**Service Modeling Language (SML)**

SML is a metamodel-based language designed for service-oriented computing. Its models, called service mograms, are focused on service-orientation similarly to UML models that are focused on object-orientation. The basic modeling style and rules of SML reflect functional composition. The simplified SML syntax is sketched below using the customized BNF style (see the *Comments* section). The SORCER [project side](http://sorcersoft.org/project/site/) contains a few hundred simple examples that allow you to learn modeling patterns along with syntax rules of SML modeling. SML models are executable by the SORCER platform when sufficiently configured.

**Service Items and Mograms**

In computing a service is the work performed in which a service provider (one that serves) exerts acquired abilities to execute a computation. A service federation is a runtime collection of all service providers needed to actualize service collaboration expressed in a service-oriented language. An SML expression called a *service item* corresponds to a single *service provider* – also called a *microservice* and a *service mogram* corresponds to a *service federation – also called a macroservice*.

Microservices implement one of the standard or custom service types (practically Java interface). SORCER standard interfaces are follows: *Evaluation* (evaluator), *Invocation* (invoker), *Callable* (caller), *Multifidelity* (fidelity, in short fi), *Signature* (a service provider handle associated with a custom service type), and *Provider* interface.

Remote service providers are remote objects that implement the *Provider* interface and expose any service types directly (by implementation) or indirectly (by delegation to their internal components called service beans). That means that all standard and custom microservices can be used as remote services referred by service types specified in service signatures.

**Service Signatures**

A *provider* *signature* is a service provider reference (handle) specified by a *service type*. The role of provider signatures that declare provider services is similar to constructors in object-oriented programming. An *operation signature* expends a *provider signature* and is an executable provider service - *exec*(*signature*). An operation signature can be customized with the following options: signature name, signature operation name (selector), provider name, implemented types, groups, locators, data context, return result format, input and output connectors.

srvSignature ::= prvSignature | opSignature | bldrSig| **sig(**srvSignature**,** signatureOp**)**

prvSignature ::= **sig(**[name**,**] prvSpec**)**

ntlSignature ::= **sig(**[name**,**] **filePath(**netletFilename | artifactName**))**

opSignature ::= **sig(**[name**,**] opSpec, prvSpec**)**

| **sig(**prvSignature**,** selector**)** | **sig(**prvSignature**,** signatureOp**)**

bldrSig ::= **sig(**[name**,**] classSelector, classType**)**

prvSpec ::= (srvType | **type(**srvTypeName)) [**,** **types(**type+**)**] [**,** (prvId] [**,** prvDeployment]

| bldrSig | prvInstance

opSpec ::= (selector |signatureOp) [**,** srvResult]

[**,** **inConn(**mapEntry+**)**] [**,** **outConn(**mapEntry+**)**] [**,** dataContext]

srvType ::= classType | interfaceType

signatureOp := **op(**selector**,** srvArg\***)** | **op(**opSignature**)**

prvId ::= **srvName(**serviceName [**,** **locators(**locatorName+**)** ] [**,** groupName+]**)**

| **prvName(**providerName**)** |

srvResult ::= **result(**[pathName**,** ][**,** inputPaths] [**,** outputPaths] [**,** dataContext]**)**

prvDeployment ::= **deploy(**(**configuration(**configName**) |** prvImpl)**,** deployOptions**)**

prvImpl ::= **implementation(**providerClassName**),**

**classpath(**jarName\***),** **codebase(**jarName\***)**

deployOptions ::=[**, maintain(**intNumber**)**][**,** **perNode(**intNumber**)**] [, **idle(**intNumber**)**]

inputPaths ::= **inPaths(**srvPath+**)**

outputPaths ::= **outPaths(**srvPath+**)**

**Service Multifidelities**

*Fidelity* is defined to be “the degree to which something matches or copies something else” (webster.com) or in general “adherence to fact or detail” (dictionary.com). In SML, a degree of adherence, matching, or accuracy has the same meaning, though it is acknowledged they have different meanings in some circles. Similarly, *service multifidelity* in a SML perspective refers to a modeling environment with multiple fidelity levels for a given computing process, meaning there are different computing components or functions to choose from. Fidelity and cost (or similarly accuracy and time) are positively correlated; this represents a fundamental trade in modeling and design. Multifidelities in SML provide a basic mechanism for agility and emergent behavior of adaptive service-oriented processes.

*Morph fidelities* are observable service fidelities by a *fidelity manager* associated with compound services (mograms). A fidelity manager takes into account observable results of executed multifidelities and by updating the existing mogram fidelities upgrades the structure of a mogram as needed. When morph fidelities are associated with functions called morphers, then a fidelity manager redirects its execution to morphers.

multiFi ::= entFidelity | sigFidelity | morphFi | varFidelity | reqFidelity | fiMogram

entFidelity ::= **eFi(**[fiName**,** ]contextEntry+**)**

sigFidelity ::= **sFi(**[fiName**,** ]opSignature+**)**

morphFidelity ::= **mFi((**[fiName**,** ][srvMorpher**,**] srvRequest+**)**

srvMorpher::= MorpherLambdaExpression

varFidelity ::= **vFi(**fiName**,** (value |opSignature | contextEntry)**)**

| **vFi((**[fiName**,** ][**,** srvRoutine] [**,** entGetter] [**,** entSetter]**)**

reqFidelity ::= **rFi(**[fiName**,** ] srvRequest+**)**

**Provider Services**

A *service* *provider* is an instance of a local or remote concrete service specified by a signature

prvInstance ::= **prv(**srvSignature**)**

Signatures, signature entries and elementary tasks are bounded to single service providers but service mograms are bound to federations of service providers by the SORCER platform at runtime.

**Service Requests**

Elementary *service requests* are called *items* and *compound requests* are called *mograms*. For example, signatures, context entries, and service fidelities are items. Context models and exertions are mograms.

srvRequest ::= srvSignature | multiFi | contextEntry | srvMogram

**Context Entries**

An *entry* is a functional association of a *path* and a *function body* of an underlying context model. A *path* is a function name as a sequence of attributes that define a modeling namespace. The body of an entry specifies a return value of the entry. A body defining a function composition depends on paths of other entries in the model scope.

entType ::= **in** | **out** | **inout** | **db**

annotatedPath ::= **path(**pathName [**,** pathTag]**)**

mapPath ::= **map(**toPath, fromPath**)**

srvPath ::=pathname| annotatedPath | mapPath

dataEntry ::= **val(**srvPath**,** value**)** | entType**Val(**srvPath**,** value**)**

contextEntry ::= dataEntry | procEntry | srvEntry | varEntry | fiEntry

| entType**(**[name**,** ]contextEntry**)**

procEntry ::= **ent(**pathName**,** srvRoutine [**,** entModel]**)** | sigEntry | lambdaEntry

srvRoutine ::= srvEvaluator | srvInvoker

sigEntry ::= **ent**([name, ]opSignature**)**

mapEntry ::= **ent(**fromPathName**,** toPathName**)**

lambdaEntry ::= **lambda(**pathName**,** (EntryCallableLambdaExpression**)**

| ServiceLambdaExpression

| CallableLambdaExpression

| ClientLambdaExpression]

| ValueCallableLambdaExpression) [**,** srvArgs]**)**

srvEntry ::= **ent(**pathName**,** (opSignature[**,** entModel] [**,** cxtSelector]**)**

| srvRoutine| srvMogram)**)**

varEntry ::= **var(**pathName**,** (value|opSignature|varFidelity+|morphFidelity

| srvRoutine|srvEntry|varProxy)**)** | entVar | objectiveVar | constraintVar

cxtSelector ::= **selector**([componentName**,**]pathName+**)**

fiEntry ::= **ent(**pathName**,** entFidelity**)**

varProxy ::= **proxy(**pathname**,** opSignature**)**

srvInvoker ::= **invoker(**JavaExpression**,** srvArgs [**,** dataContext]**)**

| **invoker(**opSignature)| srvExertion | **inc(**srvInvoker, double | int**)**

| **methodInvoker(TODO)**

| **cmdInvoker(TODO)**

| **invoker(**[name**,**]ValueCallableLambdaExpression[**,** contextModel]**,** srvArgs**)**

|procEntry | conditionalInvoker

conditionalInvoker ::=::= **loop**(srvCondition**,** srvInvoker**)**

| **loop(**min**,** max**,** [srvCondition**,**] srvInvoker**)** | **alt(**invokeOption\***)** |invokeOption

invokeOption ::= **opt(**srvCondition**,** srvInvoker**)**

srvCondition ::= **condition(**ConditionCallableLambda**)**

| **condition(**conditionExpression**,** parameterName\***)**

srvArgs ::= **args(**argName+**)**

dependentVars ::= **vars(**dependentVarName\***)**

srvEvaluator ::= objectImplemntingEvaluation | procEntry | lambdaEvaluator

srvInvoker ::= objectImplemntingInvocation

entGetter ::= objectImplemntingGetter

entSetter ::= objectImplemntingSetter

**Service Mograms**

*Service mograms* *are compound requests* that specify service federations. A context model is a declarative specification and an exertion is a procedural specification for a dynamically bound federation of collaborating service providers.

contextModelType ::= **procModel** | **srvModel** | **varModel** | **model**

srvExertionType ::= **task** | **block** | **job** | conditionalExertion | **exertion**

conditionalExertionType ::= **loop** | **alt** | **opt**

srvMogramType ::= contextModelType | srvExertionType | **mogram**

**model** == **mdl**

**context** == **cxt**

**exertion** == **xrt**

**mogram** == **mog**

srvMogram ::=dataContext | contextModel | srvExertion | fiMogram

| **mogram(** (contextModelParameters | srvExertionParamters) **)**

fiMogram ::= **fiMog(**[name**,**] (morphFidelity | reqFidelity**)**

**Context Models**

A *model* is an aggregation of entries representing service federations as functionals. A data context is composed of entries of the dataEntry type and a context model of entries of the contextEntry type.

entModel ::= dataContext | contextModel | structuredVarModel | contextSnapshotResult

dataContext ::= **context**([name,] dataEntry+ [**,** srvResult][**,** inputPaths] [**,** outputPaths]**)**

**| tag(**dataContext**,** annotatedPath**)** | **tagAssociation(**dataContext**,** newTagAssociation**)**

contextModel ::= contextModelType **(**[name**,** ] contextEntry+

[**,** **response(**pathname+**)** [**,** srvDependency]**)**

parTypes ::= **types(**Class+**)**

parArgs ::= **args(**object+**)**

srvDependency::= **dependsOn(ent(**pathName, **paths(**pathname+**)**+**)**

**Tasks**

A task specifies an action of a provider service or concatenation (batch) of provider services processing a data context.

srvTask ::= **task(**[name**,**] [opSignature\* | sigFideliy\* | sigMorphFidelity]**,** [dataContext]**)**

**Exertions**

An exertion is a hierarchical composition of tasks and other exertions – a compound exertion. Concatenated exertions (blocks), workflow exertions (jobs) and conditional exertions are compound exertions that are specified accordingly by signature, data context, and component mograms with optional an control strategy and execution dependencies.

srvExertion ::= srvTask | compoundExertion | **exertion(**srvExertionParamters**)**

compoundExertion ::= srvJob | srvBlock | conditionalExertion

srvJob ::= **job(**[name**,**] [opSignature | sigFideliy]**,** dataContext**,** srvMogram\***,**

contextPipe\* [**,** exertionStrategy] [**,** dependency], metaFiSelector\***)**

srvBlock = **block**([name**,**] [opSignature | sigFideliy]**,** [dataContext**,**]

srvMogram\***,** metaFiSelector \***)**

conditionalExertion ::= **loop**(srvCondition**,** srvMogram**)**

| **loop(**min**,** max**,** [srvCondition**,**] srvMogram**)** | **alt(**srvOption\***)** |srvOption

srvOption ::= **opt(**srvCondition**,** srvMogram**)**

contextPipe ::= **pipe(outPoint(**srvExertion, contextPathName**),**

**inPoint(**srvExertion, contextPathName**))**

exertionStrategy ::= **strategy(**[accessType**,**] [flowType**,**] [monitorable**,**] [provisionable]**)**

flowType::= **Flow.PAR** | **Flow.SEQ**

monitorable ::= **Monitor.YES | Monitor.NO**

**Variable-oriented Modeling**

Var-oriented models are structured contextModels with additional specialized aggregations of multifidelity varEntries (for example inputs, outputs, constraints, objectives variables, etc.), The structured var-models are associated with specialized modeling tasks, for example, a response, parametric, or exploration tasks. A result of executing a modeling task is, for example, a response vector for a vector of design inputs, a response table for a parametric table, and exploration context for an optimization task. When declared, a structured var-model can be more or less concrete. To be executed, to some degree an abstract model has to be configured by specifying all vars as fully declared in a model. Aggregated var-entries in structured var-models collaborate in the model accordingly to a declared type of structured modeling. Structured var-models can be used as local or remote service providers. In either case a modeling task specifies a required modeling provider with its modeling context and returns a corresponding result.

structuredVarModel ::= responseModeling | parametricModeling | optimizationModeling

| streamingParametricModeling

responseModeling ::= **responseModel(**[modelName**,** ]

[modelingInstance**,** ] basicVars+**,** varRealization\***)**

parametricModeling ::= **paramericModel(**[modelName**,** ]

[modelingInstance**,** ] basicVars +**,** varRealization\***,**

**table(**varParametricTable**,** mdlResponseTable**))**

streamingParametricModeling ::= **streamingParametricModel(**[modelName**,**

modelingInstance**) TODO streamer/reader?**

optimizationModeling ::= **optimizationModel(**[modelName**,** ] basicVars +**,** varRealization\***,**

**objectiveVars(**objectiveVar+**), constraintVars(**constraintVar+**)** [**,** modelingInstance]**)**

modelingInstance ::= **instance(**bldrSig**)**

varType ::= **input** | **output** | **linked** | **constant**

baseVars ::= varType**Vars(**basicVar+**)**

mdlParametricTable ::= **parametricTable(**(tableURL | filename ) [**,** tableSeparator]**)**

| dataTable| **parametricTable(**instanceofModelTable.class**)**

mdlResponseTable ::= **responseTable(**(tableURL | filename ) [**,** tableSeparator]**)**

dataTable := **table(header(**varName+**)**, **row(**value+**)**+**)**

basicVar ::= entVar **var(**varName**,** count#**)** | **var(**varName**,** from#, to#**)**

entVar ::= **var(**[name**,**] srvEntry**)**

objectiveVar ::= **var(**varName**,** outputVarName**,** optiTarget**)**

optiTarget ::= **Target.min** | **Target.max**

constraintVar ::= **var(**varName**,** outputVarName**,** **Relation.**relationSuffix**)**

relationSuffix ::= **lt** | **lte** | **eq** | **gt** | **gte**

varRealization ::= **realization(**varName**,** **fi(**fiName**,** varComponent+**)**\***,**

**fi(**fiName**,** **differentiation(wrt(**varName+**))**\***)**

varComponent ::= **evaluator(**evaluatorName**)** | **getter(**getterName**)** | **setter(**setterName**)**

**Structured-Var Modeling Tasks**

modelingTask ::= mdlResponseTask | mdlParamericTask | mdlOptimizationTask

mdlResponseTask ::= **responseTask (outerSig(**selector, bldrSig**)**

[**,** **modelingContext(**[**inputs(**dataEntry+**),** ] [**responses(**varName\***),** ]

[**,** **result(**pathName**)**]**))**

mdlParamericTask ::= **parametricTask(outerSig(**selector**,** bldrSig**),**

**modelingContext(**varParametricTable**,** varResponseTable**,**

[**, parameters(**varName\***)**] [**, responses(**varName+**)**] [**,** **result(**pathName**)**]

[**,** mdlParametricTable] [**,** mdlResponseTable] [**,** parStrategy]**))**

mdlOptimizationTask ::= **optimizationTask (**explorerSignature**,** optiStrategy**,**

**modelingContext(initialDesign(**dataEntry+**)** [**,** **result(**pathName**)**]**))**

parStrategy ::= **parallel(queue(**int**),** **pool(**int**))**

optiStrategy ::= **strategy(**optiTarget**,** **dispatcherSig(**prvSignature**),**

**modelSig(**prvSignature**),** **optimizerSig(**prvSignature**))**

explorerSignature ::= opSignature

**Accessing Values and Getting Results**

contextValueResult ::= **value(**dataContext, (pathName | outputPaths)**)**

|(**valueAt** | **valuesAt**) **(**dataContext, (index | pathTag)**)**

srvValueResult ::= **exec(**srvRequest**,** arg\***)** |

| **eval(**contextEntry, srvArg\***)** | **eval(**entModel, pathName, srvArg \***)**

| **eval (**srvExertion, srvArg\***)** | **returnValue(**srvMogram**)**

| **get(**contextModel, pathName)

srvMogramResult ::= **exert(**srvMogram, srvArg\***)**

dataContextResult ::= **response(**entModel, srvArg\***)** | **result(**entModel [, pathName]**)**

| **context(**srvMogram**)** | **upcontext(**compoundExertion**)**

srvExertionResult ::= **get(**srvExertion, componentPathName**)**

dataEntryResult ::= **getEntry(**contextModel, pathName**)**

entryResult ::= **setValue(**contextEntry**,** value**)**

contextModelResult ::= **setValue(**contextModel**,** pathname**,** value**)**

| **setValue(**contextModel**,** contextEntry+**)**

| **setValue(**contextModel**,** contextModel**)**

| **append(**contextModel**,** dataContext**)**

contextSnapshotResult ::= **snapshot(**structuredVarModel [**,** responseContext]**)**

varInformation ::= **varInfo(**varsType, varName\***)**

varsType ::= **INPUTS** | **CONSTANTS** | **INVARIANTS** | **DESIGN** | **PARAMETERS**

| **ALL\_OUTPUTS** | **OUTPUTS** | **RESPONSES** | **LINKED** | **OBJECTIVES**

| **CONSTRAINTS** | **WATCHABLE** | **ALL** | **NONE** | **NULL** |

srvArg ::= instanceofArg.class | dataEntry | srvMogram | fiSelector | metaFiSelector

| cxtSelector | inputPaths | outputPaths |srvResult | opSignature | accessType

| provsionable | fiList

fiSelector ::= **fi(**pathName, fiName**)**

metaFiSelector ::= **fi(**fiName**,** fiSelector+**)**

fiList ::= fis((fiSelector | fiList)+])

accessType ::= **Access.PUSH** | **Access.PULL**

provisionable ::= **Provision.YES** | **Provision.NO**

**Comments**

* Not all parts of SML language implementation are available as open source
* The simplified BNF notation is used to enhance functional composition notation for SO metamodeling with type-based arguments. That means that in most cases the order or function arguments does not matter and number of arguments depends on the context used.
* Each rule is of the form nonterminal ::= metaexpression. Other metasymbols are: | for alternation, brackets [ … ] for options, parenthesis ( ... ) for grouping, postfix + for 1 or more occurrence, postfix \* for 0 or more occurrences, and ; for the rule termination unless the line delimiter indicates termination. For simplicity, postfix \* and + includes the terminal separator **,** for each function argument. The equivalence == is for aliasing and abbreviations. Note the dual use of parenthesis as BNF metasymbols and SML terminal symbols in bold.
* Terminals appear in bold and non-terminals in plain;
* Non-terminals name, \*Name, annotation, and selector are of String type
* srvType is a class or interface type (\*.class)
* value is an instance of Object, instanceofClassName.class in an object of ClassName type
* \*LambdaExpression is a Java 8 lambda expression for the functional interface named by a prefix to “LambdaExpression”
* conditionExpression is a Groovy style closure with parameterNames as paths in its block context (scope)
* **loop** min times, then while condition is true, loop (max - min) times.
* (UML semantics of the loop operator)
* in netlets
  + classpath specification examples
  + @Load('org.sorcer:sml:jar:${sorcer.version}')
  + @Load(group='org.sorcer', module='sml', version='${sorcer.version}')
  + codebase specification examples
  + @Codebase('org.sorcer:sml:jar:dl:${sorcer.version}')
  + @Codebase(group='org.sorcer', module='sml', version='${sorcer.version}', classifier='dl')
* *Tell me and I forget. Show me and I remember. Involve me and I understand.* Get involved - the SORCER project website: <http://sorcersoft.org/project/site/>