,
        result("op3/x2", Direction.IN)),
    sig("subtract", SubtractorImpl.class,
        result("result/y", at("op3/x1", "op3/x2"))),
    context(in("op1/x1", 10.0), in("op1/x2", 50.0),
        in("op2/x1", 20.0), in("op2/x2", 80.0)));

assertEquals(value(batch, 400.0);
```

Job: Compound Service

f3(f4(x1, x2), f5(x3, x4)) as a service composition f1(f2(f4(x1, x2), f5(x1, x2)), f3(x4, x5))
with pipes from t4 and t5 to t3

```
Task f4 = task(“f4", sig("multiply", Multiplier.class),
   context("multiply", input("arg/x1", 10.0d),
      input("arg/x2", 50.0d), result("result/y")));

Task f5 = task(“f5", sig("add", Adder.class),
   context("add", input("arg/x3", 20.0d),
      input("arg/x4", 80.0d), result("result/y")));

Task f3 = task(”f3", sig("subtract", Subtractor.class),
   context("subtract", input("arg/x5”),
      input("arg/x6”), result("result/y")));

Job f1= job(“f1", job(“f2", f4, f5,
   strategy(Flow.PAR, Access.PULL)), f3,
   pipe(output(f4, "result/y"), input(f3, "arg/x5")),
   pipe(output(f5, "result/y"), input(f3, "arg/x6")));

assertEquals(value(f1), 400.0);
```

Local Service Composition

A service composition f1(f2(f4(x1, x2), f5(x1, x2)), f3(x4, x5))

```
Task f4 = task(“f4",
    sig("multiply", new Multiply(), double[].class),
    context("multiply", args(new double[] { 10.0, 50.0 }),
        result("result/y")));

Task f5 = task(”f5", sig(expression("x2 + x3",
        vars(var("x2", 20.0), var("x3", 80.0))),
        result("result/y"));

Task f3 = task(“f3", sig(var("x3",
        expression(“f3-e", "x1 - x2", vars("x1", "x2"))),
        result(path("result/y"));

Job f1= job(”f1", sig("execute", ServiceJobber.class),
    job(”f2", t4, t5), t3,
    pipe(out(f4, "result/y"), in(f3, "arg/x1")),
    pipe(out(f5, "result/y"), in(f3, "arg/x2")));

// using the return value
assertEquals(value(f1), 400.0);
```

Hybrid Service Composition

```
Task f4 = task(“f4", sig("multiply", new Multiply(), double[].class),
    context("multiply", args(new double[] { 10.0, 50.0 }),
        result ("result/y")));

Task f5 = task(”f5", sig(expression("x2 + x3",
        vars(var("x2", 20.0), var("x3", 80.0))),
        result("result/y"));

Task f3 = task(“f3", sig("subtract", Subtractor.class),
    context("subtract", in("arg/x1", null), in("arg/x2", null),
        result ("result/y")));

Job f1= job(”f1", job("j2", t4, t5), t3,
    pipe(out(f4, "result/y"), in(f3, "arg/x1")),
    pipe(out(f5, "result/y"), in(f3, "arg/x2")));

job = exert(job);
// using the global path
assertEquals(get(job, “f1/f2/f4/result/y"), 100.0);
```

Context & Exertion Closure

- Context Closure – closing over x1 and x2

```
Context<?> cxt = context(in("x1"), in("x2"),
    out("y", var("y",
        expr("e1", "x1 * x2", vars("x1", "x2")))));

assertEquals(value(cxt, "y",
    entry("x1", 10.0),
    entry("x2", 50.0)),
 500.0);
```

- Exertion Closure – closing over x1, x2, and signature

```
Exertion task = task("add",
 sig("add"),
    context(in("arg/x1"), in("arg/x2"),
        result("result/y")));

assertEquals(value(task,
    in("arg/x1", 20.0),
    in("arg/x2", 80.0),
    strategy(sig("add", AdderImpl.class),
        Access.PUSH, Wait.YES)),
 100.0);
```

Hybrid "Hello Arithmetic" Job

```
Task f4 = task("f4", sig("multiply", Multiplier.class),
    context("multiply",
        in("super/arg/x1"), in("arg/x2", 50.0),
        result ("result/y")));
Task f5 = task("f5", sig("add", Adder.class),
    context("add", in("arg/x3", 20.0), in("arg/x4", 80.0),
        result ("result/y")));
Task f3 = task("f3", sig(var("vf3",
        expression("vf3-e", "x5 - x26", vars("x5", "x6"))),
        result(path("result/y"));
 Job f1 = job("f1",
    context(in("arg/x1", 10.0), result("f3/result/y")),
    job("f2", t4, t5,
        strategy(Flow.PARALLEL, Access.PULL) ),
    t3,
    pipe(out(f3, "result/y"), in(f5, "arg/x5")),
    pipe(out(f4, "result/y"), in(f5, "arg/x6")));

assertEquals(get(exert(f1), "f1/f3/result/y"), 400.0);
```

Hybrid “Hello Arithmetic” Model

```
VarModel vm = model("Hybrid Hello Arithmetic",
   inputs(
      var("x1"), var("x2"), var("x3", 20.0), var("x4”)),
   outputs(
      var("f4",
         expression("x1 * x2", args(vars("x1", "x2")))),
      var("f5", task("t5",
         sig("add", Adder.class),
         context("add",
            in("arg/x3", var("x3")),
            in("arg/x4", var("x4")),
            result("result/y")))),
      var("f1", expression("f4 - f5",
         args(vars("f4", "f5"))))));
```

R-S Parametric Model Task

```
Signature msig = sig(ParametricModeling.class,
       "Rosen-Suzuki Model");
    String outURL = Sorcer.getWebsterUrl()
       + "/rs-model/rs-out.data";
    String inURL = Sorcer.getWebsterUrl()
       + "/rs-model/rs-in.data";

ModelTask mt = task(sig("calculateOutTable", msig),
       context(inTable(inURL),
           outTable(outURL, inputs("x1", "x2"),
              outputs("f", "g1", "g2")),
           result("table/out"),
           par(queue(20), pool(30))));

Table table = value(mt);
```

R-S Optimization Model Task

```
Context exploreContext = exploreContext("Rosen-Suzuki context",
    inputs(
        in("x1", 1.0), in("x2", 1.0),
        in ("x3", 1.0), in("x4", 1.0)),
    strategy(new ConminStrategy(
        new File(System.getProperty(
            "conmin.strategy.file")))),
    dispatcher(sig(null, RosenSuzukiDispatcher.class)),
    model(sig("register", OptimizationModeling.class,
            "Rosen-Suzuki Model")),
    optimizer(sig("register", Optimization.class,
            "Rosen-Suzuki Optimizer")));

// Create a task exertion
Task opti = task("opti",
    sig("explore", Exploration.class,
        "Rosen-Suzuki Explorer"),
    exploreContext);

// Execute the exertion and log the output context
logger.info(">>>>>>>>>>>>> results: " + context(exert(opti));
```

R-S Optimization Result

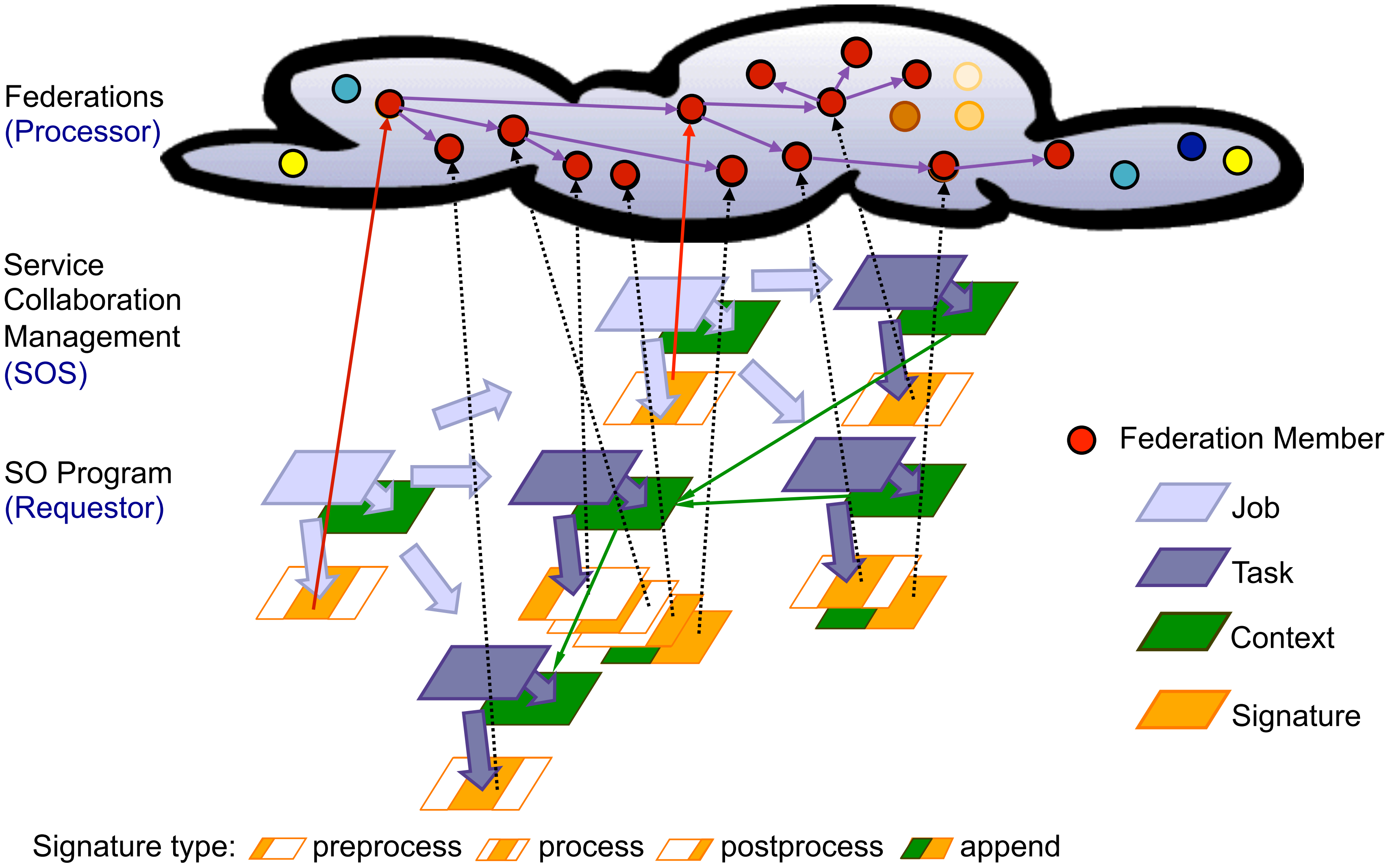
```
Objective Function fo = 6.002607805900986
Design Variable Values
   x1 = 2.5802964087086235E-4
   x2 = 0.9995594642481355
   x3 = 2.000313835134211
   x4 = -0.9986692050113675
Constraint Values
   g1c = -0.002603585246998996
   g2c =-1.0074147118087602
   g3c = 4.948009193483927E-7
Iterations
   Number of Objective Evaluations = 88
   Number of Constraint Evaluations = 88
   Number of Objective Gradient Evaluations = 29
   Number of Constraint Gradient Evaluations = 29
```

Netlets – Interpreted Mograms

```
#!/usr/bin/env nsh -f
import sorcer.arithmetic.provider.Multiplier;
import sorcer.service.Strategy.Monitor
import sorcer.service.Strategy.Wait

task("net-multiply",
   sig("multiply", Multiplier.class),
   context(
      input("arg/x1", 10.0d),
      input("arg/x2", 50.0d),
      output("result/y")),
   strategy(Monitor.YES, Wait.NO));
```

Exerting Dynamic Federations



Federations (Processor)

Service Collaboration Management (SOS)

SO Program (Requestor)

● Federation Member

Job

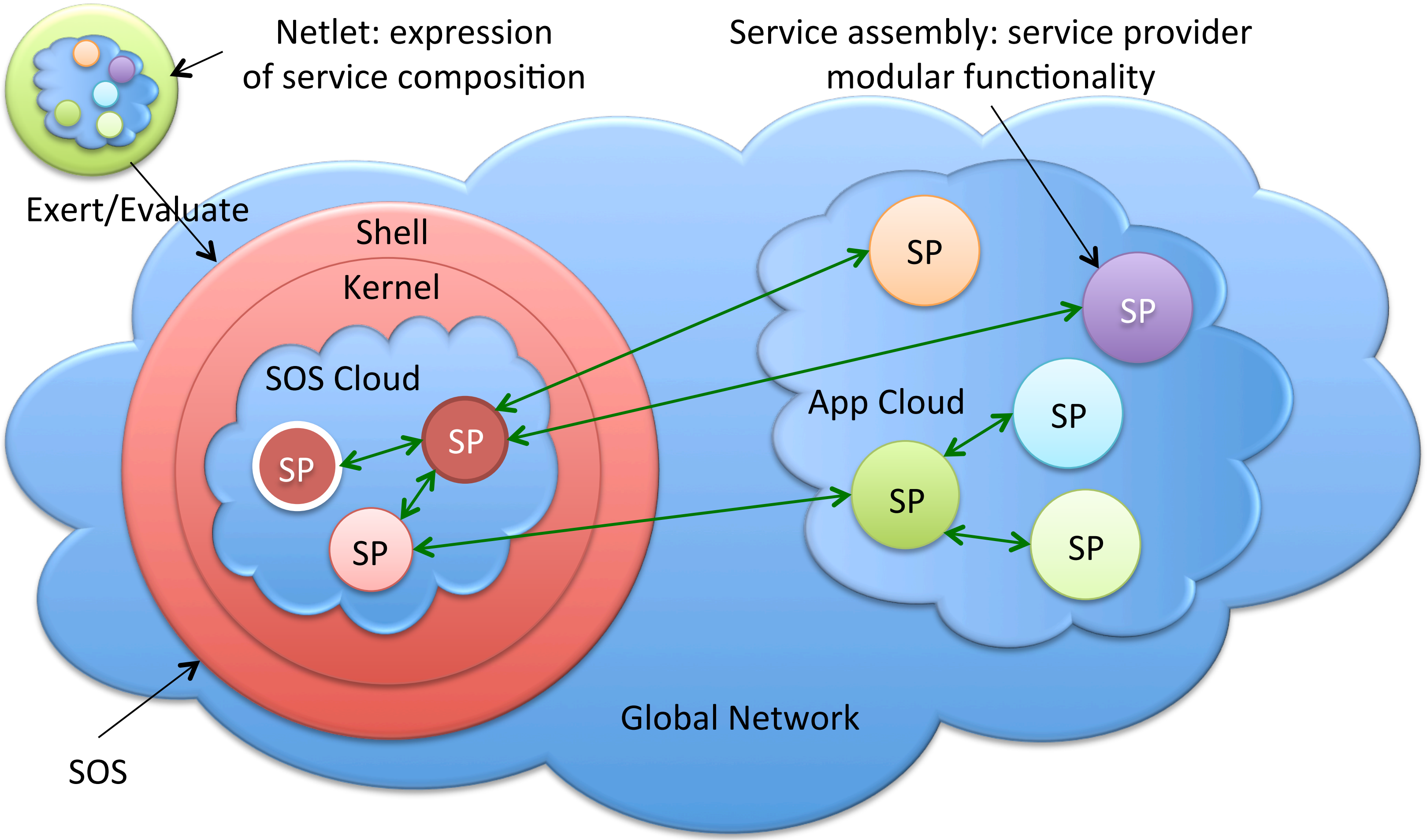
Task

Context

Signature

Signature type: preprocess process postprocess append

Netlets Run Everywhere



SP – Service Provider

SORCER Architecture

- An exertion is an expression of a federation of service providers; it exerts the local/distributed service collaboration.
- Connectivity
  - Dynamic with service provisioning
  - No static connections
- Interoperability
  - P2P (S2S) – operating system
    - Servicer#service(Exertion):Exertion
  - Data – service providers
    - <serviceType>#<selector>(Context):Context
- Collaborations (netlets)
  - Exertions (exerting collaborations of service providers)
  - Var-models (modeling connections)
  - SO mograms (hybrid both of them)

Service Engineering

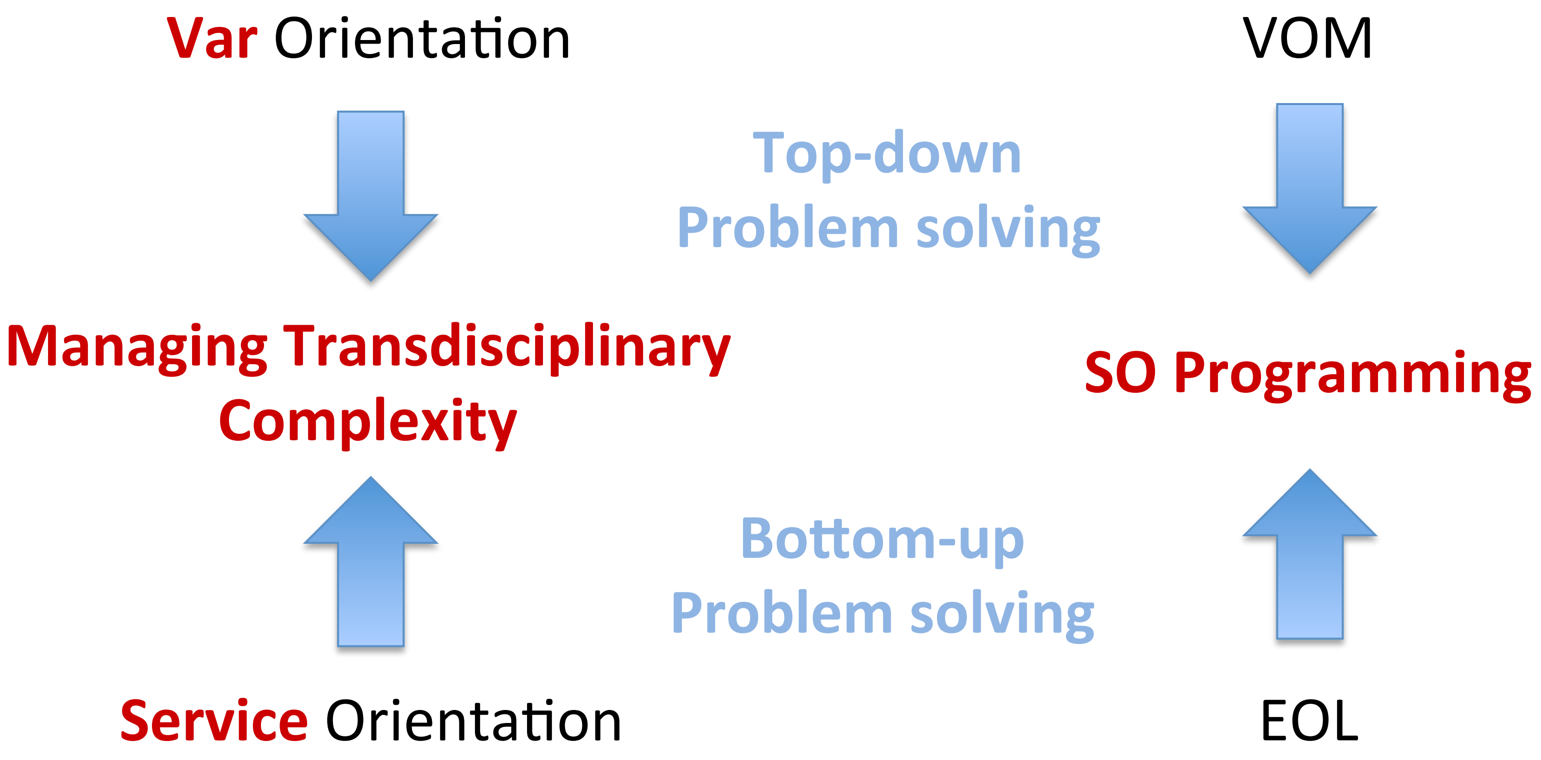
- Net service provider
  - Assembly from service beans
  - Inheritance from ServiceTasker.class
- Multiple service types per a service bean or per a ServiceTasker instance
- Multiple beans per provider
- Hybrid of multiple beans with a DS ServiceTasker
- Multiple providers per service node (JRE)
- Configuration
  - Provider deployment configuration
    - Multiple ways of proxying
    - Multiple endpoints (JERI)
    - SORCER invocation layer (Servicer#service(Exertion)
  - Provider's DS specific properties
  - SORCER env properties

UNIX Platform vs. SORCER Platform

	UNIX	SORCER
Data	File - file system	Data context - objects
Data flow	Pipes	Data context pipes
Cohesion	Everything is a file	Everything is a service
Processor	Native (instruction set)	Service providers net
Interpreter	UNIX Shell	Network shell (nsh)
System language	C	Java/Jini/Rio/SORCER API
Command language	UNIX shell scripting	EOL/VOL/VML scripting
Process control strategy	Command flow logic	Exertion control context & control flow exertions
Executable codes	Many choices	Many choices

Unix pipes – processes; SORCER pipes – data contexts
Command concatenation vs. Service federation
Local shell vs. network shell

Top Down or/and Bottom Up Mogramming



Var Orientation

VOM

Top-down Problem solving

Managing Transdisciplinary Complexity

SO Programming

Bottom-up Problem solving

Service Orientation

EOL

Today's Main "Take Away" Points

- Var-oriented modeling
  - Top-down SO problem solving
  - Var compositions (models)
    - Emphasis on var connectivity
    - Var services are specified by an evaluator/filter pair
    - Var evaluators can run locally or in the network
- Exertion-oriented programming
  - Bottom-up SO problem solving
  - Service compositions (exertions)
    - Emphasis on net services
    - Exertion providers are specified by service signatures
    - Service providers can in the network or locally
- In either case vars or exertions embrace *heterogeneous local/distributed service*