PEF
Protocol Engineering Framework

The Open Protocol Notation Programming Guide v0.1

Document Version v0.1 (1/7/2017)

Abstract. Open Protocol Notation (OPN) is a text-based notation for describing the architecture, messages, and behavior of network protocols.

Version	Editor	Change	Date
0.1	v-cepark	Initial formal release version	9-06-2013
		-	-
		-	-
		-	-

Table of Contents

1. Introduction

The Open Protocol Notation (OPN), is the fidelity textual notation for the Protocol Engineering Framework (PEF). OPN provides mechanisms to specify the architecture, messages, and behavior of network communication protocols.

1.1. Scope

OPN is a domain-specific language that enables a model-based development process for network protocols. The main usage scenarios of OPN include the following:

- Describe protocol architecture, messages, and behavior in a human-readable format.
- Simulate and validate protocol behavior and architecture.
- Generate rigorous and precise protocol documentation.
- Generate protocol network parsers and runtime monitors.
- Generate test suites (model-based tests), which can validate an implementation against the related protocol specification.
- Generate code-stubs.

An important requirement to accommodate this scope is to have the capability to capture and extend existing protocol implementation technologies, for example, RPC/COM, SOAP, block-based protocols, and so on. While some of these domains are supported as legacy protocols, OPN particularly supports modern protocols based on XML, JSON, and others that are similar.

OPN is supported by a set of tools that enable these scenarios. At the core of this tool set is a representation of OPN in abstract syntax and a framework to work with such syntax. The overall system is referred to as PEF.

1.2. Approach

OPN is influenced by numerous approaches to specification and modeling, which includes the following:

- Z Notation
- Vienna Development Method (VDM)
- Abstract State Machines (ASM)
- Functional programming languages such as Haskell, ADLs (Architectural Description Languages), Process Algebra (CSP, CCS), and others.

However, the notation takes a pragmatic approach in regardsto its appearance, system of values, and computation, by aligning some of its core concepts with mainstream programming languages such as C# and Java. In addition, OPN adds domain-specific concepts to these core concepts to deal with constrained data structures (messages), data structure transformation, behavior specification, and architecture.

1.3. About this Document

This document defines OPN to provide reference material for developers who wish to create OPN protocol descriptions for use with Microsoft Message Analyzer. The material begins with an informal overview that contains the relevant conceptual information to understand the notation. It then continues on to describe the full reference for the notation. The reader is expected to have working knowledge of computer languages such as C# and Java, and to be familiar with the network communication protocol domain.

2. Overview

This section provides an overview of OPN. Where the notation aligns with languages such as C# and Java, fewer details are provided; where it diverges or adds new concepts, more details are provided. The reader is advised that this section does not substitute as a reference for the OPN language. The full definition of OPN, that includes syntax and static semantic rules, is specified in section 4 Language Reference. The full set of predefined types and available overloaded operators that work with them is part of the standard Library Reference specified in section 5.

2.1. Types

OPN is a strongly and statically typed language, with an extended notion of types that is referred to as patterns. Patterns are defined separately from types and processed dynamically at runtime. This section describes the static contructs of the type system.

The OPN language provides a set of predefined types, a mechanism to import externally defined types, and a mechanism to define user types. All types expose three fundamental properties as follows:

- **Nullability –** a type is nullable if the special `null` value is part of its domain.
- **Mutability –** a type is mutable if it supports selective update of its component value.
- **Identity –** a type has identity if each of its values has a unique identifier that makes it distinct from every other value, and this identifier is used to reference instances of the type.

The presence of the identity property has some further implications:

- If a type has identity and it is mutable, then updates to it will be visible to all other references to that value.
- Moreover, if a type has identity, then equality on values will be based solely on this identity, whereas for a type without identity, equality will be solely based on its content value.

Examples of non-nullable and non-mutable types consist of scalars such as integers. Examples of nullable, mutable, but non-identity types consist of strings, arrays, and so forth. Examples of nullable, mutable, and identity types consist of user-defined types, which are also called reference types in OPN.

2.1.1. Generic Types

Both predefined and user-defined types and methods support type parameters. Type parameters are subject to type inference, as described in section 2.1.8 Type Inference. The OPN language does not support co-variance or contra-variance, except in some special cases that are described in section 2.1.7 Conversions.

2.1.2. Predefined Types

Common to all predefined types is that they are supported by special literal denotations in the language, or they have special type checking rules. The predefined types are listed in the following table, along with examples of literal notation. By convention, predefined types are denoted by the use of keywords or special syntactic constructs of the language.

Type Name	Description	Nullable	Mutable	Identity	Example
byte	Unsigned 8-bit integer	No	No	No	`byte val = 24;`
sbyte	Signed 8-bit integer	No	No	No	`sbyte val = 24;`
ushort	Unsigned 16-bit integer	No	No	No	`ushort val = 24;`
short	Signed 16-bit integer	No	No	No	`short val = 24;`
uint	Unsigned 32-bit integer	No	No	No	`uint val = 24;`
int	Signed 32-bit integer	No	No	No	`int val = 24;`
ulong	Unsigned 64-bit integer	No	No	No	`ulong val = 24;`
long	Signed 64-bit integer	No	No	No	`long val = 24;`
float	Single precision floating point	No	No	No	`float val  = 1.23;`
double	Double precision floating point	No	No	No	`double val = 1.23;`
decimal	A 128-bit precision floating point	No	No	No	`decimal val = 10.2m;`
bool	Boolean value	No	No	No	`bool val = false;`
char	Character value	No	No	No	`char val = 'a';`
string	String value	Yes	Yes	No	`string val = "hello";`
stream	A stream of characters	Yes	Yes	Yes	`stream val = $[AAFE];`
binary	Binary value (array of bytes)	Yes	Yes	No	`binary val = $[AAFE];`
guid	A GUID value	No	No	No	`Guid val = {01020000-0000-`

					`0000-0000-000000000000};`
array<int>	Array type is a collection type	Yes	Yes	No	`array<int> val = [1,2];`
set<int>	Set type is a collection type	Yes	Yes	No	`set<int> val = {1,2};`
map<int, char>	Map type is a collection type	Yes	Yes	No	`map<byte, char> val = {65 -> 'A', 66 -> 'B'};`
int?	Nullable type	Yes	No	No	`int? val = null; int? val = 1; int x = val as int;`
optional int	Optional type	No	No	No	`optional int = nothing; optional int val = 1; int x = val;`
xml	XML type	Yes	Yes	Yes	`xml x = xml{…}; xml y = x select xpath{//Node};`
json	JSON type	Yes	Yes	Yes	`json x = json{…}; json y = x select xpath{//Node};`
treedata	A treedata type, represents a tree model for hierarchical data	Yes	Yes	Yes	`treedata x = xml{…}; treedata y = x select xpath{//Node};`
any	Encapsulates a value of an arbitrary type	Yes	No	No	`any x = 1; any y = [1,2]; int z = x as int;`
int(int)	Function type	Yes	No	No	`int(int) adder = x => x+1;`
void	Absence of a value	-	-	-	`void M(){ … }`

Additional notes about the predefined types and the available operators are briefly described as follows, however, to review a full reference, see the standard Library Reference specification in section 5.

- The usual arithmetic, bit-wise, and relational operators are available on the scalar types. Operators are overloaded for various type combinations. Implicit conversions are applicable for certain scalar types, as described in section 2.1.7 Conversions.

- The predefined string, array, binary, set, and map collection types are not identity types in OPN. However, these types are mutable, that is they support component updates, as in the statement **`array[index] = value`**. The effect of mutation on these types is limited to the location where the value is updated, thus that guarantees immunity from the side-effects of mutation on such values.

- Functions are overloaded for manipulating strings, binaries, and arrays. For example, you can use the plus sign (+) for concatenation and you can select array elements by using the format **`s[i]`** with zero-based indexing. Standard library functions are also available, such as the **`IndexOf`** method.

- For sets, union (+), intersection (*), and difference (-) are available. Membership test is written as **`x in S`**. For maps, union (+) can be used to merge two maps. The right operand overrides keys from the left operand. In particular, to add an entry, one can write **`m + {key1 -> value1, key2 -> value2, …, keyn -> valuen}`**.

- The **`stream`** type provides sequential access data. This type supports domain specific infrastructure to parse such data.

- The **`optional`** type constructor is the preferred way in OPN to describe the presence or absence of a value or feature. The **`optional`** key word can be applied to any type, in contrast to nullable. The **`nothing`** keyword represents the value for an absent optional field, see section 2.8.2 Message Annotations.

- The **`any`** type is used to represent a polymorphic value that encapsulates a value of a concrete type. All values can be converted implicitly into a value of the **`any`** type, and converted back provided the original type is known. The **`any`** type corresponds to the object type in languages such as C# or Java. Type test is written as **`x is T`**, type conversion is written **`x as T`**.

- The **`xml`** type represents XML values as a first-class citizen of the language. XML values are nullable, mutable, and have identity. Special notation is available to denote XML values, as well as to project them by the use of the XPath syntax.

- In OPN function types are used to represent computations performed in certain environments. Function types are stored together with their function value (the so-called closure). While these types are nullable, they do not have identity and are not mutable. Equality on function values is mapped to equality on the environment plus equality on the code address of the function; for the later, indeed equality is not extensional.

- The pseudo-type **`void`** is used to represent the absence of a function return type.

- The **nothing** and **null** values are only equal to **nothing** and **null**, respectively, and not equal to any other value. They also act as the absorbing element for all arithmetic operators, for example **nothing + 5 == nothing**.

2.1.3. External Types

OPN provides a mechanism to import types and functions from a hosting runtime environment. For example, the standard library contains declarations for external types that represent time and duration, see section 5.10 Date and Time.

```
module Standard;
extern type DateTime;
extern type TimeSpan;
extern DateTime DateTimeFromSeconds(double seconds);
extern TimeSpan TimeSpanFromSeconds(double seconds);
// ...
```

The way external types are declared in OPN is part of the language. How they are implemented in the runtime environment is dependent on a particular implementation.

2.1.4. Reference Types

Users can define new types by the use of a type declaration, which resembles a class declaration in languages such as C# or Java. A type introduced this way is called a reference type, and it is nullable, mutable, and has identity. The following example shows a simple reference type declaration.

```
type Frame
{
    int Length;
    array<int> Data;
}
```

In addition to fields, a reference type can also contain methods. Methods can be instance-based or static as in the following code example.

```
type Frame
{
    // ...
    string SummarizedInfo()
    {
       return (Length as string) + ":" + (Data as string);
    }

    static Frame Make(array<int> data)
    {
        // ...;
    }
}
```

By default, all members of a reference type have the same visibility as the type itself. The default visibility of types and other container declarations is public; this shows the higher level of abstraction (with less private implementation details) often found in models.

You can create a reference value either by the call of a user-defined constructor or by the provision of field initialization at construction time. This is shown in the following example.

```
type Frame
{
     // ...
     Frame(array<byte> data)
     {
         this.Data = data;
         this.Length = data.Length;
     }
}
Frame frame1 = new Frame([1,2,3]);
Frame frame2 = new Frame{Length = 3, Data = [1,2,3]};
```

Reference types do support single inheritance and can be declared abstract, just as in languages C# and Java. They can also implement interfaces, which is described in section 2.2.6 Interface Patterns. Methods can be declared virtual or abstract, and can be overridden in subclasses. The default binding behavior of a method is non-virtual.

Reference types can have invariants. An invariant is denoted as shown in the following example.

```
type Frame<T>
{
     int Length;
     array<T> Buffer;
     invariant Buffer.Count == Length;
}
```

In addition to explicitly specified invariants, invariants can also be implicitly imposed by the use of patterns in field declarations, as described in section 2.2.1 Basic Pattern Forms. Invariant checking is described in more detail in section Invariant Checking.

While a reference value is by default mutable, it can be converted into an immutable form by *freezing* if it is part of a message. This is described in section 2.8.1 Messages.

Reference types can have type parameters, as shownin the following example.

```
type Frame<T>
{
      int Length;
      array<T> Data;
}
```

Type parameters do not support co-variance or contra-variance, for more detail see the discussion about conversions in section 2.1.7. Conversions

Reference types can also have value parameters. These parameters provide a way to pass a value at the declaration point of the type, which can play a role in field initialization or invariants as seen in the following example.

```
type BoundedIntArray[int Bound]
{
     array<int> Values;
     invariant Values.Count <= Bound;
}

type Frame
{
     int Length;
     BoundedIntArray[Length] Data;
}
Frame x = new Frame{Length = 2,
     Data = new BoundedIntArray{Values = [1,2,3]}};
```

As seen from the preceding example, the value parameter is only provided at the declaration point of a type, not at the point where an instance is constructed. Semantically, one can think of type value parameters as implicit fields that receive their value at the point where an instance of the reference type is assigned to a location. In the example, it is at the point where the **`Data`** field is initialized. Consequently, these values are not part of type compatibility rules, for example, **`BoundedIntArray[1]`** and **`BoundedIntArray[2]`** are the same types.

Inheritance preserves value parameter information, and a child type inherits its father value parameters.

```
type BoundedIntArrayWithMinimum[int Minimum]: BoundedIntArray
{
     invariant all (var x in Values) x >= Minimum;
}

type Frame
{
     int Length;
     int Minimum;
     BoundedIntArrayWithMinimum[Minimum, Length] Data;
}
Frame x = new Frame{Length = 2, Minimum = 3,
     Data = new BoundedIntArrayWithMinimum{Values = [3,4,5]}};
```

All reference types can **`override`** the **`ToString()`** method to provide a customized string representation.

```
type Frame<T>
{
    int Length;
    array<T> Data;

    override string ToString()
    {
        return "I am a message Frame";
    }
}
```

The **Equals** and **GetHashCode** methods can also be overridden for reference types in order to define how value equality behaves as in the following example.

```
type Frame
{
      int Length;
      array<T> Data;

      // Two frames are the same if their length is the same.
      override bool Equals(any frame)
      {
            if (!frame is Frame) return false;
            Frame f = frame as Frame;
            return f.Length == this.Length;
      }
      override int GetHashCode()
      {
            return this.Length;
      }
}
```

2.1.5. XML Type

OPN supports a built-in type that resembles a generic XML tree. XML values are constructed by the use of the special constructor `xml{...}` and inspected by the use of a set of methods from the standard library. In addition to that functionality, OPN supports XPath query expressions directly in the language by the use of the `select` expression.

For illustration, consider the following XML document.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<bookstore>
  <book>
    <title lang="eng">Harry Potter</title>
    <price>29.99</price>
  </book>
  <book>
    <title lang="eng">Learning XML</title>
    <price>39.95</price>
  </book>
  <something/>
</bookstore>
```

Now suppose the document elements are bound to OPN variables as shown in the following example. These variables will be used in the next example to describe the meaning of the **select** expression.

```
   xml bs ← the bookstore XML element
   xml b1 ← the harry potter book element
   xml e1 ← the text node "en" for harry potter title element attribute
   xml b2 ← the learning XML book element
   xml e2 ← the text node "en" for learning XML title element attribute
   xml s ← the something element
```

The following examples illustrate the usage and meaning of the **select** expression form. Note that the expression enclosed by **xpath{...}** at the right-hand side of the select form is a standard XPath expression.

```
// Selects the root element, returning [bs].
bs select xpath{/bookstore};
// Selects  children of bookstore named "book", returning [b1, b2].
bs select xpath{bookstore/book};
// Selects  children named "book" anywhere, returning [b1, b2].
bs select xpath{//book};
// Selects attribute named "lang" anywhere, returning [e1, e2].
bs select xpath{//@lang};
// Selects  all children of "bookstore", returning [b1,b2,s].
bs select xpath{/bookstore/*};
```

The way to specify the use of a namespace that applies to all XPath selections in a document is with a **using** statement as follows.

```
protocol Windows.P1;

using xmlns:bk="urn:loc.gov:books";
using xmlns:isbn="urn:ISBN:0-395-36341-6";
```

These **using** statements have either protocol or module scope, and apply globally to all XPath selections within the document.

```
// ...
xml bs = ...;

// Returns the node <isbn:number>1568491379</isbn:number>.
```

```
return bs select xpath{bk:book/isbn:number};
```

An OPN compiler does not perform more than syntactic checks of an XPath expression. The actual evaluation of the expression will happen at execution time.

2.1.5.1. Applying Xpath Operator to Reference Types

The XPath operator is not restricted to XML values, and it can be applied to reference type values as well. An implicit mapping takes place in this case to transform the view of an arbitrary OPN structure to an XML structure. After the implicit transformation, the operator works as performing a regular XPath on XML values.

The general idea of the mapping from an OPN structure to an XML structure is the following:

- The initial reference value is treated as an XML document (`#document`) node.
- Basic types are treated as XML element nodes.
- Arrays are treated as multiple XML element nodes with the same name, to preserve the same order.
- Sets are treated as arrays, in the sense that an arbitrary order is established for the elements.
- Maps introduce two special elements `Key` and `Value` as a way to provide the required structure.

Some examplesthat illustrate these concepts are given the following OPN code.

```
type T
{
    int AnInt;
    string AString;
    xml AnXml;
}

void Run(){
{
    T data =
       new T{AnInt = 10,
             AString = "hi",
             AnXml = xml{<A>
                          <B>100</B>
                          <C>200</C>
                          <B>300</B>
                        </A>}
            };
    var items = data select xpath{ ... };
    // ...
}
```

The implicit XML representation for **data** is:

```
<T>
  <AnInt>10</AnInt>
  <AString>hi</AString>
  <AnXml>
    <A>
      <B>100</B>
      <C>200</C>
      <B>300</B>
    </A>
  </AnXml>
</T>
```

The return type of the XPath operator is always an array of xml, so the following assertions are valid:

```
var items = data select xpath{AnInt};
assert items[0].Value == "10";

var items = data select xpath{./AnXml/A/C};
assert items[0].Value == "200";

var items = data.AnXml select xpath{.A/B/../../../AString};
assert items[0].Value == "hi";
```

For a more involved example to understand how the mapping works with arrays and maps, consider the following OPN code.

```
type Q
{
    int F1;
    string F2;
}

type T
{
    array<int> MyArray;
    array<Q> MyOtherArray;
    map<string, int> MyMap;
    map<string, array<int>> MyOtherMap;
}

void Run(){
{
      T data = new T{ MyArray = [1,2,3],
      MyOtherArray = [new Q{F1 = 1, F2 = "One"}, new Q{F1 = 2, F2 = "Two"}],
      MyMap = {"hi" -> 1, "bye" -> 2},
      MyOtherMap = {"hi" -> [1,2], "bye" -> [3,4]}
};

var items = data select xpath{ ... };
    // ...
}
```

The implicit XML representation for **data** is the following.

```
<T>
  <MyArray>1</MyArray>
  <MyArray>2</MyArray>
  <MyArray>3</MyArray>
  <MyOtherArray>
    <F1>1</F1>
    <F2>One</F2>
  </MyOtherArray>
  <MyOtherArray>
    <F1>2</F1>
    <F2>Two</F2>
  </MyOtherArray>
  <MyMap>
    <Key>hi</Key>
    <Value>1</Value>
  </MyMap>
  <MyMap>
    <Key>bye</Key>
    <Value>2</Value>
  </MyMap>
  <MyOtherMap>
    <Key>hi</Key>
    <Value>
      <MyOtherMap>1</MyOtherMap>
      <MyOtherMap>2</MyOtherMap>
    </Value>
  </MyOtherMap>
  <MyOtherMap>
    <Key>bye</Key>
    <Value>
      <MyOtherMap>3</MyOtherMap>
      <MyOtherMap>4</MyOtherMap>
    </Value>
  </MyOtherMap>
</T>
```

Observe that all collections are represented as a flattened enumeration of its elements and reuse the field name for each one. For the case of maps, the special **Key** and **Value** tags are introduced to provide the required structure. When the structure represented is a nested collection, the original field name is reused to represent the needed structure, as seen in the case of **MyOtherMap** that has arrays as values.

It is worth mentioning that an arbitrary OPN structure can contain references that define a loop. In this case, the XML representation will be an infinite tree. The implicit conversion is performed in a lazy manner; so as long as the returned XML is traversed in a finite way, it can be handled appropriately.

2.1.6. JSON Type

Similar to XML, OPN supports a built-in type that resembles a generic JSON tree. JSON values are constructed by the use of the special constructor **`json{...}`** and inspected using a set of methods from the standard library. The following is an example of how JSONvalues can be used.

```
// Construct first a JSON value. A library function to construct JSON
// values is also available.
json aValue = json { "firstName": "John",
                     "lastName": "Smith",
                     "age": 25};

// You can apply almost the same set of functions that are available to
// XML values, with similar results.
assert aValue.ChildCount == 3;
assert aValue.GetChild(new Name{LocalName == "firstName"}).value == "John";
assert (aValue select xpath{\lastName}).Count == 1;

// There is also a JSON decoder, similar to the XML decoder.
// Interprets the stream as a JSON-formatted string and returns SomeType
// if the decoding process succeeds, nothing otherwise.
stream s = ...;
var result = JsonDecoder<SomeType>(s);

// Same as the preceding, but it takes a JSON value instead of a stream
var anotherResult = JsonDecoder(aValue);
```

The only built-in operator available for JSON values is an XPath query. The syntax is the same as the one used for XML values and the operator returns an array of JSON values.

```
json aValue = ...;
var results = aValue select xpath{/firstName};
var otherResults = aValue select xpath{//phoneNumber};
```

An XML interpretation of a JSON value is needed to be able to apply an XPath. To formally define the JSON->XML mapping is unnecessarily cumbersome, so the following code example provides an example that exercises all the needed cases. The baseline is that child nodes are used instead of attributes for the XML representation, since that makes XPath expressions clearer. Note the given JSON value.

```
{
      "firstName": "John",
      "lastName": "Smith",
      "age": 25,
      "address": {
            "poBox": null,
            "streetAddress": "21 2nd Street",
            "city": "New York",
            "state": "NY",
            "postalCode": "10021"
            },
     "phoneNumber": [
     {
         "type": "home",
         "number": "212 555-1234"
      },
            {
              "type": "fax",
              "number": "646 555-4567"
            }
      ]
}
```

The equivalent XML for the JSON value is the following.

```
<json>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
      <poBox xsi:nil="true"/>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumber>
    <phoneNumber>
      <type>home</type>
      <number>212 555-1234</number>
    </phoneNumber>
    <phoneNumber>
      <type>fax</type>
      <number>646 555-4567</number>
    </phoneNumber>
  </phoneNumber>
</json>
```

Observe that a fixed `json` tag represents the root element of the XML document. Also notice how arrays are encoded (arrays are not a type in XML, so some encoding is needed). Array members are grouped with an element that uses the original array name, and that same name is used for the members in the underlying level.

As stated before, this mapping goes from JSON values to XML values (and not the other way around), but given that the mapping function is injective, of course, an inverse function is defined for the pre-image of the mapping. This means we can talk about going from XML to JSON for these cases.

The semantics of the XPath operator is very straightforward considering the preceding mapping. The result of the application of an XPath query is equivalent to the transformation of a JSON value into an XML value. This applies the previously defined XPath query and transforms the resulting array of XML elements into JSON elements. For more details on XPath syntax please see XPath Syntax by W3C. The remaining operators on JSON values are simply library functions.

2.1.6.1. A Common Type for JSON and XML

Considering that the JSON and XML types have many characteristics in common, it is likely the case that a value needs to be treated in the same way in both cases, to share the same piece of code. To facilitate this scenario, another primitive type **treedata** is available in OPN, that represents a tree model for hierarchical data. This type shares exactly the same characteristics as XML and JSON types do, in that it is mutable, nullable and has identity. Assignment works as a reference assignment. In a way, the **treedata** type behaves as the **any** type, to represent a polymorphic value that encapsulates a value of concrete type.

The OPN language does not provide a way to construct a specific **treedata** literal, since its intention is to provide a common representative for more specific types. Of course literals for more specific types can be constructed and assigned to **treedata** values.

In terms of OPN type system; there is an implicit conversion from the XML type to the **treedata** type and from the JSON type to the **treedata** type as in the following example.

```
json aJsonValue = ...;
xml anXmlValue = ...;
treedata myValue = aJsonValue;
treedata myOtherValue = anXmlValue;
```

Conversely, an explicit cast is provided and enforced from **treedata** values to JSON and XML values.

```
treedata aJsonOrXml = ...;
// Yields and exception if the conversion is not valid.
json aJsonValue = aJsonOrXml as json;
// Yields and exception if the conversion is not valid.
xml anXmlValue = aJsonOrXml as xml;
```

These relationships imply that the least common type in an **OR** pattern should resolve into a treedata type.

```
(xml | json) aValue = ...;
```

The only built-in operator available for **`treedata`** values is the same operator for XML and JSON values; which is the XPath operator. The semantics of this is operator either applies XML XPath or JSON XPath, depending on the case.

```
treedata aJsonOrXml = ...;
var results = aJsonOrXml select xpath{/A/B/C};
```

2.1.7. Conversions

OPN supports implicit conversions for predefined types as well as explicit conversions that use the **`as`** expression form. Explicit conversions are part of the standard library and can also be user-defined. Implicit conversions are fixed. The following table describes implicit conversions.

Source	Implicitly converts to
byte	short, ushort, decimal
sbyte	short, decimal
ushort	int, uint, decimal
short	int,decimal
uint	long, ulong, decimal
int	long, decimal
float	double
long	float, decimal
ulong	float, decimal
bool	int
string	stream
binary	stream
T (where T is not nullable)	T?
T	optional T
selection of XML element	xml (may yield runtime error)
T (where T is a reference type	S

deriving from S)	
T	any

Implicit conversions can be chained, for example, a `byte` can directly convert to an **`int`** value.

Conversions are not implicitly lifted over generic types; therefore, co-variance or contra-variance is not generally supported. However, if conversion for a collection type can be computed by the propagation of it over the elements of the construction, it is supported. For example, the following is legal.

```
byte x = 1;
byte y = 2;
array<int> a = [x,y];  // The same as: array<int> a = [x as int, y as int]
```

In turn, the following is illegal.

```
array<byte> a1;
array<int> a2 = a1;
// Type error, array<byte> cannot be converted to array<int>
```

The second example is produces an error because it would require applying a conversion over an array of arbitrary size that is potentially a change of representation. The first example is allowed because it can be dealt with at compile time and always results in constant execution time.

2.1.8. Type Inference

OPN uses inference so that an author would not have to provide explicit typing information. The following declarations show some examples where the type will be inferred.

```
var a = [1];                 // Inferred type: array<int>
var b = [1,"a"]              // Inferred type: array<any>
var c = []; var d = c + a;   // Inferred type (both c,d): array<int>
```

Note that the preceding example of **`c`** and **`d`** indicates that type inference is context-sensitive within a given set of declarations, following for example what C# provides.

For the sake of readability, fields of types cannot use inference and cannot use the **`var`** keyword. The **`var`** keyword can only be used in local declarations inside statement blocks. Global variables are also included in this restriction, since they can be thought of as fields declared at the document level.

Type inference combines with subtyping by the attempt to infer the most specific type that can satisfy a certain set of type constraints. The most general type is the **any** type. Therefore, for **b** the inferred type is an array with the **any** type as content, as there is no type more specific that can satisfy the constraint.

In some situations type inference cannot determine a unique type, for example, when an expression such as the empty array (**[]**) is used without any context that could indicate the type of its content. In that case the author must supply a conversion to provide type information.

2.1.9. Aliases for Types

OPN enables a way to define an alias for a type by declaring a typedef, resembling the C++ construct.

```
type MyType {...};
typedef MyOtherType = MyType;
```

So every time **MyOtherType** is used, that can be understood as using **MyType**.

```
var a = new MyOtherType{...}  // Equivalent to var a = new MyType{...}
```

A **typedef** can be declared in the same places a type is declared and an **aspect** containing metadata may be attached to it. For a description of aspects see section 2.6 Aspects. A typedef can reference a type, pattern, interface, message, or a typedef construction.

The **typedef** declaration has module scope, in the same way a regular type declaration has. The same name resolution rules apply as well, see section 2.7 for more details.

2.2. Patterns

While types represent the basic structure (or DNA) of values, patterns provide for a detailed categorization of values. Patterns can associate semantic constraints with types; can use regular-expression style operations to describe the content of collections, and more.

A conceptual difference between patterns and types is apparent in that values have a unique type, but there is no unique pattern that can be associated with a value, since a value can match more than one pattern. Therefore, while it is possible to infer types, but it is not possible to infer patterns.

Each pattern is associated uniquely with a type, referred to as the *pattern type*, and is the least type that every matching value can be converted. For many patterns this will be a plain value or reference type. For some it is the **any** type.

2.2.1. Basic Pattern Forms

The most common kind of pattern is a constraint pattern that uses the **where** keyword. For example, the following pattern describes a positive integer domain.

```
pattern PosInt = int where value >= 0;
```

A pattern can be used wherever a type is used in OPN. You can think of the syntax of types as being a subset of the syntax of patterns.

```
type Frame
{
       PosInt Length;
       array<byte> Payload;
}
```

If a pattern is used in a field declaration as in the preceding example, this can be interpreted as a shortcut for an invariant.

```
type Frame
{
       int Length;
       invariant Length >= 0;
       array<byte> Payload;
}
```

The type test (is) operator (section 2.1.2) is extended to pattern matching, as is the type conversion (as) operator.

```
assert 1 is PosInt; assert !(-1 is PosInt);
assert !(new Frame{Length = -1} is Frame);
```

While a pattern can be given a name, as shown in the preceding declarations, it can also be directly provided in a field or variable declaration. In the following constraint pattern example, the constraint can be placed after the declared name.

```
int x where value >= 0;  // Same as: (int where value >= 0) x;
```

An expression can be converted into a pattern by prefixing it with the dollar sign (**$**).

```
pattern One = $1;
```

In the case of a literal expression, the dollar sign can be omitted.

```
pattern One = 1;
```

2.2.2. Regular Expression Patterns

A regular expression can be used to define a pattern that matches a string to the expression. The regular expression is enclosed by braces.

```
pattern Ident = regex {[a-zA-Z_][a-zA-Z0-9_]*};
```

2.2.3. Pattern Conjunction, Disjunction and Negation

Patterns can be combined by Boolean algebra operators.

```
pattern PosInt = int where value >= 0;
pattern EvenInt = int where value % 2 == 0;
pattern PosAndEvenInt = PosInt & EvenInt;
pattern PosOrEvenInt = PosInt | EvenInt;
pattern OddInt = !EvenInt;
```

2.2.4. Collection Patterns

Patterns can be defined for the array, set, and map collection types. Multiplicity constraints can be provided and a wildcard can be used to match the undetermined part of a collection that remains. The following examples illustrates some of these capabilities.

```
pattern P = int where value > 0;
// Any non-empty array that starts with an integer greater than 0.
pattern A1 = [P, ...];
pattern A2 = [P?, ...];  // Empty array or array similar as A1.
pattern A3 = [] | A1;   // Same as A2.
pattern A4 = [P*];     // Array that contains zero or more positive integers.
pattern A5 = [P+];     // Array that contains one or more positive integers.
pattern A6 = [P#1..];   // Same as A5.
pattern A7 = [P#1..10]; // Array that contains 1 to 10 positive integers.
```

2.2.5. Reference Type Patterns

A reference type pattern specifies the name of the type and the patterns for the values of the fields. If fields are omitted their values are ignored for a match. The following is an example of how to use the previously declared `Frame` type that matches frames where the value of the length field is greater than zero.

```
pattern NonEmptyFrame = Frame{Length is int where value > 0};
```

If the match is for a concrete value, the equality sign can be used instead of the `is` keyword.

```
pattern Length2Frame = Frame{Length == 2};
```

For a reference type pattern in a repetition context, one can use the **in**-form to match fields. In this case, the pattern is an array and for each repetition the next element in the array is matched. For the following example, the pattern matches an array of frames where the length of the value of the field is first 0, then 1, then 2.

```
pattern ArrayOfFrames = [Frame{Length in [$0,$1,$2]}*];
```

The **in**-form of reference form of field matching is in particular useful for capturing variables (referred to as capture variables), as discussed in section 2.2.7 Interface Patterns.

2.2.6. Interface Patterns

An interface pattern defines a set of method signatures. A reference value matches an interface pattern if and only if it has been declared to implement the interface. Thus interface patterns are similar to the equally named concept in C# and Java.

Interfaces are considered patterns and not types in OPN because of their property that a given value can match more than one interface. They can be used transparently with all other pattern operators, for example, conjunction (AND (&)) and disjunction (OR (|)).

In the following example, an interface pattern is introduced and implemented by a reference type that describes the feature of comparison.

```
interface IComparable<T>
{
      int Compare(T other);
}

type MyType : IComparable<MyType>
{
      int Compare(MyType other) { ... }
}
```

2.2.7. Capture Patterns

A capture pattern can be used to bind an identifier to the value matched by a pattern or sub-pattern. The value of the bound identifier can then be referred to in subsequent execution paths. The following example matches a frame with non-empty data and delivers the length.

```
switch (x)
{
      case Frame{Length is l:int where value > 0} => return l;
}
```

The general syntax of a capture pattern is `name:pattern`. The value matches if the pattern matches. In that case, the result of the match is bound to the given name. For the case that the pattern in a capture is unconditional, instead of writing `name:any`, you can use the more descriptive `var` notation.

```
switch (x)
{
      case Frame{Length is var l} => return l;
}
```

Capture patterns can only be used in the context of patterns, namely where they build a scope of variable bindings for an expression or statement block. This is, for example, the case for a switch statement or an actor or binding rule pattern (as defined later in this section). The occasions where pattern variables can be used will be described together with the according constructs.

To capture a collection of values associated to quantified patterns (repetitions), the special **`in`**-form to match reference fields can be used. This match extracts an array of values from the same field. For example, the following pattern captures the length field values of an array of frames.

```
switch (x)
{
    case [Frame{Length in var ls}*] => return ls;  // The ls is array<int>.
}
```

The scope of pattern bound variables is the entire pattern. The same variable name may appear twice, but not on the same matching path. For example, in the pattern **`[x:int,x:int]`**, the variable **`x`** appears twice on the same matching path, which is not allowed and will result in a compiler error.

If a pattern variable appears on more than one distinct path, its type will be the least upper bound of all path types. If a pattern variable that is used in one path is not used in another one, its type will be optional. For example, in the pattern **`x:int|x:byte|string`**, the type of the capture variable **`x`** is **`optional int`**, as one path does not bind the variable, and the least upper bound of the two other paths is **`int`**.

2.2.8. Parameterized Patterns

Just as types, patterns can be parameterized by types or patterns, as shown in the following example.

```
pattern BoundedArray<P> = [P#1..10];
```

2.2.9. Enumeration and Flags Patterns

Enumerations and bitmasks are commonly used in protocol design. OPN supports the definition of enumeration patterns that can be used to introduce named constants as well as constrain the values to those constants. By default the constants introduced by an enumeration pattern are of type int. They are consecutively numbered and start at zero if not specified otherwise. The following definition introduces a name for a pattern that matches the integer values 0, 1, or 2. It also introduces the constants **Red**, **Blue**, and **Green**.

```
pattern Colors = enum {Red, Blue, Green};
```

As in other languages, an explicit value can be specified as shown in the following example.

```
pattern Colors = enum {Red = 2, Blue = 3, Green = 4};
```

Enumeration patterns can specify the type that they are derived from. These types are not restricted to integers, but can be any type which has a literal notation as in the following example.

```
pattern TextColors = enum string {Red = "r", Blue = "b", Green = "g"};
```

The ellipsis (…) modifier can be introduced to allow values outside the specified range as the following example demonstrates.

```
pattern ExtraColors = enum {Red = 0, Blue = 1, Green = 2, ...};
```

The final ellipsis (…) allows ExtraColors to contain any value that belongs to the enumeration base class, an integer in this case, even if the value is not listed explicitly as a case. The ellipsis changes the way the implicit invariant for enunerations (and flags) behave. They are not related to default values.

Enumerations are strict by default, and mean that an implicit invariant is associated to the type that enforces the value to be in one of the specified cases, see section 2.10. The **InRange** library function is provided to test if an enumeration value is in the range of explicitly listed values.

```
ExtraColors c = 3;
Assert(!InRange<ExtraColors>(c))  // The c is not in range.
c = 2;
Assert(InRange<ExtraColors>(c))  // The c is in range.
```

In the **ExtraColors** example, '3' is not an explicit case of the enum, but a '4' can be considered to be an **ExtraColors** value because the ellipsis is there. That makes the enumeration non-strict.

2.2.9.1. Flag Patterns

A flag pattern works similar to an enumeration pattern; however, the condition that matches is different. It matches any value that can be constructed by bitwise combination of the declared constants. The following pattern matches the values **0b0**, **0b01**, **0b10**, and **0b11**.

```
pattern Mode = flags {Read = 0b01, Write = 0b10};
```

As enumerations, flag patterns can specify the type from which they are derived. By default, its constants are of type **int**. The selected type must always be an integer type: **ulong**, **long**, **int**, **uint**, **short**, **ushort**, **byte** or **sbyte** .

2.2.10. From Patterns

The **from** pattern can be used to describe the transformation of data as part of a match. It is typically used to describe the decoding of protocol payloads into logical message values.

In the following example, a binary blob is decoded and converted to a framethat is matched against a frame pattern. One of the frame field values is then returned.

```
switch ($[0100000000000000])
{
      case f:Frame{Length is int where value > 0} from BinaryDecoder =>
            return f.Length;
      default:
            return -1;
}
```

The left-hand side of the **from** pattern is a pattern itself, in the example a capture pattern, whereas the right-hand is the name of a method with a particular signature. In the example the method **BinaryDecoder** is declared in the current scope as follows.

```
optional T BinaryDecoder <T>(stream input);
```

The method is supposed to return the **nothing** value if the decoding is not successful, otherwise returns the decoded value. The match will succeed if the method returns a value and its result can be matched against the left pattern operand of the **from** operator.

An OPN compiler constructs the signature of the method from the left pattern of the **from** operator. If a matching method does not exist, the compiler will generate an error.

2.2.11. Localized Strings

A special pattern is available to handle localization, to enable a transparent way to reference localized strings based on the system current locale. A localized string is simply a string pattern that uses an internal **aspect** that contains metadata to provide extra compiler support and it behaves in the same way as a regular string in terms of nullability, mutability and identity. All expressions that take regular strings can take a **LocalizedString**.

```
module MyModule;

LocalizedString DecoderError =
      "Message {0} couldn't be successfully decoded";
// ...
process e accepts m:M{}
{
    switch(m)
    {

        case m1:M1 from BinaryDecoder => ...;
        default => throw Format(DecodedError, m1.ToString());
        }
}
```

In the previous case, the system default locale is assumed. A LocaleInfo aspect, plus a Locale type, is defined in Standard.opn to provide additional information to a localized string.

```
// This aspect can be attached to a LocalizedString or to an OPN module.
aspect LocaleInfo
{
   // Module that the localized string belongs to
   string Module;
   // Locale of the string
   Locale Locale;
};

// Enumeration of all available locales
pattern Locale = enum {EN_US, EN_UK,
      // Complete list follows...
      };
```

The **LocaleInfo** can be attached to a module or to an individual **LocalizedString**. When attached to the module, all **LocalizedStrings** in that module will inherit the specified attributes. To attach this **aspect** to an individual **LocalizedString** overrides the attributes specified at module level, if present.

Following the first example in this section, to reference a **LocalizedString** the implementor should consider the existence of multiple potential localizations for the same identifier. If there are other **LocalizedString** declarations that have the same name identifier and that have been identified as belonging to **MyModule**, declaring them under module **MyModule** or through **LocaleInfo aspect** attribute, then these declarations should also contemplated. For example, we could have a German version of the same string declared in another module.

```
module MyModule.de_DE;

LocalizedString DecoderError = "Nachricht {0} konnte nicht erfolgreich
      decodiert werden"
      with LocaleInfo{Locale == Locale.de_DE, Module == "MyModule"};
```

This is another version of **MyModule** declared in a third module.

```
module YetAnotherModuleWithMoreLocales;

LocalizedString DecoderError = "Le message {0} n'a pu être décodage avec
      succès"
      with LocaleInfo{Locale == Locale.fr_FR, Module == "MyModule"};
```

In the following example, -three different locales for **DecoderError** for module **MyModule** are present. There is the default one in MyModule, with no locale information, the German one in **MyModule.de_DE**, and an additional French one in **YetAnotherModuleWithMoreLocales**.

That means that to reference **MyModule.DecoderError** will actually return one of the three options, depending on the current system locale. The following reference made will be dynamically bound to the right locale.

```
 default => throw Format(DecodedError, m1.ToString());
```

2.3. Grammars

The syntax construct can be used to introduce a grammar for data matching and parsing. Grammars are commonly used in protocol descriptions, for example, to describe the content of HTTP headers, or to describe complex binary data formats.

The syntax construct has some similarity with patterns. However, while a pattern can be understood as a predicate over a value, syntax describes a function that maps from data into some result value as specified by the syntax rule. This function has exactly the same signature as expected in the **from** pattern, as described in section 2.2.10, and can therefore be embedded into patterns using the **from** keyword.

2.3.1. Basic Syntax Forms

The syntax construct uses a variation of EBNF (extended Backus-Naur form), with space ( ) for sequencing, vertical bar (**|**) for alternates, asterisk (*****) for repetition, and question mark (**?**) for option. Terminal symbols like these are EBNF expressions that can be patterns or literals. These operators take EBNF expressions and return new EBNF expressons. Both string and binary data can be processed by a syntax rule by specifying an appropriate decoder for terminal symbols. The decoder is responsible for taking the input stream and parsing it to match the specified pattern. If the pattern is matched, the parsed value is returned by the terminal symbol. Otherwise, the value **nothing** is returned.

There are two ways of specifying a decoder: an explicit decoder that is associated to a specific terminal symbol using a **from** construct, or a default decoder that applies to all terminals in scope, unless overridden. The following example shows a grammar that describes a textual message. The syntax rules for **MessageBody**, **RequestLine**, and **StatusLine** are defined elsewhere.

```
syntax decoder = TextDecoder;

syntax Message = StartLine (Header CrLf)* CrLf ( MessageBody )?;
syntax StartLine = RequestLine | StatusLine;
syntax Header = Name ":" Content;
pattern Name = regex {[a-zA-Z]*};
pattern Content = regex {[^\r\n]*};
pattern CrLf = "\r\n";
```

The **decoder** rule has a well-known name and specifies the default decoding function for terminal symbols. The right-hand side must be a name reference to a method with the same signature as the ones used in the right-hand side of **from** patterns. Default decoding applies to every syntax definition within the same scope that they are placed in, or to an inner scope, unless overridden in this inner scope.

It is also possible to attach specific decoders to terminal symbols that locally override default decoding.

```
syntax Header = (Name from BinaryDecoder) ":" Content;
```

In this case, **Name** will be decoded using the **BinaryDecoder** function. The other two patterns in the expression will use the default encoding.

Binary syntax can also be processed in the same way, by specifying the appropriate decoders. For example, a default binary decoder can be specified.

```
syntax decoder = BinaryDecoder;

syntax Elements = ($[ff] Element)*;
pattern Element = !$[ff]*;
```

Alternatively, local decoders can be attached to syntax symbols.

```
pattern Int4 = int with BinaryEncoding{Width = 4};
syntax TwoInt4 = x:(Int4 from BinaryDecoder)
                 y:(Int4 from BinaryDecoder) => (x<<4+y);
```

In general, if `M` is the name of the method used on the right-hand side of the **`from`** construct, and **`T`** is the type of the pattern used on the left-hand side, an OPN compiler will attempt to resolve this method with the following type.

```
optional T M(stream s);
```

Usually, decoders take into consideration aspect metadata information in order to decide how to parse the input stream. The **`StreamEncoding`** aspect can be attached locally to a syntax construct to affect the decoding behavior of that specific syntax statement.

```
syntax decoder = BinaryDecoder;
syntax Header = Name ":" Content with
      StreamEncoding{TextEncoding = TextEncoding.ASCII};
```

Please refer to section 5.16.1.4 for specifications about the **`StreamEncoding`** aspect.

2.3.2. Parsing

For describing parsers, not only the grammar of the accepted input needs to be specified, but also the output to be produced. By default, syntax rules produce no output beside a value indicating whether the input is accepted or not. By adding output transformations to productions, this can be changed. Taking the preceding example of the message syntax, the following example shows where the arrow (**=>**) operator indicates an output transformation and the right-hand side of this operator is an expression. The modified syntax produces as output an array of array of strings, where each inner array represents a **`name`**/**`content`** pair for a header value.

```
syntax Message =
      StartLine hs:(h:Header CrLf => h)* CrLf ( MessageBody )? => hs;
syntax Header = n:Name ":" c:Content => [n,c];
```

In the **`Message`** rule, the inner part of the repetition produces an array of strings that is bound during each iteration to the variable `h`, whereas the outer part produces an array of such arrays, bound to the variable **`hs`**. The overall production delivers this array.

Captured variables such as `h` and `hs` are scoped to be accessible from the point they are introduced in a sequence to the end of that sequence. If a sequence is nested, for example in alternatives, the scope will be limited to that nesting.

In general, each production is considered to have a type, describing which value it produces. The type is derived from the transformations in the production and the operators applied in the production. The typing rules are the following:

- A literal, regular expression, or pattern name has the type of the **`literal`**/pattern; it produces as output the consumed input.
- The production **`p => E`** has the type of the expression **`E`**. If no transformation is given, it has the type of **`p`**.
- The production **`p1 p2`** (sequential composition) has the **`p2`** type of production.
- If **`p`** is a production of type **`T`** then **`( p )*`** has the type **`array<T>`**.
- If **`p`** is a production of type **`T`** then **`( p )?`** has the type **`optional T`**.
- If **`p1,p2`** are productions of type **`T1,T2`** then **`T1 | T2`** has as a type the least upper bound of **`T1`** and **`T2`**, which may be the **`any`** type.

2.3.3. Tokenization

The input of syntax is tokenized on the fly using the definitions of the terminal symbols, and applying longest prefix match. Tokenization can be influenced by providing special rules using well-known names.

The **`ignore`** rule specifies that all input matching the given pattern should be ignored (skipped) during parsing.

```
syntax ignore = regex {[ \t\r\n]*};
```

The **`separator`** rule specifies that all input matching the given pattern should be considered as a separator, breaking the default of the longest prefix match.

```
syntax separator = regex {[(){\}.]};
```

These rules apply to every syntax definition in the same scope they are placed within, or an inner scope, unless overridden in this inner scope.

2.3.4. Look-Ahead and Semantic Condition

Grammars use by default LL(1) parsing, meaning the next symbol in the input is used to decide which production to take, as described in section 2.3.2. If there is ambiguity, then the first matching production according to textual order is taken. A *semantic condition* can be used to disambiguate a selection of alternatives further. A *semantic condition* is a Boolean expression embedded in a production that can access the input and for example perform an arbitrary look-ahead.

Access to the input is provided by a variable named `input` available in the *semantic condition* that represents a function. The input function has type **`optional uint input(int bitPosition,int bitWidth)`**, and delivers the next 0-32 bits in the **`input`**, relative to the current bit position. Thus **`input(7,32)`** extracts 32 bits starting at bit position 7 starting from the unconsumed input. For interpreting the underlying stream as textual data, the function **`token`** is provided with type **`optional string token(int position)`**, and delivers the **`token`** at the given position relative to the current unconsumed input. See section 2.3.3 for tokenization rules. Thus **`token(0)`** delivers the first unconsumed token, **`token(1)`** the next, and so on. The function delivers the value **`nothing`** if no token is available for the given position. The default encoding is considered for interpreting the stream as textual data.

As an example, the following grammar defines context sensitive parsing of names and uses a *semantic condition* (in square brackets) to disambiguate.

```
set<string> types = ...;
syntax TypeName = [|token(0) in types|] n:Name;
syntax ValueName = n:Name;
syntax Declaration = TypeName TypeDef | ValueName ValueDef;
```

2.3.5. Syntax Rules as Methods

A syntax rule is in fact a method that takes as input a **`stream`** and produces an optional output, representing on success the parsed input. For example, the rule `Message` introduced in the last section represents the following method.

```
optional array<array<string>> Message(stream s);
```

The representation of syntax rules as methods is transparent, and they can be simply called as such, or used from patterns as described in the next section.

2.3.6. Using Syntax in Patterns

Syntax can be used in patterns by using the **`from`** pattern form. The following example illustrates the syntax rule **`Message`** from the preceding definition and is used to recognize and extract **`headers`** from the message content.

```
message Frame
{
      string Content;
}
// ...
void M(Frame x)
{
     switch (x)
     {
         case Frame{Content is var headers from Message} => ...;
     }
}
```

2.3.7. Parameters in Syntax Rules

Syntax rules also support input parameters that can be used later to specify *semantic conditions* or to pass to inner rules. When passing an actual parameter to a rule, any valid expression can be used as long as the type matches the type declaration.

Thinking in syntax rules as functions, parameters are just additional information besides the incoming data stream that can be used to drive the parsing, for example.

```
// The header is just an integer that represents the size of the body.
syntax header = x:int body[x];
syntax body[int x] = [|x <= 8|] smallBlob | [|x > 8|] bigBlob
```

2.4. Behavioral Scenarios

OPN provides behavioral scenarios that can be used to match message sequences that follow a user-definedstructure. A **scenario** can be used later in the context of the following:

- Virtual operations to be able to abstract an identified message sequence into a single operation, as descrbed in section 2.8.5.
- Actor observation rules to simplify tracking sequence validation constraints by specifying sequences of messages that trigger ADM changes as a unit, and is described in section 2.9.2.

Please refer to these specified sections to see how scenarios can be used in those contexts.

Behavioral scenarios have some similarities with grammars that are described in the previous sections. While grammars deal with binary and textual data, behavioral scenarios are specificsally targeted for dealing with messages. They also resemble an EBNF-style construction.

2.4.1. Basic Syntax and Semantics

For a good starting point and a list of real-life examples of how scenarios are implemented see section 4.5. The following is an example of a **scenario**, to show how it is implemented.

```
scenario RequestReadResponse = Request{Id is id:int}
                               (Read{Id == id})+ Response{Id == id}
```

The available operators used to define a **scenario** are the following:

- Terminal symbols match atomic messages with an optional direction keyword **issues** or **accepts** that can include capture variables. When the direction is omitted, both **issues** and **accepts** match. For example, assume message **Frame** is defined, with a field **Length**. The following expression is valid.

  ```
  Frame{Length is int where value > 0, Length - 2 is l:int}
  ```

  Wildcards noted by the underscore (_) can also be used to match any single message.

- Composition operators are shown in the following table. Both **A** and **B** are arbitrary scenarios. An alternative textual syntax is provided for each symbolic operator.

Operator syntax	Semantics
A B **A next B**	Sequential composition. A is matched, immediately followed by B.
A | B **A or B**	Alternate composition. Either **A** or **B** is matched. Shortcut semantics is used: if one of the branches matches, the other one is not explored. The branch to be explored first is not guaranteed and it is left to the implementation.
A || B **A fork B**	Alternate composition without shortcut semantics. Both branches are explored, and the result is the union of the two potential matches. See the clarification notes following this table.
A? **A optional**	Optional match. A is matched, or no messages are consumed. The behavior is eager: if A can be matched, then the input stream is always consumed. If A cannot be matched, then the input stream is not consumed, but considered to match the **scenario**.
A* **A repeat**	Kleene star. Represents zero or more repetitions of A, with no interleaving.
1. **A+** 2. **A repeat [n]** 3. **A repeat [,m]** 4. **A repeat [n,]** 5. **A repeat [n,m]**	A is repeated, respectively: 1. At least one time (equivalent to A repeat{1,}) 2. Exactly n times 3. At most m times 4. At least n times 5. At least n and at most m times

	Both n and m are arbitrary integer expressions.

`!A`
`not A`
	Negation. See the clarification notes following this table.

`A & B`
`A permute B`
	Permutation. Equivalent to A B | B A.

`[|Boolean exp|]`	Semantic condition. If the Boolean expression holds, the match succeeds. If it doesn't, the match fails. The Boolean condition can refer to any visible variable in the scope where it occurs. No input is consumed.
`(A)`	Subexpression. Groups an expression to specify its precedence with respect to surroundings expressions.

The following gives some clarifications about the semantics:

- The Kleene star does eager matching. That means that it consumes as many characters as it can. For example, matching the message sequence **`AAA`** with the scenario **`A*`** will only return **`AAA`** as result. Observe that matching **`AAA`** with **`A*A`** will fail (**`AAA`** is consumed by **`A*`**, and the sequence is over).
- Negation can **only** be applied to:
- Terminal symbols. **`A`** message matches a negated terminal **symbol *A*** if and only if the message does not match ***A***
- Terminal symbols sequentially composed. **`!(AB)`** is equivalent to **`!A | A!B`**.
- Terminal symbols loosely sequentially composed (see section 2.4.3 for definition of loose). **`!(A ->B)`** is equivalent to **`!A | A -> !B`**.

Negation cannot be applied to scenarios that involve other operators. The following describes the grammar details.

- The scope and type for variables follow the same design as for patterns. If a scenario variable appears on more than one distinct path, its type when referenced outside a particular path will be the least upper bound of all path types that apply to the referencing context. If a **`scenario`** variable that is used in one path is not declared in another one, its type will be optional. Re-declaring parameter names or variables is not allowed and will result in a compiler error.
- A special Boolean function **`eos()`** (end of sequence) is available in the context of a **`scenario`** to be used in semantic conditions. This function returns true if the input sequence of messages is completely consumed by the **`scenario`** at that point.

- The **fork** operator creates up to two completely independent matching branches. Given two scenarios **A** and **B**, the semantics of the overall matching behavior for **A || B** is equivalent to match **A** and **B** independently against the same input stream and then to **union** the corresponding results.
- Every time an expression that depends on OPN state is evaluated the current state environment is taken. This applies to operators such as semantic conditions and numeric expressions used in repeat statements. That means that, for example, in a **repeat** context the same expression can yield a different value, since capture variables may change. In practice that means that if the act of matching affects the result of these numeric expressions, the overall result is hard to predict.
- A semantic condition is a **scenario** that does not consume input messages and succeeds or fails depending on the result of evaluating its Boolean function. Since input is not consumed, applying quantifying operators (**star**, **plus**, **repeat**, **optional**,and so on) to a semantic condition does not have any observable effect.
- Scenarios can only reference parameters or capture variables. They cannot reference external variables. This restriction includes both semantic conditions and numeric expressions used in, for example, the **repeat** operator. If functions invoked in semantic conditions or numeric expressions change the value of parameters or capture variables, the matching result is nondeterministic.

2.4.2. Matching Semantics

Scenarios describe a nondeterministic matching function. This means that for a given sequence of messages, the result of matching it with a given **scenario** returns a set of sequences of messages. Each sequence in that set represents a matched sequence. Every time a subsequence matches a **scenario**, the messages that conform to that sequence are not analyzed again for the same **scenario**. A more formal definition is given in the following paragraph.

Given a message sequence **msgs** and a scenario **s**, each message **msgs[i]** is analyzed in sequential order as the first message of a potential match. If a subsequence **msgs[i, j]** is found to match **S**, then **msgs [i, j]** is added to the result (unless **drop** is present, to be explained in section 2.4.6) and the analysis continues from **msgs[j+1]. If msgs [i, j]** does not match **S**, the analysis continues from **msgs[i+1]**.

2.4.3. Strict vs. Loose

By default, scenarios specify a strict order between messages. For example, the scenario **A B** does not allow anything different from **B** after matching **A**. To be able to specify a loose order, where any message can occur in between, two special operators are available. These are the **later (->)** and the **interleave** operators.

For the case of the Kleene star, an exit criterion can be optionally provided. Observe that a loose Kleene star without an exit criterion will always consume the entire sequence.

The following table describes loose operators given two arbitrary scenarios **`A, B`**:

Operator syntax	Semantics

`A -> B`
`A later B`
	Loose concatenation. A is matched; any sequence of messages that do not match B can occur after A, and then the first sequence of messages that matches B, which completes the match.

`A interleave`
`A interleave until B`
	Loose Kleene star. A is matched as many times as possible, allowing interleaved messages that do not match A. When B is specified, the matches stop when the first match for B occurs. See the examples following this table.

`A interleave [n]`
`A interleave [,m]`
`A interleave [n,]`
 	
`A interleave [n, m]`
	A is repeated (allowing interleaving), respectively:
	1. Exactly n times
	2. At most m times
	3. At least n times. In this case “until B” can be appended as stop criteria.
	4. At least n and at most m times
	Both n and m are arbitrary integer expressions.

The following are some examples:

- Given the scenario **`A -> B`**:
  - Sequence $A_1C_2C_3B_4B_5$ matches, and the result is $\{A_1B_4\}$.
  - Sequence $A_1C_2C_3B_4A_5B_6$ matches, and the result is $\{A_1B_4, A_5B_6\}$.
- Given the scenario **`A interleave until B`**:
  - Sequence $A_1C_2C_3A_4D_5D_6A_7B_8$ matches, and the result is $\{A_1A_4A_7\ B_8\}$.
  - Sequence $A_1C_2C_3A_4$ does not match.
- Given the scenario **`A -> B || C -> D`**:
  - Sequence $A_1C_2C_3A_4D_5C_6D_7A_8B_9$ matches, and the result is $\{A_1B_9, C_2D_5, C_6D_7\}$.

Observe that messages skipped during a loose match are not part of the result.

1.1.4 Setting Backtrack Points

As mentioned in section 2.4.2, every time a subsequence matches a **`scenario`**, the collection of messages that conform to that sequence is not analyzed again for the same **`scenario`**. We provide a way to have a finer-grained control on this behavior by defining a **`backtrack`** operator that sets backtracking points as the input stream is consumed, for example.

```
scenario DataPerId = backtrack(Data{Id != 3})
                              (Data{Id is var id} -> Data{Id > id})
```

In the preceding example, every time the input message under consideration matches the pattern (that is, when it is a **Data** message with an **Id** field different than 3) that point in the input stream will define a **backtrack** point. When the process of matching the provided **scenario** is completed, the matching will continue from the defined **backtrack** point and that will be done in order, if more than one **backtrack** point is defined.

For example, consider the following message sequence.

```
Data$_1${Id = 1} Data$_2${Id = 2} Data$_3${Id = 1} Data$_4${Id = 3} Data$_5${Id = 2}
                                                              Data$_6${Id = 1}
```

Given the provided pattern, all messages except from **Data4{Id = 3}** define **backtrack** points. The first run of **DataPerId** over the sequence will return the following.

```
R1 = {Data$_1${Id = 1} Data$_2${Id = 2}, Data$_3${Id = 1} Data$_4${Id = 3}}
```

But since **Data2{Id = 2}** defined a **backtrack** point, the whole pattern is run again from that point, so in the second pass the result will be the following.

```
R2 = {Data$_2${Id = 2} Data$_4${Id = 3}}
```

The third pass is defined by **Data3{Id = 1}**, therefore the following will be the result.

```
R3 = {Data$_3${Id = 1} Data$_4${Id = 3}}
```

The fourth packet is ruled out by the pattern, so the next point is defined by **Data5{Id = 2}** and its result is empty .

```
R4 = {}
```

The last point is defined by **Data6{Id = 1}**, whose result **R5** is also empty. That means that the result of applying this **scenario** is:

```
R1 U R2 U R3 U R4 U R5 = {Data₁{Id = 1} Data₂{Id = 2}, Data₃{Id = 1}
      Data₄{Id = 3}, Data₂{Id = 2} Data₄{Id = 3}, Data₃{Id = 1} Data₄{Id = 3}}
```

So if **`A`** is an arbitrary **`scenario`**, the new construction to build a **`scenario`** is the following.

Operator syntax	Semantics
`backtrack (pattern) A`	Returns the union of evaluating A against the input stream starting the evaluation from all the messages that matched **`pattern`**.

The following shows another example.

```
scenario ReadWrite = backtrack(Read)
                              (Read{User is user} -> Write{User == user})
```

How it behaves is demonstrated when applied to the following sequence.

```
Read{User = "Ning"} Read{user = "Paul"} Read{user = "Andrey"}
      Write{user="Paul"} Write{User = "Yiming"} Write{User="Andrey"}
```

The outcome will be the following.

```
{Read{User = "Paul"} Write{User = "Paul"}, Read{User = "Andrey"}
      Write{User = "Andrey"}}
```

The following lists some considerations when defining **`backtrack`** points:

The first message of a sequence is never flagged as a **`backtrack`** point, even if the provided pattern matches. Flagging the first message as a **`backtrack`** point will always define an infinite loop and doesn't change the result, since neither the input sequence nor the pattern changed. Some side effects used in function invocations could impact this, but these side effects are not recommended to be used for scenarios.

For the same reason as the previous point, if a message is flagged more than once as a **`backtrack`** point, this doesn't have any effect and can be considered as being flagged just once. The following lists rules for defining backtrack points :

- The provided pattern can use any parameter passed to the **`scenario`**.

- Bound variables established by a provided pattern cannot be used in the body of the **scenario**. The same holds for **scenario out**parametersthat are not visible for the provided pattern.
- The **backtrack** operator can only be used as a top level operator. Allowing this operator in other places doesn't have clear semantics, so those constructions are directly forbidden. Consider that the **backtrack** operator acts as a global switch for a given top level pattern since it affects the overall matching semantics with respect to the input stream. The exception is when combining scenarios (see section 2.4.7 **Error! Reference source not found.**), since this operation involves combining independent scenarios.

2.4.5. Scenario Parameters

Scenario parameters can have **in** or **out** parameter modifiers that can be used at declaration time. These modifiers are optional, and the **in** modifier is assumed when no modifiers are specified. The **ref** modifier is not allowed and will result in a compiler error. An **out** parameter is committed when a match is found. If a particular branch does not assign a value for an **out** parameter, the default value is given for that branch, for example.

```
scenario RequestResponse[int id] = Request{Id == id} -> Response{Id == id}

scenario ResponseResult[int id, out binary payload] = Request{Id == id} ->
      Response{Id == id, Payload is payload:binary}

scenario RequestData[out array<string> users] =
      Request{Id is id:int} Data{Id == id, User in users}*

scenario AB[int i, out int j] = A{Field1 == i} B{Field1 == i, Field 2 is j}
scenario CS1D = C{Field3 is var i} S1[i, var j] D{Field4 == j}
```

Expressions such as **var j** can only be present when passing an **out** parameter to a **scenario**.

2.4.6. Result of the Match

By default, if a subsequence matches a **scenario**, the result of the match is the sequence itself. For example, if the message sequence **ABBCABC** is matched against the scenario **B*C**, the result will be the set **{BBC, BC}.** The function **drop**, as shown in the following example, can be used to specify that a given subpattern should be used for matching, but no messages should be associated as part of the result. For example, note the given messages **Req**, **Read**, **Write** and **Res**.

```
scenario SimpleScenario = Req drop((Read | Write)+) Res
```

Also note the following given sequence.

```
Req Read Write Read Read Write Res
```

The result of the match will be the following.

```
{Req, Res}
```

Observe that to decide if a sequence matches or not, the **drop** function is not taken into consideration. For example, the following sequence will not match **SimpleScenario**, since there should be at least one **Read** or **Write**.

```
Req Res
```

2.4.7. Combining Scenarios

Scenarios can be used in other scenarios as expressions. Specified input or output parameters must be provided appropriately.

```
scenario MyScenario[int i, out DateTime time] = Req{Id == i}
                              Read+ Res{Id == i}
scenario MyOtherScenario[out DateTime time] = ConnectRequest{Id is id:int}
                              MyScenario[id, time] ConnectResponse{Id == id}
```

No circular references are allowed when referencing scenarios. This includes self-recursion.

Scenarios can also be *piped* together, where the output of a scenario can be used as the input to a subsequent scenario. Given two scenarios **A** and **B**, the output of the scenario **A |> B** is defined as follows.

Operator syntax	Semantics
A |> B	Given an input sequence, scenario **A** returns a set of sequences **Sa = {S1, S2,…, Sn}**. For each **Si** in **Sa**, **Si** is taken as the input for **B**, so for each **Si**, **B** returns a set **Sbi**. The output for **A |> B** is the set union of **Sbi** for each **i**.

The following shows an example.

```
// The pattern "connect, read/write, close" is matched. A "connect" message
// sets a backtrack point, so intermingled conversations can be identified.

scenario ReadWrite = backtrack(Connect)
                        Connect{Id is var sessionId} ->
                        ((Read{Id == sessionId} | Write{Id = sessionId})
                        interleave until Close{Id == sessionId});

// After applying ReadWrite, each sequence of read/writes within the same
// session is returned as an individual result. Then check that for each read
// request there is a write response that sets a backtrack point for every
// read.
scenario ReadLaterWrite == backtrack(Read)
      Read{ReqId is var Id} -> Write{RespId == id}

// The following puts everything together.
scenario ReadLaterWritePerSession = ReadWrite |> ReadLaterWrite
```

Piped scenarios may share capture variables or parameters as long as they are in scope.

2.5. Methods, Expressions and Statements

OPN provides method definitions for the purpose of building abstractions for modeling. The typical expression and statement forms can be used. Some of the important differences to other languages are described in this section.

2.5.1. Method Definitions

Method declarations have a similar syntax as in languages C# or Java. OPN supports **`ref`** and `out` parameters in method declarations. A method that has no body will throw a not-implemented exception if executed at runtime. The following are some examples.

```
int Succ(int x) { return x+1; }
void Incr(ref int x) { x++; }
void Decr(ref int x);  // Will throw not-implemented exception if called
```

Methods that are declared at module level are implicitly static. Methods declared at reference type level are implicitly instance-based, unless preceded with the `static` keyword.

Static methods can use the `this` designation on their first parameter to enable receiver style notations.

```
int Add(this int x, int v) { return x+v; }  // Enables x.Add(v) notation
```

A property-style notation can be introduced by prefixing a method declaration with the get or set modifier, as in the following example.

```
type T
{
      int x;
      int get X() { return x; }
      void set X(int v) { x = v; }
}
```

On instances of type **T**, the notation **t.X** can now be used to call the getter method **X**, and **t.X = v** to call the setter of **X**.

Properties can also be defined using the `this` designator. The previous declaration is therefore equivalent to the following.

```
type T
{
    int x;

}
int get X(this T t) { return t.x; }
void set X(this T t, int v) { t.x = v; }
```

Operators and conversions can be defined using notation asfollows.

```
int operator + (int x, int y) { return x + y; }
int operator as (long x) { ... }
```

2.5.2. Patterns in Methods and Statements

As already mentioned, patterns can be used in all places where types do appear. Also, method declarations can use patterns for parameters and return values.

```
pattern PosInt = int where value >= 0;
PosInt M(PosInt x) { return x - 1; }
```

The meaning of the preceding example is identical to the following example.

```
PosInt M(int x)
{
      assert x is PosInt;
      int r = x - 1;
      assert r is PosInt;
      return r;
}
```

OPN allows omitting types and patterns in local declarations using the **var** keyword. Type inference is used to infer the types of such declarations, as discussed in section 2.1.8.

```
int Sum(int x, int y, int z)
{
      var t = x + y;
      return t + z;
}
```

The switch statement in OPN is extended to use not only constants but also patterns; those patterns can use capture variables to extract values. In difference to C-like languages, the variables introduced in the scope of a switch case are local to the case. Also, the case is implicitly terminated by a break, a break can be given if desired, but it is redundant.

```
int M(Frame x)
{
      switch (x)
      {
            case Frame{Length is var l} => return l;
            default => throw "Expected a Frame";
      }
}
```

OPN supports exception, as apparent from the previous example. Any value can be thrown as an exception, similar as in C++. A catch clause can use a pattern to match the exception value.

```
try
{
    // ...
}
catch (int x where value > 0) { ... }
```

2.5.3. Binders

OPN supports the **foreach** statement. The **foreach** statement uses a general concept of a *binder* (by the use of the **in** keyword) that is shared between other constructs. In the simplest form, a binder looks, together with **foreach**, as known from languages such as C# and Java.

```
var s = [1,2,3,4];
var r = 0;
foreach (int x in s) r += x;
assert r == 10;
```

Instead of using a type when declaring the bound variable, the binder can also use patterns designated by the **where** expression. In that case only those values will be iterated that match the pattern.

```
foreach (int x where x % 2 == 0 in s) r += x;
assert r == 6;
```

As in C#, the type of the bound variable can be inferred.

```
foreach (var x where x % 2 == 0 in s) r += x;
assert r == 6;
```

Indeed, one can also use a named pattern.

```
pattern EvenInt = int where value % 2 == 0;
foreach (EventInt x in s) r += x;
assert r == 6;
```

The **foreach** binder exprssion can iterate over more than one variable, and variables can be introduced by ranging over a collection or by assigning a value, as in the following example.

```
var s = [1,2,3];
var t = [2,3,5];
var r = 0;
foreach (var x in s, var y = x + 1, var z where value == y in t) r += x;
assert r == 3;
```

As in other modeling languages, the concept of a binder can be used in a few other constructs, where it has the same meaning as the **foreach-in** expression that regards the range of values being iterated. The difference is in what is done with the iterated values. First, universal and existential Boolean quantifier expressions are available as those have very common usage in expressing constraints over collections.

```
var s = [1,2,3];
assert all (var x in s) x > 0;
assert some (var x in s) x > 0;
```

Finally, there is a simplified form of LINQ selection.

```
var s = [1,2,3,4]
var t = from (var x where value % 2 == 0 in s) x+1;
assert t == [3,5];
```

The behavior for **from** is similar to **foreach** when iterating over more than one variable:

```
var s = [1,2];
var t = [3,4];
var r = from (var x in s, var y int t, x + y != 6) new Tuple(x,y);
assert r == [(1,3),(1,4),(2,3)];
```

The domain of OPN is not in querying data sets, therefore full LINQ functionality is not directly supported. However, with anonymous functions that are not explicitly named (such as lambda functions) being part of the notation and library functions makes the functionality indirectly available.

In order to enable iteration over collections in binders, the value does not need to implement a particular interface, but just needs to support methods matching the following signature.

```
S GetIterator(this T x);
V GetNext(this T x, S s);
bool MoveNext(this T x, ref S s);
```

Here, the type `S` is arbitrary, representing the state of the iterator. All collection types in the standard library support this pattern. For example, the array type uses the following methods.

```
int GetIterator<T>(this array<T> x) { return -1; }
T GetCurrent<T>(this array<T> x, int s) { return x[s]; }
bool MoveNext<T>(this array<T> x, ref int s)
{
  if (s < x.Length-1) { s++; return true; } else return false;
}
```

2.6. Aspects

In OPN, the use of metadata for describing additional contructs of the specification is ubiquitous. Therefore, the notation supports rich means to attach metadata to declarations. The notation construct to achieve this is referred to as *aspects*. Aspects are similar to attributes in C# and annotations in Java, yet there are differences in expressiveness and syntax as to the central role metadata has for the OPN domain.

Conceptually, as in C# and Java, metadata is information not required for the core semantics of a specification. The core semantics consists of the mathematical model required to describe raw data values and computations on that data. Metadata can be erased from a specification without changing its meaning with respect to the core semantics. However, interpretations of OPN by specific tools may require specific metadata to be present for proper functionality. Also, runtime behavior of library functions may be dependent on interpretation of metadata. Such dependencies are documented together with these tools. In some documented exceptions, aspects may also indirectly influence execution semantics of OPN, as they describe configuration of the execution environment.

In contrast to C# and Java, aspects are introduced by a special declaration form. This shows that they are not values present at execution time, and the declaration form enables customized notations. Aspects are *attached* to language elements by using the affix (**with**) notation instead of the prefix notation as in C# and Java. This demonstrates the fact that they represent additional and not core information about that declaration. The following shows an example of an aspect declaration and usage.

```
aspect Info
{
       string Description;
       int Version;
}

type Person { string Name; int Age; }
     with Info{Description = "The type of persons.", Version = 1};
```

Syntactically an aspect declaration is very similar to a type declaration: it consists of a set of fields, and it is constructed by initializing those fields.

An aspect can use the well-known name **this** in a field declaration in order to relate constraints within the declaration to which an aspect is attached. The **this** field then represents the value to which the aspect associates. This concept is referred to as *context binding* of aspects. The following is an example of an aspect with *context binding* that describes the width in bits of an integer, ensuring that the actual value fits in that width.

```
aspect IntegerEncoding
{
     int this;
     int WidthInBits;
     invariant this >= MaxValue(WidthInBits) && this <= Int.Max(WidthInBits);
}

type Frame
{
       int Value with IntegerEncoding{WidthInBits = 6};
}
```

While an aspect value is not actually constructed at runtime, the constraint it imposes is propagated to its application context. For the execution semantics, the preceding example is therefore equivalent to thefollowing example.

```
type Frame
{
       int Value;
       invariant Value >= Int.Min(WidthInBits) &&
                 Value <= Int.Max(WidthInBits);
}
```

2.7. Modules, Name Resolution, Visibility, and Lifetime

OPN supports a simple concept of modularization, which is similar to namespaces in C# or packages in Java. An OPN document can be prefixed by a declaration indicating a module name. All names declared in the document do then implicitly belong to that module. The fully-qualified name of the entities in the module consists of the module name separated by a dot (.) from the declaration name.

In the following example, the type **`Person`** has the fully qualified name **`DataModel.Core.Person`**, and the method **`Summary`** has the fully qualified name **`DataModel.Core.Summary`**.

```
module DataModel.Core;

public type Person { ... }
public string Summary(Person p) { ... }
```

In order to use the declarations from another module in a document, one has to supply a using clause to include the class to be used. The names from another module can then be referred with or without qualification. Names are not implicitly available in fully qualified form without a using clause, making usage relations explicit, for example.

```
module DataModel.Extensions;
using DataModel.Core;

type Employee : Person { ... }
```

A module can be defined using multiple documents. The using clauses are only relevant for the document that they appear in and are not inherited from other documents.

Modules can contain global methods and fields that are implicitly static. These are methods declared directly under the document scope. Types can contain both static and instance methods and fields. A declaration with static lifetime can only access other declarations that are static as well.

OPN supports two levels of visibility: **`public`** and **`internal`**. Internal declarations are only visible in a project (a set of coherent documents) to which the declaration belongs. As a modeling language, the default in OPN is **`public`**, not **`internal`**.

Name resolution is based on lexical scoping and context information. In a lexical scoping context, inner declarations shadow outer declarations, but outer declarations can be accessed using full name qualification. The using clauses of a document constitute an outer lexical scope relative to the document, such that they are implicitly shadowed by document declarations.

In some cases, name resolution is influenced by the expected resolved entity. For example, in a context where a pattern or type is expected, only names for such declarations are considered.

In a context where a value is expected, pattern and type names are not considered. If a method name can resolve to different declarations, parameter types, stripping off any patterns, are used to disambiguate the declaration. Disambiguation does not consider global type inference.

2.8. Protocol Architecture

The purpose of OPN is the definition of architecture and behavior of communicating systems. While the constructs that have been described in the previous sections are available to generically support this purpose, the ones described here are specific to the domain.

The core concepts for describing protocols are *messages, endpoints*, and *actors*. These are augmented by *contracts* that cluster *messages*, and by *roles* that cluster *endpoints* with shared properties. A *protocol namespace* is a namespace that can contain these elements. *Endpoints* and *roles* can have a *data model* that is used to describe their behavior using *actor* processing rules.

2.8.1. Messages

A message declaration is a reference type declaration with specific properties. The predefined type `any` **`message`** is the base type of all message types.The following, two messages **`Request`** and **`Response`** are introduced as examples.

```
message Request
{
     int Op;
     string Arg;
}
message Response
{
      int Result;
}
```

The following lists the distinct features of messages compared to general reference values:

- Messages can be dispatched and received via endpoints, as discussed in section 2.8.7.
- A message value can be changed from being mutable to being immutable; a process that is referred to as *freezing*. When a message value is *frozen*, all updates on its components will throw a runtime exception. *Freezing* happens at the point of time the message is dispatched to an endpoint. Once a message is travelling via a network, it cannot be updated posterior.
- Messages support the concept of *annotations.* These are values that can be dynamically attached to an existing message value. When a message is *frozen*, annotations can still be updated. Annotations are discussed in section2.8.2.

- Value parameters are also supported for messages.

2.8.2. Message Annotations

Annotations are used to attach data to message values that are not directly related to the message type contract. This can be data required for a particular message processing implementation, for relation with other messages in a network stack, or for comments provided by the user.

In OPN the type and name of an annotation need to be declared before it can be used. Annotations can be applied to all message types or to specific ones. Once declared, any message value can be queried to check whether it has the given annotation, and what its value is. An annotation declaration takes the following form.

```
// Introduces the comment annotation for all message types.
annotation string Comment;

// Introduces the RequesterComment annotation for the Request message type.
annotation string Request#RequesterComment;
```

In any scope where this annotation is known, one can use the annotation name as a field selector. To differentiate an annotation from a regular field and to access an annotation, the number sign (**#**)operator is used instead of the standard dot (**.**) operator.

```
Response rs;
rs#Comment = "Hello!";
assert rs#Comment == "Hello!";

// RequesterComment is only available for Request messages.
Request rq;
rq#RequesterComment = "Hello!";
assert rq#RequesterComment == "Hello!";
```

The result of selecting an annotation has the logical type `optional T` (with `T` representing the annotation type). Before explicitly assigning a value, an annotation has the value **`nothing`.** One can use an expression such as **`rs#Comment != nothing`** to check for this value, or **`rs#Comment = nothing`** to clear the value of an annotation.

To try to assign or access a value of an annotation on a message for a type that it was not declared will result in a compiler error. A set of annotations are predefined for all message types. A couple of them are in the following list. The full set is given in the Library Reference.

```
module Standard;
annotation DateTime TimeStamp;
annotation guid SessionId;
// And so on.
```

2.8.3. Message Stacking

Messages are constructed based on other messages using processing rules, as described in section 2.9.1. This dependency creates a relation between messages referred to as the *stacking relation*. OPN supports *reflecting* over the stacking relation using library methods, as well as using special forms of patterns in matching; internally the stacking information is an integral part of a message value, but its representation is not directly visible.

The library exposes the **Origins** property method that is available for the **any message** value. This method provides the ability to query and update a set of messages that a given message is built from, for example.

```
array<any message> get Origins(this any message m);
void set Origins(this any message m, array<any message> origins);
```

Note that there are scenarios where the **origins** may not be available or complete: messages may have been garbage collected in the course of circular message buffering, or a message may have been injected into the runtime framework instead of being constructed from other messages. Therefore, protocol models should not rely on the availability of message **origins**. The previous method will return the currently available **origins**, which may change over time and may possibly be an empty array.

The stacking information is also exposed via special pattern syntax Aaligned with the XPath syntax. One can use the double backslash (\\) prefix for a message reference pattern to indicate searching for any message of the given type with given field constraints immediately in the matching input or by traversing the stacking relation. A single backslash is used to move just one step down.

For example, if **TCP.Segment** is the type of the matching input, and if **IP.Packet** is the type of a message indirectly reachable via the **Origins** relation, then from that input the match **\\IP.Packet** will succeed. The following examples are more specific.

```
\\M          // Matches if any message in the stacking relation has type M.
\\M{F is P}  // Matches if any message in the stacking relation has type M
             // with field F matching P.
\\M.F is P   // Same as the preceding message.
\\M.F == E   // Matches if any message in the stacking relation has type M
             // with field F equal to value of E.
M1\M2        // Matches if current message in the stacking relation has
             // type M1 and if there is a message with type M2 directly under
             // the current message in the stacking relation.
```

2.8.4. Operations

The message declarations seen so far introduce one-way messages. Both one-way and two-way messages are referred to as *operations*, and are declared as follows. When no modifier is specified **in** is the assumed default.

```
operation Order
{
      in string Item;
      out bool Success;
}
```

An **operation** declaration can use one or more exception clauses to specify and enable throwing exceptions as in the following example.

```
operation Order
{
      in string Item;
      out bool Success;
}
exception UserNotAuthorized;
```

An **operation** is shorthand for declaring a set of one-way messages: an **in** call message, an **out** return message, and an **exception** message if the operation has exception clauses. They can be thought of as regular messages that add **in** or **out** direction to the fields and optionally an exception case. When creating an operation with a declared exception part, the exception can be optionally included. The exception clause uses a pattern to describe the values that can be thrown as seen in the following example.

```
var op = new Order{Item = "MyItem",
                  Success = true, exception = new UserNotAuthorized() }
```

The exception clause is matched later.

```
process e accepts Order{Item == 0, exception is var ex}
{
      // ...
}
```

An operation can also use a special modifier on one of its parameters, marking it as the **result** parameter, and then use **error** and **success** clauses to indicate what a particular result means. Recall that the dollar sign (**$**) turns any expression into a pattern that matches exactly the value of the expression as the following example shows.

```
operation Order
{
      in string Item;
      result bool Success;
}
success $true with Description("The operation succeeded")
error $false  with Description("The operation failed")
```

The **success** and **error** clauses are only allowed if a result parameter has been declared. If **success** and **error** clauses are given, then the value of the result parameter must match one of them. The **success** and **error** clauses cannot be mixed with **exception** clauses.

2.8.5. Virtual Operations

The virtual keyword can add to operations the ability to abstract messages sequences as a single operation. This construction is useful when individual messages in a contract are intended to be used as a unit. The following example shows how the sample operations shown in the previous section can be turned into virtual operations.

```
virtual operation Order
{
      in string Item = item;
      out bool Success = success;
} =
issues OrderRequest{RequestId is var id1, Item is item:string }
accepts OrderResponse{RequestId == id1, Success is success:bool};
```

In this example, the **request/response** order is specified by placing the two corresponding messages one after the other. In this case the **virtual operation** specifies that an **OrderRequest** is issued first and then an **OrderResponse** is accepted. Notice how **requestId** is used to ensure that a response matches its immediately preceding request, but it is not relevant at the upper operation level.

The body of a virtual operation that follows the equal sign (**=**) is a behavioral scenario. Refer to the sections in 2.4 for details. Scenarios can be anonymous and thus not explicitly named, for an example see section 2.9.2.2. A previously defined scenario can be used as in the following example.

```
scenario MyScenario[out array<string> items, out array<bool> successes] =
(
      issue OrderRequest{RequestId is var id1, Item in items:array<string>}
      accept OrderResponse{RequestId == id1, Success in
      successes:array<bool>}
) interleave until issues OrderClose{};

virtual operation Orders{
      in array<string> Items = items;
      out bool Success = all (var s in successes) s;
} = MyScenario[out var items, out var successes];
```

When using named scenarios in the body of virtual operations, the scenario can only have **out** parameters or constant **in** parameters that are allowed in this context and they can be bound to virtual operation fields. The following is another example of a virtual operation, using repetition operators.

```
virtual operation Orders
{
     in array<string> Items = items;
     out bool Success = all (var s in successes) s;
} =
(
     issues OrderRequest{RequestId is var id1,
                         Item in items:array<string>}
     accepts OrderResponse{RequestId == id1,
                         Success in successes:array<bool>}

)*
issues OrderClose{}?;
```

This virtual operation represents a sequence of request/response messages (with undetermined length) ending with an optional **OrderClose** message.

Additional constraints can be specified by using a **where** clause as in the following example.

```
virtual operation Orders
{
     in array<string> Items = items;
     out bool Success = all (var s in successes) s;
} =
(
     issues OrderRequest{RequestId is var id1,
                  Item in items:array<string>}
     accepts OrderResponse{RequestId == id1,
                  Success in successes:array<bool>}

)*
issues OrderClose{}?
where (items.Length == 0) ==> !Success;
```

The Boolean condition specified in the where clause acts as an invariant: if the condition is not satisfied, the virtual operation is not matched. The evaluation of the **where** clause happens after the matching is done.

Virtual operations can also use exception clauses, as regular operations do as in the following example.

```
virtual operation Orders{
      in array<string> Items = items;
      out bool Success = all (var s in successes) s;
}
exception optional int = reason
=
(
      issues OrderRequest{RequestId is var id1, Item in items:array<string>}
        (
            accepts OrderResponse{RequestId == id1}
            |
            accepts OrderFailure{RequestId == id1, Reason is reason:int}
         )
);
```

Success and error rules can also be specified with virtual operations. The restrictions are similar to regular operations: the result parameter value must match one of the success or error clauses, and if these clauses are used, they cannot be mixed with exception clauses, for example.

```
virtual operation Order
{
      in string Item = item;
      result bool Success = success;
}
success $true with Description("The operation succeeded")
error $false  with Description("The operation failed")
=
issues OrderRequest{RequestId is var id1, Item is item:string}
accepts OrderResponse{RequestId == id1, Success is success:bool};
```

Virtual operations are not implicit, in the sense that they should be exposed by individual endpoints at declaration time. Endpoints can *issue* or *accept* virtual operations in the same way they do with regular messages or operations, and virtual operations can also be part of contracts.

With respect to virtual operation directionality, note the following endpoint example.

```
endpoint SomeEndpoint accepts Order issues Order;
```

The endpoint **`SomeEndpoint`** issues the virtual operation **`Order`** when it issues an **`OrderRequest`** and accepts an **`OrderResponse`**. In the same way, an endpoint accepts this virtual operation if it accepts an **`OrderRequest`** and issues an. The inferred virtual operation direction has to match the direction declared by the endpoint.

2.8.6. Contracts

Contracts are patterns that can match endpoints that they provide or consume. Thus they behave as endpoints, in the way that interfaces behave as reference types. However, in difference to interfaces, contracts are not nonminal: any endpoint that provides the same set tof messages (or more) as specified in the contract can match the contract, independent of whether the contract is mentioned in constructing the endpoint.

The directionality of a message is of relevance when a message is associated with an endpoint, as discussed subsequently. A number of messages or operations together with their directionality can be clustered in a contract, for example.

```
contract Service
{
      accepts Request {int Op; string Arg;}
      issues Response {int Result;}
}
```

Contracts can inherit from other contracts. Depending on the use of the keyword **`consumes`** or **`provides`**, the directionality of messages flips. This is done using the following notation.

```
contract ServiceWithNotify provides Service
{
      accepts Notify {int Op;}
}

contract ServiceConsumer consumes Service {}
contract ServiceWithNotify2 consumes ServiceConsumer{}
```

In the contract **`ServiceWithNotify`**, the same messages with same directionality are provided as in **`Service`**, except that the message **`Notify`** is added. In the contract **`ServiceConsumer`**, all directionalities are flipped. The contract **`ServiceWithNotify2`** is identical with **`ServiceWithNotify`**. In general, the **`consumes`** keyword flips directionality, whereas the **`provides`** keywords maintains it.

2.8.7. Endpoints

Messages can be issued (sent) and accepted (received) via an endpoint. An endpoint represents a communication port for message exchange. Endpoints can be declared based on messages and contracts. In its simplest form an endpoint declaration lists a set of messages or operations together with the direction offlow.

```
endpoint Server accepts Request issues Response;
```

Instead or in addition to directly listing messages, an endpoint can also refer to contracts.

```
endpoint Server provides Service consumes OtherService;
```

Referring to a contract is equivalent to referring to all the messages from the given contract with the according (original or flipped) directionality.

An endpoint can be indexed, meaning that multiple instances of the same endpoint can exist, one for each index. Endpoint instances are first-class values, which can be denoted in various ways as described later in this section. The name of an endpoint behaves as a type name, matching all instances of that endpoint. The index is given as a list of declarations between brackets (**[…]**) as shown in the following example.

```
endpoint Server[string Address, int Port] accepts Request issues Response;
```

The indices behave as fields of the data state of the endpoint. Additional data state can be provided as part of the body of the endpoint.

```
endpoint Server[string Address, int Port] accepts Request issues Response
{
      public map<int, ConnectionInfo> ConnectionTable = {};
}
```

Given an endpoint instance value, one can use the dot notation to access fields, provided they are public. Note that index fields are always public.

```
Server server;
// ...
var port = server.Port;
var table = server.ConnectionTable;
```

Endpoints can be stacked on each other, using the **over** notation as the following example shows.

```
endpoint Server[string Address] provides ServerMessages;
endpoint Connection[int Id] over Server provides ConnectionMessages;
```

Stacking of endpoints means that for every instance of the underlying endpoint (**Server** in this case) a set of instances for the stacked endpoint exists (**Connection** in this case). This set can be a singleton if the underling endpoint does not have an index. Thus via stacking, endpoints can build a tree-like hierarchy.This demonstrates concepts such as sessions nested within sessions, for example.

A stacked endpoint can have constraints on its underlying endpoint, expressed by a **where** clause. If constraints are given, stacking is only possible if they are satisfied. One typical constraint is that the underlying endpoint is in turn stacked on other endpoints. To declare that a given endpoint is stacked on another one, the **over** form can be used as an operator between endpoint instances and endpoint names.

```
endpoint Service over Soap where value over Http provides Messages;
```

In the preceding declaration, **Service** is stacked on top of an endpoint named **Soap**, which in turn is constrained to be stacked on top of an endpoint declared **Http.**

Constraints on stacking become useful, in particular, if combined with stacking variations. To that end, an endpoint can be declared to have alternative stacking configurations by giving multiple **over** clauses.

```
endpoint Soap over Http | over Rpc provides Messages;
```

This declaration means that endpoint **Soap** can be either stacked on **Http** or on **Rpc**. Note that the previous declaration of endpoint `Service` requires **Soap** to be restricted to the **Http** variant.

If an endpoint is stacked on another one, in order to access index and public data fields from the underlying endpoint instance a capture variable can be explicitly introduced.

```
endpoint T1 { ... int Address; ...}
endpoint T2 { ... int Address; ...}

endpoint P over t1:T1 | over t2:T2 {...}
```

Here **t1** and **t2** provide access to the underlying endpoint instance. At execution time, an endpoint instance will have exactly one of the specified configurations: either **t1** or **t2** will be a **null** value, representing which of the two transports are actually present. For example, to simplify the access to the current transport we can define a property that returns the underlying endpoint address.

```
endpoint P over t1:T1 | over t2:T2
{
      observe this // ...
```

```
        {
                int transportAddr = this.TransportAddress;
                // ...
        }

        int get TransportAddress()
        {
                assert t1 != null || t2 != null;
                return t1 != null ? t1.Address : t2.Address;
        }
}
```

Alternatively, the underlying transport instance can be accessed through the **GetTransport** library function (section 5.6), which does not need an explicit capture.

```
endpoint P over T1 | over T2
{
        observe this // ...
        {
                int transportAddr = 0;
                var t1 = this.GetTransport<T1>();
                if (t1 != null)
                        transportAddr = t1.Address;
                else
                {
                        var t2 = this.GetTransport<T2>();
                        assert t2 != null;
                        transportAddr = t2.Address;
                }
        }
}
```

2.8.7.1. Client Endpoints

A **client** endpoint represents a communication port that sits on the client side of a protocol. It is logically connected to a given **server-side** endpoint (referred to as **regular** endpoint) indicating that messages flow between them.

```
// A regular endpoint, representing a server endpoint.
endpoint MyAppServer accepts Request issues Response;

// A client endpoint, attached to MyServer.
client endpoint MyAppClient connected to MyAppServer;
```

A **client** endpoint is always attached to a given regular endpoint. Client endpoints have an implicit dispatching logic that is dictated by its corresponding regular endpoint: every time a message **m** arrives to a regular endpoint, the same message arrives to all its connected client endpoints with the opposite direction. The order in which the message arrives to each endpoint for both servers and clients is non-deterministic.

Observe that client endpoints do not need to declare which messages they accept or issue: these messages are declared exactly the same as in its corresponding server endpoint with flipped directions.

The **client** endpoints can have actors (both explicit and implicit) in the same way regular server-side endpoints do. For example, we can define an implicit actor for **MyAppClient**.

```
client endpoint MyAppClient connected to MyAppServer
{
    // MyAppClient ADM
    // ...

    observe this accepts Response{}
    {
        // ...
    }

    observe this issues Request{}
    {
        // ...
    }
}
```

Following the preceding example, in the following example an **actor** decodes a message from the transport layer of **MyApp** and dispatches it to **MyAppServer**.

```
actor TransportToApp(MyTransport t)
{
    process t accepts // ...
    {
        Request m = ...;
        dispatch (endpoint MyAppServer) accepts m;
    }
}
```

This means that **MyAppServer** will get message **m** with **accepts** direction as usual. Additionally, **MyAppClient** will get the same message with **issues** direction. The second rule we specified previously the implicit actor for **MyAppClient** will fire.

In this way, actors listening to **MyAppClient** can describe sequence validation rules from a client point of view. These are regular server-side actors, and client state can be tracked independently of the server through client ADMs. Actors involved in the decoding process do not need to be aware of the existence of aclient in the upper level protocol description.

Client endpoints have some of the following particularities.

- Only **`observe`** rules are allowed to be applied to client endpoints. Actors listening to client endpoints are not designed to consume arriving messages.

- Capture variables can be used to bind the server endpoint to the client endpoint and access their ADM.

```
client endpoint MyAppClient connected to s:MyAppServer
{
      // The s can be used here to access the ADM of MyAppServer.
}
```

- A call by the standard library function **`GetServerEndpoint`** can be used to retrieve the corresponding server endpoint.

```
client endpoint MyAppClient connected to MyAppServer
{
      endpoint myAppServer = this.GetServerEndpoint()
}
```

- Additionally, a call by the standard library function GetClientEndpoints is available to retrieve all client endpoints that a server endpoint is connected to.

```
endpoint MyAppServer accepts Request issues Response;
{
      array<any endpoint> myAppClients = this.GetClientEndpoints()
}
```

- Recall that the order in which messages arrive to client and server endpoints is non-deterministic, so no assumptions can be made on the ADM state of the opposite endpoint in terms of the message that just arrived.

- Indexes and transport protocols can be retrieved from the corresponding server endpoint. Client endpoints don't have indexes or transport on its own. They inherit the ones declared on their associated server endpoint.

- Partial order declaration **`follows`** and **`precedes`** can be included in client endpoints as described in section 2.9.3.

2.8.8. Roles

A role defines a cluster of a number of endpoints that can share a given data model, and can optionally include a common base index. The data model is a set of variable declaratons (the state of the entity). It is defined in section 2.8.7. The following is an example of a role that introduces a server with several endpoints.

```
role Server[Address Addr]
{
      int connectionCount;
      endpoint ServiceAccessPoint provides ServiceMessages;
      endpoint ServiceConnection[int Session] provides ConnectionMessages;
}
```

In the preceding example, both endpoints inherit and share address and data model from the role. The variables from the data model of the role can be used anywhere in the scope of the role. There is one instance per role-index of this data model.

If the data model of a role is public, it can be accessed from the outside via any of the endpoint instances.

A role may also declare stacking uniformly, by specifying an endpoint on which all its endpoints are stacked.

```
role Server[Address Addr] over Node
{
      // ...
}
```

2.8.9. Protocol Namespaces

Roles and endpoints can only be placed in a special variation of a namespace, referred to as a *protocol namespace*. In a compilation unit, one uses the keyword `protocol` instead of **`module`** to indicate a protocol name space as in the following example.

```
protocol Windows.P with Documentation{Name = "MS-P", Version = 2.3, ... };
role Server { ... }
role Client { ... }
```

Technically, the protocol namespace behaves as a regular namespace; however, only within such a namespace, roles and endpoints can be declared. In order to support reuse, messages, contracts and interfaces can also be declared outside of a protocol namespace.

2.9. Protocol Behavior

In OPN the description of operational behavior is based on a variation of the *actor* model, providing an event-driven approach. Actors listen to events on endpoints, for instance that a message is accepted or issued. In reaction to this they execute a rule that can update state and may dispatch new messages to other endpoints.

Actors virtually execute atomically, with no interaction between rules of different actors, therefore avoiding the related concurrency problems, and providing high potential for parallelization.

2.9.1. Actors

An actor declaration is parameterized by one or more endpoints and zero or more values. It consists of a local state and a set of rules.

```
endpoint MyEndpoint accepts M1 issues M2;
actor A(MyEndpoint e)
{
      int state;
      observe e accepts M1 { /* statements */ }
      observe e issues  M2 { /* statements */ }
}
```

One of the preceding rules fires if a message is *accepted* or *issued* on the endpoint that matches the pattern **`M1/M2`.** If a rule fires, then the given statements are executed.

The pattern provided in a rule must be based on one of the messages declared in the endpoint. The directionality must be provided, and must match that of the message on the endpoint. The pattern typically uses capture variables (see section 2.2.7) that can be used inside of the statement block.

Rules can be also directly placed in endpoints and in roles, where they represent an implicit actor, as explained in section 2.9.6.

2.9.2. Observe and Process rules

An actor may either *consume* a message on an endpoint or it may passively *observe* it. Consumption is typically used when the actor defines how messages are travelling on the network stack; observation is typically used when the actor enforces a behavioral contract, or collects diagnostic information.

When the **`observe`** keyword is used the rule does not consume the messages on the endpoint. That means if the rule fires, then the message is still available and other actors can fire on the same message. Also, a given actor can have more than one observation rule that matches. Then all of those observation rules will be executed in an undetermined order.

An actor that consumes messages as they arrive uses the following notation, where the rule is prefixed by the **`process`** keyword instead of the **`on`** keyword.

```
actor A(MyEndpoint e)
{
      int state;
      process e accepts M1 { /* statements */ }
```

```
        process e issues  M2 { /* statements */ }
}
```

The processing rule is the same as an observation rule, but differs in its meaning only in that the message will be consumed when the rule fires. Consumption here means that no other rule that attempts to process the message can fire (in the given actor or in another actor). A non-deterministic choice is made if more than one rule could potentially process the message. Independent of that, all observations will be guaranteed to fire before any message is consumed. A finer-grained execution priority can be defined among actors; please refer to section 2.9.3 .

Note that the use of the directionality `issues` or `accepts` is not related to whether the actor reacts on a message or dispatches one. It is related to the direction a message travels to an endpoint. This can be understood as having an independent channel for each direction of travel.

An actor can mix both kinds of rules. This is particularly useful in scenarios where an actor implements a network diagnosis, observing some traffic passively, and participating in a diagnosis related protocol actively. Observe rules and process rules can be given a name by providing an identifier in square brackets.

```
process [P] e accepts M1 { /* statements */ }
```

2.9.2.1. Rejecting and Releasing Messages

A message can be rejected by an actor after its message pattern has matched as in the following example.

```
process e issues  M2 { /* statements */; if (Cond) reject; }
```

A *rejected* message will not be consumed, and made available for consumption by other processing rules. Note that the effect of any global state updates the actor did until rejection is undefined; implementations may roll-back those effects or not. The following example shows the previous actor **`process`** with a condition to **`reject`**.

```
process e issues  M2 { /* statements */; if (Cond) reject; }
```

An actor can also **`release`** a message to indicate that it should no longer be matched against actor rules (see implentation note section 6). After indicating this, the given message is considered to be out of the actor processing chain.

The typical case where an actor releases a message is when the message was initially buffered in the actor internal state waiting for some additional information to come, but later the actor determines that the expected information will never arrive. Under that situation, the actor can indicate that the message will not be further processed so, for example, a UI can display it properly. These situations are explained in the following examples.

Consider a protocol where **request/response** message pairs can arrive out of order, until a **ConnectionClose** arrives, that signal the end of the communication.

```
message Request{int id; ...}
message Response{int id; ...}
message ConnectionClose{}

endpoint MyEndpoint accepts Request, issues Response,
      accepts ConnectionClose;
```

A **request/response** pair is matched using the **id** field, present in both messages. Every time that a **request/response** pair has arrived, the protocol issues an operation to the upper layer. To simplify the example, consider that when a response arrives the corresponding request has always already arrived at some point in the past.

```
actor A(MyEndpoint e)
{
    // Store all the arrived requests that don't have their corresponding
    // responses yet.
    map<int, Request> arrivedRequests = new map<int, Request>();

    process e issues rq:Request{}
    {
        // If the request is not in the map, we store it.
        if (rq.id in arrivedRequests.keys)
        {
            arrivedRequests += {rq.id -> rq};
        }
    }

    process e accepts rs:Response{}
    {
        // The response always corresponds to an already arrived request.
        Request rq = arrivedRequest[rs.id];
        // The operation is built from the request and the response.
        ReqResOperation op = buildOperation(rq, rs);
        // The request is removed from the map.
        arrivedRequests.Remove(rs.id);
        // The operation representing the request/response
        // pair is dispatched.
        dispatch (endpoint UpperEndpoint) accepts op;
    }
}
```

When MyEndpoint gets a **`ConnectionClose`**, no further **`request`**/**`response`** messages can arrive there. That means that all the arrived **`requests`** that still don't have the associated **`response`** will never be paired, and the operation representing the pair will never be dispatched to the upper layer. Here is a case when the actor releases those messages.

```
actor A(MyEndpoint e)
{
    // ...
    process e accepts ConnectionClose{}
    {
      // When MyEndpoint is deleted, all orphan requests are released.
      foreach(Request rq in arrivedRequests.Values)
        {
            release rq;
        }
    }
}
```

Releasing a message is a way to specify that the information that the message is carrying has not been propagated to the actor process chain and that will never be propagated.

It is worth noting that:

- A message is *frozen* when *released*, and all updates on its components will throw a runtime exception after the message is *released*.

- A *released* message cannot be dispatched, and a runtime exception will be generated under this situation.

2.9.2.2. Behavioral Scenarios in Observation Rules

Observation rules allow not only individual messages to be matched, but also sequences of messages, using behavioral scenario constructions to be matched, for example.

```
endpoint E accepts M1 accepts M2
{
    // Define an anonymous scenario.
    observe this m:(accepts M1{Id is var id} issues M2{Id == id}*)
    {
        // The m is an array<message>, with the matched messages.
    }
}
```

It is also possible to reuse already defined scenarios.

```
scenario MyScenario = M1{Id is var id} M2{Id == id}*

endpoint E accepts M1 accepts M2
```

```
{
    // Use a named scenario.
    observe this m:MyScenario
    {
        // The m is an array<message>, with the matched messages.
    }
}
```

The semantics of these constructions is the following: when a sequence of messages that match the scenario specified in a rule occurs on the listening endpoints, the rule is fired.

2.9.3. Establishing Actor Priorities

A partial order can be established between actors in order to have a more refined control in the execution rule sequence. Actors (including implicit actors in endpoints) can declare that they *precede* or *follow* other actorsto defines a strict partial order between them.

```
// The actor A must wait for B to finish before processing.
actor A(MyEndpoint e) follows B {...}

// The endpoint Socket must wait for both B and C to finish
// before processing.
endpoint Socket accepts M issues M follows B, C {...}

// After actor X finishes, Y must be executed.
actor X(MyEndpoint e) precedes Y {...}

// After actor X finishes, rules in endpoint MyOtherEndpoint
// must be executed.
actor X(MyEndpoint e) precedes MyOtherEndpoint
```

The execution order between observation rules and process rules are maintained, and independently of the specified partial order, all observations will be guaranteed to fire before any message is consumed. The execution order between actors (including implicit actors in endpoints) is determined in this way:

1. If an actor A **precedes** an actor **B**, or an actor **B follows** an actor **A**, rules in **A** are executed first then rules in **B**
2. Rules in implicit actors are executed before rules in normal actors, unless specified otherwise with a **follows** or **precedes** modifier.
3. Rules in actors where **follows** or **precedes** modifiers are present are executed before the rules in actors without any of these modifiers.

This set of rules are first applied to observation rules to determine the execution order, and then to process rules.

2.9.4. Dealing with Endpoints

An actor can bind to an endpoint to access its public data state and to dispatch messages. An endpoint expression is introduced by the **endpoint** keyword, followed by an **optional** modifier, followed by the name of the endpoint, followed by optional indices and transport. the following examples denote endpoint values.

```
var a = endpoint A;          // Binds or creates a singleton endpoint.
var b = endpoint create A;   // Creates an endpoint;
                             // exception if endpoint exists.
var c = endpoint bind A;     // Binds to existing endpoint;
                             // exception if not existent.
var d = endpoint exists A;   // Tests whether endpoint exists;
                             // returns Boolean value.
var e = endpoint B[22];      // Binds or creates endpoint with index.
var f = endpoint C[1] over a; // Binds or creates endpoint
                              // with index over transport.
```

The **dispatch** statement uses an **endpoint** value, directionality, and a message value. The following example shows an actor that consumes a message from one endpoint and dispatches it to another endpoint. This is a typical network stack parsing scenario. Note that on **dispatch** the directionality is provided, because it indicates the logical direction that the message flows to an endpoint.

```
endpoint E[int Id] over T accepts ProtocolMessage;
endpoint T[string Id] accepts TransportMessage;

actor Parser(T t)
{
      process t accepts m :TransportMessage
      {
            var e = endpoint E[GetId(m)] over t;
            dispatch e accepts DecodeMessagePayload(m);
      }
}
```

Endpoints are first-class values that can be passed around as parameters of functions, or stored in messages and travel via the network. Binding endpoints, and dispatching messages and accessing data are not syntactically restricted to appear only inside of an actor rule. However, an actor rule must be currently executing to allow a binding to proceed; otherwise an exception will be thrown.

2.9.4.1. Endpoint Deletion and Destructors

An endpoint stays in existence until it is explicitly deleted. When an endpoint is deleted, all further messages dispatched to that endpoint will result in a runtime error. Deletion of endpoints is achieved with the following statement.

```
delete E;
```

An explicit destructor is available so that user-defined code can run upon endpoint deletion. Destructors are exclusively associated to endpoints, but actors (both implicit and explicit) are the ones that get notified, for example.

```
actor A(MyEndpoint e)
{
    array<User> connectionQueue = [];

    process e accepts m:M
    {
        // ...
    }

    ~endpoint(MyEndpoint e)
    {
       // Clean-up data. For example, the connectionQueue is emptied.
    }
}
```

A destructor cannot be explicitly called, they are always invoked automatically. Every actor can declare a method with the following signature.

```
~endpoint(ADeclaredEndpoint e)
```

Where **`ADeclaredEndpoint`** must be one of the endpoint types the actor declares to be listening to. In this way, an actor can declare multiple destructor methods, one for each endpoint it is listening to. The endpoint that is about to be deleted is passed as a parameter.

The following are some destructor considerations:

- If more than one actor is listening to the same endpoint, the order that the destructors will be called is not deterministic.
- Destructors can also be declared in the body of implicit actors.
- For instance-based actors and indexed endpoints, the index of the endpoint that is passed as a parameter to the destructor corresponds to the one particular endpoint that actor instance is listening to. Static actors automatically listen to all endpoint indexes, so deleting any index will fire the destructor invocation.
- The endpoint being destructed is still alive when the destructor is called in the sense that if A is the endpoint being destructed then the check for **`endpoint exists A`** will return true. After the destructor is executed, the endpoint will be effectively disposed, but when exactly that happens is implementation dependent.

Arbitrary code can be present in the body of a destructor. However, the usual scenarios that an actor accomplishes upon endpoint deletion are: cleaning-up data, reestablishing structural invariance of internal states, or dispatching messages. Observe that deleting an endpoint means that no further data is going to be received on that endpoint, so actors can make decisions based upon that fact.

2.9.5. Explicit Activation and Deactivation of Actors

An actor can be created by a special statement that invokes the name of the actor and its parameters. This can only happen inside of a rule execution context. The actor will not start running until the current rule has terminated execution.

```
start Actor(parameter1, parameter2, ...);
```

An actor can only be deactivated by one of its own rules, using the following special statement.

```
stop;
```

2.9.6. Associating Actors with Endpoints and Roles

Actors can be declared such that they attach automatically to endpoints whenever instances of those endpoints exist. Actors can be also declared implicitly as part of an endpoint or a role, which is syntactic sugar to make declarations easier (usually more concise) than declaring them explicitly.

Often, it is desirable that an actor starts processing automatically as soon as one or more endpoints are created. The most general syntax for this allows the actor to be defined in a different module than the endpoint itself. This is important to support the scenario where protocols are added that stack on existing transports. To achieve auto-starting, the actor must have only endpoint parameters, and use the following syntax.

```
autostart actor Parser(T t) { ... }
```

Another way to implicitly start an actor is to associate it with an endpoint. In this usage scenario, the body of the endpoint directly contains the actor processing rules.

```
endpoint P provides Messages
{
      observe P accepts M1 { ... }
      observe P issues M2 { ... }
}
```

The actor will be started whenever the endpoint is created. A similar notation can be used for roles, by placing the processing rules in the **role** container.

A new instance of an actor is automatically created for every tuple of instances of indexed endpoints. That means that every time a new index is created for a given endpoint, new instances for all the actors that listen to that endpoint will be created,for example.

```
endpoint E[int i, int j] accepts M;

actor A(E e)
{

}
```

For each instance of **E[i, j]**, an instance of **A** will be created that will listen to that particular endpoint index. This enables actors to track messages coming from particular indexes automatically. The default behavior can be changed by prefixing an actor with the **static** keyword.

```
static actor A(E1 e1, E2 e2)
{

}
```

When an actor is declared as **static**, only one instance will be created. This single instance will listen to messages arriving to any of the endpoints indexes passed as parameters. The following are some considerations:

- Static actors can have arbitrary parameters, not only endpoint type (as regular actors can).
- Static actors can also be autostart actors. The same restrictions hold for regular actors hold here: an autostart actor can only have endpoint-type parameters.
- A non-autostart static actor can be explicitly started (as with regular actors). Further starts applied on the same actor do not have any effect.

2.9.7. Bindings

A **binding** provides a declarative way to define a mapping between two endpoints by relating patterns of messages on each endpoint. Bindings can be thought of as simple actors, with no state, that translate messages from one endpoint to the other applying a set of rewriting rules, for example.

```
endpoint P[int Port] accepts ProtocolMessage issues ProtocolMessage;
endpoint T accepts TransportMessage issues TransportMessage;

binding MyBinding: P over t:T
{
      rule t accepts
                  TransportMessage{Payload is m:ProtocolMessage from
                  ProtocolDecoder} => P[t.Port] accepts m
      rule t issues
                  TransportMessage{Payload is m:ProtocolMessage from
                  ProtocolDecoder} => P[t.Port] issues m
}
```

The preceding rules state that for every **TransportMessage** the endpoint **T** accepts or issues, it should be decoded and matched to a **ProtocolMessage** and accepted/issued by **P**. Observe that these rules do not involve actual dispatching. The semantics should be read as (taking the first rule) "**P** accepting a **TransportMessage m** is translated as **T** accepting **m**, decoding it as a **ProtocolMessage**".

The left-hand side of the rule implicitly binds the lower layer (**T** in the example), while the right-hand side of the rule binds the upper layer (P in the example). Observe that the capture variable **t** allows the access to the transport endpoint instance.

As we mentioned before, bindings can be thought of as simple actors. In fact, bindings can be rewritten using actors directly. The previous example can be translated by creating an actor with an equivalent behavior.

```
endpoint P[int Port] accepts ProtocolMessage issues ProtocolMessage;
endpoint T accepts TransportMessage issues TransportMessage;

autostart actor P_T(P p, T t)
{
      process t accepts
            TransportMessage{Payload is m:ProtocolMessage from
                              ProtocolDecoder}
            {dispatch (endpoint P[t.Port]) accepts m;}
      process t issues
            TransportMessage{Payload is m:ProtocolMessage from
                              ProtocolDecoder}
            {dispatch (endpoint P[t.Port]) issues m;}
}
```

Nevertheless, for describing simple behaviors bindings provide a more concise and readable syntax and does not involve declaring actors explicitly.

2.10. Invariant Checking

Types, endpoints, roles, as well as global static contexts such as modules and protocols can have constraints associated with the values stored in their fields. Those constraints result from either field declarations that via a pattern constraint the admissible values, or from the explicit declaration of an invariant. This section outlines the semantics of field constraints and invariants.

First, pattern constraints on fields can be normalized into invariants. Consider the following declaration.

```
pattern PosInt = int where value >= 0;
PosInt x;
```

This is equivalent to the following OPN fragment.

```
pattern PosInt = int where value >= 0;
int x;
invariant x is PosInt;
```

Based on that reduction, the discussion can be focused on invariants proper.

A well-known problem with invariant checking is that it must be allowed to temporarily violate an invariant. Consider the following OPN fragment.

```
type T { int X; int Y; invariant X > Y; };
var t = new T{};
t.X = 1; t.Y = 0;
```

Here, during the construction of the value, the invariant is violated. This is necessary because it would be too hard of a burden to a modeler to always construct admissible values from scratch.

OPN takes the approach to this problem to defer invariant checking until some well-defined point, which guarantees that the invariant is checked. It also guarantees that invariants are not checked *earlier* than certain points, effectively leaving an implementation the freedom to perform checking in-between the following points:

- During the execution of a constructor of a type message, all invariant checking is deferred.
- At the moment a field is *accessed*, it is guaranteed that the invariants related to the field are satisfied as a part of the exception for constructor execution.
- When a rule is activated on a message, it is guaranteed that all invariants to all the fields of the message,including invariants for embedded reference values, are satisfied.
- It is guaranteed that in a given particular rule execution, invariants are not checked *earlier* than a related field access *or* dispatching of the message.

This practically means that an implementation is allowed to realize checking of invariants anytime in-between dispatching by an actor and processing by an actor, that is during the time the message *travels*. In particular, if the message will never be processed, then invariants may never be checked, for example, because it has filtered out in some configuration.

2.10.1. Presence indicators

OPN provides specific syntax to specify conditions under which **optional** or **union** values are present. The following is a code example.

```
type MyType
{
   int flag;
   optional [|flag > 1|] string user;
   optional [|value > 0|] int count;
   invariant flag == 2 => count > 10;
}
```

The Boolean condition specified between **[|...|]** defines an invariant for the enclosing type, that determines when the associated **optional** field is present (that is, when it is not the **nothing** value). This does not add additional expressivity to the language, and it is just a handy way to specify these constraints. The preceding example can be rewritten, with exactly the same meaning, as the following.

```
type MyType
{
   int flag;
   optional string user;
   optional int count;
   invariant flag == 2 => count > 10;
   invariant (flag > 1 <=> user != nothing);
   invariant (count > 0 <=> count != nothing);
}
```

Similar syntax is provided for **union** types, in order to determine which branches are enabled.

```
type MyOtherType
{
    int flag;
    ([|flag > 1|] int | [|flag > 2|] bool | [|value > 0|] float) someField;
}
```

Presence indicators are especially useful when consumed by codecs. The **optional** and **union** values always pose a complication to codecs since, in an arbitrary case, codecs have to resolve a potentially complex constraint system to determine if an **optional** field (or if a specific branch in a **`union`** type) has to be decoded under certain conditions. Using presence indicators helps codecs isolate these conditions. Codecs shipped with PEF will normally consume presence indicators to guide the parsing process in the expected way.

3. Walkthrough: TCP/IP

Basic concepts of protocol modeling in OPN are illustrated in this section using as an example a *largely simplified* version of the TCP/IP stack with a simple application protocol running on top of it. In order to not confuse the model with the real TCP/IP protocol, we call it **MiniIP** and **MiniTCP**, respectively. This example will focus on describing the server side of the involved protocols.

3.1. MiniIP Layer

The stack described here starts at the **MiniIP** level. The link level is omitted, that is the way IP packets are synthesized from a lower level is notdisscussed. That means that **process** rules are omitted for this protocol, and we are going to focus on **observe** rules. Therefore, any message encoding detailsthat would usually be part of such a protocol definition are ignored. The focus is on the logical structure.

First define the protocol name and a simplified version of an IP address.

```
protocol MiniIP;

// An Address is just a four-byte length binary value.
pattern Address = binary where value.Count == 4;
// This constant represents a broadcast address for the MiniIP protocol.
const Address BroadcastAddress = $[FFFFFFFF];
```

Next, define the message type that is going to be exchanged at the **MiniIP** level.

```
message Packet
{
      Address SourceAddress;
      Address DestinAddress;
      int Version;
      binary Payload;
}
```

To represent the message exchange point for **MiniIP** we declare a **Node** endpoint. This endpoint will accept and issue **Packet** messages. The endpoint is indexed by an internet address. Thus, for every internet address, there will be one instance of this endpoint.

To use an implicit actor attached to **Node** will do the job in this simple scenario. We just need two rules here to match a **Packet** with its two possible directions. We also want to be sure that a message with the right address is being observed.

```
endpoint Node[Address NodeAddress] accepts Packet issues Packet
{
      // A message is accepted if it matches the Packet.
      observe this accepts p:Packet{}
      {
            assert p.DestinAddress == NodeAddress
                  || p.DestinAddress == BroadcastAddress;
      }

      // Same case for the other direction
      observe this issues p:Packet{}
      {
            assert p.SourceAddress == NodeAddress;
      }
}
```

The endpoint has installed a contract that enforces that the accepted packets are either addressed to this endpoint, or are broadcasted, and that the issued packets use the right source address. Observe that the pattern used in both rules determine when rules are fired and imposes that the observed message must match **Packet**. That means that a Packet sent with the wrong address will be observed by this endpoint, but the assertion in the body of each rule will make sure that such a message will be marked as problematic, so other tools can consider this information to decide how to react.

3.2. MiniTCP Layer

Next the **MiniTCP** level is described. **MiniTCP** deals with fragmentation, and it has the responsibility to build a complete message out of fragments. We are going to use two endpoints for this both of them under **MiniTCP** protocol. The first endpoint, called **Connection**, will collect the decoded IP fragments from the underlying Node. Once all the fragments have arrived the **Connection** will dispatch the complete message to the second endpoint, called **Socket**. The endpoint **Socket** will act as a viewpoint that provides the proper abstraction to upper layers.

The messages used by **MiniTCP** are called segments.

```
protocol MiniTCP;
using MiniIP;

// MiniTCP flags
pattern SegmentKind = enum byte {SYN, SYNACK, DATA, FIN};

// A Segment will model both the fragments and the complete MiniTCP message.
message Segment
{
      SegmentKind Kind;
      Port SourcePort;
      Port DestinPort;
      int SeqNo;
      binary Payload;
}
```

As we mentioned previously, the endpoint provided to upper layers is **`Socket`** that is stacked on top of a **`MiniIP`** Node. A socket is identified by the address it inherits from the node, by a port, and by a destination address and port.

```
pattern Port = short;

// This endpoint acts as a viewpoint for upper layers.
endpoint Socket[Port Port, Address DestinAddress, Port DestinPort]
      over Node
      accepts Segment issues Segment;
```

`Socket` endpoint may have its own specific internal state and logic, but we are not going to go into details here and will keep the example simple. **`Socket`** will just get the reassembled messages from **`Connection`**. Upper level protocols will **`stack over Socket`**.

The endpoint **`Connection`** keeps track of the connection state of the protocol. It defines a set of observer rules that queries the internal state of the endpoint, consider the current message being evaluated, and update the state accordingly. In contrast to real-world TCP, the handshake for tearing down a connection is simplified, and also the sequence number synchronization in the initial handshake is omitted.

```
// The enumeration that defines the possible logical states of the protocol.
Pattern ConnectionState = enum {SynSent, SynReceived, Established,
              FinWait1, FinWait2, CloseWait, Closing,
              LastAck, TimeWait, Closed}

// Connection endpoint defines rules to update the state in terms of the
// incoming and outgoing messages. It also handles reassembly.
endpoint Connection[Port Port, Address DestinAddress, Port DestinPort]
      over Node
      accepts Segment issues Segment
{
      // The abstract data model of the endpoint is just a variable keeping
      // the connection state.
      ConnectionState state = Closed;

      // Each of the rules matches a message with a particular direction and
      // Kind. The endpoint state is updated accordingly. The assertions
      // add sequence validation to the protocol, checking that messages
      // arrive in good order.
      observe this issues Segment{Kind == SYN}
      {
            assert state == Closed;
            state = SynSent;
      }
      observe this accepts Segment{Kind == SYN}
      {
            assert state == Closed;
            state = SynReceived;
```

```
        }
        observe this issues Segment{Kind == SYNACK}
        {
                assert state == SynReceived;
                state = Established;
        }
        observe this accepts Segment{Kind == SYNACK}
        {
                assert state == SynSent;
                state = Established;
        }
        observe this accepts Segment{Kind == DATA}
        {
                assert state == Established;
        }
        observe this issues Segment{Kind == DATA}
        {
                assert state == Established;
        }
        observe this issues Segment{Kind == FIN}
        {
                assert state == Established;
                state = Closed;
        }
        observe this accepts Segment{Kind == FIN}
        {
                assert state == Established;
                state = Closed;
        }
        // ... to be continued
}
```

It is worth noting that these *observe* rules are intended to keep track of the protocol state and validate that messages arrive in the right sequence. *Process* rules are the ones that can consume messages and there is where we place the reassembly logic. The actual reassembly and sequence number handshake is abstracted.

```
endpoint Connection
{
        // ... continued

        // Store the fragments as they arrive.
        array<Segment> orderedSegments = [];

        // This rule consumes the fragments, and when all the pieces are
        // together, reassembles the whole message and dispatches it to the
        // Socket endpoint.
        process this accepts s:Segment{Kind == DATA}
        {
                // The incoming message is added to the array. The function
                // returns if there is some set of fragments that reassembles
                // a full message.
```

```
            bool aFullMessageIsAvailable = InsertSegment(s, orderedSegments);
            if (aFullMessageIsAvailable)
            {
                  // The socket endpoint is bound.
                  var socket =
                  endpoint Socket[Port, DestinAddress, DestinPort]
                        over this.Node;

                  // Reassembles the full message and removes the appropriate
                  // fragments from the array.
                  Segment fullMessage = ReassembleFragments(orderedSegments)

                  // The full message is dispatched to the socket endpoint.
                  dispatch socket accepts fullMessage;
            }
      }

      // Other process rules to handle the other casesor directions.
      //...
}
```

Note that since the **Socket** endpoint is directly defined to be stacked on top of a **MiniIP** Node, and not a connection, the binding of a socket used in the preceding example needs to provide full addressing information as well as the node (**this.Node** can serve as a node because **Connection** is also on top of node). In an alternative design, the socket would have been defined on top of the connection; however, one may argue that such a design exposes internals of the **MiniTCP** model, as the endpoint **Connection** can be considered as auxiliary abstraction.

The last piece we need to describe is how **MiniIP** messages are decoded and dispatched to **MiniTCP Connection** endpoint. A parser actor is defined that is attached to a **MiniIP** Node that recognizes segments and dispatches them.

```
// The Parser actor listens to Node endpoint. Since Node is an indexed
// endpoint and no specific index is specified; it will listen to all
// Node instances.
autostart actor Parser(Node node)
{
      // MiniTCP is only interested in MiniIP messages whose version is 4.
      const int LastMiniIPVersion = 4;

      process node accepts p:Packet{Version == LastMiniIPVersion}
      {
            switch (p.Payload)
            {
                  // Bind connection and dispatch.
                  case s:Segment from BinaryDecoder =>
                        var conn = Connection[s.SourcePort,
                              p.DestinAddress, s.DestinPort]
                        over node;
                        dispatch conn accepts s;
```

```
                    // Report payload as invalid, dropping packet.
                    default =>
                          ReportMatchFailureDiagnosis();
              }
        }

        // Similar rule for the issues direction
        //...
}
```

This actor processes every **MiniIP** message whose version matches the constraint. Then it attempts to decode the **Segment** from the **Packet** payload, and then forwards it to the **MiniTCP Connection**.

3.3. Application Layer

Next a simple application layer protocol is described that travels on top of **MiniTCP**. The protocol, called **Echo**, simply echoes data it receives in arbitrary order, but without any lost. Incoming messages carry the string that contains the message to be echoed, plus the user name that initiated the message.

```
protocol Echo;
using MiniTcp;

// The MiniTCP port is used by the Echo protocol.
const Port EchoPort = 99;

// Post received by the server with the message to be echoed and the user
// that generated the message.
message Post {
      string Content;
      string UserName;
}

// The echoed message.
message Echo {
      string Content;
}
```

The **EchoServer** accepts **Post** messages (sent by the clients) and issues **Echo** messages, representing the echoed data. The endpoint also keeps a list of users that are allowed to send **Post** messages. If a message from a user that is not in the list is received, the message is ignored.

```
// The EchoServer is stacked on top of a Socket with the proper ports.
endpoint EchoServer
      over Socket where value.SourcePort == EchoPort
```

```
            &&  value.DestinPort in EphemeralPorts;
      accepts Post issues Echo
{
      // Messages that arrived and were not echoed yet.
      array<string> unechoed = [];

      // List of users that are allowed to request an echo. How this list
      // is retrieved is abstracted, and can be thought of as being a static
      // list.
      array<string> allowedUsers = getAllowedUsers();

      // The message is accepted only if the user is allowed. In case it is
      // not, the message will not match the rule and will be just ignored.
      observe this accepts m:Post{UserName in allowedUsers}
      {
            unechoed += [m.Content];
      }

      // When an Echo is issued, to be sure its content is unechoed
      // add an assertion to validate this logic.
      observe this issues m:Echo{}
      {
            assert m.Content in unechoed;
            unechoed = unechoed.Remove(m.Content);
      }
}
```

The contract on the endpoint assures that only data that has been posted can be echoed. Observe that since this is our top-level protocol, we do not need process rules to consume messages and dispatch them up the stack.

Next, the parsing actor is defined that maps from a TCP socket to the echo endpoint. This parser will be automatically started on a socket that uses the port for the **`Echo`** protocol.

```
// This is the actor that listens on the server socket indexed by
// the EchoPort.
actor autostart EchoServerParser(
      Socket serverSocket where value.SourcePort == EchoPort)
{
      // A MiniTCP Segment is matched, the Payload is decoded to a Post
      // message and dispatched to EchoServer.
      process serverSocket accepts Segment{Payload is m:Post
                                             from BinaryDecoder}
      {
            dispatch (endpoint EchoServer over socket) accepts m;
      }

      // A similar rule for the other direction.
      //...
}
```

4. Language Reference

4.1. Lexis

4.1.1. Lexical Unit

```
LexicalUnit ::= LexicalElement*
LexicalElement ::= Whitespace | Comment | Token
Token ::= Identifier | Keyword | Literal | OperatorOrPunctuator
```

4.1.2. Whitespace

```
Whitespace ::= any charater with Unicode class Zs
             |  tab characters (U+0009, U+000B)
             |  formfeed character (U+000C)
             |  Newline
NewLine ::=     carriage return (U+000D)
             |  line feed  (U+000A),
            |  next line  (U+0085)
```

4.1.3. Comments

```
Comment ::= SingleLineComment | MultiLineComment
SingleLineComment ::= //  (! NewLine)*
MultiLineComment ::= /* (  ! */  )* */
```

4.1.4. Literals

```
Literal ::= BooleanLiteral | IntegerLiteral | FractionalLiteral | CharacterLiteral | StringLiteral
          | NullLiteral | NothingLiteral | BinaryLiteral | GuidLiteral | DecimalLiteral
BooleanLiteral ::= true | false
IntegerLiteral ::= Sign? DecimalDigit+ | ( 0x | 0X ) HexDigit+ |  ( 0b | 0B ) BinaryDigit+
            | (0o | 0O) OctetDigit+
FractionalLiteral ::= Sign? DecimalDigit* . DecimalDigit+ Exponent?
DecimalLiteral ::= FractionalLiteral (m | M) | Sign? DecimalDigit+ (m | M)
Exponent ::= ( e | E ) Sign? DecimalDigit+
Sign ::= + | -
BinaryDigit ::= 0 | 1
OctetDigit ::= BinaryDigit | 2 |3 |4 |5 |6 |7
DecimalDigit ::= OctetDigit |8 |9
HexDigit ::= DecimalDigit | a | A |b | B |c | C |d |D |e |E|f |F
```

```
CharacterLiteral ::= ' ( ! ' & VisibleCharacter | \' | CharacterEscape ) '
CharacterEscape ::= \xHexDigit#4  | \xHexDigit#8 | \CharacterEscapeControl
CharacterEscapeControl ::= 0 | a |b | f | n | r |t |v
StringLiteral ::= SingleLineStringLiteral | MultilineStringLiteral
SingleLineStringLiteral ::=  " ( ! " & VisibleCharacter | \" | CharacterEscape )*  "
    MultiLineStringLiteral ::=
        @ " ( ! " & VisibleCharacter | \" | Newline | CharacterEscape )*  "
BlockLiteral ::= { ( \} | \{ |  \\  | VisibleCharater | NewLine | BlockLiteral )* }
NullLiteral ::= null
NothingLiteral ::= nothing
BinaryLiteral ::= $[ ( HexDigit HexDigit )* ] | Base64Literal
Base64Literal ::= TO BE DONE
GuidLiteral ::= { HexDigit#8 - HexDigit#4 - HexDigit#4 - HexDigit#4 - HexDigit#12 }
VisibleCharacter ::= TO BE DONE (specify unicodes)
Newline ::= TO BE DONE (specify unicodes for CR or CR LF)
```

The intention of the *BlockLiteral* is to allow embedding of an alien grammar within OPN. It allows nesting braces without escaping as long as the braces are balanced. If this is not the case, writers have to escape unbalanced braces. The same rule holds for the backslash (\ ), and can be escaped, that is (\\), if needed.

4.1.5. Identifiers and Keywords

```
QualifiedIdentifier ::= Identifier ( . Identifier )*
Identifier ::= KeywordOrIdentifier without Keyword | @ KeywordOrIdentifier | $ StringLiteral
KeywordOrIdentifier ::= IdentifierBegin IdentifierContinue* IdentifierEnd | IdentifierSingle
IdentifierBegin ::= one of _ Letter
IdentifierContinue ::= one of  Letter  DecimalDigit
IdentifierEnd = one of  _ Letter DecimalDigit
IdentifierSingle = Letter
Letter ::= unicode character of class Lu, Ll, Lt, Lm, Lo, Nl, Mn, Mc, Pc, Cf
Keyword ::= any of:
    abstract accepts actor annotation any array as aspect assert  autostart
    binary binding bool break byte
    case catch change char connection const consumes continue contract
    decimal default delete dispatch do double
    enum else endpoint error exception extends extern
    false finally flags float from for foreach follows
    get guid
  if in int interface internal invariant is issues
  json
    long
    map metadata message module
  new nothing null
    observe operation operator over override
    pattern precedes process protocol provides public
    regex ref return role rule
    sbyte select set short static string success switch syntax
    throw treedata true try type typedef
    uint ulong using ushort
    value var virtual void
    where while with
  xml xpath
WeakKeyword::= any of:
  all bind create exists freeze optional out result some start stop this ignore separator
```

4.1.6. Operators and Punctuators

Note that multi-character operators (for example `&&, >=, #..,` and so on) should be considered as single tokens, and therefore they do not allow spaces or tabs interleaved.

```
OperatorOrPunctuator ::= any of:
    { } [ ] ( ) . , + -  * / %  !  =  <  >  &  ^ ;  # ` ?
     && ||  ==  !=  <=  >=  => | & ^ ==> <=> ~ .. ...  <<  >> += -=  ++ -- ?? #? #..
```

4.2. Expressions

4.2.1. Function Expressions

```
Expression ::= FunctionParameterList => (Expression | Block)
FunctionParameterList ::= Identifier | ( ( FunctionParameter ( , FunctionParameter )* )? )
FunctionParameter ::= PrimaryPattern Identifier | Identifier
```

4.2.2. Conditional Expressions

```
Expression ::= Expression ? Expression : Expression
```

4.2.3. Binder Expressions

```
Expression ::= from Bindings  Expression
Expression ::= all  Bindings Expression
Expression ::= some Bindings Expression
Bindings ::= ( Binding ( , Binding )* )
Binding ::= ( var | PrimaryPattern ) Identifier Constraint? ( in | = ) Expression
```

4.2.4. Logical Operator Expressions

```
Expression[3] ::= Expression[4] ?? Expression[3]
Expression[4] ::= Expression[4] ||  Expression[5]
Expression[5] ::= Expression[5] && Expression[6]
Expression[16] ::=  !  Expression[16]
```

4.2.5. Endpoint Expression

```
Expression[6] ::= endpoint ( create | bind  | exists )?  EndpointReference
EndpointReference ::=
  ReferencePatternWithoutArguments
  ( [ Expression ( ,  Expression )* ]  )? ( over Expression[6] )?
```

4.2.6. Logical and Bitwise Operator Expressions

```
Expression[7] ::= Expression[7]  | Expression[8]
Expression[8] ::= Expression[8] ^  Expression[9]
Expression[9] ::= Expression[9] & Expression[10]
Expression[6] ::= Expression[6] ==> Expression[7]
Expression[6] ::= Expression[6] <=> Expression[7]
Expression[16] ::=  ~  Expression[16]
```

```
Expression[16] ::=  !  Expression[16]
```

4.2.7. Relational Operator Expressions

```
Expression[10] ::= Expression[10] == Expression[11]
Expression[10] ::= Expression[10] != Expression[11]
Expression[11] ::= Expression[11] > Expression[12]
Expression[11] ::= Expression[11] >= Expression[12]
Expression[11] ::= Expression[11] <  Expression[12]
Expression[11] ::= Expression[11] <= Expression[12]
```

4.2.8. Bit Shift Operator Expressions

```
Expression[12] ::= Expression[12] <<  Expression[13]
Expression[12] ::= Expression[12] >> Expression[13]
```

4.2.9. Range Expression

```
Expression[12] ::= Expression[13] ..  Expression[13]
```

4.2.10. In Expression

```
Expression[12] ::= Expression[13] in  Expression[13]
```

4.2.11. Is Expressions

```
Expression[12] ::= Expression[13] is Pattern
```

4.2.12. As Expressions

```
Expression[12] ::= Expression[13] as Pattern
```

4.2.13. Arithmetic Operator Expressions

```
Expression[13] ::= Expression[13] + Expression[14]
Expression[13] ::= Expression[13] -  Expression[14]
Expression[14] ::= Expression[14] * Expression[16]
Expression[14] ::= Expression[14] / Expression[16]
Expression[14] ::= Expression[14] % Expression[16]
Expression[16] ::=  +  Expression[16]
Expression[16] ::=  -  Expression[16]
Expression[16] ::=  Expression[16] ++
Expression[16] ::=  Expression[16] --
```

4.2.14. Assignment Expressions

```
Expression[16] ::= NonInvokeStatementExpression = Expression
Expression[16] ::= NonInvokeStatementExpression += Expression
Expression[16] ::= NonInvokeStatementExpression -= Expression
```

4.2.15. Reference Expressions

```
PrimaryExpression ::= Expression[16]
Expression[16] ::= ReferenceExpression
ReferenceExpression ::= QualifiedIdentifier GenericArguments?
GenericArguments ::= <  Pattern ( , Pattern )* >
```

4.2.16. Access Expressions

```
Expression[16] ::= Expression[16]  [ Expression ]
Expression[16] ::= Expression[16]  .  Identifier
Expression[16] ::= Expression[16]  #  Identifier
Expression[16] ::= Expression[16] select xpath{BlockLiteral}
```

4.2.17. Invoke Expressions

```
Expression[16] ::= InvokeExpression
InvokeExpression ::= Expression[16]  InvokeArguments
InvokeArguments ::= ( ( InvokeArgument ( , InvokeArgument )* )?  )
InvokeArgument ::= (ref  | out)?  Expression
```

4.2.18. Parenthesized Expressions

```
Expression[16] ::= ( Expression )
```

4.2.19. Literal Expressions

```
Expression[16] ::= Literal
```

4.2.20. Creation Expressions

```
Expression[16] ::=  { ( KeyInitializer ( ,  KeyInitializer )* ,? )? }
Expression[16] ::=  { ( Expression  ( ,  Expression  )* ,? )? }
Expression[16] ::=  [ ( Expression  ( ,  Expression  )* ,? )? ]
Expression[16] ::= new ReferencePatternWithoutArguments InvokeArguments
Expression[16] ::= new InitializerExpression
InitializerExpression ::=
    ReferencePatternWithoutArguments { ( FieldInitializer ( ,  FieldInitializer)* ,? )? }
KeyInitializer ::= Expression  =  Expression
FieldInitializer ::= (Identifier | exception) =  Expression
```

4.2.21. This Expression

```
Expression[16] ::= this
```

4.2.22. Value Expression

```
Expression[16] ::= value
```

4.2.23. Freeze Expression

```
Expression[17] ::= freeze  Expression[16]
```

4.3. Patterns

Note that types and proper patterns are unified with one non-terminal operator, called Pattern, reflecting that types are subset of patterns.

4.3.1. Composed Patterns

```
Pattern ::= ([|Expression|])? Pattern  |  ([|Expression|])? Pattern[1]
Pattern[1] ::= Pattern[1] &  Pattern[2]
Pattern[2] ::= Pattern[2] +  Pattern[3]
```

4.3.2. Count Patterns

```
Pattern[3] ::= Pattern[4] *
Pattern[3] ::= Pattern[4] +
Pattern[3] ::= Pattern[4] #?
Pattern[3] ::= Pattern[4] # PatternCountValue
Pattern[3] ::= Pattern[4] # PatternCountValue  ..  PatternCountValue?
Pattern[3] ::= Pattern[4] #..  PatternCountValue
PatternCountValue ::= PrimaryExpression
```

4.3.3. From (Decoding) Pattern

```
Pattern[3] ::= Pattern[4]  from Reference
```

4.3.4. Constraint Patterns

```
Pattern[4] ::= Pattern[5] Constraint
Constraint ::= where Expression
```

4.3.5. Capture Patterns

```
Pattern[5] ::=  Identifier : Pattern[6]
Pattern[5] ::= var Identifier
PrimaryPattern ::= Pattern[6]
```

4.3.6. Not Patterns

```
Pattern[6] ::= ! Pattern[6]
```

4.3.7. Nullable or Optional Patterns

```
Pattern[6] ::= Pattern[7] ?
Pattern[6] ::= optional ([|Expression|])? Pattern[7]
```

Note: in the context of a sequential pattern, this operator denotes 0 or 1 repetitions. In the context of a singleton pattern, it denotes nullable.

4.3.8. Parenthesized Patterns

```
Pattern[7] ::=  ( Pattern )
```

4.3.9. Function Patterns

```
Pattern[7] ::=  PrimaryPattern(( Pattern ( , Pattern )* )? )
```

4.3.10. Reference Patterns

```
Pattern[7] ::= ReferencePattern
ReferencePattern ::= ReferencePatternIdentifier GenericArguments? TypeArguments?
TypeArguments ::= [ ( Expression ( , Expression )*  )? ]
ReferencePatternWithoutArguments ::= ReferencePatternIdentifier GenericArguments?
ReferencePatternIdentifier ::= QualifiedIdentifier | PredefinedTypeIdentifier
PredefinedTypeIdentifier ::=
  byte | sbyte | short | ushort | int | uint | long | ulong | float | double |
   decimal | guid | bool | char | string | binary | any | any aspect | any
   endpoint | any message | any type | array | set | map | | json | xml | treedata |
   void
```

4.3.11. Literal and Expression Patterns

```
Pattern[7] ::= $ PrimaryExpression
Pattern[7] ::= Literal
Pattern[7] ::= regex BlockLiteral
```

4.3.12. Range Patterns

```
Pattern[7] ::= $ PrimaryExpression .. PrimaryExpression
Pattern[7] ::= Literal .. Literal
```

4.3.13. Membership Patterns

```
Pattern[7] ::= in Expression
```

4.3.14. Compound Patterns

```
Pattern[7] ::=  (\\ | \)? ReferencePattern FieldPatterns? ((\\ | \) ReferencePattern
    FieldPatterns?)* )
FieldPatterns ::= { (FieldPattern ( , FieldPattern )* )? }
FieldPattern ::= (Identifier | exception) is Pattern | Identifier == Expression | Expression
```

4.3.15. Array Patterns

```
Pattern[7] ::= [ SequentialPattern? ]
SequentialPattern::= Pattern ( ,  Pattern )* ( , WildcardPattern? )?
WildcardPattern ::= ... ( Identifier )?
```

4.3.16. Map Patterns

```
Pattern[7] ::= { ( EntryPattern ( ,  EntryPattern )* ( , WildcardPattern? )? )? }
EntryPattern ::= Expression -> Pattern
```

4.3.17. Set Patterns

```
Pattern[7] ::= { SequentialPattern? }
```

4.3.18. Enum Patterns

```
Pattern[7] ::= ( enum | flags ) EnumBase? { (EnumField ( , EnumField )* )? ,? }
EnumBase ::= ReferencePatternWithoutArguments
EnumField ::= Identifier ( = Expression)? Aspects?
```

4.3.19. Reference Patterns with Capture

```
Pattern[7] ::= ReferencePatternWithCapture
ReferencePatternWithCapture ::= (Identifier:)? ReferencePattern
```

4.4. Productions

4.4.1. Alternation Productions

```
Production ::= Production[1] | Production
```

4.4.2. Sequencing Productions

```
Production[1] ::= Production[2] Production[1]
```

4.4.3. Capture Productions

```
Production[2] ::= Identifier : Production[3]
```

4.4.4. Quantifier Productions

```
Production[3] ::= Production[4] *
Production[3] ::= Production[4] +
Production[3] ::= Production[4] ?
```

4.4.5. Transformation Productions

```
Production[4] ::= => Expression
```

4.4.6. Semantic Condition Productions

```
Production[4] ::= [| Expression |]
```

4.4.7. Parenthesize Productions

```
Production[4] ::= ( Production )
```

4.4.8. Pattern Reference Productions

```
Production[4] ::= ReferencePattern TypeArguments
Production[4] ::= ReferencePattern from Reference
```

4.4.9. Literal Productions

```
Production[4] ::= StringLiteral
Production[4] ::= regex BlockLiteral
Production[4] ::= BinaryLiteral
```

4.5. Scenarios

4.5.1. Combination

```
TopScenario ::= BacktrackScenario
TopScenario ::=  TopScenario  |> TopScenario[0]
TopScenario[0] ::=  ( TopScenario )
```

4.5.2. Backtracking

```
BacktrackScenario ::= Scenario
BacktrackScenario ::= backtrack (Pattern) Scenario
```

4.5.3. Alternation

```
Scenario[0] ::=  Scenario[0] ( | | or ) Scenario[1]
Scenario[1] ::=  Scenario[1] ( || | fork ) Scenario[2]
```

4.5.4. Sequencing

```
Scenario[2] ::=  Scenario[2] (next)? Scenario[3]
Scenario[3] ::=  Scenario[3] ( -> | later ) Scenario[4]
Scenario[4] ::=  Scenario[4] ( & | permute ) Scenario[5]
```

4.5.5. Quantifying

```
Scenario[5] ::=  Scenario[6] ( * | repeat ) Range?
Scenario[5] ::=  Scenario[6] +
Scenario[5] ::=  Scenario[6] interleave UnboundedRange?  (until Scenario)?
Scenario[5] ::=  Scenario[6] interleave BoundedRange
Range ::=  BoundedRange | UnboundedRange
BoundedRange ::= [Expression ,Expression] | [,Expression]
UnboundedRange ::=[Expression ,]
Scenario[5] ::=  Scenario[6] ( ? | optional )
```

4.5.6. Semantic Condition

```
Scenario[6] ::=  [| Expression |]
```

4.5.7. Drop

```
Scenario[6] ::= drop(Scenario)
```

4.5.8. Parenthesis

```
Scenario[6] ::= ( Scenario )
```

4.5.9. Negation

```
Scenario[6] ::= (! | not) Scenario[6]
```

4.5.10. Terminals

```
Scenario[6] ::= \\ ? (issues | accepts)? ReferencePattern FieldPatterns?  ( (\ |\\)
    (issues | accepts)?ReferencePattern FieldPatterns? )*
Scenario[6] ::= _
Scenario[6] ::= QualifiedIdentifier ScenarioTypeArguments?
ScenarioTypeArguments ::= [ ( ScenarioTypeArgument ( , ScenarioTypeArgument )*   )?
    ]
ScenarioTypeArgument ::= Expression | ScenarioParameter
ScenarioParameter ::= out ( var | PrimaryPattern ) Identifier Constraint? Aspects?
```

4.6. Statements

```
Statement ::= VariableDeclaration ; | EmbeddedStatement
```

4.6.1. Variable Declarations

```
VariableDeclaration ::= ( PrimaryPattern | var ) Identifier Constraint? ( = Expression )?
```

4.6.2. Empty Statements

```
EmbeddedStatement ::= ;
```

4.6.3. Break Statements

```
EmbeddedStatement ::= break ;
```

4.6.4. Continue Statements

```
EmbeddedStatement ::= continue ;
```

4.6.5. Return Statements

```
EmbeddedStatement ::= return Expression? ;
```

4.6.6. Throw Statements

```
EmbeddedStatement ::= throw Expression ;
```

4.6.7. Assertion Statements

The optional expression after the comma, before the question mark (?) follows C# standards and its intended for documenting the reason of the assertion. The optional expression will be automatically converted to string and attached to the assertion.

```
EmbeddedStatement ::= assert Expression ( , Expression )? ;
```

4.6.8. Dispatch Statements

```
EmbeddedStatement ::= dispatch TransmitExpression ;
TransmitExpression ::= Expression ( issues | accepts ) Expression
```

4.6.9. Start Statement

```
EmbeddedStatement ::= start Identifier InvokeArguments  ;
```

4.6.10. Stop Statement

```
EmbeddedStatement ::= stop ;
```

4.6.11. Reject Statement

```
EmbeddedStatement ::= reject ;
```

4.6.12. Delete Statement

```
EmbeddedStatement ::= delete Expression ;
```

4.6.13. Expression Statements

```
EmbeddedStatement ::= StatementExpression ;
StatementExpression ::=
                NonInvokeStatementExpression = Expression
              | NonInvokeStatementExpression += Expression
              | NonInvokeStatementExpression -= Expression
StatementExpression ::=
                NonInvokeStatementExpression InvokeArguments
              | NonInvokeStatementExpression ++
              | NonInvokeStatementExpression --
NonInvokeStatementExpression ::= (NonInvokeStatementExpression[1])
NonInvokeStatementExpression[1] ::=
                ReferenceExpression
              | PrimaryStatementExpression . Identifier
              | PrimaryStatementExpression # Identifier
              | PrimaryStatementExpression [ Expression ( , Expression )* ]
PrimaryStatementExpression ::=
                NonInvokeStatementExpression
              | PrimaryStatementExpression InvokeArguments
```

4.6.14. Block Statements

```
EmbeddedStatement ::= Block
Block ::= { Statement* }
```

4.6.15. For Statements

```
EmbeddedStatement ::=
    for ( ForInitializer? ; Expression? ; ( StatementList ,? )? )   EmbeddedStatement
ForInitializer ::= VariableDeclaration ( , VariableDeclaration )* | StatementList ,?
StatementList ::= StatementExpression ( ,  StatementExpression )*
```

4.6.16. Foreach Statements

```
EmbeddedStatement ::= foreach Bindings EmbeddedStatement
```

4.6.17. If Statements

```
EmbeddedStatement ::= if ( Expression ) EmbeddedStatement
                      ( else EmbeddedStatement )?
```

4.6.18. While Statements

```
EmbeddedStatement ::= while ( Expression ) EmbeddedStatement
```

4.6.19. Do Statements

```
EmbeddedStatement ::= do EmbeddedStatement while ( Expression ) ;
```

4.6.20. Switch Statements

```
EmbeddedStatement ::= switch ( Expression ){ SwitchCase* DefaultCase? }
SwitchCase ::= case Pattern => Statement*
DefaultCase ::= default => Statement*
```

4.6.21. Try Statements

```
EmbeddedStatement ::= try Block ( catch ( Pattern ) Block )+ ( finally Block )?
EmbeddedStatement ::= try Block finally Block
```

4.6.22. Release Statements

```
EmbeddedStatement ::= release Expression;
```

4.7. Declarations

4.7.1. Aspects

```
Aspects ::= with  InitializerExpression ( , InitializerExpression )*
AspectsAfterBlock ::= Aspects ;
```

4.7.2. Field Declarations

```
FieldDeclaration ::=
     ExtensionModifier?
      FieldModifier* PrimaryPattern Identifier Constraint?  Aspects?
      ( = Expression  Aspects? )? ;
ExtensionModifier ::= change | delete
FieldModifier ::= VisibilityModifier | static | const
VisibilityModifier ::= public | internal
```

4.7.3. Annotation Declarations

```
AnnotationDeclaration ::=
     ExtensionModifier?
      AnnotationModifier* annotation PrimaryPattern Identifier Aspects? ;
AnnotationModifier ::= VisibilityModifier
AnnotationDeclaration ::=
     ExtensionModifier?
      AnnotationModifier* annotation PrimaryPattern QualifiedIdentifier#Identifier Aspects? ;
AnnotationModifier ::= VisibilityModifier
```

4.7.4. Method Declarations

```
MethodDeclaration ::=
     ExtensionModifier?
     MethodModifier* PrimaryPattern PropertyModifier? MethodIdentifier GenericParameters?
    Parameters  Aspects?  ( Block AspectsAfterBlock? | ; )
MethodDeclaration ::=
    extern VisibilityModifier? PrimaryPattern PropertyModifier? MethodIdentifier
     GenericParameters?
    Parameters  Aspects?  ;
MethodDeclaration ::= ~endpoint (PrimaryPattern Identifier Aspects?) Aspects?  ( Block
     AspectsAfterBlock? | ; )
PropertyModifier ::= get | set
```

```
MethodModifier ::= VisibilityModifier | static | virtual | override | abstract
GenericParameters ::= < Identifier ( , Identifier)* >
Parameters ::= ( (this? Parameter ( , Parameter )*  )? )
Parameter ::= ParameterModifier? PrimaryPattern Identifier Constraint? Aspects?
ParameterModifier ::= ref | out
MethodIdentifier ::= Identifier | operator DefinableOperator
DefinableOperator ::= one of
    [] + - * / %  ~ ^ & | << >> == != <= >= < > as in
```

4.7.5. Constructor Declarations

```
ConstructorDeclaration ::=
    ExtensionModifier?
    VisibilityModifier? Identifier Parameters  Aspects?  ( Block AspectsAfterBlock?  |  ; )
```

4.7.6. Invariant Declarations

```
InvariantDeclaration ::=
    invariant (Identifier : )?  Expression ( , Expression )? ;
```

4.7.7. Pattern Declarations

```
PatternDeclaration ::=
    ExtensionModifier? VisibilityModifier?  pattern Identifier GenericParameters? Aspects?
    = Pattern   Aspects?  ;
```

4.7.8. Syntax Declarations

```
SyntaxDeclaration ::=
    ExtensionModifier? VisibilityModifier?  syntax Identifier TypeParameters? Aspects?
    = Production   Aspects?  ;
SyntaxDeclaration ::=
    ExtensionModifier? VisibilityModifier?  syntax  ignore  Aspects?
    = regex  BlockLiteral Aspects?  ;
SyntaxDeclaration ::=
    ExtensionModifier? VisibilityModifier?  syntax  separator  Aspects?
    = regex  BlockLiteral Aspects? ;
```

4.7.9. Type Declarations

```
TypeDeclaration ::=
    ExtensionModifier?
    TypeModifier*  type Identifier GenericParameters?
    TypeParameters? Aspects? TypeBase?
    { TypeMember* } AspectsAfterBlock?
TypeDeclaration ::=
    extern VisbilityModifier *  type Identifier GenericParameters? Aspects? ;
TypeParameters ::= [ ( Parameter ( , Parameter )*  )? ]
TypeBase ::= : ReferencePattern (  , ReferencePattern )*
TypeModifier ::= VisbilityModifier | abstract
TypeMember ::= FieldDeclaration | ConstructorDeclaration |
                  MethodDeclaration | PatternDeclaration  | InvariantDeclaration
```

4.7.10. Interface Declarations

```
InterfaceDeclaration ::=
    VisibilityModifier?  interface Identifier GenericParameters? Aspects? TypeBase?
    { InterfaceMember* } AspectsAfterBlock?
InterfaceMember ::= MethodDeclaration
```

4.7.11. Message Declarations

```
MessageDeclaration ::=
    ExtensionModifier?
    VisibilityModifier?  message
    Identifier  TypeParameters? Aspects? TypeBase?
    { MessageMember* } AspectsAfterBlock?
MessageDeclaration ::=
    ExtensionModifier? VisibilityModifier? operation
    Identifier  TypeParameters? Aspects?
    { OperationMember* } ( Aspects ResultDeclaration* ; | ResultDeclaration+ ; )?
MessageDeclarationInContract ::=
    ExtensionModifier?
    VisibilityModifier? ( accepts | issues )? message
    Identifier  TypeParameters? Aspects? TypeBase?
    { MessageMember* } AspectsAfterBlock?
MessageDeclarationInContract ::=
    ExtensionModifier? VisibilityModifier? ( accepts | issues )? operation
    Identifier  TypeParameters? Aspects?
    { OperationMember* } ( Aspects ResultDeclaration* ; | ResultDeclaration+ ; )?
```

```
OperationFieldDeclaration ::=  ( in | out | result |in out |in result )? FieldDeclaration
ResultDeclaration ::= ( error | success | exception ) Pattern Aspects?
MessageMember ::= TypeMember
OperationMember ::=  OperationFieldDeclaration | MethodDeclaration |
                     PatternDeclaration | InvariantDeclaration
```

4.7.12. Virtual Declarations

```
MessageDeclaration ::=
    ExtensionModifier? VisibilityModifier? virtual operation
    Identifier  Aspects?
    { OperationMember* } (  Aspects VirtualResultDeclaration* | VirtualResultDeclaration+
    )?
    = (Scenario | QualifiedIdentifier TypeArguments?);
MessageDeclarationInContract ::=
    ExtensionModifier? VisibilityModifier? ( accepts | issues )? virtual operation
    Identifier  Aspects?
    { OperationMember* } (  Aspects VirtualResultDeclaration* | VirtualResultDeclaration+
    )?
    = (Scenario | QualifiedIdentifier TypeArguments?);
VirtualResultDeclaration ::= ( error | success ) Pattern Aspects? | (exception
    PrimaryPattern Identifier Constraint?  Aspects?  = Expression  Aspects?)
```

4.7.13. Aspect Declarations

```
AspectDeclaration ::=
    ExtensionModifier?
    TypeModifier*  aspect Identifier Aspects? TypeBase?
    {  AspectMember* }  AspectsAfterBlock?
AspectMember ::= ThisFieldDeclaration | FieldDeclaration | InvariantDeclaration
ThisFieldDeclaration ::=
     FieldModifier* PrimaryPattern this Constraint?  Aspects?
     ( = Expression  Aspects? )? ;
```

4.7.14. Actor Declaration

```
ActorDeclaration ::=
    ExtensionModifier?
    VisibilityModifier? static? autostart? actor Identifier  Parameters
    PartialOrderDeclaration* Aspects?
    { ActorMember* } AspectsAfterBlock?
ActorMember ::= FieldDeclaration | MethodDeclaration | PatternDeclaration |
    TypeDeclaration | InvariantDeclaration | RuleDeclaration | ActorDeclaration |
    ScenarioDeclaration
ActorModifier ::= VisibilityModifier | autostart
RuleDeclaration ::=
    (observe  | process) ( Identifier : )?  TransmitPattern { Statement* }
RuleDeclaration ::=
    observe ( Identifier : )?  ScenarioTransmitPattern { Statement* }
ScenarioTransmitPattern ::=  Expression (Identifier:)? Scenario
ScenarioTransmitPattern ::=  Expression (Identifier:)? QualifiedIdentifier

TransmitPattern ::=  Expression Direction Pattern
Direction ::= issues | accepts
PartialOrderDeclaration ::= (follows | precedes) QualifiedIdentifier GenericArguments? (,
    QualifiedIdentifier GenericArguments?)*
```

4.7.15. Binding Declaration

```
BindingDeclaration ::=
    ExtensionModifier?
    BindingModifier* binding  Identifier Aspects?
    :  ReferencePattern over  ReferencePatternWithCapture
    (  { BindingMember* }  AspectsAfterBlock? | Aspects?  ;  )
BindingMember ::= FieldDeclaration | MethodDeclaration | PatternDeclaration |
    TypeDeclaration | InvariantDeclaration | BindingRuleDeclaration
BindingModifier ::= VisibilityModifier
BindingRuleDeclaration ::=
  rule  ( Identifier : )?  Direction Pattern =>  Expression Direction Expression Aspects? ;
```

4.7.16. Contract Declarations

```
ContractDeclaration ::=
    ExtensionModifier?
    VisibilityModifier? contract Identifier Aspects? ( ContractElement+ )?
    { ContractMember* } AspectsAfterBlock?
ContractMember ::= MessageDeclarationInContract
ContractElement ::= ExtensionModifier?
    ( ( consumes | provides ) | accepts | issues ) ReferencePattern
```

4.7.17. Endpoint Declarations

```
EndpointDeclaration ::=
    ExtensionModifier?
    VisibilityModifier? endpoint Identifier EndpointAddress? EndpointTransports?
    PartialOrderDeclaration? Aspects?
        ContractElement+
    ( { EndpointMember* } AspectsAfterBlock?   |   Aspects? ; )
EndpointDeclaration ::=
    ExtensionModifier?
    VisibilityModifier? client endpoint Identifier PartialOrderDeclaration? Aspects?
    ( { EndpointMember* } AspectsAfterBlock?   |   Aspects? ; )
EndpointAddress ::= [ EndpointIndex ( , EndpointIndex )* ,? ]
EndpointIndex ::= Pattern Identifier Aspects?
EndpointTransports ::= over  ReferencePatternWithCapture TransportConstraint? ( |
    over ReferencePatternWithCapture TransportConstraint?)*
TransportConstraint ::= where  value EndpointTransports
EndpointMember ::= FieldDeclaration | MethodDeclaration | PatternDeclaration |
    TypeDeclaration | InvariantDeclaration |  RuleDeclaration | ActorDeclaration |
    ScenarioDeclaration
```

4.7.18. Scenario Declarations

```
ScenarioDeclaration ::=
    ExtensionModifier? VisibilityModifier? scenario  Identifier ScenarioTypeParameters?
    Aspects?      = Scenario Aspects? ;
ScenarioTypeParameters ::= [ ( ScenarioTypeParameter ( ,ScenarioTypeParameter)* )? ]
```

```
ScenarioTypeParameter ::= out? PrimaryPattern Identifier Constraint? Aspects?
```

4.7.19. Role Declarations

```
RoleDeclaration ::=
    ExtensionModifier?
    VisibilityModifier? role Identifier EndpointAddress? EndpointTransports? Aspects?
    ( { RoleMember* } AspectsAfterBlock? | Aspects? ; )
RoleMember ::= EndpointMember | EndpointDeclaration
```

4.7.20. Typedef Declarations

```
TypedefDeclaration ::=
    VisibilityModifier? typedef Identifier Aspects? = ReferencePatternIdentifier Aspects? ;
```

4.7.21. Documents

```
Document ::=
    module QualifiedIdentifier Extensions Aspects? ; UsingDeclaration* ModuleMember*
Document ::=
    protocol QualifiedIdentifier Extensions Aspects? ; UsingDeclaration* ProtocolMember*
ModuleMember ::= FieldDeclaration | MethodDeclaration | TypeDeclaration |
                 PatternDeclaration | InvariantDeclaration | ActorDeclaration |
                 MessageDeclaration | ContractDeclaration |
                 AnnotationDeclaration | AspectDeclaration | InterfaceDeclaration |
                 SyntaxDeclaration | BindingDeclaration | ScenarioDeclaration |
                 TypedefDeclaration

ProtocolMember ::= ModuleMember | EndpointDeclaration | RoleDeclaration |
    ConnectionDeclaration
Extensions ::= extends QualifiedIdentifier ( , QualifiedIdentifer )*
UsingDeclaration ::= using (QualifiedIdentifier Aspects?) | XMLNamespace;
XMLNamespace ::= please refer to the XML namespaces grammar definition at
    Namespaces in XML 1.0
```

5. Library Reference

The OPN standard library provides definitions that are common to all technologies supported by the OPN tool chain. This includes the following:

- Constants, patterns and functions on primitive and structured values.
- Aspects that are technology independent, related to encoding, documentation, and protocol metadata.

5.1. Any

These operators apply to any type.

```
module Standard;

bool operator ==(any x, any y);
bool operator !=(any x, any y);
T operator ??<T> (T x, T y);

string operator as(any x);
```

5.2. Scalars

The semantics of OPN operators is aligned to C# conventions.

5.2.1. Integers

All arithmetic operations happen on 32 bits, 64 bits, or 128 bits. Therefore, on 8 and 16 bit integers, only conversion operators are defined. See also the implicit conversion rules in section 2.1.7.

5.2.1.1. byte

```
module Standard;

byte operator as (sbyte x);
byte operator as (short x);
byte operator as (ushort x);
byte operator as (int x);
byte operator as (uint x);
byte operator as (long x);
byte operator as (ulong x);

byte operator as (string s);
```

5.2.1.2. sbyte

```
module Standard;

sbyte operator as (byte x);
sbyte operator as (short x);
sbyte operator as (ushort x);
sbyte operator as (int x);
sbyte operator as (uint x);
sbyte operator as (long x);
sbyte operator as (ulong x);

sbyte operator as (string s);
```

5.2.1.3. ushort

```
module Standard;

ushort operator as (short x);
ushort operator as (int x);
ushort operator as (uint x);
ushort operator as (long x);
ushort operator as (ulong x);

ushort operator as (string s);
```

5.2.1.4. short

```
module Standard;

short operator as (ushort x);
short operator as (int x);
short operator as (uint x);
short operator as (long x);
short operator as (ulong x);

short operator as (string s);
```

5.2.1.5. int

```
module Standard;
int operator + (int x);
int operator - (int x);
int operator ~ (int x);

int operator + (int x, int y);
int operator - (int x, int y);
int operator * (int x, int y);
int operator / (int x, int y);
int operator % (int x, int y);
int operator & (int x, int y);
int operator | (int x, int y);
int operator ^ (int x, int y);

bool operator == (int x, int y);
bool operator != (int x, int y);
bool operator <= (int x, int y);
bool operator < (int x, int y);
bool operator >= (int x, int y);
bool operator > (int x, int y);

int operator << (int x, int y);
int operator >> (int x, int y);

int operator as (uint x);
int operator as (long x);
int operator as (ulong x);
int operator as (bool x);

int operator as (string s);

int Min(int x, int y);
int Max(int x, int y);
int Abs(int x);
```

5.2.1.6. uint

```
module Standard;
uint operator + (uint x);
uint operator - (uint x);
uint operator ~ (uint x);

uint operator + (uint x, uint y);
uint operator - (uint x, uint y);
uint operator * (uint x, uint y);
uint operator / (uint x, uint y);
uint operator % (uint x, uint y);
uint operator & (uint x, uint y);
uint operator | (uint x, uint y);
uint operator ^ (uint x, uint y);

bool operator == (uint x, uint y);
bool operator != (uint x, uint y);
bool operator <= (uint x, uint y);
bool operator < (uint x, uint y);
bool operator >= (uint x, uint y);
bool operator > (uint x, uint y);

uint operator << (uint x, int y);
uint operator >> (uint x, int y);

uint operator as (int x);
uint operator as (long x);
uint operator as (ulong x);
uint operator as (string s);

uint Min(uint x, uint y);
uint Max(uint x, unit y);
```

5.2.1.7. long

```
module Standard;
long operator + (long x);
long operator - (long x);
long operator ~ (long x);

long operator + (long x, long y);
long operator - (long x, long y);
long operator * (long x, long y);
long operator / (long x, long y);
long operator % (long x, long y);
long operator & (long x, long y);
long operator | (long x, long y);
long operator ^ (long x, long y);

bool operator == (long x, long y);
bool operator != (long x, long y);
bool operator <= (long x, long y);
bool operator < (long x, long y);
bool operator >= (long x, long y);
bool operator > (long x, long y);

long operator << (long x, int y);
long operator >> (long x, int y);

long operator as (ulong x);
long operator as (string s);

long Min(long x, long y);
long Max(long x, long y);
long Abs(long x);
```

5.2.1.8. ulong

```
module Standard;

ulong operator + (ulong x);
ulong operator - (ulong x);
ulong operator ~ (ulong x);

ulong operator + (ulong x, ulong y);
ulong operator - (ulong x, ulong y);
ulong operator * (ulong x, ulong y);
ulong operator / (ulong x, ulong y);
ulong operator % (ulong x, ulong y);
ulong operator & (ulong x, ulong y);
ulong operator | (ulong x, ulong y);
ulong operator ^ (ulong x, ulong y);

bool operator == (ulong x, ulong y);
bool operator != (ulong x, ulong y);
bool operator <= (ulong x, ulong y);
bool operator < (ulong x, ulong y);
bool operator >= (ulong x, ulong y);
bool operator > (ulong x, ulong y);

ulong operator << (ulong x, int y);
ulong operator >> (ulong x, int y);

ulong Min(ulong x, ulong y);
ulong Max(ulong x, ulong y);

ulong operator as (long x);
ulong operator as (string s);
```

5.2.2. Enums

```
// The EnumToString function attempts to interpret the given value as the
// given enum pattern name, and returns the corresponding friendly enum name.
// If the given value is not an enum, returns the normal string
// representation of the value instead.
string EnumToString(any @value, string enumPatternName);

// Tests if the t value is in the range of explicitly defined enum
// values of T.
bool InRange<T>(T t);
```

5.2.3. Floating Point

5.2.3.1. Float

```
module Standard;
float operator + (float x);
float operator - (float x);
float operator + (float x, float y);
float operator - (float x, float y);
float operator * (float x, float y);
float operator / (float x, float y);
float operator % (float x, float y);

bool operator <= (float x, float y);
bool operator < (float x, float y);
bool operator >= (float x, float y);
bool operator > (float x, float y);

float operator as (byte x);
float operator as (short x);
float operator as (ushort x);
float operator as (int x);
float operator as (uint x);
float operator as (long x);
float operator as (ulong x);
float operator as (double x);
float operator as (decimal x);

float operator as (string s);
```

5.2.3.2. Double

```
module Standard;

double operator + (double x);
double operator - (double x);
double operator + (double x, double y);
double operator - (double x, double y);
double operator * (double x, double y);
double operator / (double x, double y);

bool operator == (double x, double y);
bool operator != (double x, double y);
bool operator <= (double x, double y);
bool operator < (double x, double y);
bool operator >= (double x, double y);
bool operator > (double x, double y);

double operator as (byte x);
double operator as (short x);
double operator as (ushort x);
double operator as (int x);
double operator as (uint x);
double operator as (long x);
double operator as (ulong x);
double operator as (float x);
double operator as (decimal x);

double operator as (string s);

double Min(double x, double y);
double Max(double x, double y);
double Abs(double x);
```

5.2.3.3. Decimal

```
module Standard;
decimal operator + (decimal x);
decimal operator - (decimal x);
decimal operator + (decimal x, decimal y);
decimal operator - (decimal x, decimal y);
decimal operator * (decimal x, decimal y);
decimal operator / (decimal x, decimal y);
decimal operator % (decimal x, decimal y);
```

```
bool operator <= (decimal x, decimal y);
bool operator < (decimal x, decimal y);
bool operator >= (decimal x, decimal y);
bool operator > (decimal x, decimal y);

decimal operator as (byte x);
decimal operator as (short x);
decimal operator as (ushort x);
decimal operator as (int x);
decimal operator as (uint x);
decimal operator as (long x);
decimal operator as (ulong x);
decimal operator as (float x);
decimal operator as (double x);


decimal operator as (string s);

decimal Min(decimal x, decimal y);
decimal Max(decimal x, decimal y);
decimal Abs(decimal x);
```

5.2.4. Any Number

The **AnyNumber** is a pattern that represents all possible numeric types available in OPN.

```
module Standard;
pattern AnyNumber = byte | sbyte | short | ushort | int | uint | long |
                  ulong | float | double | decimal;
```

5.2.5. Booleans

```
module Standard;
bool operator ! (bool x);
bool operator ~ (bool x);

bool operator & (bool x, bool y);
bool operator ^ (bool x, bool y);
bool operator | (bool x, bool y);
bool operator && (bool x, bool y);
bool operator || (bool x, bool y);
bool operator <=> (bool x, bool y);
bool operator ==> (bool x, bool y);

bool operator == (bool x, bool y);
bool operator != (bool x, bool y);
bool operator <= (bool x, bool y);
bool operator < (bool x, bool y);
bool operator >= (bool x, bool y);
bool operator > (bool x, bool y);

bool operator as (int x);
bool operator as (string s);
```

5.2.6. Characters

```
module Standard;
bool operator == (char x, char y);
bool operator != (char x, char y);
bool operator <= (char x, char y);
bool operator < (char x, char y);
bool operator >= (char x, char y);
bool operator > (char x, char y);

char operator as (string s);

char ToUpper(this char c);
char ToLower(this char c);
```

5.3. Collections

As for scalars, collections API semantics aligns to C# conventions.

5.3.1. Strings

```
module Standard;
string operator + (string x, string y);
char operator [] (string x, int y);
bool operator in (char x, string y)

bool operator == (string x, string y);
bool operator != (string x, string y);
bool operator <= (string x, string y);
bool operator < (string x, string y);
bool operator >= (string x, string y);
bool operator > (string x, string y);

int IndexOf(this string x, string y);
int IndexOf(this string x, string y, int start);
int LastIndexOf(this string x, string y);
int LastIndexOf(this string x, string y, int start);

string Segment(this string x, int start);
string Segment(this string x, int start, int count);

string Replace(this string x, string phrase, string replacement);

int get Count(this string x);

string operator as (array<char> a);
string operator as (char c);

string ToUpper(this string s);
string ToLower(this string s);
string Trim(this string s);

int GetIterator(this string x);
char GetCurrent(this string x, int s);
bool MoveNext(this string x, ref int s);
```

5.3.2. Binary

```
module Standard;
binary operator + (binary x, binary y);
byte operator [] (binary x, int y);
bool operator in (byte x, binary y);

bool operator == (binary x, binary y);
bool operator != (binary x, binary y);
bool operator <= (binary x, binary y);
bool operator < (binary x, binary y);
bool operator >= (binary x, binary y);
bool operator > (binary x, binary y);

int IndexOf(this binary x, binary y);
int IndexOf(this binary x, binary y, int start);
int LastIndexOf(this binary x, binary y);
int LastIndexOf(this binary x, binary y, int start);

binary Segment(this binary x, int start);
binary Segment(this binary x, int start, int count);

int get Count(this binary x);

binary operator as (array<byte> x);
binary operator as (string s);

int GetIterator(this binary x);
byte GetCurrent(this binary x, int s);
bool MoveNext(this binary x, ref int s);
```

5.3.3. Arrays

```
module Standard;
array<T> operator + <T>(array<T> x, array<T> y);
T operator [] <T>(array<T> x, int y);
bool operator in <T>(T x, array<T> y);

bool operator == <T>(array<T> x, array<T> y);
bool operator != <T>(array<T> x, array<T> y);

int IndexOf<T>(this array<T> x, array<T> y);
int IndexOf<T>(this array<T> x, array<T> y, int start);
int LastIndexOf<T>(this array<T> x, array<T> y);
int LastIndexOf<T>(this array<T> x, array<T> y, int start);

array<T> Segment<T>(this array<T> x, int start);
array<T> Segment<T>(this array<T> x, int start, int count);

int get Count<T>(this array<T> x);

array<char> operator as (string s);
array<byte> operator as (binary b);

array<R> Select<T,R>(this array<T> x, R(T) f);
R Accumulate<T,R>(this array<T> x, R(T,R) f);
array<T> Filter<T>(this array<T> x, bool(T) f);
void Do<T>(this array<T> x, void(T) f);

// Returns if all elements in an array are different.
bool distinct<T>(this array<T> x);
// Returns the intersection of two arrays.
array<T> intersection<T>(this array<T> x, array<T> y);
```

```
// Returns the difference of two arrays.
array<T> except<T>(this array<T> x, array<T> y);
// Applies an accumulator function over a sequence. The specified seed value
// is used as the initial accumulator value, similar to "fold" function in
// functional languages.
Q aggregate<T, Q>(this array<T> x, Q seed, (Q, array<T>) => Q func);

// Returns the sum of all the elements of an array.
AnyNumber sum<AnyNumber>(this array<AnyNumber> x);
// Returns the minimum of the array, or nothing if the array is empty.
optional IComparable min<IComparable>(this array<IComparable> x);
// Returns the maximum of the array, or nothing if the array is empty.
optional IComparable max<IComparable>(this array<IComparable> x);
// Returns the average of the array, or nothing if the array is empty.
optional float average<AnyNumber>(this array<AnyNumber> x);
// Returns an array where each element is the absolute value of the
// corresponding element in the original array.
array<AnyNumber> abs<AnyNumber>(this array<AnyNumber> x);

int GetIterator<T>(this array<T> x);
T GetCurrent<T>(this array<T> x, int s);
bool MoveNext<T>(this array<T> x, ref int s);
```

5.3.4. Maps

```
module Standard;
map<K,D> operator + <K,D>(map<K,D> x, map<K,D> y);
D operator [] <K,D>(map<K,D> x, K y);
bool operator in <K,D>(K x, map<K,D> y);

bool operator == <K,D>(map<K,D> x, map<K,D> y);
bool operator != <K,D>(map<K,D> x, map<K,D> y);

int get Count<K,D>(this map<K,D> x);

set<K> get Keys<K,D>(this map<K,D> x);
array<D> get Values<K,D>(this map<K,D> x);
map<K,D> Remove<K,D>(this map<K,D> x, K y);

map<K,R> Select<K,D,R>(this map<K,D> x, R(K,D) f);
R Accumulate<K,D,R>(this map<K,D> x, R(K,D,R) f);
map<K,D> Filter<K,D>(this map<K,D> x, bool(K,D) f);
void Do<K,D>(this map<K,D> x, void(K,D) f);

int GetIterator<K,D>(this map<K,D> x);
Pair<K,D> GetCurrent<T>(this map<K,D> x, int s);
bool MoveNext<K,D>(this map<K,D> x, ref int s);
type Pair<K,D>;
K get Key(this Pair<K,D> x);
D get Value(this Pair<K,D> x);
```

5.3.5. Sets

```
module Standard;
set<T> operator + <T>(set<T> x, set<T> y);
set<T> operator + <T>(set<T> x, T y);
bool operator in <T>(T x, set<T> y);

bool operator [] <T>(set<T> x, T y);

bool operator == <T>(set<T> x, set<T> y);
bool operator != <T>(set<T> x, set<T> y);

int get Count<T>(this set<T> x);

set<R> Select<T,R>(this set<T> x, R(T) f);
R Accumulate<T,R>(this set<T> x, R(T,R) f);
set<T> Filter<T>(this set<T> x, bool(T) f);
void Do<T>(this set<T> x, void(T) f);

int GetIterator<T>(this set<T> x);
T GetCurrent<T>(this set<T> x, int s);
bool MoveNext<T>(this set<T> x, ref int s);
```

5.4. Streams

The stream API allows the OPN author to access the stream directly, as a way to implement parsers. Function names are self explanatory.

```
module Standard;

bool get CurrentBit(this stream s);
int get BitPosition(this stream s);
int get BitLength(this stream s);
binary PeekBits(this stream s, int bitOffset, int bitCount);

byte get CurrentByte(this stream s);
int get BytePosition(this stream s);
int get ByteLength(this stream s);
binary PeekBytes(this stream s, int bitOffset, int byteCount);

char get CurrentChar(this stream s);
int get CharPosition(this stream s);
int get CharLength(this stream s);
string PeekChars(this stream s, int bitOffset, int charCount);
```

5.5. Messages

5.5.1. Annotations

All the provided annotations are defined for all messages. As explained before, a user can add new annotations at will.

```
module Standard;

// Timestamp of the message, assigned by runtime when the message reaches
// the message source.
annotation DateTime Timestamp;
// EndTimestamp refers to the completed timestamp of the operation. Usually
// this is the timestamp of the last origin of the operation (or a
// reassembled message).For a message, the EndTimestamp value is the same as
// its Timestamp value.
annotation DateTime EndTimestamp;
// TimeElapsed is the difference of EndTimestamp and Timestamp annotation
// values for a particular message.
annotation double TimeElapsed;
// This is a GUID to identify a message session. This value is assigned
// by runtime.
annotation guid SessionId;
// A unique number assigned to the message by runtime, guaranteed unique
// per session.
annotation uint MessageNumber;
// Generic comment attached to a message, available for the user.
annotation string Comment;
// Embedded protocols use this annotation to attach embeddedspecific data.
// The key of the map is the embedded protocol name.
annotation map<string, any> Embedded;
// General diagnosis errors/warnings. Sequence and data validation problems
// are identified here.
annotation DiagnosisType DiagnosisTypes;
annotation DiagnosisLevel DiagnosisLevels;
```

The **DiagnosisLevel** annotation represents the severity of the error, and the possible values are the following.

```
pattern DiagnosisLevel = enum { Error = 1, Warning = 2, Information = 4 };
```

The **DiagnosisType** represents the category of the error.

```
pattern DiagnosisType = enum  DiagnosisType {
        Application = 0x01,
        Validation = 0x02,
        InsufficientData = 0x04,
        Parsing = 0x08
}
```

5.5.2. Origins

These functions expose the **origins** of a message. The array of messages represents the associated direct child messages of a given message.

```
module Standard;

array<any message> get Origins(this any message m);
void set Origins(this any message m, array<any message> origins);
```

5.6. Endpoint Transport

These functions allow accessing the underlying endpoint of a given endpoint.

```
any endpoint GetTransport(this any endpoint x);
T GetTransport<T>(this any endpoint x);
```

5.7. XML

The following **DeclarationInfo** aspect is for internal purposes only, and it is not supposed to be used by the end user.

```
//////////////////////////////////////////////
// Part 1. XML Data Types

// Represents a name of an XML element or attribute.
type XmlName : TreedataName {
    // Gets the namespace part of the fully qualified name.
    // Returns nothing if there is no namespace;
    // Returns empty string "" if there is a prefix but the namespace ca not
    // be resolved.
    optional string NamespaceUri;
}

//////////////////////////////////////////////
// Part 2. XML Properties

// Gets the name of an XML element or attribute.
// Returns nothing if x.Kind is not Element or Attribute.
optional XmlName get Name(this xml x) with DeclarationInfo {
            Handcoded = true };

// Gets the kind of an XML element or attribute.
XmlKind get Kind(this xml x) with DeclarationInfo {
            Handcoded = true };

// Gets the concatenated text contents of an XML structure,
// as described in this list:
//   [XmlKind]         [Value property]
//   Attribute         value of the Attribute
//   Namespace         namespace URI
//   CData             {EscapedText}
//   Comment           {Comment}
//   Declaration       nothing
//   Document          nothing
//   Element           nothing (according to dot net APIs)
//   Text              textual content
optional string get Value(this xml x) with DeclarationInfo {
            Handcoded = true };

// Gets the string representation of the inner content within an
// XML Element.
optional string get InnerXml(this xml x) with DeclarationInfo {
            Handcoded = true };

// Gets the string representation of the entire XML structure.
optional string get OuterXml(this xml x) with DeclarationInfo {
            Handcoded = true };

// Gets the string representation of the text content within the
// XML structure.
// Element: the concatinated text within the Element.
// Others: the Value.
optional string get Text(this xml x) with DeclarationInfo {
            Handcoded = true };

// Gets the resolved namespace URI for an XML Namespace.
// Returns nothing if x has no namespace or its kind is not Element.
optional string get NamespaceUri(this xml x) with DeclarationInfo {
            Handcoded = true };

// Gets the count of children.
// Returns 0 if x.Kind is not Element or Document.
int get ChildCount(this xml x) with DeclarationInfo {
            Handcoded = true };

// Gets the children elements as array of XML elements.
// Returns array contains only one XML elment of kind XmlKind.
// Text for Attribute. Returns empty array
//  if x.Kind is not Element, Attribute or Document.
array<xml> get Children(this xml x) with DeclarationInfo {
            Handcoded = true };

// Gets the count of attributes.
// Returns 0 if x.Kind is not Element or Declaration.
int get AttributeCount(this xml x) with DeclarationInfo {
            Handcoded = true };

// Gets attributes.
// Returns empty map if x.Kind is not Element or Declaration.
array<xml> get Attributes(this xml x) with DeclarationInfo {
```

```
            Handcoded = true };

// Sets the name of the XML element.
// Throws exception if x.Kind is not Element.
void set Name(this xml x, XmlName name) with DeclarationInfo {
            Handcoded = true };

// Sets the value of the XML Attribute.
// Throws exception if x.Kind is Declaration or Document.
void set Value(this xml x, string s) with DeclarationInfo {
            Handcoded = true };

// Sets the Namespace URI of the XML Namespace.
// Throws exception if x.Kind is not Element or Attribute.
void set NamespaceUri(this xml x, string uri) with DeclarationInfo {
            Handcoded = true };

///////////////////////////////////////////////
// Part 3. XML Query/Update methods

// Returns a single XML element matched to the given name from all child
// elements of this XML element, or returns an arbitrary one if more than
// one child matched. Returns nothing if x.Kind is Attribute, CData,
// Comment, Declaration or Text.
optional xml GetChild(this xml x, XmlName name) with DeclarationInfo {
            Handcoded = true };

// Returns a single  XML element matched to the given name from all
// descendant elements of XML element, returns an arbitrary one if more
//  than one descendant matched. Returns nothing if
//  x.Kind is CData, Comment, Declaration or Text.
optional xml GetDescendant(this xml x, XmlName name) with DeclarationInfo {
            Handcoded = true };

// Returns a filtered array of XML elements matched the given name from
// all children elements of the XML element, in document order.
// Returns empty array if x.Kind is CData, Comment, Declaration or Text.
array<xml> GetChildren(this xml x, XmlName name) with DeclarationInfo {
            Handcoded = true };

// Returns a filtered array of XML elements matched to the given name from
// all descendant elements of the XML element, in document order.
// Returns empty array if x.Kind is CData, Comment, Declaration or Text.
array<xml> GetDescendants(this xml x, XmlName name) with DeclarationInfo {
            Handcoded = true };

// Returns an attribute matched the given name of the XML element.
// Returns nothing if x.Kind is CData, Comment, Document or Text.
optional xml GetAttribute(this xml x, XmlName name) with DeclarationInfo {
            Handcoded = true };

// !FIXME: XPath query is not supported so far.
// Returns an XML element using the specified XPath-like expression with the
// specified mapping to resolve namespace prefixes.
// Returns an arbitrary one if more than one XML element matches the path.
// Returns nothing if x.Kind is not Document or Element.
optional xml Get(this xml x, string @xpath, map<string,string> resolver) with
            DeclarationInfo { Handcoded = true };


// !FIXME: XPath query is not supported so far.
// Returns an array of XML elementsusing the specified XPath-like expression
// with the specified mapping to resolve namespace prefixes.
// Returns empty array if x.Kind is not Document or Element.
array<xml> GetMany(this xml x, string @xpath, map<string,string> resolver)
            with DeclarationInfo { Handcoded = true };

// Adds a child XML element or attribute under the given XML element.
// Throws exception if x.Kind is not Document or Element.
// Throws exception if x is Kind of Document and
// already has one child with kind of Element.
void Add(this xml x, xml y) with DeclarationInfo { Handcoded = true };

// Adds a namespace declaration under the given XML element as attribute.
// Throws exception if x.Kind is not Element.
void AddNamespace(this xml x, string prefix, string uri) with
            DeclarationInfo { Handcoded = true };

// Parses string containing the XML element and builds an XML object from it
// Returns null if the data is not in valid XML format.
xml BuildXml(string data) with DeclarationInfo { Handcoded = true };
```

5.8. JSON

The following **DeclarationInfo** aspect is for internal purposes only, and it is not supposed to be used by the end user.

```
// Explicit cast operator is provided from treedata to JSON.
json operator as (treedata t) with DeclarationInfo { OperatorKind =
      OperatorKinds.Casting };

// Gets the name of an element.
optional TreedataName get Name(this json j) with DeclarationInfo {
            Handcoded = true };

// Gets the kind of an element.
TreedataKind get Kind(this json j) with DeclarationInfo {
            Handcoded = true };

// Gets the value of an element.
optional string get Value(this json j) with DeclarationInfo {
            Handcoded = true };

// Gets the count of children.
int get ChildCount(this json j) with DeclarationInfo {
```

```
            Handcoded = true };

// Gets the children elements.
array<json> get Children(this json j) with DeclarationInfo {
            Handcoded = true };

// Returns a single child matched the given name from all child elements.
optional json GetChild(this json j, TreedataName name) with DeclarationInfo {
            Handcoded = true };

// Returns a single element matched the given name from all descendant
// elements.
optional json GetDescendant(this json j, TreedataName name) with
            DeclarationInfo { Handcoded = true };

// Returns a filtered array of treedata matched the given name from all
// children elements, in document order.
array<json> GetChildren(this json j, TreedataName name) with
            DeclarationInfo {Handcoded = true };

// Returns a filtered array of treedata matched the given name from all
// descendant elements, in document order.
array<json> GetDescendants(this json j, TreedataName name) with
            DeclarationInfo { Handcoded = true };

// Returns a treedata using the specified XPath-like expression with
// the specified mapping to resolve namespace prefixes.
// Returns an arbitrary one if more than one matches the path.
optional json Get(this json j, string @xpath, map<string,string> resolver)
            with DeclarationInfo { Handcoded = true };

// Returns an array of treedata using the specified XPath-like expression
// with the specified mapping to resolve namespace prefixes.
array<json> GetMany(this json j, string @xpath, map<string,string> resolver)
            with DeclarationInfo { Handcoded = true };
// Parses string containing JSON and builds JSON object from it
// Returns null if the data is not valid JSON format.
      json BuildJson(string data) with DeclarationInfo { Handcoded = true };
```

5.9. Treedata

The following **DeclarationInfo** aspect is for internal purposes only, and it is not supposed to be used by the end user.

```
// Represents a name for tree-structured data, backward compatible
// with XmlName to make it uniform with treedata and JSON.
type TreedataName {
    // Gets the local (unqualified) part of the name.
    string LocalName;
}
```

```
// Gets the name of an element.
optional TreedataName get Name(this treedata x) with DeclarationInfo {
            Handcoded = true };

// Just a renaming to avoid making explicit references to xml.
// The same type makes sense to JSON as well.
typedef TreedataKind = XmlKind;

// Gets the kind of an element.
TreedataKind get Kind(this treedata t) with DeclarationInfo {
            Handcoded = true };

// Gets the value of an element.
optional string get Value(this treedata t) with DeclarationInfo {
            Handcoded = true };

// Gets the count of children.
int get ChildCount(this treedata t) with DeclarationInfo {
            Handcoded = true };

// Gets the children elements.
array<treedata> get Children(this treedata t) with DeclarationInfo {
            Handcoded = true };

// Returns a single child matched the given name from all child elements.
optional treedata GetChild(this treedata t, TreedataName name) with
            DeclarationInfo { Handcoded = true };

// Returns a single element matched the given name from all descendant
// elements.
optional treedata GetDescendant(this treedata t, TreedataName name) with
      DeclarationInfo { Handcoded = true };

// Returns a filtered array of treedata matched the given name from all
// children elements, in document order.
array<treedata> GetChildren(this treedata t, TreedataName name) with
            DeclarationInfo { Handcoded = true };

// Returns a filtered array of treedata matched the given name from all
// descendant elements, in document order.
array<treedata> GetDescendants(this treedata t, TreedataName name) with
            DeclarationInfo { Handcoded = true };

// Returns a treedata using the specified XPath-like expression with the
// specified mapping to resolve namespace prefixes.
// Returns an arbitrary one if more than one matches the path.
optional treedata Get(this treedata x, string @xpath, map<string,string>
            resolver) with DeclarationInfo { Handcoded = true };

// Returns an array of treedata using the specified XPath-like expression
// with the specified mapping to resolve namespace prefixes.
array<treedata> GetMany(this treedata x, string @xpath, map<string,string>
            resolver) with DeclarationInfo { Handcoded = true };
```

5.10. Date and Time

Both **DateTime** and **TimeSpan** are extern types for OPN.

```
module Standard;

extern type DateTime;
extern type TimeSpan;

optional DateTime ToDateTime (this string s);
optional TimeSpan ToTimeSpan(this string s);
```

5.11. Modalities

The following declarations address the representation in OPN of modality keywords (MUST, MAY, SHOULD), as defined in RFC 2119.

A Boolean method is used to represent a modality. The semantics for parsing, validation and simulation would be to allow both interpretations (DOES, DOES NOT) unless a value of true or false has been specified or inferred.

```
module Standard;

pattern Modality = enum
{
	MAY,
	SHOULD
}

bool Option(Modality modality, string identifier, Map<string, bool> perVendor
	);

bool Option(Modality modality, string identifier)
{
	return Option(modality, identifier, {});
}

bool Option(Modality modality, Map<string, bool> perVendor)
{
	return Option(modality, "", perVendor);
}

bool Option(Modality modality)
{
	return Option(modality, "", {});
}

bool May(string identifier, Map<string, bool> perVendor)
```

```
{
        return Option(Modality.MAY, identifier, perVendor);
}

bool May(string identifier)
{
        return May(identifier, {});
}

bool May(Map<string, bool> perVendor)
{
        return May("", perVendor);
}

bool May()
{
        return May("", {});
}

bool Should(string identifier, Map<string, bool> perVendor)
{
        return Option(Modality.SHOULD, identifier, perVendor);
}

bool Should(string identifier)
{
        return Should(identifier, {});
}

bool Should(Map<string, bool> perVendor)
{
        return Should("", perVendor);
}

bool Should()
{
        return Should("", {});
}
```

5.12. Error handling

Standard library provides a way to handle non-fatal validation and protocol errors. When processing a message, must-stop (fatal) errors can be already handled by specifying a **`where`** clause describing the conditions under which the message is not valid. To deal with non-fatal errors, standard library provides a set of functions that can be used in **`where`** clauses to just signal that some problem occurred during parsing, but that current message can continue processing. These Boolean functions always return *true*, so from a **`where`** clause perspective, no constraints are violated. When non-fatal error conditions are met, the compiler provides diagnosis information to the runtime so an error message can be displayed appropriately.

5.12.1. ValidationCheck

The **ValidationCheck** function is provided to check whether non-fatal errors occurred.

```
module Standard;

bool ValidationCheck(bool expression, string ErrorDescription);
bool ValidationCheck(bool expression, any message context, string
            ErrorDescription);
bool ValidationCheck(bool condition, any message context, DiagnosisLevel
            level, string description);
```

When the message context is **null**, or when using the overloaded function, the error will be attached to the current message that is being processed, which is the lower layer message in the current runtime implementation. Otherwise, it will be attached to the message instance passed as a parameter. As mentioned before, **ValidationCheck** always returns true.

When **DiagnosisLevel** is not specified, **Warning** is the default.

Typical value validations could be the following:

- A value is in a certain range or one of the specified values.
- A value must contain certain characters.
- A value has a certain length.

To give an example, note the following.

```
type SomeType
{
      int Checksum where ValidationCheck(value > 1, null, "A value greater
            than 1 was expected");
      // ...
}
```

If the value assigned to **Checksum** is not greater than 1, then a notification will be sent to runtime, attached to the message currently being processed.

5.12.2. ErrorCodeIf

The **ErrorCodeIf** function represents a legitimate protocol error (such as SMB_ACCESS_DENIED for SMB protocol). These kinds of errors are usually encoded in a special type of message, or indicated by a message with special values.

```
bool ErrorCodeIf(Boolean expression, any message context, string
            ErrorDescription);
```

```
bool ErrorCodeIf(Boolean expression, string ErrorDescription);
```

Message context works in the same way as for the previous function, using **`null`** to represent that the error will be attached to the message currently being processed.

```
type SomeType
{
Code AckCode where ErrorCodeIf(value != S_OK, null,
      "The ACK code is not ok");
      // ...
}
```

In this example, the runtime will be notified that the current message being processed represents a protocol error if the **`AckCode`** field is not equal to **`S_OK`**.

5.12.3. ReportInsufficientData

The **`ReportInsufficientData`** function represents the case when additional data was expected, but it is known that this data will never arrive.

```
bool ReportInsufficientData(any message context, DiagnosisLevel level,
      string description);
```

Message context works in the same way as for the previous function, using **`null`** to represent that the error will be attached to the message currently being processed.

5.13. Logging Messages

There is a special type of endpoint to which log messages can be dispatched. The runtime listens to this special logging endpoint and from the processing perspective can be considered as a regular endpoint.

```
pattern LogLevel = enum
{
      Error = 1,    // All errors
      Warning = 2,  // All warnings, including errors.
      Info = 4      // All informational, including warnings.
}

// Dispatch the logMessage to the logging endpoint with a specific logLevel.
void DispatchLog(LogLevel logLevel, string logMessage);
```

```
// Converts a value to string and logs the result.
void DispatchLog(LogLevel logLevel, any logValue);

// If there is no log level specified, by default log level = LEVEL_INFO.
void DispatchLog(string logMessage);
void DispatchLog(any logValue);
```

An example of use is the following.

```
process this accepts s:Segment{Kind = $DATA}
{
      InsertSegment(s);
      while (segmentsReady > 0)
      {
            var socket = endpoint Socket[Port, DestinAddress, DestinPort]
                        over this.Node;
            DispatchLog("Accepted data segment:");

            String message ;
            foreach (var byte in orderedSegments[0])
            {
                  message += byte + "|";
                  dispatch socket accepts byte;
            }
            DispatchLog(message);
            segments = orderedSegments.RemoveAt(0);
            segmentsReady -= 1;
      }
}
```

5.14. Associating On-the-Wire Data to OPN Values

OPN Standard library offers a set of functions to associate on-the-wire data to OPN values. This association is automatically made when invoking PEF decoders such as the **`BinaryDecoder`** or the **`XMLDecoder`**, but it can be explicitly invoked in OPN code when a custom parser is being written. This association can be later consumed by any artifact (the UI, for example).

5.14.1. Basic Associations

The most common decoding cases involve associating a particular field of a container (the container can be a message or a reference type) to a collection of data chunks. That functionality is exposed with the following function.

```
// Associate a field that is part of a container (message or reference type)
// to data chunks.
void AssociateField(this any type containerInstance, string fieldName,
      DataChunks chunks);
```

The following list describes the fields in the preceding function:

- The **containerInstance** should be either a message or a reference type.
- The **fieldName** should be a valid field name of the container .
- The **chunks** is the information representing the data source fragments where the container field was decoded from.
- All association functions (this and the ones definedfollowing) overrides previously defined associations if invoked for the same container instance and field name.

Field names are case-sensitive. The field name that is passed must match a field name of a container in a case-sensitive way. If the field name that was passed does not match any of the field names in the container, an execution exception will be thrown.

A field is decoded from a given source, and several fragments of that source could have been used to decode that field. The source is set at the field level, and that implies that a container (that is a message or reference type) can have fields decoded from different data sources.

```
type DataChunks
{
    any DataSource;
    array<Chunk> Chunks;
}
```

Observe that a source can be from any type. Nevertheless, there are three expected cases, representing the three most common sources: a binary stream, a character stream, and an XML structure. This is not enforced by the API and it is left as a contract between the decoder writer and the consumer, which is usually the UI. Therefore, a **Chunk** will represent the fragments of each case in a different way.

```
interface Chunk{}

type BinaryChunk: Chunk
{
    int bitPosition where value >= 0;
    int bitLength where value >= 0;
}

type CharacterChunk: Chunk
{
    int charPosition where value >= 0;
    int charLength where value >= 0;
}

type XMLChunk: Chunk
{
    string xpath;
}
```

The following describes each type of **`Chunk`**:

- The **`BinaryChunk`** is a pair representing a position in the binary stream and the number of bytes from that position.
- The **`CharacterChunk`** is a pair representing a position in the character stream and the number of characters from that position. Observe that this type is equivalent to **`BinaryChunk`** from a structural perspective, but they are modeled separately to make the distinction explicit: magnitudes are different, and the way to measure bits is not the same as the way to measure characters.
- The **`XMLChunk`** is a string representing an XPath that selects from a source with XML type a fragment of XML that was used to decode the field.
- In case of a mismatch between the **`Chunk`** subtype and the type of information decoded (for example an **`XMLChunk`** is associated to binary data, decoded by **`BinaryDecoder`**) an execution exception is thrown.

For an example that uses this function consider this declared message.

```
message MyMessage
{
    string UserName;
    int Flags;
}
```

A user writing a custom parser to decode this message can invoke the association function in this way.

```
MyMessage customDecoder(stream s)
{
    MyMessage m = new MyMessage();
    int currentPosition = s.BitPosition;

    // Decode the username and associate the corresponding data chunks
    // and source.
    m.UserName = decodeUserName(s);
    DataChunks usernameChunk = new DataChunks();
    usernameChunk.DataSource = s;
    usernameChunk.Chunks = {new BinaryChunk{bytePosition = currentPosition,
            byteLenght = s.BitPosition - currentPosition}}
    m.AssociateField("UserName", usernameChunk);

    // Decode the Flags and associate the corresponding data chunks
    // and source.
    int currentPosition = s.BitPosition;
    m.Flags = decodeFlags(s);
    DataChunks flagsChunk = new DataChunks();
    flagsChunk.DataSource = s;
    flagsChunk.Chunks = [new BinaryChunk{
            bytePosition = currentPosition, byteLenght = s.BitPosition -
            currentPosition}]
    m.AssociateField("Flags", flagsChunk);

    return m;
}
```

In the preceding example we assumed that the stream is treated as a binary stream, and that each field of the message has a unique chunk of data.

5.14.2. Associating Collections

It is also possible to associate data to collections.

5.14.2.1. Arrays

The first one deals with arrays specificallythose that are the most common case for collections.

```
// Associates an array that is part of a container
// (message or reference type) to an array of data chunks.
// There is an implicit one-to-one association between each
// element of the array field and each element of the array of chunks.
void AssociateArrayField(this any containerInstance, string arrayFieldName,
      array<DataChunks> arrayOfChunks);
```

The following are some restrictions that will be enforced at execution time.

- The provided **arrayFieldName** should be a valid field name of **containerInstance** and must have type **array<T>.**
- The size of **arrayOfChunks** must be the same as the size of **arrayFieldName**.

The other available function is the following.

```
// Selectively associates some elements of an array that is part of
// a container (message or reference type) to data chunks.
// The map keys stores positions of the array,
// and the map values of the associated chunks.
void AssociateArrayField(this any containerInstance, string arrayFieldName,
      map<int, DataChunks> mapOfChunks);
```

The previous function allows associating just some positions of the array by providing a map from positions (zero-based) to data chunks. This case is useful when the decoded array is big and the data to associate to is only present for sparse elements.

Observe that in this way, for each array element, a different source and data chunks can be associated to it,for example.

```
message MyMessage
{
    array<int> Options;
}

MyMessage MyMessageDecoder(stream s)
{
    MyMessage m = new MyMessage();
    int currentPosition = s.CurrentByte;

    // Decodes the array of options.
    m.Options = decodeArray(s);
    // Assumes each decoded element is a 4 byte integer.
    array<DataChunks> chunks = [];
    for(int i = 0; i < chunks.Count; i++)
    {
       chunks+= new DataChunks{DataSource = s, Chunks =
       [new BinaryChunk{bytePosition = currentPosition + i*4,
       byteNumber = 4}]};
    }
    m.AssociateArrayField("Options", chunks);

    return m;
}
```

5.14.2.2. Sets

To deal with the case of sets the following function is provided.

```
// Similar as the preceding example, but for sets. Each element of the set
// that the user wants to associate has to be the key of the chunk map in
// order to  establish the association.
void AssociateSetField(this any containerInstance, string
      collectionFieldName, map<any, DataChunks> mapOfChunks);
```

The following are the restrictions:

- The provided **`collectionFieldName`** should be a valid field name of **`containerInstance`** and must have type **`set<T>`**.
- The provided map of chunks must have type **`map<T, DataChunks>`**.

Not all elements in the set need to be associated. For each element **`e`** in the specified container set, an association is made if **`e`** is a key of **`mapOfChunks`**. Take the previous example and use this function to associate *data chunks* to the first element of the array.

```
MyMessage MyMessageDecoder(stream s)
{
    MyMessage m = new MyMessage();
    int currentPosition = s.CurrentByte;

    // Decodes a set of options.
    m.Options = decodeSet(s);

    // Creates a data chunk for the first element.
    DataChunks chunks = new DataChunks{DataSource = s, Chunks =
      [new BinaryChunk{bytePosition = currentPosition, byteNumber = 4}]};

    // Associates just the element MAIN_ELEMENT of the set, if that exists.
    if (m[MAIN_ELEMENT])
    {
        m.AssociateSetField("Options", {MAIN_ELEMENT -> chunks);
    }
    return m;
}
```

5.14.2.3. Maps

Finally, a function to deal with maps is provided.

```
// Similar as the preceding example, but for maps. Each element of the
// decoded map is a key-value pair, so two maps are provided, one for keys
// and the other for pairs.
void AssociateMapField(this any containerInstance, string
      collectionFieldName, map<any, DataChunks> mapOfChunksForKeys, map<any,
      DataChunks> mapOfChunksForValues);
```

The following are the restrictions:

- The provided **`collectionFieldName`** should be a valid field name of **`containerInstance`** and must have type **`map<T,Q>.`**
- The provided map of chunks **mapOfChunksForKeys** must have type **`map<T, DataChunks>.`**
- The provided map of chunks **`mapOfChunksForValues`** must have type **`map<Q, DataChunks>.`**

The decoded keys and values that the user wants to associate should be keys of **`mapOfChunksForKeys`** and **`mapOfChunksForValues`** respectively.

5.14.3. Retrieving Associations

Associated chunks can be retrieved in the following way.

```
// Retrieve the array of data chunks associated to a value.
// Returns null if no values are associated yet.
```

```
any GetFieldAssociation(this any containerInstance, string fieldName);
```

One of the following is the returning type of the function:

- The **`DataChunks`** if **`AssociateField`** was used for the given container instance and field name.
- The **`array<DataChunks>`** if **`AssociateArrayField`** was used for the given container instance and field name.
- A pair of **`map<any, DataChunks>`** (containing one map for keys and another one for values) if **`AssociateMapField`** was used for the given container instance and field name.

If no association was made for that particular container and field, **`null`** is returned.

5.14.4. Summary of the API

```
// Associate a field that is part of a container (message or reference type)
// to data chunks.
void AssociateField(this any type containerInstance, string fieldName,
      DataChunks chunks);

// Associate an array that is part of a container (message or reference type)
// to a array of data chunks. There is an implicit one-to-one
// association between each element of the array field and each element of
// the array of chunks.
void AssociateArrayField(this any containerInstance, string arrayFieldName,
      array<DataChunks> arrayOfChunks);

// Selectively associates some elements of an array that is part of
// a container (message or reference type) to data chunks.
// The map keys stores positions of the array,
// and the map values of the associated chunks.
void AssociateArrayField(this any containerInstance, string arrayFieldName,
      map<int, DataChunks> mapOfChunks);

// Similar as the preceding example but for sets. Each element of the
// collection that the user wants to associate has to be the key of the
// chunk map in order to establish the association.
void AssociateSetField(this any containerInstance, string
      collectionFieldName, map<any, DataChunks> mapOfChunks);

// Similar as the preceding example but for maps. Each element of the decoded
// map is a key-value pair, so two maps are provided, one for keys and
// the other for pairs.
void AssociateMapField(this any containerInstance, string
      collectionFieldName, map<any, DataChunks> mapOfChunksForKeys, map<any,
      DataChunks> mapOfChunksForValues);

// Retrieves the array of data chunks associated to a value. Returns null
// if no values are associated yet.
any GetFieldAssociation(this any containerInstance, string fieldName);

// Data chunks has a data source and a set of chunks.
```

```
type DataChunks
{
    any DataSource;
    array<Chunk> Chunks;
}

// A Chunk is just an interface, and we currently provide Binary,
// Character and XML chunk type.
interface Chunk{}

type BinaryChunk: Chunk
{
    int bitPosition where value >= 0;
    int bitLength where value >= 0;
}

type CharacterChunk: Chunk
{
    int charPosition where value >= 0;
    int charLength where value >= 0;
}

type XMLChunk: Chunk
{
    string xpath;
}
```

5.15. Trace Library

This library adds a first level of debugging to OPN by enabling authors to write information to a log at run time. The destination of log messages is specified by the tool that consumes OPN, and this API is agnostic at this respect.

```
module Diagnostics;
int LogIndentLevel = 0 ;  // Get or set the current log indent level.
int LogIndentSize = 4;    // Get or set the number of spaces in a log indent.
// Gets or sets whether log Flush should be performed on every write.
bool AutoFlushLog = false;

// Flushes the log output buffer and causes buffered data to be written.
void FlushLog();
void IndentLog();    // Increases the current log indent level by one.
void UnindentLog();  // Decreases the current log indent level by one.

void Log(string m);      // Logs a message.
void Log(any v);         // Converts a value to string and logs the result.
void LogLine(string m);  // Logs a message followed by a line break.
void LogLine(any v);     // Logs string version of a value and a line break.

void LogIf(bool c, string m);  // Logs a message if a condition is true.
void LogIf(bool c, any v);     // Logs the string version of a value.
// Conditionally logs a message followed by a line break.
void LogLineIf(bool c, string m);
```

```
// Conditionally logs the string version of a value and a line break.
void LogLineIf(bool c, any v);

void LogError(string m);    // Logs an error message.
void LogWarning(string m);  // Logs a warning message.

any Log(any v, string m);    // Returns a value and logs a message.
// Returns a value, and logs a message and a line break.
any LogLine(any v, string m

// String format is provided as a way to customize a string
// with specific values.
string Format(string m, any v);
string Format(string m, any v1, any v2);
string Format(string m, any v1, any v2, any v3);
string Format(string m, array<any> v);
```

Formatting functions take the same conventions used by C# String.Format. See Formatting Types (string) for details.

5.16. Aspects

The standard library provides the following set of predefined aspects that support a rich means to attach metadata to declarations.

5.16.1. Aspects for Data Mapping

This group of aspects enables attaching information on how to map data (typically protocol payloads) and transform it to OPN logical values. The standard library provides aspects for describing how to transform binary, textual and XML data. This mapping information will be usually queried by codecs, responsible for parsing the data.

As a general design guideline, the use of codecs guided by aspects is a way to specify a declarative parsing mechanism intended to cover simple and common cases. They are not intended to solve situations with a complex parsing logic. When these situationsarrise, writers should switch to syntax constructs or to directly access the stream using the stream API.

As a general rule, aspects attached to a field with a collection-like type (such as an array, for example) do not propagate its elements, unless specified otherwise.

5.16.1.1. General Encoding

These aspects specify encoding characteristics that can be shared among different aspects. The **`ignore`** attribute specifies that the entity they are attached to should be ignored (when the**`ignore`** attribute is set to true). These types of aspects usually come in two flavors: the first can be attached to containers to specify defaults for elements, after the container, the second can be attached to a field via the **this** field.

```
module Standard;

aspect EncodingDefaults
{
    bool ignore = false;
}

aspect Encoding : EncodingDefaults
{
      any this;
}
```

5.16.1.2. Binary Encoding

The binary encoding aspect aggregates all attributes related to binary encoding of values.

5.16.1.2.1. BinaryEncoding Declaration

```
module Standard;

aspect BinaryEncoding : BinaryEncodingDefaults
{
    // CommonBinaryEncodingAttributes
    uint LeadPadding;   // Number of leading bits to ignore.
    uint TrailPadding;  // Number of trailing bits to ignore.

    // List Binary Encoding Attributes
    uint Length;


    // Width is used to specify the number of bits for
    // on-the-wire representation. Width can be only be applied
    // to field with numeric types such as int, short, and so on.
    uint Width;

    // DecodeAsUnit is used to specify a reference type is considered
    // as a unit when decoding or encoding, and the length in bits
    // of the unit must be specified by the property WidthForComposedType.
    // The endianess of the encoding for the unit is affected by the current
    // local endianess.
    bool DecodeAsUnit = false;

    // WidthForComposedType is used to specify the length in bits for a
    // composed type. A composed type refers to reference type, array, set,
    // or map. The specific functionality that WidthForComposedType defines
    // is:
    //  1) If WidthForComposedType is applied on a field, the field will be
    //  decoded until the provided number of bytes has been consumed.
    //  If the field contains types that do not fill up the entire range,
    //  padding is assumed. If Length is present, both need to be satisfied.
    //  2) If WidthForComposedType is applied to a reference type
    //  declaration,it indicates the number of bits this container
    //  will consume as a whole.
   int WidthForComposedType;

} with DeclarationInfo { AspectCompileToAttribute = true, AspectDefaults =
      "BinaryEncodingDefaults" };

aspect BinaryEncodingDefaults
{
    // IntegerBinaryEncodingAttributes
    Endian Endian = Standard.Endian.Little;

    // NumberBinaryEncodingAttributes
    NumberFormat NumberFormat;

    // TextBinaryEncodingAttributes
    TextEncoding TextEncoding = Standard.TextEncoding.UTF16;
    string TextTerminator = "\0";

    // DateTimeBinaryEncodingAttributes
    TimeZone TimeZone = Standard.TimeZone.Universal;
    ulong BaseTime      = 0;
    uint NanoSecondsPerTick = 100;
}

pattern NumberFormat = enum {
    IEEE64
    // TBD
};

pattern TimeZone = enum { Local,Universal};

// Well-known encoding enumerators
pattern Endian = enum { Little, Big };

// Text Binary Encoding Attributes
pattern TextEncoding = enum {
      None, ASCII, UTF7, UTF8, UTF16, UTF32, BigEndianUnicode, Base64, DNS, N
      BT, MBCS };
```

5.16.1.2.2. BinaryDecoder Behavior

Standard library provides a decoder function for binary data that takes a stream, interpreted as binary, and returns an OPN type. The type of the function is such that can be used with a **from** pattern.

```
optional T? BinaryDecoder<T>(stream x);
```

The following is a simple example in which binary encoding aspects guide the codec parsing behavior.

```
protocol P with BinaryEncodingDefaults{Endian = Endian.Big};

message M1
{
      // Will be decoded as big endian.
      int F1;
      // Will be decoded as little endian.
      int F2 with BinaryEncoding{Endian = Endian.Little};
}

endpoint E accepts M1 accepts M2
{
      process this accepts M1{} m
      {
            switch (m.F1)
            {
                  // Value for i decoded as big endian
                  case i:int from BinaryDecoder =>
                  // ...
            }
            switch (m.F2)
            {
                  // Value for i decoded as little endian
                  case i:int from BinaryDecoder =>
                  // ...
       }
      }
}
```

The following is another example that shows how the encoding aspect can help guiding the decoder.

```
type BoundedString
{
    uint Size;
    string Str with BinaryEncoding{Length = Size,
                                   TextEncoding = TextEncoding.ASCII};
}

endpoint E // ...
{
    // ...
    case s:BoundedString from BinaryDecoder => // ...
}
```

In this case the decoder will parse the first field **Size** (using the default behavior for parsing an uint, that is, it will parse 32 bits from the wire, with the default endian. Then it is going to parse the second field, and will use the **Length** attribute to know how many bytes to consume. In this case, it is going to parse as many bytes as specified by **Size**, using ASCII encoding.

Now for an example that shows how the decoder behaves when decoding bit level structures. Consider the following OPN type.

```
type TwoByteStructure
{
      bool F0 with BinaryEncoding{Width = 1};
      bool F1 with BinaryEncoding{Width = 1};
      // ...
      bool F15 with BinaryEncoding{Width = 1};
}
```

This is a structure with 16 Boolean fields. Observe that the **Width** attribute is specifying that each field is represented on the wire with just one bit. Now invoke **BinaryDecoder** to decode a stream and return an instance of **TwoByteStructure**.

```
 binary MyPayload = ...;
 switch (MyPayload)
 {
     case t:TwoByteStructure from BinaryDecoder =>
     // ...
 }
```

The most significant bit of **MyPayload** will be decoded into **F0**, the following one into **F1**, and so on. For example, consider **MyPayload** contains the following bits (expressed in binary format, most significant bit first).

```
1100011100001110
```

The returned instance **t** will have the following values.

F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
true	true	false	false	false	true	true	true	false	false	false	false	true	true	true	false

The following rules apply in general for **BinaryDecoder**:

- The stream is consumed starting from the first byte of the stream.
- The consumed stream is decoded starting from the first field of the returning structure and continuing with the subsequent fields, from top to bottom, following the OPN declaration.
- The endian attribute does not affect how bits are interpreted. Endian attribute only applies when decoding bits into numerical values (for example int or long) and dictates the way a sequence of bits are interpreted, it does not affect the order in which bits are consumed.

Consider that if the same logical field is represented using multiple fields in a message declaration, then **DecodeAsUnit** and **WidthForComposedType** can be used to specify this situation. See the aspect declaration for more details section 5.15.1.2.1.

5.16.1.2.2.1. Dealing with Optional Fields and Branching

The **BinaryDecoder** also takes into consideration presence indicators attached to fields to help decide its behavior under certain parsing situations,for example.

```
type Test
{
	byte Flag;
	optional [|Flag == 1|] int Field;
}
```

When **Field** is being decoded, the decoder will inspect the constraint to determine whether it should parse **Field** or not. In the case that the constraint is not satisfied, the value **nothing** will be assigned to **Field** and the stream cursor will not be moved.

As mentioned before, codecs are designed to handle simple parsing situations. Binary codec does not guarantee that it will successfully parse a stream when aspect information is missing or underspecified,for example.

```
type T
{
	int a;
	(int | long) x;
	int y;
	invariant y > 0;
}
```

In this example, depending on the election between **int** or **long** for field **x**, a different number of bits are going to be consumed from the stream. Therefore, field **y** may have a different value assigned. Observe that the decision could cause the type invariant to hold or fail.

The **BinaryDecoder** will not try all possible parsing combinations in order to guarantee that a value can be successfully created. It will eagerly try constructing a value of type **int** for field **x**, and after succeeding, it will consume the following bits in the stream to construct a value for **y**. At this point, if the type invariant holds, it will return the appropriate value. If it does not hold, it will return the value **nothing** and will not revise its decision about **int** vs. **long**.

It is worth noting than from a protocol specification perspective, the type **T** in the preceding example shows signs of being a poorly specified data format. Usually for real-case protocols the bit format to expect on the wire is guided by previous parsed information, such as flags. From an OPN perspective, this means that probably type **T** is underspecified and, for example, field **a** carries the information about how to parse field **x**.

```
type T
{
      int a;
      ([|a ==1|]int | [|a != 1|] long) x;
      int y;
      invariant y > 0;
}
```

In the preceding example, the **BinaryDecoder** will inspect the discriminators of the type **union** to determine how many bits it should parse.

Note that aspects are not automatically *inherited* in **OR** patterns, so in the following case the field **Afield** won't have any particular aspect attached to it.

```
type T{...} with BinaryEncoding{Width = 1};
type Q{...} with BinaryEncoding{Width = 2};
type R
{
      bool flag;
      ([|flag|] T | [|!flag|] Q) AField;
}
```

In case this is needed, an aspect has to be explicitly attached to the **AField** field. There is however one case where the aspect attached to an **OR** pattern is considered to be attached to individual components of the **OR** pattern. This is when the component of the **OR** pattern does not have an explicit declaration. This applies to simple types, for example in the following case.

```
type T
{
      bool flag;
```

```
    ([|flag|] float | [|!flag|] int) AField with BinaryEncoding{Width = 1};
}
```

The attribute **Width** will be considered as attached to both cases of the **OR** branch. The reason is that it is not possible to attach an aspect to entities that do not have an associated declaration.

5.16.1.3. Xml Encoding

The xml encoding aspect aggregates all attributes related to xml value encoding.

5.16.1.3.1. XML Encoding Declaration

```
module Standard;

aspect XmlEncoding : XmlEncodingDefaults
{
      // XML name encoding attribute to be attached to types and fields.
      string Name;  // XML serialization/deserialization name.

      // Indicates whether the type should be an element or be part of an
      // enclosing element to be attached to types.
      bool AnonymousType = false;

      // XML array item encoding attributes.
      string ArrayItemName;  // The serialization name for array item.

      // XML namespace for array item.
      // The serialization namespace, for array item WCF will use
      // the following by default
      // "http://schemas.microsoft.com/2003/10/Serialization/Arrays"
      string ArrayItemNamespace;

} with DeclarationInfo {AspectCompileToAttribute = true, AspectDefaults =
        "XmlEncodingDefaults"};

aspect XmlEncodingDefaults
{
      // XML namespace encoding attribute to be attached to module, protocol,
      // type, or pattern field declarations.
      string Namespace;  // TargetNamespace

      // XML kind encoding attribute to be attached to fields.
      XmlKind Kind = XmlKind.Element;  // The kind of the XML element.

      // The order indicator
      XmlOrderIndicator Order = XmlOrderIndicator.Sequence;

      // Specifies whether the field is Wildcard Schema Components, which is
      // <any> or <anyAttribute>. The usage of Wildcard Schema Components is
      // subject to the same ambiguity constraints as other content model
      // particles: If an instance element could match either an explicit
      // particle  and  a wildcard, or one of two wildcards within the
      // content model of a type that model is in error. For details see
      // Schema Component Constraint.
      bool IsAny = false;
}

aspect XmlPrimitiveValueEncoding : XmlPrimitiveValueEncodingDefaults
{
      any this;
} with DeclarationInfo { AspectCompileToAttribute = true, AspectDefaults =
      "XmlPrimitiveValueEncodingDefaults" };

aspect XmlPrimitiveValueEncodingDefaults
{
      // XML string encoding attribute to be attached to string
      // structured patterns. Mapped directly to the XSD Length
      // restriction on string value.
      uint Length;  // Length of the string.
      // Mapped directly to the XSD MaxLength restriction on string value.
      uint MaxLength;  // Maximum allowable length for the string.
      // Mapped directly to the XSD MinLength restriction on string value.
      uint MinLength;  // Minimum allowable length for the string.
      //Mapped directly to the XSD Pattern restriction.
      string Pattern;   // Regular expression restriction on the string.

      // XML numeric encoding attributes
      // Mapped directly to the XSD MinInclusive restriction.
      AnyNumber MinInclusive; // Can be realized as >= in OPN.
      // Mapped directly to the XSD MaxInclusive restriction.
      AnyNumber MaxInclusive; // Can be realized as <= in OPN.
      // Mapped directly to the XSD MinExclusive restriction.
      AnyNumber MinExclusive;  // Can be realized as > in OPN.
      // Mapped directly to the XSD MaxExclusive restriction.
      AnyNumber MaxExclusive; // Can be realized as < in OPN.
      // Mapped directly tothe XSD FractionalDigits restriction on
      // fractional value. Refers to the number of fraction digits
      // the value is allowed to have.
      uint FractionalDigits;
      // Mapped directly  tothe XSD TotalDigits restriction on fractional
      // value. Refers to the total number of digits allowed in the value.
      uint TotalDigits;

      //XML char encoding attribute
      // If true, char will encode as number (ushort).
      // If false, value is ASCII character and encoce as is.
      bool EncodeCharAsNumber = true
}

pattern AnyNumber = byte | sbyte | ushort | short | uint | int | ulong | long
       | float | double | decimal ;

// Specifies the kind of an XML element.
pattern XmlKind = enum {
    // Attribute
    Attribute,
    // A CDATA section <![CDATA[{EscapedText}]]>.
    CData,
    // A comment <!-- {Comment} -->.
    Comment,
    // An XML declaration node <?xml version='1.0'...?>.
    Declaration,
    // A root of the document tree
    Document,
    // An element <E>{Content}</E>.
    Element,
    // A Namespace.
    Namespace,
    // The text content of a XML element.
    Text,
};

// Order indicators are used to define the order of the elements.
pattern XmlOrderIndicator = enum {
      // Specifies that the child elements can appear in any order,
      // and that each child element must occur only once.
      All,
      // Specifies that either one child element or another can occur.
      // Didn't implement since it can be done by using the OR pattern.
      Choice,
      // Specifies that the child elements must appear in a specific order.
      Sequence,
}

// Namespace indicators are used to define the namespace of any
// element/attribute.
pattern XmlNamespaceIndicator = enum string {
      // Any namespace
      Any = "##any",
      // Any namespace that is not the target namespace of the parent element
      Other = "##other",
      // Not qualified with a namespace.
      Local = "##local",
      // target namespace of the parent element.
      TargetNamespace = "##targetNamespace",
}


// Primitive types for XML schema definition.
pattern XsdPrimitiveTypes = AnyNumber | bool | string | binary | DateTime |
      TimeSpan;
```

5.16.1.3.2. XMLDecoder Behavior

Standard library provides a decoder function for XML data that takes a **stream**, interpreted as XML type, and returns an OPN type.

```
optional T XmlDecoder<T>(stream s);
optional T XmlDecoder<T>(xml x);
```

These functions inspect the **XmlEncoding** aspect and use the information specified there to guide the decoder parsing.

Following are some examples of use. An OPN reference type can be mapped to an XML type, usually by providing the XML type name and an order indicator to specify the order of its elements from an XML perspective.

```
type SomeType
{
     SomeOtherType field1;
     int field2;
}
with XmlEncoding {Name = "MyType", Order = XmlOrderIndicator.Sequence}
```

In the following example the **AnonymousType** attribute is set to true to specify that the structure for **SomeOtherType** should be a complex type declared within the enclosing element, rather than an element by itself.

```
type SomeOtherType
{
     // ...
}
with XmlEncoding {AnonymousType = true};
```

The preceding declaration is equivalent to the following XML schema.

```
  <xs:complexType name="MyType">
    <xs:sequence>
      <xs:element name="field1">
        <xs:complexType>
          <!-- ... -->
        </xs:complexType>
      </xs:element>
      <xs:element name="field2" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
```

The aspect also provides attributes to specify string properties, which can be attached to patterns with the string type structure.

```
type Type
{
     int IntAttribute with XmlEncoding { Kind= XmlNodeKind.Attribute };
     int IntElement;
}
with XmlEncoding {OrderIndicator = XmlOrderIndicator.Sequence}
```

This is equivalent to the following xml schema definition.

```
<xs:complexType name="Type">
    <xs:sequence>
      <xs:element name="IntElement" type="xs:int" />
    </xs:sequence>
    <xs:attribute name="IntAttribute" type="xs:int" />
</xs:complexType>
```

Xml encoding defines a handy way to deal with arrays. The simplest case is when all the elements of an array have the same type.

```
<Content>
<Array xmlns:a="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
<a:Item>1</a:Item>
      <a:Item>89.32</a:Item>
      <a:Item>13</a:Item>
      <a:Item>33.456</a:Item>
      <a:Item>21</a:Item>
      <a:Item>0.123456789</a:Item>
</Array>
</Content>
```

This type of structure can be represented with the following OPN code.

```
type ArrayOfPrimitive
{
    array<int> Array with XmlEncoding{
    ArrayItemName = "Item",
    ArrayItemNamespace = "http://schemas.microsoft.com/2003/10/Serialization
                        /Arrays"};
}
```

This is not restricted to primitive data types, such as integers in the preceding example, but can also be used with arbitrary OPN patterns.

Observe that attributes **ArrayItemName** and **ArrayItemNamespace** are propagated to the collection elements of the fields they are attached to.

5.16.1.3.2.1. XML Extensibility

OPN can also be used to specify extensible types for XML based protocols. XML can be used to extend types using **<xs:any>** and **<xs:anyAttribute>**. Please refer to W3C XML schema reference for more details about these tags.

To specify a field in OPN as XML extensible element or attribute, the **IsAny** attribute must be set to **true**. The **XmlEncoding.Kind** XML element indicates if the field represents **<xs:any>** or **<xs:anyAttribute>.**

For example, observe the following XML schema.

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:any minOccurs="0" namespace ="##local http://url" />
    </xs:sequence>
    </xs:anyAttribute>
  </xs:complexType>
  </xs:element>
```

A possible representation of the element Person in OPN would be the following.

```
type Person
{
      string firstname;
      string lastname;
      optional xml anyElement with XmlEncoding{ IsAny = true,
         Namespace = "##local http://url"};
      optional array<xml> anyAttribute with XmlEncoding{ IsAny = true,
         Kind = XmlKind.Attribute};
}
```

When **IsAny** is set to **true**, the namespace attribute expects a list of namespace URIs. In this list, two values have a specific meaning: **##targetNamespace** stands for the target namespace that the **XmlDecoder** will pick up from the container type, and **##local** stands for local elements (without namespaces). These values can be mixed in the list with regular namespace URIs. The whole list may also be replaced by two other special values: **##any** stands for any namespace at all that is the default value for the namespace attribute, and **##other** stands for any namespace other than the target namespace. When **##other** is used, all the namespaces except the target namespace are allowed, and elements without namespaces are forbidden.

It is worth noting that if a field is specified as any element (using **IsAny = true**), and the element kind is **XmlKind.Element**, the container type must have the order indicator set to **XmlOrderIndicator.Sequence**, which is the default value.

There is no requirement for the location of any elementthat means it can be at any position among other fields. It is also possible to have multiple any elements in one type, as long as the declaration satisfies the ambiguity constraints of XML schema content model particles. For details about the ambiguity constrains, please see Schema Component Constraint: Unique Particle Attribution.

5.16.1.4. StreamEncoding

This aspect is to be attached to syntax constructs in order to guide the way a **stream** is parsed.

```
module Standard;

// This aspect should be only attached to syntax construct.
// It will be ignored if it is attached to other construct.
aspect StreamEncoding : StreamEncodingDefaults
{
} with DeclarationInfo { AspectCompileToAttribute = true, AspectDefaults =
        "StreamEncodingDefaults" };

// It will affect the entire scope of the syntax construct that
// it is attached to.
aspect StreamEncodingDefaults
{
    // The encoding used to retrieve string from the stream.
    // All the encodings are little endian, except
    // TextEncoding.BigEndianUnicode.
    TextEncoding Encoding;
    // The endianess used to retrieve multiple bytes type from the
    // stream, that isInt16. Set Endian will not affect TextEncoding,
    // since we only support Big Endian for Unicode.
    Endian Endian = Endian.Little;
}
```

Standard Library exposes a decoder function that can parse a stream and return string values.

```
optional T TextDecoder<T>(stream s);
```

This function inspects the **`StreamEncoding`** aspect to determine the encoding and endian to be used during decoding.

5.16.2. Aspects for Documentation

This module groups definitions that are related to documentation aspects.

5.16.2.1. Documentation

```
module Standard;

aspect Documentation
{
    // Documentation
    // Whether the declaration should be ignored in documentation or not.
    // Default value: false.
    optional bool Ignore;


    // Text that specifies the description for the item.
    optional string Description;

    // Protocol that uses Technical Document (TD) templates.
    // Determines which TD template should be used for document
    // generation, for example SOAP.
    optional string ProtocolType;

    // Long name of the protocol, for example,
    // Access Services Data Server Protocol.
    optional string ProtocolName;

    // Short name of the protocol, used mostly for inventory
    // purposes, for example ADS.
    optional string ShortName;

    // Name of the doucment,similar to ShortName.
    optional string DocumentName;

    // The set of glossary terms specific to this protocol.
    optional array<GlossaryTerm> GlossaryTerms;

    optional string Prerequisites;
    optional string Preconditions;
    optional string Applicability;

    // To be attached only to a using clause. If the value is Include
    // the TD must include all the items from the referenced namespace
    // otherwise the TD should only have a reference to the protocol
    // corresponding to the namespace. If the no UsingKind is attached
    // to a using clause then the default value for module would be
    // Include and for protocol would be Reference.
    optional ReferenceKind UsingKind;

   // To be attached only to a using clause when the UsingKind
   // is set to Reference. This is the link/inventory ShortName
   // of the reference protocol TD.
    optional string ReferencedProtocolShortName;
}

pattern ReferenceKind = enum as string
{
      Include = "Include",
      Reference = "Reference"
};

type GlossaryTerm
{
    string Name;
    string Description;
}
```

5.16.2.2. EmbeddedIn

This aspect is used to document and embed the relationship between two protocols. The **`ParentProtocolName`** field specifies the entity that hosts the embedded protocol.

```
module Standard;

aspect EmbeddedIn
{
      optional string ParentProtocolName
}
```

5.16.3. Visualization

This aspect provides information to be consumed by the user interface.

```
module Standard;

aspect Visualization
{
      // Defines an alias name for an entity
      optional string AliasName;
}
```

The **`Visualization`** aspect is attached to a message field or protocol to define an alternative name to be displayed in the UI. The intended use is to provide a way to group a set of entities under the same name, so that the UI filtering language can refer to these groups within a filter.

5.16.4. DisplayInfo

The **`DisplayInfo`** aspect provides a customized way to obtain a textual representation of an object. This is usually consumed by the UI.

```
module Standard;

aspect DisplayInfo
{
	// Defines the name of a function that is responsible for
	// providing a textual representation of the entity that
	// this aspect is attached to.
	optional string ToText;
}
```

The following example shows the use of the **`DisplayInfo`** aspect.

```
message IPv4Datagram
{
   // ...
   IPv4Address SourceAddress with DisplayInfo{ToText = AddressDisplay};
   IPv4Address DestinationAddress with DisplayInfo{ToText = AddressDisplay};
}

string AddressDisplay(any data)
{
    IPv4Address address = data as IPv4Address;
    return (address.Octets[0] as string) + "." +
           (address.Octets[1] as string) + "." +
           (address.Octets[2] as string) + "." +
           (address.Octets[3] as string);
}
```

5.16.5. StopProcessing

When an endpoint has the **`StopProcessing`** aspect, any messages dispatched to that endpoint will be dropped by runtime. The net effect is that runtime will stop processing those messages.

```
aspect StopProcessing {};
```

The following is an example of a **`StopProcessing`** aspect in use.

```
endpoint NodeSource[MacAddress Address] issues Frame with StopProcessing{};
```

This aspect can be used in the context of message echoing. Message echoing describes the case when actors dispatch a message from message sources (known as chokepoints) to both source and destination endpoints. The message is dispatched with the **accept** direction to the destination endpoint and with the **issues** direction to the source endpoint. Message Echoing enables sequence validation to run on both source and destination sides.

When an endpoint has enough information to determine that certain types of messages do not correspond to the side that endpoint is modeling, it can use the **StopProcessing** aspect to indicate that. This improves the performance and avoids duplicated processing.

For the following case an echoed message bubbles up to the top level endpoint on both sides, the function **MarkEchoMessage** prevents the UI from showing the same message twice.

```
void MarkEchoMessage(any message m) with DeclarationInfo { Handcoded = true }
      ;
```

The following example uses **MarkEchoMessage**.

```
autostart actor CapFileToEthernet(CapFile.CapFileEndpoint cap)
{
    process cap accepts cf : CapFile.CapFrame{ MediaType is MediaTypes.Ethern
      et }
    {
        switch(cf.Payload)
        {
            case f:Ethernet.Frame from BinaryDecoder<Ethernet.Frame> =>
                        MarkEchoMessage(f);
                  var en = endpoint Ethernet.Node[f.DestinationAddress];
                  dispatch en accepts f;

            var enIssue = endpoint Ethernet.NodeSource[f.SourceAddress];
                        dispatch enIssue issues f;
            default =>
                throw "error";
        }
    }
}
```

5.16.6. OPNAuthoring

This aspect is intended to document information related to the authoring process itself, such as version, references, and so on. It complements **Documentation** aspect. This aspect is not currently consumed by any tool, but defines a standard for future uses.

```
module Standard;

aspect OPNAuthoring
```

```
{
    optional string Copyright;
    optional array<Reference> References;
    optional array<Revision> RevisionSummary;
}

aspect Reference
{
    optional string Name;
    optional string Version;
    optional string Link;
    optional string Date;
    optional ProgramName ProgramName;
}

aspect Revision
{
    optional RevisionClass Class;
    optional string Version;
    optional string Date;
}

pattern RevisionClass = enum {Editorial, Major, Minor};
pattern ProgramName = enum {MCPP, WSPP};


// This aspect is to be attached to a module or protocol,for example.
protocol SampleProtocol with Documentation
{
    ProtocolName = "Sample Protocol",
    ShortName = "SAMPLE",
    Description = "This is a very simple sample protocol"
},
OPNAuthoring
{
   Copyright = "(c) 2011 Microsoft Corporation",
   References =
  [
       new Reference{Name = "SOAP", Version = "1.2",
        Link = "http://www.w3.org/TR/soap", ProgramName = ProgramName.MCPP},
       new Reference{Name = "HTTP", Version = "1.1",
        Link = "http://www.w3.org/"},
       new Reference{Name = "MS-ISTM", Version = "0.3.1"}
  ]
   ,
   RevisionSummary =
      [
         new Revision{Class=RevisionClass.Editorial,
            Version="1.0.1", Date="01/03/2011"},
         new Revision{Class=RevisionClass.Major,
            Version="2.0.0", Date="01/10/2011"},
         new Revision{Class=RevisionClass.Minor,
            Version="2.1.0", Date="02/13/2011"},
         new Revision{Class=RevisionClass.Editorial,
            Version="2.1.1", Date="02/23/2011"},
         new Revision{Class=RevisionClass.Major,
            Version="3.0.0", Date="03/04/2011"}
      ]
};
```

The revisions must be placed in temporal order, with the last revision at the end of the list. Each element has type **`Revision`**, whose fields are the following:

- **Class** – this is an enumeration defined with the values {Major, Minor, Editorial}.
- **Version** – a String with the protocol version, with the format "x.y.z", where Major, Minor and Editorial revisions are associated with the numbers x, y and z respectively. Each revision increases the associated numbers by one and resets the less significant ones.
- **Date** – the revision date, in the format "MM/DD/YYYY".

The general guideline for using this aspect is the following:

- For a Microsoft protocol, the standard information in References should include: *Name* (for example, Name = "MS-DFSC"), *Version* and *ProgramName*. If there is no version available, *Date* should be used instead.
- For RFC, the standard information in References should include the *Name* (for example, Name = "RFC 4795").
- For Third-party, the standard information in References should include *Name* (for example, Name = "WS-Discovery"), *Link* and *Version*.

5.16.7. UsageInfo

This aspect can be attached to fields, properties or annotations, and it represents how those entities are used. This aspect is consumed by runtime and guides optimization decisions.

```
// Represents a set of properties about usage of the aspect carrier.
aspect UsageInfo
{
    // Currently, the aspect can be attached only to fields, properties and
    // annotations of primitive values: byte, sbyte, short, ushort, int,
    // uint, long, ulong, float, double, decimal, bool, char, guid, string,
    // binary and their nullable counterparts.
    any this;

    // Indicates whether the carrier of the aspect is frequently used in
    // UI or other clients. Such knowledge can help perform different
    // kinds of optimizations. The default value is false.
    bool FrequentlyUsed;

    // An optional field specifying the name of a logical group the aspect
    // carrier belongs to.
    optional string LogicalGroup where value == nothing ||
            (value != null && value.Count != 0);

    // An optional field specifying a maximum number of elements held by
    // the aspect carrier.
    optional int MaxLength where value == nothing || value > 0;

    // Currently MaxLength can be specified only for string or binary values.
    invariant MaxLength == nothing || this is binary || this is string;
}
```

6. Implementation Notes

The following features are *not expected* to be implemented for Iteration 7:

- XML literals **`(xml book = <Book owner="me"></Book>;)`** and JSON literals.
- XML pattern **`(switch (x){case ..<Address> ... </Address> a: return a;).`**
- The ability to store endpoints in messages.
- Partial ordering using **`precedes`** and **`follows.`** Modifiers cannot be declared by an endpoint, but actors can reference endpoints.
- Support for collection patterns is limited.
- Naming of invariants, rules, and other elements.
- **`Do`**/**`while`** statement.
- Actor **`start`**/**`stop`**.
- **`Expression++`** and **`Expresion--`** are not implemented. The statement version of both expressions does work.
- Operators **`==>`** and **`<=>`** are not implemented.
- Modalities.
- **`GetClientEndpoint`** and **`GetServerEndpoint`**.
- **`Min`**, **`Max`** and **`Abs`** are not implemented for numeric types.
- The ability to declare a **`destructor`** for implicit actors.
- Behavioral scenarios do not support arbitrary patterns, just reference patterns.
- The **`release`** statement.

7. References

"Formatting Types (string)," Microsoft Corporation, http://msdn.microsoft.com/en-us/library/fbxft59x(v=vs.71)

"Namespaces in XML 1.0 (Third Edition), " by the XML Core Working Group, http://www.w3.org/TR/REC-xml-names/

"Schema Component Constraint: Unique Particle Attribution,", by the W3C XML Schema Working Group, XML Schema Part 1: Structures Second Edition, 3.8.6 Constraints on Model Group Schema Components, http://www.w3.org/TR/xmlschema-1/#cos-nonambig

"W3Schools Online Web Tutorials," W3Schools, http://www.w3schools.com/

"W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures, " by the W3C XML Schema Working Group , http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/

"XML schema", by the W3C XML Schema Working Group, http://www.w3.org/XML/Schema

"XML Tutorial," W3Schools, http://www.w3schools.com/xml/default.asp

"XPath Syntax," W3Schools, http://www.w3schools.com/xpath/xpath_syntax.asp

"XPath Tutorial," W3Schools, http://www.w3schools.com/xpath/default.asp