Extensible and Dynamic Topic Types for DDS

MARS – Long Beach, CA – Dec., 2009

Presenter: Rick Warren, RTI

Submitting POCs:

- Angelo Corsaro, PrismTech: angelo.corsaro@prismtech.com
- Gerardo Pardo, RTI: gerardo.pardo@rti.com
- Robert Poth, PrismTech: robert.poth@prismtech.com
- Rick Warren, RTI: rick.warren@rti.com

Supporting POCs:

- VirginieWatine, Thales: virginie.watine@thalesgroup.com

document number: 2009-12-01

DDS Types: Assessment and Challenges

Good

- Unmatched type safety => fewer defects
- Unique type awareness => faster integration
- Unique type awareness => higher performance

Bad

- No standard API to define / access types at runtime
- Hard to change types without rebuilding, redeploying
- Hard to upgrade systems one subsystem at a time

Vendor-Specific Solutions

OMG-Standard Solutions

Where We Are

- RFP issued: Jun. 2008
- Initial submission: Nov 2008
- Revised submission extended: Feb. 2009
- Revised submission extended: Aug. 2009
- **Revised submission deadline: Nov. 2009**

Requirements: Type System

- **Extensible, Compatible**
  - 6.5.1: (*a*) *Type System* as UML meta-model.
    (*b*) Type *substitutability* rules.
  - 6.5.2: Type *extensibility*, *mutability* rules
    - 6.5.3: Type *compatibility* in presence of change
    - 6.5.17: Allow type to *evolve* without breaking interoperability
  - 6.5.5: Support *keys*. How does extensibility apply?

- **More Expressive**
  - 6.5.4: *Map* data type
  - 6.5.6, 6.5.19: Sparse data: object contains only *subset of fields* defined by its type

Requirements: Working with Types

- **Static Definition**
  - 6.5.10: *Built-in types*: map, octet sequence, string sequence
  - 6.5.11: *Programming-language-independent* type serialization format(s)
    - 6.5.9: Represent types in *IDL* and *XML*
- **Dynamic Definition**
  - 6.5.20: (*a*) *Define types dynamically*.
    (*b*) *Create topics* based on them.
  - 6.5.12: Type *discovery*
  - 6.5.13: Type *introspection*
  - 6.5.21: *All pub-sub operations* using dynamically defined types

Requirements: Working with Data

- **Dynamic Access**
  - 6.5.18: Dynamic data access
  - 6.6.1: *Optional*: Dynamic data access based on XML or JSON

- **Network Encapsulation**
  - 6.5.14: Encapsulation specification, negotiation
    - 6.5.15: Is encapsulation request-offer?
  - 6.6.2: *Optional*: Custom encapsulation API

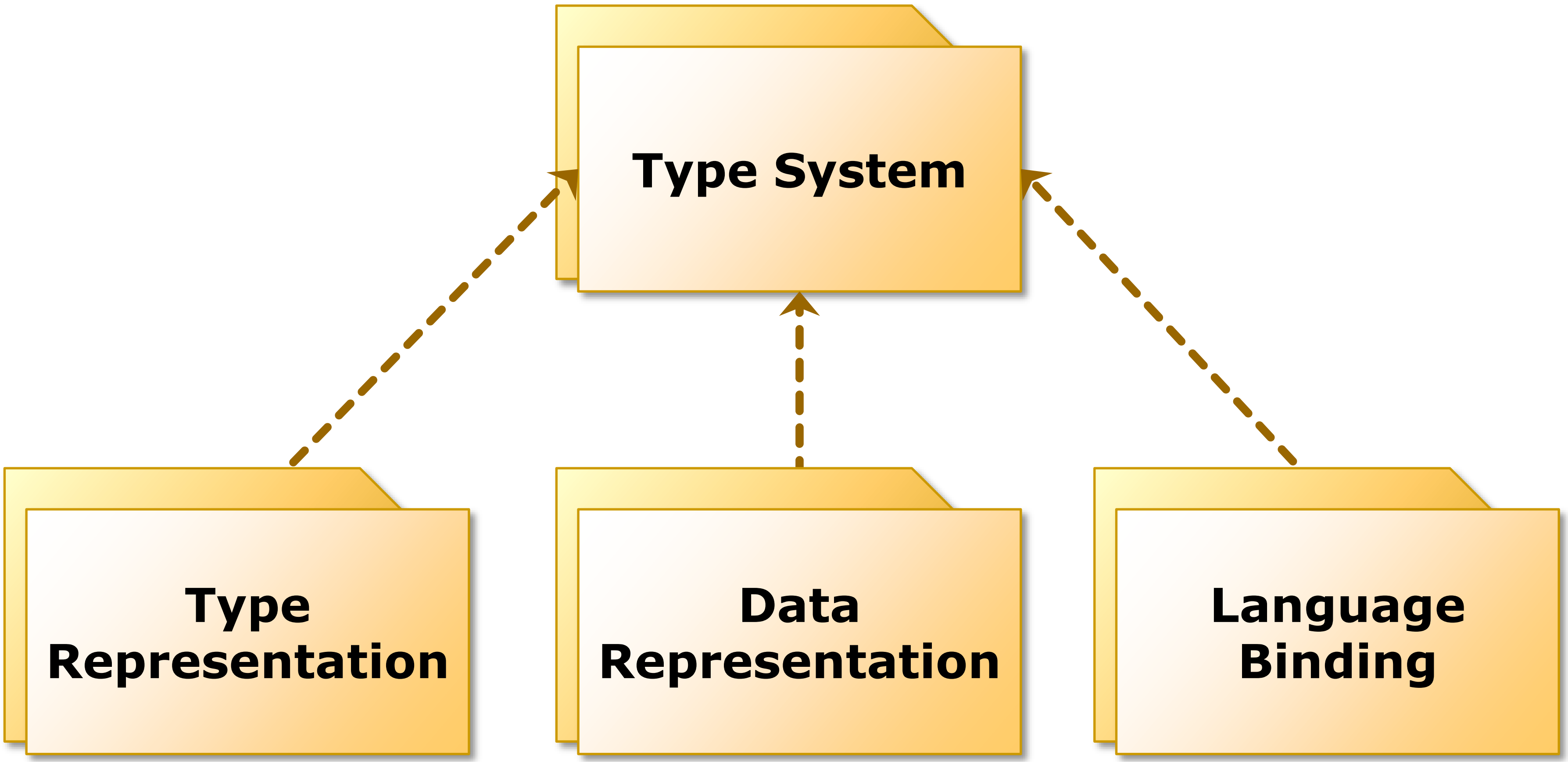
Requirements: Compatibility

- **Type Representation Level**
  - 6.5.7: Support subset of IDL used by DDS 1.2
- **Programming Level**
  - 6.5.8: APIs for C, C++, Java
- **Network Level**
  - 6.5.16: (*a*) Support CDR.
    (*b*) Support parameterized CDR.
    (*c*) Interoperate with RTPS/DDSI 2.

Design Principles

- **Don't break anybody.**
  - Many people depend on DDS
    - Don't modify existing APIs
    - Don't modify existing compliance points
  - Many people depend on RTPS/DDSI 2
    - Don't modify existing protocols or formats
    - Don't modify existing compliance points
- **Keep it fast.**
  - New capabilities must permit efficient implementations.
  - Don't use them? Don't pay for them.
- **Keep it orthogonal.** Type compatibility doesn't depend on:
  - Types' definition language(s)
  - Data encapsulation(s)
  - Programming language(s)
  - QoS configuration

Overview



Type System
Type Representation
Data Representation
Language Binding

Overview

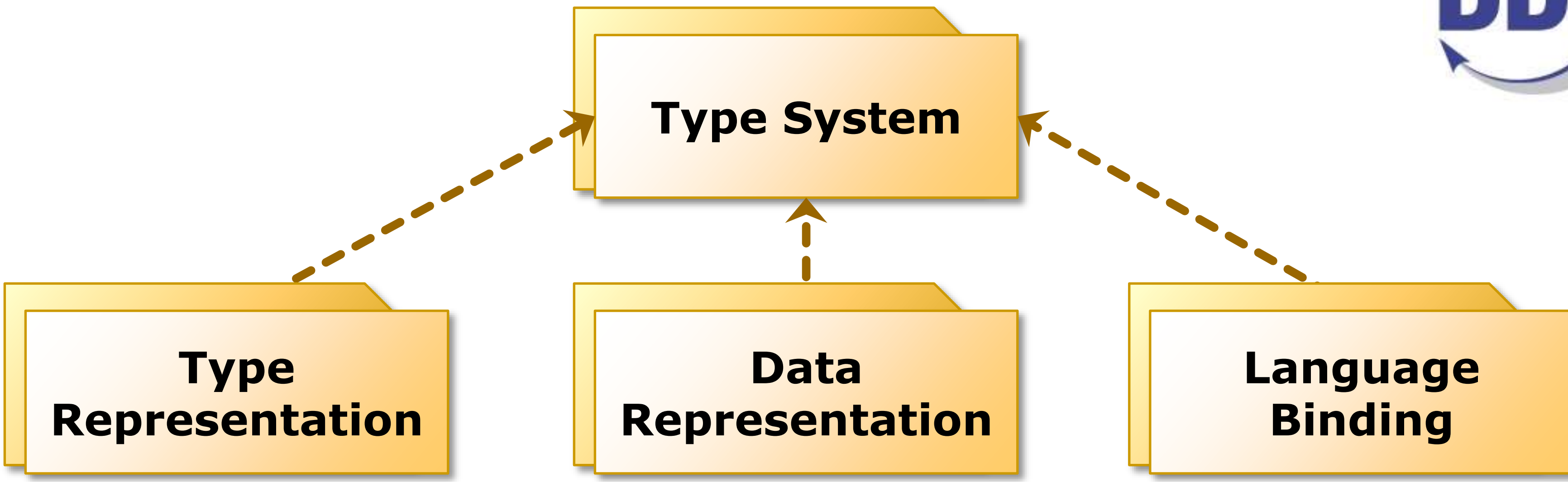
1. **Type System**: abstract definition of what types can exist
   - Expressed as UML *meta-model*
   - Including *substitutability*, *compatibility* rules
   - Mostly *familiar* from IDL
2. **Type Representations**: languages for describing types
   - *IDL*
   - *XML* and *XSD*
   - *TypeCode*
3. **Data Representations**: languages for describing data
   - *CDR*
   - *XML*

Overview

4. **Language Binding**: programming APIs
   - *"Plain Language"*: extension of existing IDL-to-*language* bindings
   - *Dynamic*: reflective API for types and objects, defined in UML (conceptual model) and IDL (API)

5. **Use by DDS**: application of type/data representations to middleware
   - Data *encapsulation*, QoS *compatibility*
   - *Type compatibility* as applied to endpoint matching
   - *Built-in types*

Overview



- **Type System**: DDS data objects have a type
- **Language Binding**: Objects are manipulated using a Language Binding to some programming language
- **Data Representation**: Objects can be serialized for file storage and network transmission

- **Language Binding**: Types are manipulated using a Language Binding to some programming language
- **Type Representation**: Types can be serialized for file storage and network transmission

Overview: Example

Type Representation

IDL:
Foo.idl

```
struct Foo {
 string name;
 long ssn;
};
```

Language Binding

IDL to Language Mapping:
Foo.h
Foo.c
FooTypeSupport.c

```
    struct Foo {
      char *name;
      int ssn;
    };

Foo f = {"hello", 2};
```

Data Representation

IDL to CDR:

```
00000006
68656C6C
6F000000
00000002
```

Type System

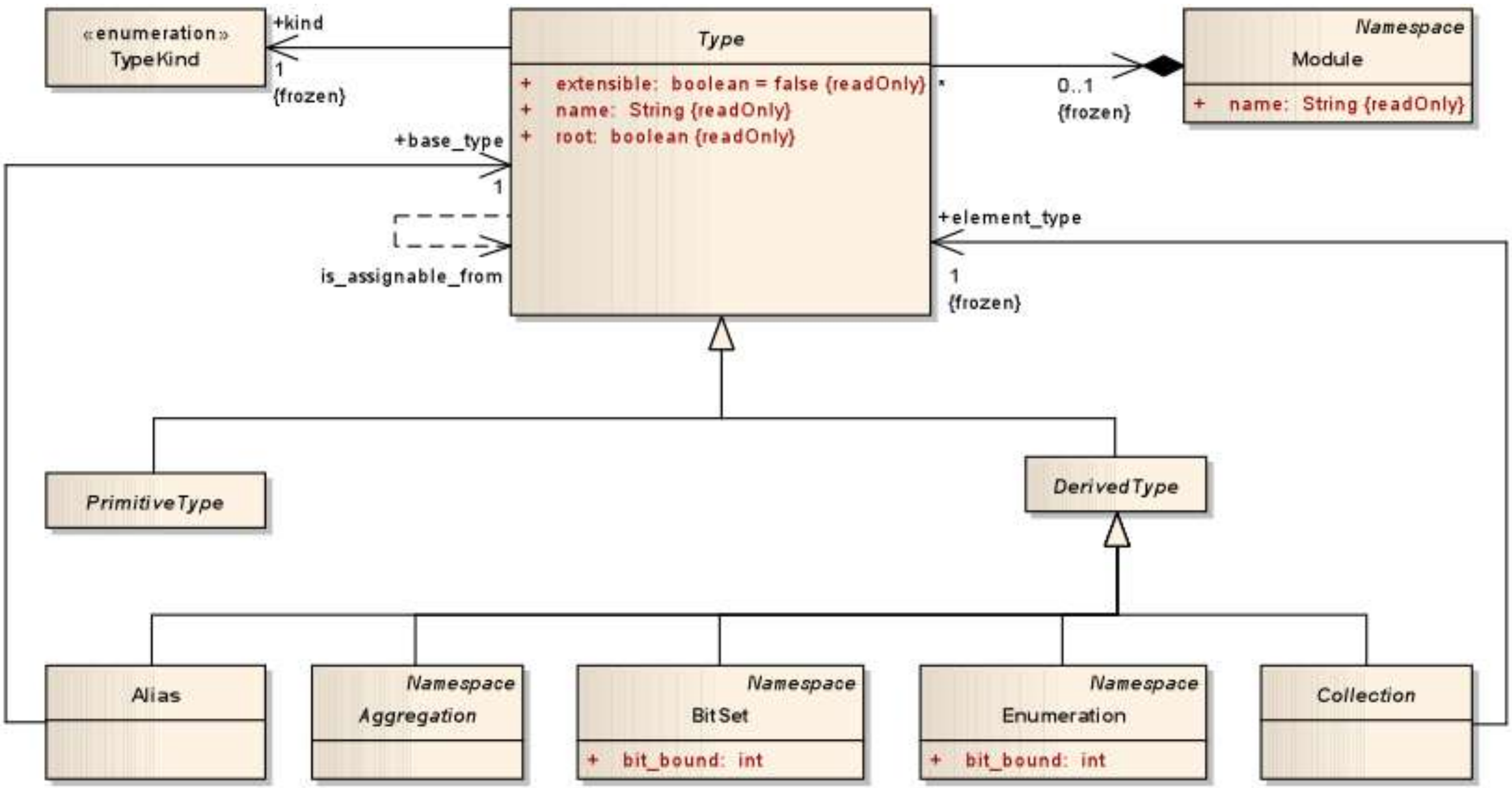
- **Extensible, Compatible**
  - 6.5.1: (*a*) *Type System* as UML meta-model.
    (*b*) Type *substitutability* rules.
  - 6.5.2: Type *extensibility*, *mutability* rules
    - 6.5.3: Type *compatibility* in presence of change
    - 6.5.17: Allow type to *evolve* without breaking interoperability
  - 6.5.5: Support *keys*. How does extensibility apply?
- **More Expressive**
  - 6.5.4: *Map* data type
  - 6.5.6, 6.5.19: Sparse data: object contains only *subset of fields* defined by its type

The Example

- Radar system uses “Original Land Data” (OLD):

```
structOriginalLandData {
    long x;
    long y;
};
```

Type System Overview



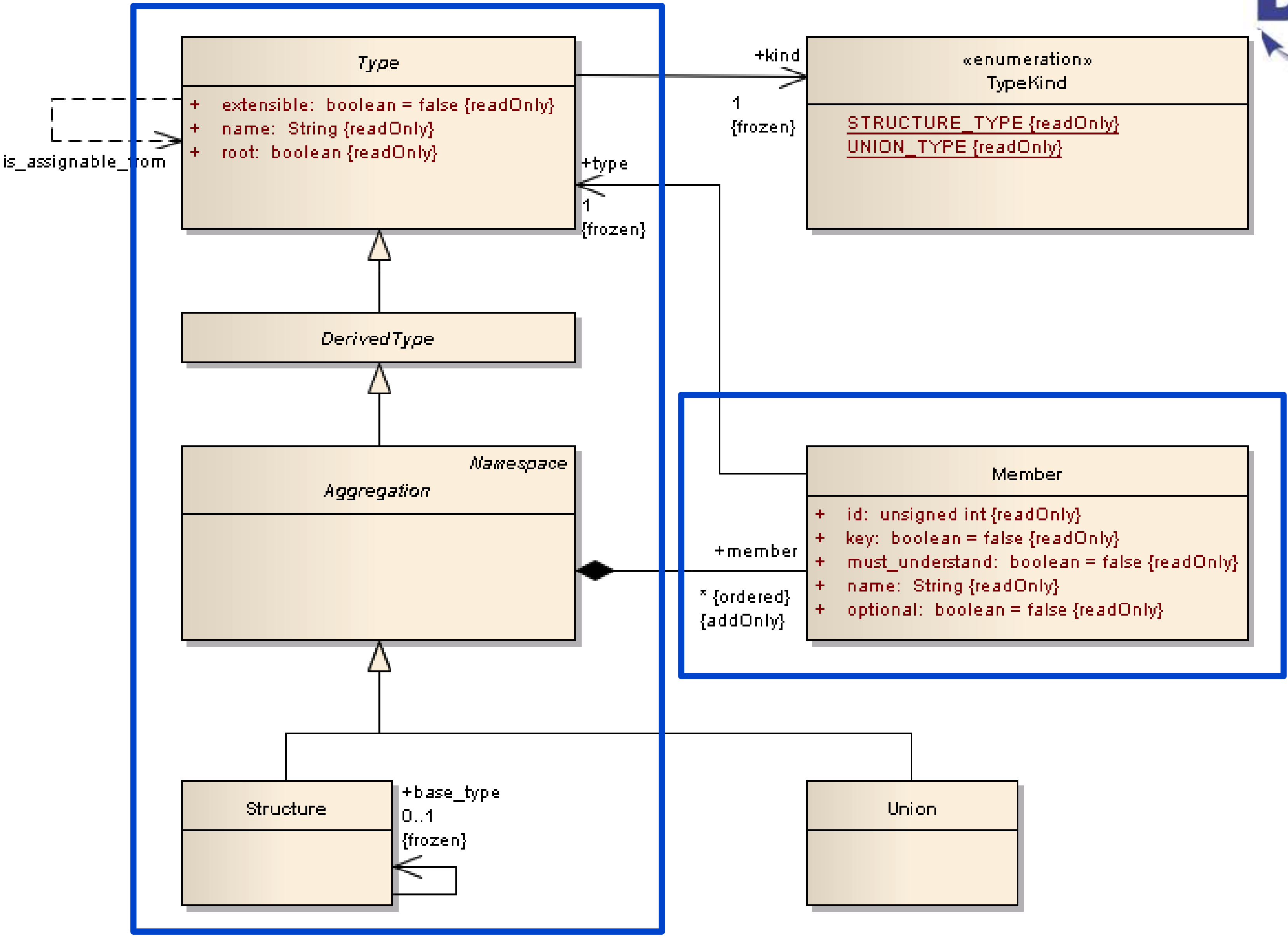
Type System Overview

* *Type System in not defined in terms of IDL. I'm explaining in terms of IDL for clarity.*

- Entirely familiar from IDL:
  - Primitives
  - Strings, narrow and wide
  - Arrays and sequences
  - Aliases (`typedef`s)
  - Unions
  - Modules
- As in IDL, but extended:
  - *Structures*—including single inheritance *(for substitutability: 6.5.1)*
  - *Enumerations*—specify bit width and constant values
- New relative to IDL:
  - *Maps*—like `std::map`or `java.util.Map`; required by RFP (*6.5.4*), fundamental OO collection
  - *Annotations*—for extensibility (*6.5.2, 6.5.3*)

```
structOriginalLandData {
    long x;
    long y;
};
```

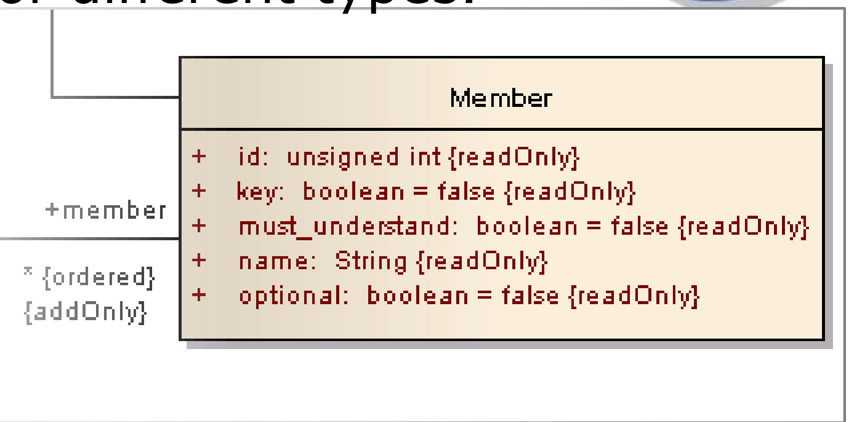
Type System Extensibility: Structures



Type System Extensibility: Structures

Collection of members, of same or different types. Each member has:

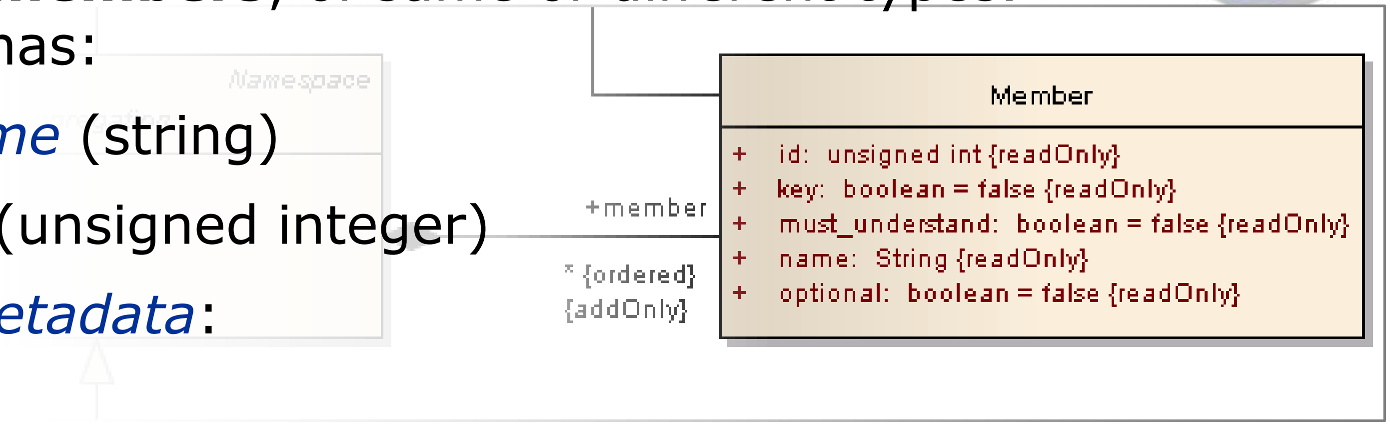
- A *name*, unique w/in type
- A *ID*, unique w/in type
  - Allows compact, extensible representation
  - Brings RTPS discovery types into type system mainstream
  - Improves introspection performance
  - Consistent with other technologies, like SNMP, FIX, TIBCO Rendezvous




```
structOriginalLandData {
    long x; // ID=0
    long y; // ID=1
};
```

Type System Extensibility: Structures

Collection of members, of same or different types.
Each member has:



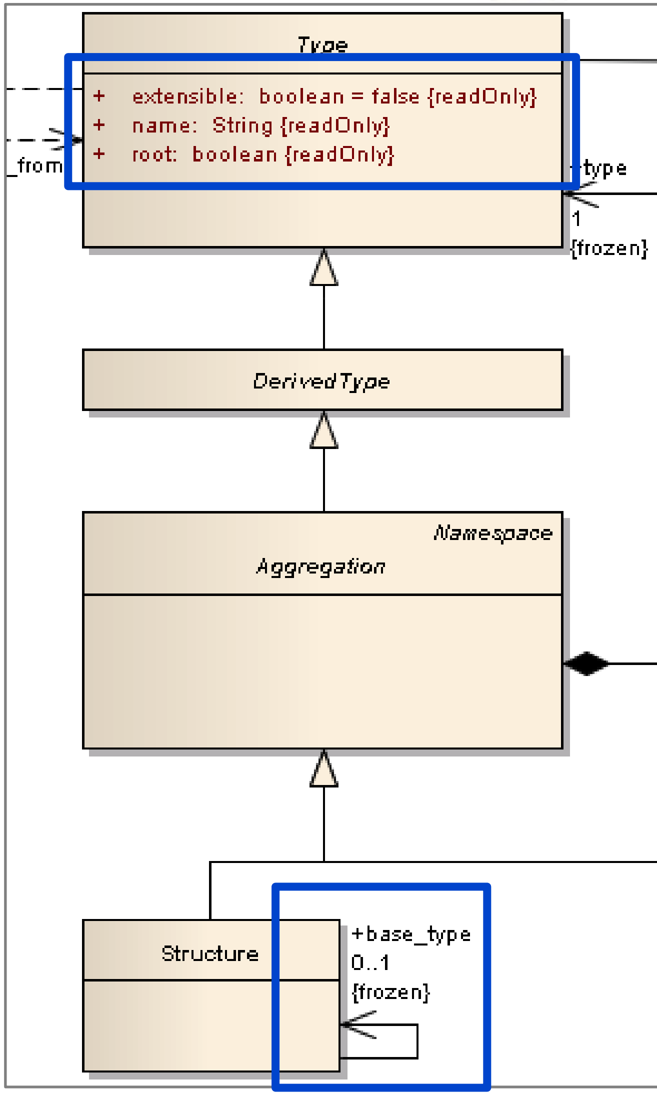
- A *unique name* (string)
- A *unique ID* (unsigned integer)
- Additional *metadata*:
  - *Key*?
  - *Optional vs. required*—Does value always exist? (*6.5.6, 6.5.19*)
    - Important semantics: Think null vs. non-null value
    - Universal concept: C, C++, Java, .Net, SQL, XSD, …
    - Important for interop: If I have a field in my type that you don't, what do I do with data from you?
    - Important for representational efficiency: Skip a field with well-defined semantics
    - *Note*: Keys always required; otherwise, identity breaks down

Q: Do Member ID's Make Integration Harder?

- **No**: Same scope as member names (per type)
- **No**: Huge range (100s millions): easy to partition
- **No**: Proven approach in large systems
  - *e.g.* SNMP
  - *e.g.* FIX
  - *e.g.* Google Protocol Buffers
  - *e.g.*Tibco Rendezvous

Type System Extensibility: Structures

- **Single inheritance**
  - Most common use case for type *substitution* (*6.5.1*)
- **Additional metadata**
  - *Extensible*?
    - Type may change in the future; be ready (*6.5.2, 6.5.3, 6.5.17*)



Type Compatibility (*6.5.3, 6.5.17*)

- **Directed relationship**: *is-assignable-from*
  - *Type1 is-assignable-from Type2* means *if I need* a Type1, *I can use* a Type2 in its place
  - *Intuition*: If Bar is a subclass of Foo, I can use Bar wherever I need Foo
- **Performance constraint**: Don't require reader to interpret data in writer-specific way
- **Implication**: is-assignable-from must be very strict:
  - Serialized object of Type2 must be interpretable *with access only to Type1*
  - Keys must agree too
  - *True*: `sequence<long,10>`is-assignable-from `sequence<long,5>`
  - *False*: `long[10]`is-assignable-from `long[5]`

Type Compatibility: Example

```
// OLD
structOriginalLandData {
    long x;
    long y;
};
```

is-assignable-from

! is-assignable-from

To Be Continued…

```
// NEW1
struct NextEnhancedWorldview1 {
    long x;
    long y;
TrackKindEnum kind;
};
```

```
// NEW2
struct NextEnhancedWorldview2 {
    long y; // out of order!
    long x; // out of order!
};
```

Type System Extensibility: Metadata

- **Question**: How can vendors attach *additional* metadata to type system elements?
- **Question**: How can users do the same?
- **Answer**: Annotations

DB Table
“Track”

Project
“Radar”

```
structOriginalLandData {
    long x;
    long y;
};
```

DB Column
“horz”

Version 1.2

DB Column
“vert”

Type System Extensibility: Metadata

- **Annotation = structured metadata** attached to type or type member
  - *Annotation type* defines annotation members, their names, and their types
  - *Annotation usage* provides values for annotation type's members

DB Table
name: string

DB Table
name = "Track"

- **Established programming practice**
  - In Java: `@MyAnnotationintx;`
  - In C#: `[MyAnnotation] intx;`

Type Representations

- IDL
- XML, XSD
- TypeCode

- **Static Definition**
  - 6.5.11: *Programming-language-independent* type serialization format(s)
    - 6.5.9: Represent types in *IDL* and *XML*
    - 6.5.7: Support IDL subset used by DDS 1.2
- **Dynamic Definition**
  - 6.5.12: Type *discovery*

IDL Type Representation (*6.5.7, 6.5.9, 6.5.11*)

Type System Overview

- Entirely familiar from IDL
  - Primitives
  - Strings, narrow and wide
  - Arrays and sequences
  - Aliases (typedefs)
  - Unions
  - Modules
- As in IDL, but extended
  - Structures
  - Enumerations
- New relative to IDL
  - Maps
  - Annotations

← *No change*

← Use *structures*.
Use *value types* if inheritance desired.

← New *keyword* (only one)
`map<key, value, bound>`

← We'll get to these…

Metadata in IDL: Non-Standard Metadata

- **Java-like annotation syntax**: already familiar
- **Annotation type**:

```
@Annotation local interface MyAnnotation {
    long value1();
    double value2();
};
```

- **Annotation usage**:

```
structMyStruct {
    @MyAnnotation(value1 = 42, value2 = 42.0)
    long my_field;
};
```

Metadata in IDL: Standard Metadata

- **Historical approach**: add *keywords* (`abstract`, `truncatable`, `public`, …)
  - Clutters up language
  - Compatibility nightmare
  - Not scalable: each new revision needs to add more
- **Alternative**: re-use *annotation* syntax
  - *All metadata looks the same*: standard, vendor-specific, user-defined
  - *Update compilers just once**; don't break syntax with every new spec (**…or not at all: see next slide*)
- **Built-in annotations**:
  - Declare annotation type: `@Annotation`
  - Modify members: `@Key`, `@ID`
  - Modify types: `@Extensible`

Metadata in IDL: Compatible Syntax

- **Scenario**: Using same IDL file with two IDL compilers; only one supports this spec
- **Solution**: Comment-like syntax
  - Primary syntax: `@ID(1) long x;`
  - BW-compatible syntax: `long x; //@ID(2)`
- **Advantage**: annotation *syntactically connected* to annotated element
  - Looks like comment: ignored by non-conformant compilers
  - As *compact* as primary syntax
  - Allows meaning of declaration to be *understood at once*
  - Simplifies *config. mgmt.*: metadata can't get out of sync

Type Compatibility: Example (Revisited)

```
// OLD
@Extensible
structOriginalLandData {
    long x;
    long y;
};
```

is-assignable-from

is-assignable-from

```
// NEW1
@Extensible
struct NextEnhancedWorldview1 {
    long x;
    long y;
TrackKindEnum kind;
};
```

```
// NEW2
@Extensible
struct NextEnhancedWorldview2 {
    @ID(1) long y;
@ID(2) long z;
@ID(0) long x;
};
```

Q: Why Aren’t All Structures “Extensible”?

- **They are**, but only in certain ways.
  - Always legal to add fields to the end
  - “Extensible” types are *more* extensible
- **There’s a price**: space, processing
  - Can the data present vary from sample to sample?
    Need sufficient metadata to detect what’s (not) there.
  - *Technical case* for not paying:
    Not all types, systems will change in deployment
  - *Business case* for not paying:
    - CDR doesn’t support extensibility by itself
      - No way to detect member addition, omission, reordering
    - Can’t abandon or break entire installed base
- **Can “extensible” be the default?**
  - *No*: IDL → CDR is well defined; would break everyone
  - Vendors could provide IDL compiler configuration option

XML Type Representations

#1: XSD

- Based on existing IDL-to-WSDL mapping
- Validates XML Data Representation
- Allows type sharing between DDS and web services

But…

- Very verbose
- Hard to read, hard to write, hard to parse

#2: XML

- Straightforward mapping of Type System into XML
- Easy to read, easy to write, easy to parse
- Suitable for embedding into other XML documents (*e.g.* QoS profile files)

XML Type Representations

#1: XSD

```
<xsd:complexType
  name="OLD">
<xsd:sequence>
<xsd:element
   name="x"
  minOccurs="1"
  maxOccurs="1"
   type="xsd:int"/>
<xsd:element
   name="y"
  minOccurs="1"
  maxOccurs="1"
   type="xsd:int"/>
</xsd:sequence>
</xsd:complexType>
```

#2: XML

```
<struct name="OLD">
<member name="x"
   type="int32"/>
<member name="y"
   type="int32"/>
</struct>
```

TypeCode Type Representation

- **Requirement** (*6.5.12*): Propagate types during discovery to enable dynamic behavior
- **Question**: Which Type Representation is appropriate?
  - Should be compact, binary, …
- **Answer**: TypeCode Type Representation
  - We know how to represent data efficiently for DDS: *describe it as a data type*
  - Name “TypeCode” has heritage in CORBA, .Net

Data Representations

- CDR
- XML

CDR Data Representation

- **Requirements**:
  - Continue supporting existing (compact) CDR used by user-defined types (*6.5.16*)
  - Continue supporting existing (extensible, parameterized) CDR used by discovery types (*6.5.16*)
  - Provide extensibility to user-defined types (*6.5.2, 6.5.3, 6.5.17*)—*all Data Representations should have equivalent expressiveness*
- **Solution**: Existing extensible, parameterized CDR available to all
  - *Same rules* for every type, built-in or user-defined
  - *100% compatible* with existing types and serialization
- **Parameterized CDR**: structs& unions marked “extensible”
  - Each member == parameter
  - Member ID == parameter ID
- **Compact CDR**: everything else

CDR: Costs & Benefits

- **Cost**: 4 bytes per member
- **Benefits**:
  - Proven: Tag-Length-Value (TLV) in DHCP, ASN.1, …
  - Consistent: Brings discovery types into mainstream
  - Type evolution: Determine which fields are (not) present;
    Skip fields you don't understand
  - Time savings: Encode/decode fast;
    Send fields in any order
  - Space savings: Cost of omitted optional field = 0 bytes;
    Cost of default required field = 0 bytes

Q: Could a Different Representation Save Space?

Short Answer: Yes

Long Answer:

- CDR *optimizes for speed*.
  - Wire representation matches machine representation:
    - Endianness
    - Alignment
- Another representation could *optimize for size*.
  - Could use variable-length encoding
    - Example: Google Protocol Buffers
    - Example: Apache Thrift (incubator; originally FaceBook)
    - *Unaligned access, bitwise operations will make it slow*
  - Could use compression—*e.g.* zip the whole UDP payload
  - *Active area of research*

Summary:

- Once fully understood, a space-optimized representation could be added to this proposed spec
- Could not *replace* CDR for speed, compatibility reasons

XML Data Representation

- **Optional Requirement** (*6.6.1*):
  Dynamic data access based on XML or JSON
- **Given**: XSD Type Representation

- Data object == XML document
- Validated by XSD Representation of its type
- SAX, DOM-based access follow naturally
- New RTPS encapsulation ID lets you use XML on the network

```
structOriginalLandData {
    long x;
    long y;
};
```

```
<OriginalLandData>
<x>5</x>
<y>42</y>;
</OriginalLandData>
```

Language Bindings

- Plain Language Binding
- Dynamic Language Binding

Plain Language Binding

- Defined by existing IDL-to-*language* mappings:
  1. Start with IDL Representation of a type
  2. Apply mapping to desired programming language
- Mappings expanded to cover new concepts
  - *e.g.* maps, annotations, optional members
  - For C, C++, Java as required by RFP (*6.5.8*)

```
// IDL
struct
OrigLandData {
    long x;
    long y;
};
```

```
// Java
public class
OrigLandData {
    public intx;
    public inty;
}
```

Dynamic Language Binding

- **Requirements**:
  - 6.5.20: (*a*) *Define types dynamically*.
    (*b*) *Create topics* based on them.
  - 6.5.13: Type *introspection*
  - 6.5.18: Dynamic *data access*
  - 6.5.21: *All pub-sub operations* using dynamically defined types

Dynamic Language Binding

- **Classes**
  - `DynamicTypeFactory`: Creates new types (*6.5.20*)
  - `DynamicType`: Introspect type definitions (*6.5.13*)
    - Informed by CORBA `TypeCode` API
  - `DynamicData`: Introspect objects (*6.5.18*)
    - Informed by CORBA `DynAny`, JMS `MapMessage`, TIBCO Rendezvous self-describing messages
  - `DynamicTypeSupport`: Registers types with DDS (*6.5.20*)
  - `DynamicDataWriter`: Writes objects of any (single) type, each represented by a `DynamicData` object (*6.5.21*)
  - `DynamicDataReader`: Reads objects of any (single) type, each represented by a `DynamicData` object (*6.5.21*)

Dynamic Language Binding

- **Works with, like the DDS you know**
  - *Modeled* in UML
  - *API* in IDL
  - *Interoperable* with statically defined types

Use by DDS

- QoS Compatibility
- Type Compatibility Revisited
- Built-in Types

QoS Compatibility

- **Reader and writer must agree** on which Data Representation to use
  - 6.5.14: Encapsulation specification, negotiation
    - 6.5.15: Is encapsulation request-offer?
  - 6.6.2: *Optional*: Custom encapsulation API
- **Solution**: *DataEncapsulationQosPolicy*
  - List of encapsulation IDs
  - Applies to both readers and writers
  - Included in corresponding built-in topic data types
  - Request-Offer semantics
    - *Writer offers single* encapsulation it will use
    - *Reader requests a list* of encapsulations
    - *Compatible* iff writer's encapsulation is in reader's list
  - *Can describe proposed encapsulations, future additions, and vendor extensions*

Type Compatibility

- **New capability**: readers and writers can use *different types*
  - *Because of type evolution*;
    e.g. one side has been upgraded, other hasn't
  - *Because of decoupled design*;
    e.g. pub subclass, sub superclass
- **Two aspects** of compatibility:
  - *Physical*: Can type be interpreted as another w/o misunderstanding? —*covered this already*
  - *Intentional*: Do I *want* to interpret that way?
    - Don't just hard-code inheritance-based rules
    - Old saw: For Shapes and Gunslingers, "draw" is different.
  - Both require type to be included in endpoint discovery
    - Propagated using TypeCode Representation
    - API is `DynamicType`

Intentional Type Compatibility

- **Expressed at type registration** with *type signature*
  - Extension of type name syntax of today

```
MyTypeSupport::register_type(
my_participant,
     "Foo");
```
←Type Signature

- **Example**: "T"
  - *My type* is "T"
  - *When writing* a topic of "T," readers' topics must also use "T"
  - *When reading* a topic of "T," writers' topics must also use "T"
- **Example**: "T1, T2, T3 : Base1, Base2"
  - *My type* is "T1"
  - *When writing* a topic of "T1,"
    readers' topics must use "T1," "T2," "T3," "Base1," or "Base2"
  - *When reading* a topic of "T1,"
    writers' topics must use "T1," "T2," or "T3"

Intentional Type Compatibility

- **Intuition**: T1, T2, and T3 are like "subclasses" of Base1 and Base2
- **Formally**: "T1, T2, T3 : Base1, Base2" means:
  - T1 is-assignable-from T2, T3
  - T2 is-assignable-from T1, T3
  - T3 is-assignable-from T1, T2
  - Base1 is-assignable-from T1, T2, T3
  - Base2 is-assignable-from T1, T2, T3
- **Mismatch** → `on_inconsistent_topic`
  1. If *names don't match* type signatures (as today)
  2. If physical *compatibility doesn't match* declared intention
  - (If type definitions unavailable—e.g. for backward compatibility—omit #2)

Built-in Types

- **Goal**: Improve out-of-box experience for new users, those with simple needs
- **Background**: Usability gap vs. competition
  - Other middlewares allow sending textual or binary data without explicit type definition, registration
- **Response**: Provide simple types built in
  - Pre-defined, pre-registered

Built-in Types

- **RFP says**: octet sequence, string sequence, map (*6.5.10*)
  - *Proposal diverges from RFP*:
  - *Octet sequence*: kept this one
  - *String sequence*: value unclear; doesn't match competing technologies
  - *Map*: requirement unclear; what are the key and value types?
- **Unkeyed Types**
  - *String*
    - Payload is single unbounded string, not a sequence
    - `FooDataWriter`, `FooDataReader` "templates" instantiated w/ basic `string`
  - *Bytes*
    - Payload is unbounded sequence of octets, as in RFP
    - `FooDataWriter`, `FooDataReader` "templates" instantiated w/ `sequence<octet>`
- **Keyed Variants**
  - *KeyedString*
    - Payload, key each unbounded string
  - *KeyedBytes*
    - Payload is unbounded sequence of octets
    - Key is unbounded string

Summary

Summary: Type System Requirements

Extensible, Compatible

6.5.1: (*a*) *Type System* as UML meta-model. (*b*) Type *substitutability* rules.	**Yes.** Is-assignable-from relationship defines substitutability.
6.5.2: Type *extensibility*, *mutability* rules	**Yes.** All structures, unions are extensible; those *marked* "extensible" are more so.
6.5.3: Type *compatibility* in presence of change	**Yes.** Is-assignable-from relationship defines compatibility.
6.5.17: Allow type to *evolve* without breaking interoperability	**Yes.** Is-assignable-from relationship defines compatibility.
6.5.5: Support *keys*. How does extensibility apply?	**Yes.** Keys are first-class concept. Object identity require compatible types to agree.

More Expressive

6.5.4: *Map* data type	**Yes.** Keys strings or ints; values anything.
6.5.6, 6.5.19: Sparse data: object contains only *subset of fields* defined by its type	**Yes.** Any structure or union may contain optional members. *Any* member may be omitted on wire if encoding is extensible.

Summary: Type Requirements

- **Static Definition**

6.5.11: *Programming-language-independent* type serialization format(s)	**Yes.** IDL, XML, XSD, and TypeCode are included.
6.5.9: Represent types in *IDL* and *XML*	**Yes.** IDL, XML, and XSDare included.
6.5.10: *Built-in types*: map, octet sequence, string sequence	**Yes. But**types are String, KeyedString, Bytes, KeyedBytes.

- **Dynamic Definition**

6.5.20: (*a*) *Define types dynamically*. (*b*) *Create topics* based on them.	**Yes.**`DynamicTypeFactory`creates types. `DynamicTypeSupport`registers them. Once registered, they behave like any other type.
6.5.12: Type *discovery*	**Yes.** Type is in pub and sub topics.
6.5.13: Type *introspection*	**Yes.**`DynamicType`provides this.
6.5.21: *All pub-sub operations* using dynamically defined types	**Yes.** Once registered, dynamically defined types behave just like statically defined ones.

Summary: Data Requirements

- **Dynamic Access**

6.5.18: Dynamic data access	**Yes.** `DynamicData` provides this.
6.6.1: *Optional*: Dynamic data access based on XML or JSON	**Yes.** Follows naturally from XML Data Representation; no additional facility necessary.

- **Network Encapsulation**

6.5.14: Encapsulation specification, negotiation	**Yes.** `DataEncapsulationQosPolicy` provides this.
6.5.15: Is encapsulation request-offer?	**Yes.** It is.
6.6.2: *Optional*: Custom encapsulation API	**Yes.** Vendors can provide them by defining additional encapsulation ID values.

Summary: Compatibility Requirements

- **Type Representation Level**

6.5.7: Support subset of IDL used by DDS 1.2	**Yes.** Same IDL supported, and it implies the same wire representation as before.

- **Programming Level**

6.5.8: APIs for C, C++, Java	**Yes.** Plain Language Binding extensions defined for these languages. Dynamic Language Binding API defined in IDL for broad applicability.

- **Network Level**

6.5.16:(*a*) Support CDR.(*b*) Support parameterized CDR.(*c*) Interoperate with RTPS/DDSI 2.	**Yes.** Fully backwards compatible with both CDR variants. Generalizes parameterized encapsulation to eliminate special cases.

Summary

The proposal:

- Satisfies all mandatory and optional requirements

  and
- Supports identified real-world use cases

- Without sacrificing backwards compatibility

  and
- Without sacrificing performance

- Is easily extensible w/ new type, data representations to handle *additional* use cases

Q & A