Working with strongly typed data models in Lua for building Industrial Internet of Things (IIoT) applications

Rajive Joshi, Ph. D.
Principal Solution Architect, Real-Time Innovations, Inc.
Agenda

• Background
• Problem
• Solution
• Tutorial | Demo
• Applications
• Next Steps
Background

What we do?
The Industrial Internet of Things

Consumer Internet of Things (CIoT)

Cyber-Physical Systems (CPS)
The Industrial Internet of Things

Consumer Internet of Things (CIoT)

Industrial Internet of Things (IIoT)

Cyber-Physical Systems (CPS)
RTI Excels at Industrial IoT (IIoT)

• ~1000 Projects
  – Healthcare
  – Transportation
  – Communications
  – Energy
  – Industrial
  – Defense

• 15+ Standards & Consortia Efforts
  – Interoperability
  – Multi-vendor ecosystems
Connect Anything, Anywhere - Intelligently!

RTI Connext DDS “DataBus”

Seamless data sharing regardless of:

- Proximity
- Platform
- Language
- Physical network
- Transport protocol
- Network topology

✓ Reliable
✓ Real-Time
✓ Scalable
✓ Available
✓ Resilient
✓ Secure
✓ Safe
✓ Composable
RTI Connext Platform is built on the DDS Standard

http://portals.omg.org/dds/
Problem Definition

“Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming. (See Brooks p. 102.)”
Data-Centric Systems: Shared Data Space

Component $X_1$ → DataWriter $\text{write}()$ → Data-Objects (Topic) → DataReader $\text{take}()$ $\text{read}()$ → Component $Y_1$
Data-Centric Systems: Shared Data Space

Component $X_1$ → DataWriter → Data-Objects (Topic) → DataReader → Component $Y_1$

... → DataWriter → Data-Objects (Topic) → DataReader → Component $Y_M$

Component $X_N$ → Data-Objects (Topic) → DataReader → Component $Y_1$
Data-Centric Systems: Shared Data Space

Multiple platforms, Multiple Languages

Pick the language most suitable to the requirements of a component
Data-Centric Systems: Data Models

struct ShapeType {
    long x;
    long y;
    double shapesize;
    string<128> color;  // @Key
};
Data-Centric Systems: Data Models

Interface Definition Language (IDL)
XML with Equivalent Schema

DDS X-Types
Specification

Data Model
(Datatype)

struct ShapeType {
    long x;
    long y;
    double shapesize;
    string<128> color;    //@Key
};

C/C++
Component \(X_1\)

.Net/C#
Component \(X_N\)

Java
Component \(Y_1\)

Lua
Component \(Y_M\)
Data-Centric Systems: Data Models

DDS X-Types Specification

Data Model (Datatype)

```
struct ShapeType {
  long x;
  long y;
  double shapesize;
  string<128> color; //@Key
};
```

Interface Definition Language (IDL)
XML with Equivalent Schema
Data Model Constrains the Data-Objects

Data Model (Datatype)

```c
struct ShapeType {
    long x;
    long y;
    double shapesize;
    string<128> color; //@Key
};
```

Language Neutral

Compiled in a statically typed language

Language Specific

Static: C/C++, Java, .NET/C#, Ada, ...
Dynamic: Lua, Python, Javascript, ...

©2015 Real-Time Innovations, Inc.
Data Model Constrains the Data-Objects

struct ShapeType {
    long x;
    long y;
    double shapesize;
    string<128> color; //@Key
};

Language Neutral

Constrains

How to enforce in dynamically typed languages?

Data-Objects (Topic)

Language Specific

Static: C/C++, Java, .NET/C#, Ada, ...
Dynamic: Lua, Python, Javascript, ...
Solution

DDSL: Data Domain Specific Language
DDSL: Data Domain Specific Language

An algebra for defining, using, and manipulating X-Types (in Lua)

struct ShapeType {
  long x;
  long y;
  double shapesize;
  string<128> color;  //@Key
};

template (Lua)

Data Model (Datatype)

Data-Objects (Topic)

DDSL
Enforces Constraints
Keeps Synchronized

Create & Modify
• Datatypes
• Instances

Index data fields

instances (Lua)

©2015 Real-Time Innovations, Inc.
Data models can be very complex....

**Primitive**
- Atomic (long, double, string, etc...)
- Const
- Enums

**Structural**
- Structs
- Unions
- Sequences (multidimensional)
- Arrays (multidimensional)

**Organizational**
- Typedefs
- Module
- Annotations
Without DDSL...

• The datatype must be maintained outside the scripting language environment
  – not accessible to program (in the language)

• The datatype must be known to the programmer, and its structure hard-coded in the script
  – error-prone due to duplication
  – brittle to datatype evolution or large/complex types

• A script cannot use or introduce new topics or endpoints (not known to the programmer)
  – impossible to build generic components
With DDSL...

• The datatype can be maintained inside the scripting environment
  — fully accessible to program
• The datatype need not be known to the programmer, and its structure need not be hard-coded in the script
  — safe, single version of truth (no datatype duplication)
  — robust to datatype evolution or large/complex types
• A script can use or introduce new topics or endpoints (not known to the programmer)
  — possible to build generic components
Quick Example...

Data Model - IDL

struct ShapeType {
    long x;
    long y;
    double shapesize;
    string<128> color; //@Key
};

Data-Object - Lua

shape = {
    x = 50,
    y = 30,
    shapesize = 20,
    color = 'GREEN'
}
Quick Example...

**Data Model - IDL**

```c
struct ShapeType {
    long x;
    long y;
    double shapesize;
    string<128> color; //@Key
};
```

**Data Model – DDSL (template)**

```c
local ShapeType = xtypes.struct{
    ShapeType = {
        { x = { xtypes.long } },
        { y = { xtypes.long } },
        { shapesize = { xtypes.long } },
        { color = { xtypes.string(128), xtypes.Key } },
    }
}
```

**Data-Object – DDSL (instance)**

```c
local shape = xtypes.new_instance(ShapeType)
shape.x = 50
shape.y = 30
shape.shapesize = 20
shape.color = 'GREEN'
```

**Data-Object – Lua**

```lua
shape = {
    x = 50,
    y = 30,
    shapesize = 20,
    color = 'GREEN'
}
```
DDSL opens up new possibilities...

• A DDSL datatype
  – Can be examined and traversed in the program
  – Can be defined and manipulated by the program
    • create|read|update|modify operations
  – Can be synthesized and manipulated by the program
  – Can be used to automate code or data generation while using the full power and flexibility of Lua

• Brings datatypes to every IoT platform
  – DDSL is platform independent
    • is written in pure Lua with no external dependencies (not even DDS!)
    • can be used on any platform on which Lua can be used (practically everywhere)
  – Even web-browsers (e.g. using lua.vm.js)
An algebra for defining, using, and manipulating X-Types (in Lua)
So, what exactly is DDSL?

1. Replacement for IDL, by allowing datatypes definitions in Lua
2. Factory of data-objects (instances) that adhere to a datatype
3. An algebra to operate on datatypes and maintain the associated data-objects (instances)
4. Generator of accessor strings for indexing members in a data-object store
5. An extensible framework for data modeling
1. IDL Replacement in Lua

- OMG DDS X-Types Specification in Lua
  - including features not yet supported in IDL
    - e.g. custom annotations
- XML → DDSL
  - Import datatypes defined in XML
- DDSL → IDL (→ XML)
  - Export datatypes to IDL
- Easy to define new kinds of datatypes
  - Useful for experimenting with new X-Types features
    - e.g. choice
1. IDL Replacement in Lua

**Declarative style (Datatype Constructor)**

```lua
local MAX_COLOR_LEN = xtypes.const{ MAX_COLOR_LEN = { xtypes.long, 128 } }

local ShapeType = xtypes.struct{
    ShapeType = {
        { x = { xtypes.long } },
        { y = { xtypes.long } },
        { shapesize = { xtypes.long } },
        { color = { xtypes.string(MAX_COLOR_LEN), xtypes.Key } },
    }
}
```

**Imperative Style (Datatype CRUD)**

```lua
local MAX_COLOR_LEN = xtypes.const{ MAX_COLOR_LEN = { xtypes.long, 128 } }

local ShapeType = xtypes.struct{ShapeType=xtypes.EMPTY}
ShapeType[1] = { x = { xtypes.long } }
ShapeType[2] = { y = { xtypes.long } }
ShapeType[3] = { shapesize = { xtypes.long } }
ShapeType[4] = { color = { xtypes.string(MAX_COLOR_LEN), xtypes.Key } }
```
2. Factory of instances

• Instances are constrained by the datatype
  – arbitrary members cannot be added to instances
  – members may be assigned nil (e.g. optional); and later re-assigned a non-nil value
  – collection (array and sequence) bounds are enforced; now allowed to exceed the max capacity

• Yet, instance performance is on par with a manually defined instance
  – no overhead in setting/getting member values
    • it was an explicit design goal to not add any overhead over raw member field setters/getters
  – only structural constraints are enforced!
    • type-checking is not enforced for member value assignment (although an application could easily do so if desired)
2. Factory of instances

-- shape is equivalent to manually defining the following --
local shape_manual = {
  x = 50,
  y = 30,
  shapesize = 20,
  color = 'GREEN',
}

local shape = xtypes.new_instance(ShapeType)
print("--- Iterate through instance members : unordered ---")
for role, _ in pairs(shape_manual) do
  shape[role] = shape_manual[role]
  print('\t', role, shape[role])
end

⇒

--- Iterate through instance members : unordered ---
color  GREEN
  x  50
shapesize  20
3. Datatype Algebra

• Datatypes can be mutated (unlike IDL/XML)
  – the “kind” (meta-datatype) is immutable and defines the structure and constraints
  – all other aspects of a datatype can be changed, e.g.: name, members, annotations, inheritance, switch

• Datatype changes are propagated to instances (unlike IDL/XML)
  – adding a member to the datatype also adds it to all the instances (with a default value)
  – removing a member from a the datatype also removes it from all the instances
  – changing a member's datatype propagates the change to all the instances
### 3. Datatype Algebra

**Models and Instances**

A DDSL instance has two faces (like a coin):
- an underlying datatype model (the blueprint)
- fields with role names dictated by the datatype

Constructors return a special “template” instance (e.g. MyType) as a handle to the underlying model.

Datatype models are manipulated using operators on instances. Any of the instances could be used, but its idiomatic to use the template instance.

Template instances link model elements to form a data model graph.
3. Datatype Algebra

A DDSL instance has two faces (like a coin):
- an underlying datatype model (the blueprint)
- fields with role names dictated by the datatype

Constructors return a special “template” instance (e.g. MyType) as a handle to the underlying model.

Datatype models are manipulated using operators on instances. Any of the instances could be used, but its idiomatic to use the template instance.

Template instances link model elements to form a data model graph.
### 3. Datatype Algebra: Model Operators

print('--- add member z ---')
ShapeType[#ShapeType+1] = { z = { xtypes.string(), xtypes.Key } }
print_datatype(ShapeType, 'ShapeType:')

print('--- remove member x ---')
ShapeType[1] = nil
print_datatype(ShapeType, 'ShapeType:')

print('--- redefine member y ---')
ShapeType[1] = { y = { xtypes.double } }
print_datatype(ShapeType, 'ShapeType:')

print('--- add a base struct ---')
local Property = xtypes.struct{
    Property = {
        { name = { xtypes.string(MAX_COLOR_LEN) } },
        { value = { xtypes.string(MAX_COLOR_LEN) } },
    }
}
ShapeType[xtypes.BASE] = Property
print_datatype(ShapeType, 'PropertyShapeType:')
print_instance(ShapeType, 'PropertyShapeType accessors:')
4. Generator of Accessor Strings

- Datatype is a special ‘template’ instance
  - whose members are initialized to the accessor strings for a DDS
    DynamicData object
  - automatically maintained to confirm with the underlying data
    model
  - for efficiency, collection member elements are allocated (at most
    once) on demand (i.e. when accessed)

```plaintext
-- ShapeType can be accessed as if it was defined as --
ShapeType = {
    x = 'x',
    y = 'y',
    shapesize = 'shapesize',
    color = 'color'
}
```
4. An extensible framework for data modeling

• Core engine supports a generic meta-model
  – Agnostic to specific data modeling language constraints
  – X-Types module extends the ddsI engine to support DDS X-Types (OMG IDL)

• Easy to extend the modeling language
  – New atomic data types, annotations, compositions, ...

• Easy to add new constraints
  – Data ranges
  – Data units
  – Semantics
Datatype Algebra
DDSL Nomenclature

**MyType API**

- i-th member: MyType[i] = { role = { ... } }
- #members: #MyType
- name: MyType[NAME]
- namespace: MyType[NS]
- kind: MyType[KIND]
- (): capacity | const value | alias
- qualifiers | base | switch: MyType[QUALIFIER | BASE | SWITCH]

**Nomenclature**

<table>
<thead>
<tr>
<th>General Concept</th>
<th>DDSL Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance (a.k.a. data-object)</td>
<td>instance</td>
</tr>
<tr>
<td>datatype</td>
<td>template</td>
</tr>
<tr>
<td>member</td>
<td></td>
</tr>
<tr>
<td>member name</td>
<td>role</td>
</tr>
<tr>
<td>member definition</td>
<td>role definition (roledef)</td>
</tr>
<tr>
<td>annotation</td>
<td>qualifier</td>
</tr>
</tbody>
</table>

**MyType**

```
new_instance()
```

```
myInstance
```

**metatable**

```
new_instance()
```

```
myInstance
```

```
member_role_1
```

```
str | tbl
```

```
member_role_M
```

```
str | tbl
```

```
: 
```

```
member_role_1
```

```
value | tbl
```

```
member_role_M
```

```
value | tbl
```

```
: 
```

```
myInstance
```

```
member_role_1
```

```
value | tbl
```

```
member_role_M
```

```
value | tbl
```

```
: 
```
Datatype Algebra - Model Constructors

Create a datatype template:

```
MyType = xtypes.<kind>{...}
```

where, `<kind>` is one of:

- annotation,
- const, enum,
- struct, union,
- module,
- typedef

and `{...}` depends on `<kind>`

E.g.

```
MyType = xtypes.struct{MyType=xtypes.EMPTY}
```

Common definition style:

```
{ <name> = { <definition> } }
```
Datatype Algebra - Model Operators

Create a member:
    MyType[i] = { role = {template, [array|sequence,] [[annotation,]...]} }

Delete a member:
    MyType[i] = nil

Namespace, Qualifiers, Base (structs), Switch (unions):
    MyType[xtypes.NS] = AnotherType
    MyType[xtypes.QUALIFIERS] = { annotation, ... }
    MyType[xtypes.BASE] = BaseType
    MyType[xtypes.SWITCH] = DiscriminatorType

Name:
    MyType[xtypes.NAME] = ‘NewTypeName’

```
tostring(MyType)
```

Kind (immutable):
    MyType[xtypes.KIND]
Iterate over the members:

```latex
for i = 1, #MyType do
  local role, roledef = next(MyType[i])
  print(role, table.unpack(roledef))
end
```

Iterate over the members:

```latex
for i = 1, #MyType do
  local case = MyType[i]

  -- case discriminators:
  print(table.unpack(case))

  -- member:
  local role, roledef = next(case, #case)
  print(role, table.unpack(roledef))
end
```
Create an instance:

```python
myInstance = xtypes.new_instance(MyType)
```

Create a collection of instances:

```python
myCollection = xtypes.new_collection(MyType, capacity)
```

NOTE: A sequence or array member in an instance is a collection
Is it a collection?
    if xtypes.is_collection(myCollection) then
        print( 'capacity', myCollection() )
        print( 'length', #myCollection )
    end

Iterate over a collection’s length:
    for i = 1, #myCollection do
        print( myCollection[i] )
    end

Iterate over a collection’s capacity:
    for i = 1, myCollection() do
        print( myCollection[i] )
    end
Datatype Algebra - Instance Iterators

Iterate over instance members (unordered):

    for role, value in pairs(myInstance) do
        print(role, value)
    end

Iterate over instance members (ordered):

    for i = 1, #MyType do
        local element_i, role = MyType[i]

        role = next(element_i)          -- struct
        role = next(element_i, #element_i)-- union

        local value = myInstance[role]
        print(role, value, table.unpack(MyType(role)))
    end
### Datatype Algebra – Template Instance

Iterate over DDS DynamicData accessor strings (unordered):

```plaintext
for role, accessor in pairs(MyType) do
    print(role, accessor)
end
```

Iterate over DDS DynamicData accessor strings (ordered):

```plaintext
for i = 1, #MyType do
    local element_i, role = MyType[i]
    role = next(element_i)    -- struct
    role = next(element_i, #element_i)-- union
    local accessor = MyType[role]
    print(role, accessor)
end
```

Field values are accessor strings for indexing members in a data-object store.
Datatype Algebra – Misc Operators

Get the fully qualified name within the namespaces

```python
print( xtypes.nsname(MyType) )
```

Resolve an alias (e.g. typedef) to its underlying datatype

```python
print( xtypes.resolve(MyType) )
```

Get an instance’s underlying datatype template

```python
print( xtypes.template(myInstance) )
```
Datatype Algebra – Utilities

- `ddsl.xtypes.utils`
  - `to_instance_string_table(instance)`
  - `to_idl_string_table(instance)`
  - `nslookup(name, ns)`

- `ddsl.xtypes.xml`
  - `filelist2xtypes(filename_array)`
  - `file2xtypes(filename)`
  - `string2xtypes(xmlstring)`

- Command line
  - `run xml2ddsl [-d] <xml-file> [ <xml-files> ... ]`
Running the tutorial...

git clone github.com/rticommunity/rticonnextdds-ddsl/

export DDSL_HOME=$(pwd -P)
export LUA_PATH+="${DDSL_HOME}tutorial/?.lua;"

cd tutorial/
lua dds1_tutorial.lua
Applications
Current Applications (that I am aware of)

• Data Emulator
  – RTI Experimental Product
    • Emulate data-centric interfaces
    • First drop to customer (2015 Jun 30)

• Data Generators for Reactive Programming
  – Contribued by Sumant Tambe
  – Useful for simulation and testing
Potential Applications

• Community of **open data models** for [Industrial] IoT
• Data serialization and deserialization
• Code generation
  – Integration with templating libraries
• Model mapping and generation
  – synthesize datatypes from **semantic** models
  – synthesize data-model mappings
• Synthesis of data models
  – machine learning
• Experimentation with New Data Modeling Ideas
Next Steps

• Getting Started Guide
  – README
    • https://github.com/rticomunity/rticonnextdds-ddsl

• Users Manual
  – API
    • http://rticomunity.github.io/rticonnextdds-ddsl/

• Unit Tests
  – test/ (for more advanced examples)
    • ddsl-xtypes-tester.lua
Thank You!

rajive@rti.com  |  @RajiveJoshi

We are Hiring!

www.rti.com/company/careers.html