



# Software Architecture in Action

Flavio Oquendo, Jair C Leite, Thais Batista



# Chapter 17

## Textually Representing Software Architectures

# Learning outcomes of this chapter

- You will learn:
  - the SysADL textual notation to express the structural, behavioral, and executable viewpoints
  - the SysADL grammar for each architectural construct
  - how to use the textual notation to define an architecture

# The structure of this chapter

1. Introduction
2. Textual notation for the structural viewpoint
3. Textual notation for the behavioral viewpoint
4. Textual notation for the executable viewpoint
5. Summary
6. For further reading

# Introduction (1/2)

- Presentation of the textual notation of SysADL
  - The SysADL constructs (structural, behavioral, and executable) in the textual notation
- The expressiveness of both diagrammatic and textual notation is equivalent as all the architectural abstractions expressed in the diagrammatic notation can be also expressed in the textual notation
  - SysADL provides both notations to allow the architect choose the most suitable to describe the different parts of the architecture
  - Typically, the textual notation is used for detailing the diagrammatic representation of the overall picture of an architecture

# Introduction (2/2)

- The questions that we will address are:
  - what is the SysADL grammar for specifying the textual description of a software architecture?
  - how to apply the SysADL grammar for describing different views of an architecture?



# Textual notation for structural viewpoint

# Textual notation

## Data

### Data and Value Types

- SysADL provides the following primitive data types: **Boolean**, **Integer**, **Real**, and **String**
- These scalar data types are completed with user-defined enumerated types, defined with the **enum** construct
- Data types, defined with the datatype construct, are compound types
- A value type represents a quantity in a dimension according to a unit of measure
- A value type may be defined by extending another defined value type

### Grammar

```

DataDef ::=
    DimensionDef |
    UnitDef |
    ValueTypeDef |
    DataTypeDef |
    Enumeration
DimensionDef ::=
    'dimension' ID
    ('{' (Property)* '}' )?
UnitDef ::=
    'unit' ID 'for' ID
    ('{' (Property)* '}' )?
  
```



# Textual notation

## Data

### Data and Value Types

#### ■ Examples:

```
dimension TemperatureDimension
unit Fahrenheit for
TemperatureDimension
unit Celsius for TemperatureDimension
```

### Grammar

```
DataDef ::=
  DimensionDef |
  UnitDef |
  ValueTypeDef |
  DataTypeDef |
  Enumeration
DimensionDef ::=
  'dimension' ID
  ('{' (Property)*'}')?
UnitDef ::=
  'unit' ID 'for' ID
  ('{' (Property)*'}')?
```

# Textual notation

## Data

### Data and Value Types

- A value type represents a quantity in a dimension according to a unit of measure

- Examples:

```
valuetype Temperature = Real
  <dimension = TemperatureDimension>;
valuetype FTemperature
  extends Temperature
  <unit = Fahrenheit>;
valuetype CTemperature
  extends Temperature
  <unit = Celsius>;
```

### Grammar

```
ValueTypeDef ::=
  'valuetype' ID (('extends' ID)
  / ('=' ID))?
  (('<' 'unit' '=' ID '>')?
  (('<' 'dimension' '=' ID '>')?
  )?
```

# Textual notation

## Data

### Data and Value Types

- A value type represents a quantity in a dimension according to a unit of measure

- Examples:

```
datatype Commands {  
  heater: Command;  
  cooler: Command;  
}
```

### Grammar

```
DataTypeDef ::=  
  'datatype' ID '{'  
    (TypeUse)*  
  '}'
```

# Textual notation

## Data

### Data and Value Types

- **Enumerated datatypes:**  
defined by literal values

- Examples:

**enum** Command {on, off}

**enum** TemperatureUnit {Celsius, Fahrenheit}

### Grammar

```
Enumeration ::=  
  'enum' ID '{'  
    (Property)*  
    (EnumLiteralValue  
      ( " , " EnumLiteralValue)* )?  
    '}'  
  
EnumLiteralValue ::=  
  ID
```

# Textual notation

## Data (example)

**dimension** TemperatureDimension

**unit** Fahrenheit **for** TemperatureDimension

**unit** Celsius **for** TemperatureDimension

**valuetype** Temperature = **Real** <**dimension** = TemperatureDimension>

**valuetype** FTemperature **extends** Temperature <**unit** = Fahrenheit>

**valuetype** CTemperature **extends** Temperature <**unit** = Celsius>

**datatype** Commands {

  heater: Command

  cooler: Command

}

**enum** Command {on, off}

**enum** TemperatureUnit {C, F}

# Textual notation

## Components

### Component Definition

- **Component** defines the ports used to interact with components of the type, and for each port the direction of the flow through the port (in or out)

### Grammar

```
ComponentDef ::=  
  'boundary'? 'component' 'def' ID  
  '{'  
    ( StructuralDef  
      | DataDef )*  
    ('ports'  
      PortUse*)  
    (Configuration)?  
  '}';
```

# Textual notation

## Components

### Component Definition

```
component def SensorsMonitorCP {
  ports
    in s1: CTemperatureIPT
    in s2: CTemperatureIPT
    out average: CTemperatureOPT
}
component def CommanderCP {
  ports
    target: CTemperatureIPT
    average: CTemperatureIPT
    heating: CommandOPT
    cooling: CommandOPT
}
component def PresenceCheckerCP {
  ports
    detected: PresenceIPT
    userTemp: CTemperatureIPT
    target: CTemperatureOPT
}
```

### Grammar

```
ComponentDef ::=
  'boundary'? 'component' 'def' ID
'{'
  ( StructuralDef
| DataDef )*
('ports'
  PortUse*)
(Configuration)?
'}';
```

# Textual notation

## Components

### Component Definition

```
boundary component def
  TemperatureSensorCP {
ports
  current: F'TemperatureOPT
}
boundary component def HeaterCP {
ports
  controller: CommandIPT
}
```

### Grammar

```
ComponentDef ::=
  'boundary'? 'component' 'def' ID
  '{'
    ( StructuralDef
    | DataDef ) *
    ('ports'
     PortUse *)
    (Configuration)?
  '}' ;
```



# Textual notation

## Ports

### Port Definition

- **Port** defines the port name, the direction and the datatype of the flow, and the protocol that characterizes the behavioral specification of the port

### Grammar

```
SimplePortDef ::=  
  'port' 'def' ID '{'  
    'flow' FlowProperty TypeUse  
    (DataDef)*  
  '}'  
enum FlowProperty ::=  
  in = 'in' | out = 'out'
```

# Textual notation

## Ports

### Port Definition

```
port def CTemperatureOPT {  
    flow out temp:CelsiusTemperature  
}  
port def CTemperatureIPT {  
    flow in temp:CelsiusTemperature  
}  
port def CommandIPT {  
    flow in cmd:Command  
}  
port def CommandOPT {  
    flow out cmd:Command  
}  
port def PresenceIPT {  
    flow in presence:Boolean  
}  
port def PresenceOPT {  
    flow out presence:Boolean  
}
```

### Grammar

```
SimplePortDef ::=  
    'port' 'def' ID '{'  
        'flow' FlowProperty TypeUse  
        (DataDef)*  
    '}'  
enum FlowProperty ::=  
    in = 'in' | out = 'out'
```

# Textual notation

## Connectors

### Connector Definition

- **Connector** specifies the ports of the components that will participate in the connector using the **participant** clause
  - It also specifies the direction of the **flow** that traverses the connector (from in to out)
  - Note that the flow itself is implicitly typed by the flow definition of participants

### Grammar

```
ConnectorDef ::=  
  'connector' 'def' ID '{'  
    DataDef*  
    'participants' PortUse_Reverse*  
    ('flow' Flow*)?  
    Configuration?  
  '}'  
Flow ::=  
  ID 'from' ID 'to' ID
```

# Textual notation

## Connectors

### Connector Definition

```

connector def CTemperatureCN {
  participants
    source: ~CTemperatureOPT
    destination: ~CTemperatureIPT
  flow Temperature from source to destination
}
connector def DetectPresenceCN {
  participants
    source: ~PresenceOPT
    destination: ~PresenceIPT
  flow Presence from source to destination
}
connector def ControlCommandCN {
  participants
    source: ~CommandOPT
    destination: ~CommandIPT
  flow Command from source to destination
}

```

### Grammar

```

ConnectorDef ::=
  'connector' 'def' ID '{'
    DataDef*
    'participants' PortUse_Reverse*
    ('flow' Flow*)?
    Configuration?
  '}'
Flow ::=
  ID 'from' ID 'to' ID

```

# Textual notation

## Composite Components

### Composite Component Def..

- **Composite components** are components too and thereby have ports
  - As the ports of composite components are always proxy ports, they need to be delegated to the ports of internal components using the delegation clause

### Grammar

```
ComponentDef ::=  
    'boundary'? 'component' 'def' ID  
    '{'  
        ( StructuralDef  
          | DataDef ) *  
        ( 'ports'  
          PortUse * )  
        ( Configuration ) ?  
    '}' ;
```

# Textual notation

## Composite Components

### Composite Component Def..

**component def** RoomTemperatureControllerCP {

#### **ports**

localTemp1: CTemperatureIPT  
 localTemp2: CTemperatureIPT  
 detected: PresenceIPT  
 userTemp: CTemperatureIPT  
 heating: CommandOPT  
 cooling: CommandOPT

#### **components**

pc: PresenceCheckerCP  
 sm: SensorMonitorCP  
 cm: CommanderCP

#### **connectors**

target:CTemperatureCN **bind**  
 source **to** pc.target,  
 destination **to** cm.target  
 average:CTemperatureCN **bind**  
 source **to** sm.target,  
 destination **to** cm.target

#### **delegations**

localTemp1 **to** sm.s1  
 localTemp2 **to** sm.s2  
 detected **to** pc.detected  
 target **to** pc.userTemp  
 heating **to** cm.heating  
 cooling **to** cm.cooling

}

# Textual notation

## Composite Connectors

### Composite Connector Def..

- **Composite connectors**
  - As components, connectors can also be composite
  - Composite connectors can be a composition of others defined connectors

### Grammar

```
ConnectorDef ::=  
  'connector' 'def' ID '{'  
    DataDef*  
    'participants' PortUse_Reverse*  
    ('flow' Flow*)?  
    Configuration?  
  '}'  
Flow ::=  
  ID 'from' ID 'to' ID
```

# Textual notation

## Composite Connectors

### Composite Connector Def..

```
port def ClientCPT {
  ports
    query:QueryOPT
    answer:AnswerIPT
}
port def ServerCPT {
  ports
    query:QueryIPT
    answer:AnswerOPT
}
connector def ClientServerQueryCN {
  participants
    client: ~QueryOPT
    server: ~QueryPortIPT
  flow Any from source to destination
}
```

```
connector def ClientServerAnswerCN {
  participants
    client: ~AnswerOPT
    server: ~AnswerPortIPT
  flow Any from source to destination
}
connector def ClientServerCN {
  participants
    client: ~QueryCPT
    server: ~ServerCPT
  connectors
    query: ClientServerQueryCN bind
      client.query to server.query
    answer: ClientServerAnswerCN bind
      server.answer to client.answer
}
```



# Textual notation

## Composite Connectors

### Composite Connector Def..

```
datatype TemperatureDT {  
  temp:Temperature[1..*]  
}
```

```
port def TemperatureIPT {  
  flow out temp:Temperature  
}
```

```
port def TemperatureDTOPT {  
  flow out tempdt:TemperatureDT  
}
```

```
component def TemperatureAggregatorCP {  
  ports  
    p1:TemperatureIPT  
    p2:TemperatureDTOPT  
}
```

```
connector def AllTemperaturesCN {  
  participants  
    source: ~TemperatureOPT[1..*]  
    destination: ~TemperatureDTIPT  
  components  
    ta:TemperatureAggregatorCP  
  flows  
    Temperature from source to ta.p1  
    Temperature from ta.p1 to destination
```

# Textual notation

## Composite Connectors

### Composite Connector Def..

```

port def FTemperatureOPT {
  flow out temp:FahrenheitTemperature
}
port def CTemperatureIPT {
  flow in temp:CelsiusTemperature
}
port def StatusOPT {
  flow out s:Status
}
port def StatusTempCPT {
  ports
    s:StatusOPT
    t:CTemperatureIPT
}
component def FaultTolerantCP {
  ports
    sensor:TemperatureIPT
    st:~StatusTempCPT

```

```

connector def StatusTempCCN {
  participants
    source: ~FTemperatureOPT
    destination: ~StatusTempCPT[2]
  components
    ft:FaultTolerantCP
  flows
    Temperature from source to ft.sensor
    Temperature from ft.st.t to destination.t
    Status from destination.s to ft.st.s
}

```

# Textual notation

## Architecture

### Architecture Definition

- **Architecture** is defined as a special kind of composite component: it is the composite component at the root of the hierarchy of composite components representing a software architecture
- As composite components, an architecture is defined in terms of instantiated components and connectors forming a configuration
- Note that the architecture represents its interaction with the environment of the system through boundary components

### Grammar

```
ArchitectureDef ::=
  'architecture' 'def' ID '{'
    (StructuralDef
    | DataDef
    | PortUse)*
  Configuration
  '}' ;
```

```
Configuration ::=
  'components' ComponentUse*
  & 'connectors' ConnectorUse*
  & 'delegation' Delegation*
```

```
ConnectorUse ::=
  ID ':' ID 'bind' ConnectorBinding
  ("," ConnectorBinding)*
```

```
ConnectorBinding ::= ID 'to' ID
```

# Textual notation

## Architecture

### Architecture Definition

```
architecture def ARCH1 {
```

#### **components**

```
  s1:TemperatureSensorCP
    {sensorTemperatureUnit =
      TemperatureUnit::Fahrenheit}
  s2:TemperatureSensorCP
    {sensorTemperatureUnit =
      TemperatureUnit::Fahrenheit}
  s3:PresenceSensorCP
  ui:UserInterfaceCP
  a1:HeaterCP
  a2:CollerCP
  rtc:RoomTemperatureController
```

#### **connectors**

```
  c1:FahrenheitToCelsiusCN bind
    source to s1.current.heating,
    destination to rtc.localTemp1
  c2:FahrenheitToCelsiusCN bind
    source to s2.current.heating,
    destination to rtc.localTemp2
  uc:CTemperatureCN bind
    source to ui.desired,
    destination to rtc.userTemp
  pc:DetectPresenceCN bind
    source to s3.detected,
    destination to rtc.detected
  cc1:ControlCommandCN bind
    source to rtc.heating,
    destination to a1.controller
  cc2:ControlCommandCN bind
    source to rtc.cooling,
    destination to a2.controller
```



# Textual notation for behavioral viewpoint

# Textual notation

## Activities

### Activity Definition

- **Activity** specifies the behavior of components
  - They possibly execute repeatedly waiting for incoming values, executes calling internal actions when these values are received and then sends the results in outgoing flows

### Grammar

```

ActivityDef ::=
  'activity' 'def' ID '('
  ((ActivityParam) (',' (Activity-Param))* )? ')'
  '{'
  ( BehavioralDef
  | DataDef
  | ConstraintUse)*
  (ActivityBody)?
  '}'
ActivityParam ::=
  ActivityParamIn | ActivityParamOut
ActivityParamIn ::=
  'flow'? 'in' ID ':' ID
ActivityParamOut ::=
  'flow'? 'out' ID ':' ID
ConstraintUse ::=
  ConstraintKind ID '(' (ID (',' ID )* ')'
enum ConstraintKind ::=
  'pre' | 'post' | 'invariant'
  
```

# Textual notation

## Activities

### Activity Definition

- **Activity** specifies the behavior of components
  - The behavior of an activity is expressed in terms of statements (such as conditional statements, assignment, sequence, loop statements)

### Grammar

```

ActivityBody ::=
    'actions' ActionUse*
    ('data' DataObject*)?
    ('delegations' ActivityDelegation*)?
    ('flows' ActivityFlow*)?
ActionUse ::=
    ID ':' ID '(' ID (',' ID) )? ')'
    ('{' Property*
    '}' )?;
DataObject ::=
    DataStore | DataBuffer;
DataStore ::=
    'datastore' ID ':' ID ('=' Expression)?;
DataBuffer ::=
    'buffer' ID ':' ID ('=' Expression)?;
ActivityDelegation ::=
    ID 'to' ID;
ActivityFlow ::=
    ID 'from' ID 'to' ID;
  
```

# Textual notation

## Activities (examples)

### Activity Definition

```

activity def CalculateAverageTemperatureAC(
  flow in t1:CelsiusTemperature,
  flow in t2:CelsiusTemperature,
  flow out average:CelsiusTemperature) {

  actions
    av:CalculateAverageTemperatureAN(x, y)

  delegations
    t1 to av.x
    t2 to av.y
    average to av.out
}

```

```

activity def FahrenheitToCelsiusAC(
  flow in f : FahrenheitTemperature,
  flow out c : CelsiusTemperature) {

  actions
    ftc:FahrenheitToCelsiusAN(f);

  delegations
    f to ftc.f
    ftc.c to c
}

```



# Textual notation

## Activities (other examples)

### Activity Definition

```
activity def DecideCommandAC(  
  flow in averageTemp:Temperature,  
  flow in targetTemp:Temperature,  
  flow out heater:Command,  
  flow out cooler:Command) {  
  actions  
    ct:CompareTemperatureAN(t1, t2)  
    cmdHeater:CommandHeaterAN(cmds)  
    cmdCooler:CommandCoolerAN(cmds)  
  delegations  
    averageTemp to ct.t1  
    targetTemp to ct.t2  
    cmdHeater.heater to heater  
    cmdHeater.cooler to cooler  
  
  flows  
    Commands from ct.cmds to cmdHeater.cmds  
    Commands from ct.cmds to cmdCooler.cmds  
}
```

# Textual notation

## Protocols

### Protocol Definition

- **Protocol** specifies the behavior of ports
  - They possibly execute repeatedly waiting for incoming values, executes calling internal actions when these values are received and then sends the results in outgoing flows

### Grammar

```

Protocol ::=
  'protocol' ID '{'
    ProtocolBody
  '}'

ProtocolBody ::=
  ('several' | 'once' | 'perhaps' | 'always')
  (ProtocolBody | PredefinedActions |
   '(' ProtocolBody ')')
  ( ';' | '/' ) ProtocolBody?

PredefinedActions ::=
  ActionSend | ActionReceive

ActionSend ::=
  ID 'send' Expression

ActionReceive ::=
  ID 'receive' TypeUse
  
```

# Textual notation

## Protocol

### Protocol Definition

```
protocol FTemperatureOPC (  
    out temp:FTemperatureOPT {  
        several send temp  
    }  
  
protocol CTemperatureOPC(  
    out temp:CelsiusTemperature) {  
        several send temp;  
    }  
  
protocol CTemperatureIPC (  
    in temp:CTemperatureIPT) {  
        several receive temp;  
    }  
  
protocol CommandIPC (  
    in cmd:CommandIPT) {  
        several receive cmd;  
    }  
  
protocol CommandOPC (  
    out cmd: CommandOPT) {  
        several send cmd;  
    }  
  
protocol PresenceIPC (in presence:PresenceIPT) {  
        several receive presence;  
    }  
  
protocol PresenceOPC (out presence:PresenceOPT {  
        several send presence;  
    }  
}
```

# Textual notation

## Actions

### Action Definition

- **Action** is the basic unit of executable functionality
  - The execution semantics of actions is the one of start-stop execution: it executes from beginning to end, atomically, only once when called
  - An action forms thereby an abstraction of a computation which is an atomic execution and therefore completes without interruption

### Grammar

```

ActionDef ::=
  'action' 'def' ID '(' (ActionPin
    (',' ActionPin)*)? ')' (':' ID)?
  '{'
  ( DataDef
    | ConstraintUse)*
  '}'

ConstraintUse ::=
  ConstraintKind ID '(' ID (',' ID)* ')'
  /* the first ID refers to a constraint ID */

ActionPin ::=
  'in'/'inout' ID ':' ID

enum ConstraintKind:
  'pre' | 'post' | 'invariant'
  
```

# Textual notation

## Actions (examples)

### Action Definition

```
action def CalculateAverageTemperatureAN (  
  t1: CelsiusTemperature,  
  t2: CelsiusTemperature,  
  result: CelsiusTemperature) {  
  post CalculateAverageTemperatureEQ(t1, t2,  
    result)  
}
```

```
action def FahrenheitToCelsiusAN (  
  f: FahrenheitTemperature,  
  c: CelsiusTemperature) {  
  post FahrenheitToCelsiusEQ(f, c)  
}
```

# Textual notation

## Constraints

### Constraint Definition

- **Constraint** is the logical expression for specifying pre- and post-conditions
- Examples:

#### **constraint def**

```
CalculateAverageTemperatureEQ(
  t1:CelsiusTemperature,
  t2: CelsiusTemperature,
  average: CelsiusTemperature) {
  average==(t1+t2)/2;
}
```

#### **constraint def** FahrenheitToCelsiusEQ (

```
  f: FahrenheitTemperature,
  c: CelsiusTemperature) {
  c == (5 * (f - 32) / 9);
}
```

### Grammar

```
ConstraintDef ::=
  'constraint' 'def' ID '('(' TypeUse
  (',' TypeUse)*')' '{'
  (BooleanExpression)
  '}'
```



# Textual notation for executable viewpoint

# Textual notation

## Executable

### Executable Definition

- **Executable** is used to define the executable body of an action

- Examples:

```
executable def
CalculateAverageTemperatureEX(
  t1: CelsiusTemperature,
  t2: CelsiusTemperature):
  CelsiusTemperature {
return ((t1 + t2)/2);
}
executable def FahrenheitToCelsiusEX(
  f:FahrenheitTemperature):
  CelsiusTemperature {
return 5*(f - 32)/9;
}
```

### Grammar

```
ExecutableDef ::=
  'executable' 'def' ID '(' (ActionPin
    (',' ActionPin)*)? ')' (':' ID)? '{'
    Statements*
  '}'
```



# Textual notation

## Executable (examples 1/3)

### Activity Definition

```
executable def AggregateTemperaturesEX(temps:Temperature[1..*]):TemperatureDT {  
    let allTemp:TemperatureDT = ;  
    let index:Integer=1;  
    for (t in temps) {  
        allTemp[index] = t;  
        index = index + 1;  
    }  
    return allTemp;  
}
```

```
executable def CalculateAverageTemperatureEX(alltemp:TemperatureDT):Temperature {  
    let sum:CelsiusTemperature=0;  
    let numOfElem:Integer=0;  
    for (t in alltemp) {  
        sum = sum + t;  
        numOfElem = numOfElem + 1;  
    }  
    return sum/numOfElem;
```

# Textual notation

## Executable (examples 2/3)

### Activity Definition

```
executable def CalculateAverageTemperatureEX(  
    alltemp: TemperatureDT): Temperature {  
    return alltemp->sum()/alltemp->size();  
}
```

```
executable def CheckPresenceToSetTemperatureEX(  
    presence: Boolean,  
    desiredTemp: CelsiusTemperature)  
    result: CelsiusTemperature) {  
    if (presence) {  
        return desiredTemp;  
    } else {  
        return 22;  
    }  
}
```

# Textual notation

## Executable (examples 3/3)

### Activity Definition

```
executable def DecideCommandEX (  
  in averageTemp:CelsiusTemperature,  
  in targetTemp:CelsiusTemperature):Commands {  
  
  let heater:Command = Command::off;  
  let cooler:Command = Command::off;  
  
  if (averageTemp > targetTemp) {  
    heater = Command::off  
    cooler = Command::on  
  } else if averageTemp < targetTemp {  
    heater = Command::on  
    cooler = Command::off  
  }  
  return new Commands(heater=>heater, cooler=>cooler)
```

# Summing up (keywords)

Keyword	Concept
property	Defines an annotation for any named element.
valuetype	Defines a quantity with a dimension and a unit.
datatype	Defines a data type
enum	Enumeration
port def	Port definition
protocol	Protocol definition
several	Specify that an action or a sequence of actions will be executed several times (none, one, or many times) in the protocol behavior
once	Specify that an action or a sequence of actions will be executed once in the protocol behavior
perhaps	Specify that an action or a sequence of actions will be executed once or not in the protocol behavior

# Summing up (keywords)

Keyword	Concept
connector def	Connector definition
participants	Participant ports of a connector
activity def	Activity definition
in, out	Data direction: input/output
invariant	Define the conditions that must be true during the execution of the action or activity
pre	Define the conditions that must be true before the execution of the action
post	Define the conditions that must be true after the execution of the action
action def	Action definition
constraint def	Constraint definition

# Summing up (keywords)

Keyword	Concept
component def	Component definition
boundary	Boundary component
architecture def	Architecture definition
bind	Define a binding between connectors' and components' ports
while	Control loop based on a boolean expression
for	Control loop based on enumerated values
if	Conditional statement
switch	Multiple-choice conditional statement
select	Multiple-choice for port synchronization

# Summary

- In this chapter, you learned how to:
  - organize architectural definitions in the form of a textual description
  - the SysADL textual notation to express the architecture provided by the executable viewpoint

# For further reading

- Concrete Syntax For A UML Action Language: Action Language For Foundational UML™ (ALF™).  
<http://www.omg.org/spec/ALF/>
- Semantics Of A Foundational Subset For Executable UML Models (FUML™). <http://www.omg.org/spec/FUML/>
- Precise Semantics Of UML Composite Structures™ (PSCS™).  
<http://www.omg.org/spec/PSCS/1.0/>