**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.

**Signatures**

A *provider* *signature* is a service provider reference (handle) specified by a *service type*. The role of provider signatures declaring provider services is similar to constructors in object-oriented programming. An *operation signature* expending a *provider signature* 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 | **sig(**srvSignature**,** signatureOp**)**

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

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

prvSpec ::= (srvType | prvInstance | **type(**srvTypeName) [**,** **types(**type+**)**] [**,** (srvTag | prvTag)]

[**,** prvDeployment]

opSpec ::= selector**,** signatureOp [**,** dataContext] [**,** srvResult]

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

srvType ::= classType | interfaceType

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

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

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

prvTag ::= **prvName(**providerName**)**

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

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

prvDeployment ::= **deploy(implementation(**providerClassName**),** **classpath(**jarName+**),**

**codebase(**jarName\***),** **configuration(**configName**),** **maintain(**intNumber**),** **idle(**intNumber**))**

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

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

**Provider Services**

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

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

Service mograms are bound to service providers at runtime by the SORCER platform.

**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 | contextEntry | multiFi | srvMogram

**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 modeling namespace. A 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]**)**

srvPath ::=pathname| annotatedPath

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

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

procEntry ::= **ent(**opSignature**)** | **ent(**pathName**,** srvEvaluator**)**

| **ent(**pathName**,** srvInvoker [**,** srvModel]**)** | procEntry | lambdaEntry

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

| **lambda(**pathName**,** ServiceLambdaExpression**,** srvArgs**)**

| **lambda(**pathName**,** CallableLambdaExpression**,** srvArgs**)**

| **lambda(**pathName**,** ClientLambdaExpression]**,** srvArgs**)**

| **lambda(**pathname**,** ValueCallableLambdaExpression**,** srvArgs**)**

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

| **ent(**pathName**,** srvRoutine**)** | **ent(**pathName**,** srvMogram**)**

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

srvRoutine ::= contextEntry | srvInvoker | srvEvaluator

varEntry ::= **var(**pathName**,** value**)** | **var(**pathName**,** opSignature**)**

| **var(**pathName**,** varFidelity+**)** | **var(**pathName**,** morphFidelity\***)**

| **var(**pathname, lambdaEvaluator**)** | **var(**pathName**,** varProxy**)**

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

entFidelity ::= **eFi(**contextEntry\***)**

contextEntry ::= dataEntry | srvEntry | varEntry | fiEntry

| entType**(**contextEntry**)**

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

morphFidelity ::= **mFi(**fiName**,** srvRequest+**)**

| **mFi(**fiName**,** srvMorpher**,** srvRequest+**)**

srvMorpher::= MorpherLambdaExpression

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

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

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

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

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

| **methodInvoker()**

| **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 ::= lambdaEvaluator | entEvaluator | objectImplemntingEvaluation

srvInvoker ::= objectImplemntingInvocation

entGetter ::= objectImplemntingGetter

entSetter ::= objectImplemntingSetter

**Mograms**

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

contextModelType ::= **entModel** | **parModel** | **srvModel** | **model** | **mdl** | **varModel**

srvExertion Type ::= **task** | **block** | **job** | conditionalExertion | **exertion** | **xrt**

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

srvMogramType ::= contextModel | srvExertion | **mogram** | **mog**

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

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

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

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

srvMogram ::=dataContext | contextModel | srvExertion | multiFiMogram

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

multiFiMogram ::= **multiFiReq(**[name**,**] morphFidelity**)** | **multiFiReq(**[name**,**] srvFidelity**)**

**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.

srvModel ::= dataContext | contextModel | varOrientedModel

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

**| tag(**dataContext**,** annotatedPath**)** | **tagContext(**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 provider service or concatenation (batch) of provider services processing data context.

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

**Exertions**

An exertion is an elementary exertion – task or a hierarchical composition of tasks – a compound exertion. A concatenated exertion (block), workflow exertion (job) and a conditional exertion are compound exertions that are specified accordingly by signature, data context, and component mograms with optional strategy and execution dependencies.

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

compoundExertion ::= srvJob | srvBlock | conditionalExertion

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

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

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

srvMogram\***,** exertionFidelitity\***)**

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

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

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

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

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

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

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

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

srvFidelity ::= **srvFi(**[fiName**,** ] srvRequest+**)**

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

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

multiFi := entFidelity | sigFi | morphFi | varFidelity | srvFidelity

**Accessing Values and Getting Results**

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

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

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

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

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

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

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

dataContext ::= **response(**srvModel, srvArg\***)**

| **result(**srvModel [, pathName]**)** | **context(**srvMogram**)**

| **upcontext(**compoundExertion**)**

srvEntry ::=srvValue | **get(**srvMograml, componentName**)**

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

| compFiSelector | cxtSelector | inputPaths | outputPaths |srvResult

**Var-Oriented Modeling**

Var-oriented models are contextModels as defined above with various aggregations of varEntries to specify, for example, a response, parametric, or exploration modeling tasks. A result is a response, for example an output vector for a vector of design inputs, a response table for a parametric table, and exploration context, for example, for an optimization task. To be executed, a var-oriented model has to be configured by specifying all vars as declared in a model. That follows the metamodeling approach such that each var-oriented model in fact is a metamodel for a modeling task. Var-entries in var-oriented models collaborate in the model accordingly to a declared type of modeling. Var-oriented models can be used as local or remote service providers. In either case an exertion task specifies a required service provider and its data context and returns a result context by the executed model.

varOrientedModel ::= responseModeling | parametricModeling | optimizationModeling

| streamingParametricModeling

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

[modelingInstance**,** ] baseVar\***,** varRealization\***)**

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

[modelingInstance**,** ] baseVar\***,** varRealization\***,**

**table(**varParametricTable**,** varResponseTable**))**

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

modelingInstance**)**

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

**objectiveVars(**objectiveVar+**), constraintVars(**constraintVar+**))**

modelingInstance::= **instance(**opSignature**)**

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

baseVar ::= varType**Vars(**varName+**)** |varType**Vars(**varName**,** count#**)**

| varType**Vars(**varName**,** from#, to#**)** | **var(**[name**,**] srvEntry**)**

varParametricTable ::= **parametricTable(**tableURL [**,** tableSeparator]**)**

| **table(header(**varName+**)**, **row(**value+**)**+**)**

varResponseTable ::= **responseTable(**tableURL [**,** tableSeparator]**)**

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**)**

**Var-Oriented Modeling Tasks**

modelingTask ::= mdlResponseTask | mdlParamericTask | mdlOptimizationTask

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

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

[**,** **result(**pathname**)**]**))**]

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

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

[**, parameters(**varName\***)**] [**, responses(**varName\***)**]

[**, parallel(queue(**int**,** **pool(**int**))**] [**,** **result(**pathname**)**]**))**

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

**exploreContext(initialDesgn(**dataEntry \***)** [**,** **result(**pathname**)**]**),**

**strategy(**optiTarget**,** **dispatcherSig(**prvSignature**),**

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

modelSig ::= opSignature

explorerSignature ::= opSignature

**Comments**

* 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
* \*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/>