AMOSSYS

Netzob

Language Specifications

Ref : Netzob API

Oct 30, 2017

Table of contents

1 Format Message Modeling **1**

1.1 Format Message Modeling Concepts 1

1.1.1 Definitions: Vocabulary, Symbol, Field, Variable 1

1.1.2 Abstraction and Specialization of Symbols 2

1.2 Modeling Data Types 4

1.2.1 Data Types API 4

1.2.2 Available Data Types 7

1.2.2.1 Integer Type 7

1.2.2.2 BLOB / Raw Type 11

1.2.2.3 HexaString Type 13

1.2.2.4 String Type 14

1.2.2.5 BitArray Type 16

1.2.2.6 IPv4 Type 18

1.2.2.7 Timestamp Type 19

1.3 Modeling Data Variables 22

1.4 Modeling Fields 24

1.5 Modeling Fields with Complex Structures 28

1.5.1 Aggregate Domain 28

1.5.2 Alternate Domain 33

1.5.3 Repeat Domain 34

1.6 Modeling Field Relationships 38

1.6.1 Value Relationships 38

1.6.2 Size Relationships 40

1.6.3 Padding Relationships 43

1.6.4 Checksum Relationships 45

1.6.5 Hash Relationships 46

1.6.6 HMAC Relationships 47

1.7 Modeling Symbols 48

1.8 Persistence during Specialization and Abstraction of Symbols 56

2 Sending and Receiving Messages **63**

2.1 Underlying Concepts 63

2.2 Communication Channel API 63

2.3 Available Communication Channels 65

2.3.1 RawEthernetChannel channel 66

2.3.2 RawIPChannel channel 66

2.3.3 IPChannel channel 67

2.3.4 UDPClient channel 68

2.3.5 TCPClient channel 69

2.3.6 UDPServer channel 70

2.3.7 TCPServer channel 71

2.3.8 SSLClient channel 72

©2017 AMOSSYS

2.3.9 DebugChannel channel 73
2.4 Abstraction Layer 73
2.5 Relationships between Messages and the Environment 78

1 Format Message Modeling

The Netzob Description Language (ZDL) is the API exposed by the Netzob library to model data structures employed in communication protocols. This textual language has been designed in order to be easily understandable by a human. It enables the user to describe a protocol through dedicated *.zdl* files, which are independent from the API and core of the library. The ZDL language has been designed with attention to its expressiveness. In this chapter, firstly, the main concepts of the ZDL language are presented, then its expressiveness in terms of data types, constraints and relationships are explained.

1.1 Format Message Modeling Concepts

This chapter covers the following requirements: 20, 21, 22.

1.1.1 Definitions: Vocabulary, Symbol, Field, Variable

The **vocabulary** of a protocol defines the set of valid messages and their formats. In the library, the vocabulary of a protocol consists of a list of symbols. A **symbol** represents all the messages that share a similar objective from a protocol perspective. For example, the HTTP_GET symbol would describe any HTTP request with the method GET being set. A symbol can be specialized into a context-valid message and a message can be abstracted into a symbol.

A **field** describes a chunk of the symbol and is defined by a **definition domain**, representing the set of values the field handles. To support complex domains, a definition domain is represented by a tree where each vertex is a **Variable**. There are two kinds of variables:

- **Leaf Variables** which accept no children.
- **Node Variables** which accept one or more children variables.

Leaf Variables are the simplest variables that contain user content. There are three main leaf variables: Data, Size and Value variables.

A **Data Variable** describes data whose value is of a given **type**. Various types are provided with the library, such as String, Integer, Raw and BitArray.

Along with Data Variables, the definition domain of a field can embed the definition of relationships. Two kinds of relationships are supported:

- **Intra-symbol relationships**, which denote relationships between the size or the value of a variable and another field in the same symbol.
- **Inter-symbol relationships**, which denote relationships with a field of another symbol.

The main relationships supported in the library are:

- **Size relationships**, which describe data whose value is the size of another field.
- **Value relationships**, which is very similar to the size relationships except that the relationship applies on the value of the targeted field.

As stated before, Leaf Variables can be combined into a tree model to produce much more complex definition domains. To achieve this, Node Variables can be used to construct complex definition domains, such as:

- **Aggregate node variable**, which can be used to model a concatenation of variables.
- **Alternate node variable**, which can be used to model an alternative of multiple variables.
- **Repeat node variable**, which can be used to model a repetition of a variable.

As an illustration of these concepts, the following figure presents the definition of a Symbol structured with three Fields. The first field contains an alternative between a String Data with a constant string and an Integer Data with a constant value. The second field is a String Data with a variable length string. The third field depicts an Integer whose value is the size of the second string.

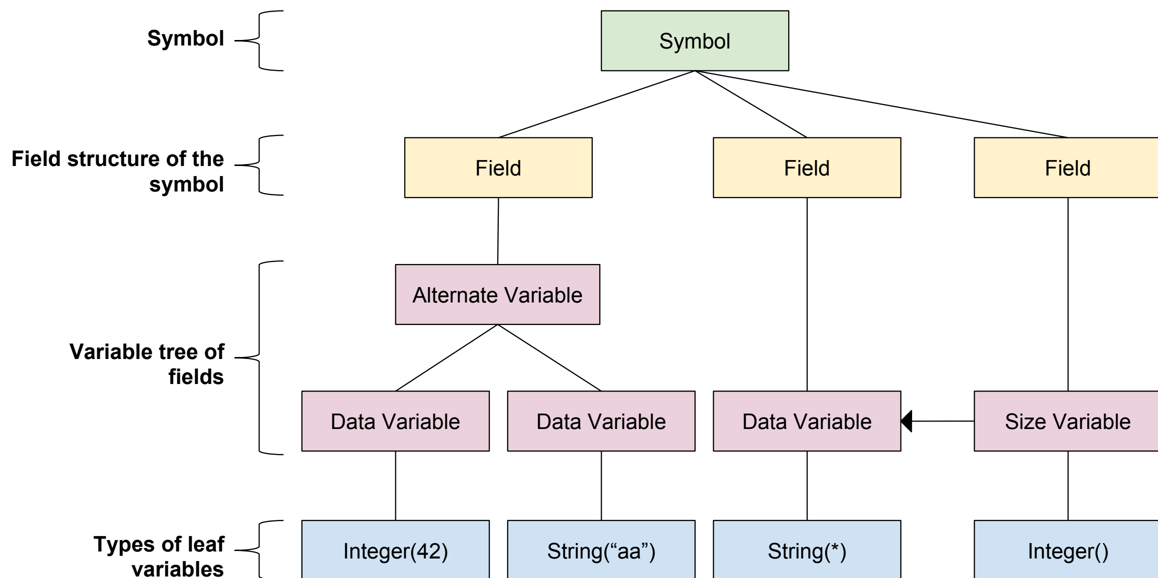

Fig. 1.1: Example of Symbol definition and relationships with Field and Variable objects.

1.1.2 Abstraction and Specialization of Symbols

The use of a symbolic model is required to represent the vocabulary of a protocol in a compact way. However, as the objective of this platform is to analyze the robustness of a target implementation, this implies that the testing tool should be able to exchange messages with this target. We therefore need to abstract received messages into symbols that can be used by the protocol model. Conversely, we also need to specialize symbols produced by the protocol model into valid messages. To achieve this, we use an **abstraction** method (*ABS*) and a **specialization** (*SPE*) method.

As illustrated in the following figure, these methods play the role of an interface between the symbolic protocol model and a communication channel on which concrete messages transit.

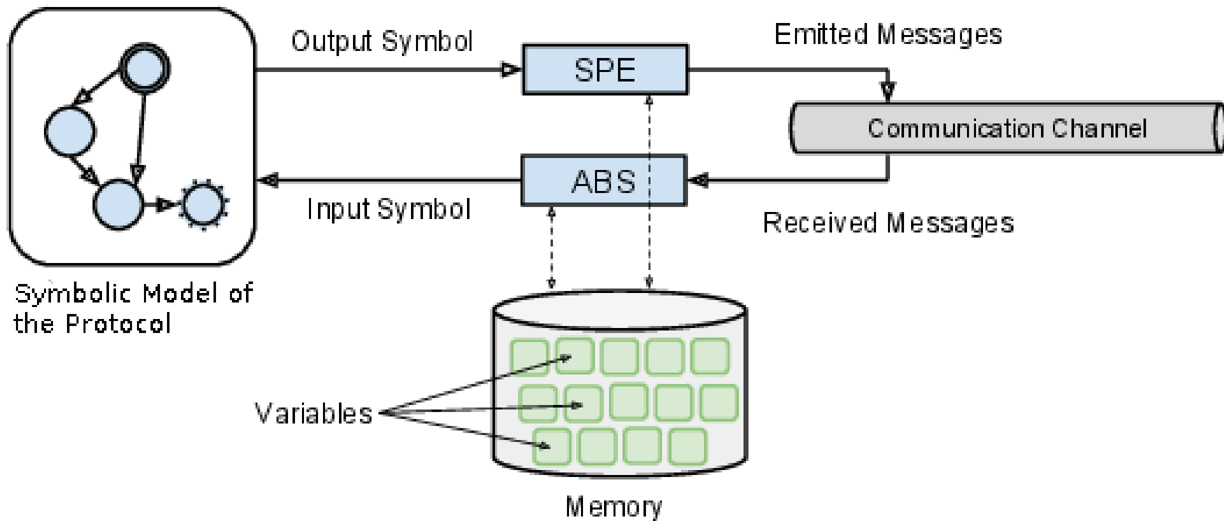

Fig. 1.2: Abstraction and Specialization methods are interfaces between the protocol symbols and the wire messages.

To compute or verify the constraints and relationships that participate in the definition of the fields, the library relies on a `Memory`. This memory stores the value of previously captured or emitted fields. More precisely, the memory contains all the variables that are needed according to the field definition during the abstraction and specialization processes.

1.2 Modeling Data Types

The library enables the modeling of the following data types:

- **Integer**: The Integer type is a wrapper for the Python integer object with the capability to express more constraints regarding the sign, endianness and unit size.
- **HexaString**: The type HexaString enables to describe a sequence of bytes of arbitrary sizes, with a hexastring notation (e.g. `aabbcc`).
- **BLOB / Raw**: The type Raw enables to describe a sequence of bytes of arbitrary sizes, with a raw notation (e.g. `\xaa\xbb\xcc`).
- **String**: The type String enables to describe a field that contains sequence of String characters.
- **BitArray**: The type BitArray enables to describe a field that contains a sequence of bits of arbitrary sizes.
- **IPv4**: The type IPv4 enables to encode a raw python in an IPv4 representation, and conversely to decode an IPv4 into a raw object.
- **Timestamp**: The type Timestamp enables to define dates in a specific format (such as Windows, Unix or MacOS X formats).

1.2.1 Data Types API

Each data type provides the following API:

class AbstractType(*typeName*, *value*, *size=(None, None)*, *unitSize=None*, *endianness=None*, *sign=None*)

AbstractType is the abstract class of all the classes that represents netzob types.

A type defines a definition domain as a unique value or specified with specific rules. For instance, an integer under a specific interval, a string with a number of chars and an IPv4 of a specific netmask.

The constructor for an AbstractType expects some parameters:

Parameters

- **typeName** – The name of the type (we highly recommand the use of __class__.__name__).
- **value** – The current value of the type instance.
- **size** – The size in bits that this value takes.
- **unitSize** – The unitsize of the current value. Values must be one of UnitSize.SIZE_*. If None, the value is the default one.

Note: `value` and `size` attributes are mutually exclusive. Setting both values raises an `Exception`.

The following unit sizes are available:

- UnitSize.SIZE_1
- UnitSize.SIZE_4
- UnitSize.SIZE_8 (default value)
- UnitSize.SIZE_16
- UnitSize.SIZE_24
- UnitSize.SIZE_32
- UnitSize.SIZE_64

Parameters

- **endianness** (*Endianness*, optional) – The endianness of the current value. Values must be Endianness.BIG or Endianness.LITTLE. If None, the value is the default one.

  The following endianness are available:

  – Endianness.BIG (default value)

  – Endianness.LITTLE

- **sign** (*Sign*, optional) – The sign of the current value. Values must be Sign.SIGNED or Sign.UNSIGNED. If None, the value is the default one.

  The following signs are available:

  – Sign.SIGNED (default value)

  – Sign.UNSIGNED

Internal representation of Type objects

Regarding the internal representation of variables, the Python module `bitarray` is used, thus allowing to specify objects at the bit granularity. As an example, the following code show how to access the internal representation of the value of an Integer object:

```
>>> from netzob.all import *
>>> i = Integer(20)
>>> print(i)
Integer=20 ((None, None))
>>> i.value
bitarray('00010100')
```

 ©2017 AMOSSYS

convert (*typeClass*, *dst_unitSize=None*, *dst_endianness=None*, *dst_sign=None*)
Convert the current data type in a destination type specified in parameter.

Parameters

- **typeClass** (*AbstractType*, required) – The Netzob type class to which the current data must be converted.
- **dst_unitSize** (*UnitSize*, optional) – The unitsize of the destination value. Values must be one of UnitSize.SIZE_*. If None, the value is the default one (UnitSize.SIZE_8).
- **dst_endianness** (*Endianness*, optional) – The endianness of the destination value. Values must be Endianness.BIG or Endianness.LITTLE. If None, the value is the default one (Endianness.BIG).
- **dst_sign** (*Sign*, optional) – The sign of the destination. Values must be Sign.SIGNED or Sign.UNSIGNED. If None, the value is the default one (Sign.SIGNED).

Returns The converted current value in the specified data type.

Return type `AbstractType`

generate (*generationStrategy=None*)
Generates a random data that respects the current data type.

Returns The value produced.

Return type `bitarray`

```
>>> from netzob.all import *
>>> a = String(nbChars=20)
>>> l = a.generate()
>>> len(l)
160
>>> a = HexaString(nbBytes=20)
>>> l = a.generate()
>>> len(l)
160
>>> a = HexaString(b"aabbccdd")
>>> a.generate()
bitarray('10101010101110111100110011011101')
```

Some data types can have specific attributes regarding their endianness, sign and unit size. Values supported for those attributes are available through Python enumerations:

class Endianness
Enum class used to specify the endianness of a type.

BIG = 'big'
Endianness.BIG can be used to specify the endianness of a type.

LITTLE = 'little'
Endianness.LITTLE can be used to specify the endianness of a type.

class Sign

Enum class used to specify the sign of a type.

SIGNED = 'signed'

Sign.SIGNED can be used to specify the sign of a type.

UNSIGNED = 'unsigned'

Sign.UNISGNED can be used to specify the sign of a type.

class UnitSize

Enum class used to specify the unit size of a type (i.e. the space in bits that a unitary element takes).

SIZE_1 = 1

UnitSize.SIZE_1 can be used to specify the unit size of a type.

SIZE_4 = 4

UnitSize.SIZE_4 can be used to specify the unit size of a type.

SIZE_8 = 8

UnitSize.SIZE_8 can be used to specify the unit size of a type.

SIZE_16 = 16

UnitSize.SIZE_16 can be used to specify the unit size of a type.

SIZE_24 = 24

UnitSize.SIZE_24 can be used to specify the unit size of a type.

SIZE_32 = 32

UnitSize.SIZE_32 can be used to specify the unit size of a type.

SIZE_64 = 64

UnitSize.SIZE_64 can be used to specify the unit size of a type.

1.2.2 Available Data Types

Supported data types are described in details in this chapter.

1.2.2.1 Integer Type

This chapter covers the following requirements: 23, 24.

In the API, the definition of an integer type is done through the Integer class.

class Integer(*value=None, interval=None, unitSize=UnitSize.SIZE_8, endianness=Endianness.BIG, sign=Sign.SIGNED*)

The Integer class enables to represent an integer, with the capability to express constraints regarding the sign, the endianness and the unit size.

The Integer constructor expects some parameters:

Parameters

- **value** – The current value of the type instance.
- **interval** – The interval of permitted values for the Integer. This information is used to compute the size of the Integer.
- **unitSize** – The unitsize of the current value. Values must be one of UnitSize.SIZE_* (see below for supported unit sizes). The default value is UnitSize.SIZE_8.

Note: `value` and `interval` attributes are mutually exclusive. Setting both values raises an `Exception`.

The following unit sizes are available:

- UnitSize.SIZE_1
- UnitSize.SIZE_4
- UnitSize.SIZE_8 (default unit size)
- UnitSize.SIZE_16
- UnitSize.SIZE_24
- UnitSize.SIZE_32
- UnitSize.SIZE_64

Parameters

- **endianness** (*Endianness*, optional) – The endianness of the current value. Values must be Endianness.BIG or Endianness.LITTLE. The default value is Endianness.BIG.

  The following endianness are available:

  – Endianness.BIG (default endianness)

  – Endianness.LITTLE

- **sign** (*Sign*, optional) – The sign of the current value. Values must be Sign.SIGNED or Sign.UNSIGNED. The default value is Sign.SIGNED.

  The following signs are available:

  – Sign.SIGNED (default sign)

  – Sign.UNSIGNED

The Integer class provides the following public variables:

Variables

- **typeName** (str) – The name of the implemented data type.
- **value** (bitarray) – The current value of the instance. This value is represented under the bitarray format.
- **size** (a tuple (int, int) or int) – The size of the expected data type defined by a tuple (min integer, max integer). Instead of a tuple, an integer can be used to represent both min and max value.
- **unitSize** (str) – The unitSize of the current value.
- **endianness** (str) – The endianness of the current value.
- **sign** (str) – The sign of the current value.

Examples of Integer object instantiations

The following example shows how to define an integer encoded in sequences of 8 bits and with a default value of 12 (thus producing `\x0c`):

```
>>> from netzob.all import *
>>> i = Integer(value=12, unitSize=UnitSize.SIZE_8)
>>> i.generate().tobytes()
b'\x0c'
```

The following example shows how to define an integer encoded in sequences of 32 bits and with a default value of 12 (thus producing `\x00\x00\x00\x0c`):

```
>>> from netzob.all import *
>>> i = Integer(value=12, unitSize=UnitSize.SIZE_32)
>>> i.generate().tobytes()
b'\x00\x00\x00\x0c'
```

The following example shows how to define an integer encoded in sequences of 32 bits in little endian with a default value of 12 (thus producing `\x0c\x00\x00\x00`):

```
>>> from netzob.all import *
>>> i = Integer(value=12, unitSize=UnitSize.SIZE_32, endianness=Endianness.LITTLE)
>>> i.generate().tobytes()
b'\x0c\x00\x00\x00'
```

The following example shows how to define a signed integer encoded in sequences of 16 bits with a default value of -12 (thus producing `\xff\xf4`):

```
>>> from netzob.all import *
>>> i = Integer(value=-12, sign=Sign.SIGNED, unitSize=UnitSize.SIZE_16)
>>> i.generate().tobytes()
b'\xff\xf4'
```

Examples of pre-defined Integer types

For convenience, common specific integer types are also available, with pre-defined values of `unitSize`, `sign` and `endianness` attributes. They are used to shorten calls of singular definitions.

For example, a *16-bit little-endian unsigned* Integer is classically defined like this:

```
>>> from netzob.all import *
>>> i = Integer(42,
...             unitSize=UnitSize.SIZE_16,
...             sign=Sign.UNSIGNED,
...             endianness=Endianness.LITTLE)
```

Could also be called in an equivalent form:

```
>>> from netzob.all import *
>>> i = uint16le(42)
```

There is an equivalence between these two integers, for every internal value of the type:

```
>>> from netzob.all import *
>>> i1 = Integer(42,
...              unitSize=UnitSize.SIZE_16,
...              sign=Sign.UNSIGNED,
...              endianness=Endianness.LITTLE)
>>> i2 = uint16le(42)
>>> i1, i2
(42, 42)
>>> i1 == i2
True
```

But a comparison between two specific integers of different kind will always fail, even if their values look equivalent:

```
>>> from netzob.all import *
>>> i1 = uint16le(42)
>>> i2 = uint32le(42)
>>> i1 == i2
False
```

And even when the concrete value seems identical, the integer objects are not:

```
>>> from netzob.all import *
>>> i1 = uint16le(42)
>>> i2 = int16le(42)
>>> i1, i2
(42, 42)
>>> print(i1, i2)
Integer=42 ((None, None)) Integer=42 ((None, None))
>>> i1 == i2
False
```

Integer raw representations

The following examples show how to create integers with different raw representation, depending on data type attributes. In these examples, we create a 16 bits little endian, a 16 bits big endian, a 32 bits little endian and a 32 bits big endian:

```
>>> from netzob.all import *
>>> int16le(1234).value.tobytes()
b'\xd2\x04'
>>> int16be(1234).value.tobytes()
b'\x04\xd2'
>>> int32le(1234).value.tobytes()
```

```
b'\xd2\x04\x00\x00'
>>> int32be(1234).value.tobytes()
b'\x00\x00\x04\xd2'
```

Representation of Integer type objects

The following examples show the representation of Integer objects with and without default value.

```
>>> from netzob.all import *
>>> i = int16le(value=12)
>>> str(i)
'Integer=12 ((None, None))'
```

```
>>> from netzob.all import *
>>> i = int16le()
>>> str(i)
'Integer=None ((-32768, 32767))'
```

Encoding of Integer type objects

The following examples show the encoding of Integer objects with and without default value.

```
>>> from netzob.all import *
>>> i = int32le(value=12)
>>> repr(i)
'12'
```

```
>>> from netzob.all import *
>>> i = int32le()
>>> repr(i)
'None'
```

1.2.2.2 BLOB / Raw Type

In the API, the definition of a BLOB type is done through the Raw class.

This chapter covers the following requirements: 25.

class Raw(*value*, *nbBytes*, *alphabet*)

This class defines a Raw type.

The Raw type describes a sequence of bytes of arbitrary sizes.

The Raw constructor expects some parameters:

Parameters

- **value** (`bitarray` or `bytes`, optional) – The current value of the type instance.
- **nbBytes** (an `int` or a tuple with the min and the max size specified as `int`, optional) – The size in bytes that this value can take.

- **alphabet** (a `list` of `bytes`, optional) – The alphabet can be used to limit the bytes that can participate in the domain value.

Note: `value` and `nbBytes` attributes are mutually exclusive. Setting both values raises an `Exception`.

The Raw class provides the following public variables:

Variables

- **typeName** (`str`) – The name of the implemented data type.
- **value** (`bitarray`) – The current value of the instance. This value is represented under the bitarray format.
- **size** (a tuple (`int`, `int`) or `int`) – The size in bits of the expected data type defined by a tuple (min, max). Instead of a tuple, an integer can be used to represent both min and max value.
- **alphabet** (a `list` of `bytes`) – The alphabet can be used to limit the bytes that can participate in the domain value.

The following example shows how to define a six bytes long raw object, and the used of the generation method to produce a value:

```
>>> from netzob.all import *
>>> from netzob.all import *
>>> r = Raw(nbBytes=6)
>>> len(r.generate().tobytes())
6
```

It is possible to define a range regarding the valid size of the raw object:

```
>>> from netzob.all import *
>>> r = Raw(nbBytes=(2, 20))
>>> 2 <= len(r.generate().tobytes()) <= 20
True
```

The following example shows the specification of a raw constant:

```
>>> from netzob.all import *
>>> r = Raw(b'\x01\x02\x03')
>>> print(r)
Raw=b'\x01\x02\x03' ((None, None))
```

The alphabet optional argument can be used to limit the bytes that can participate in the domain value:

```
>>> from netzob.all import *
>>> r = Raw(nbBytes=100, alphabet=[b"t", b"o"])
>>> data = r.generate().tobytes()
>>> data_set = set(data)
>>> data_set
{116, 111}
```

1.2.2.3 HexaString Type

In the API, the definition of a hexastring type is done through the HexaString class.

class HexaString(*value*, *nbBytes*)

This class defines a HexaString type.

The HexaString type describes a sequence of bytes of arbitrary sizes with the hexastring notation (e.g. `b'aabbcc'` instead of the raw notation `b'\xaa\xbb\xcc'`).

The HexaString constructor expects some parameters:

Parameters

- **value** (`bitarray` or `bytes`, optional) – The current value of the type instance.
- **nbBytes** (an `int` or a tuple with the min and the max size specified as `int`, optional) – The size in bytes that this value can take.

Note: `value` and `nbBytes` attributes are mutually exclusive. Setting both values raises an `Exception`.

The HexaString class provides the following public variables:

Variables

- **typeName** (`str`) – The name of the implemented data type.
- **value** (`bitarray`) – The current value of the instance. This value is represented under the bitarray format.
- **size** (a tuple (`int`, `int`) or `int`) – The size in bits of the expected data type defined by a tuple (min, max). Instead of a tuple, an integer can be used to represent both min and max value.

The following example shows how to define a hexastring object with a constant value, and the use of the generation method to produce a value:

```
>>> from netzob.all import *
>>> h = HexaString(b"aabbcc")
>>> h.generate().tobytes()
b'\xaa\xbb\xcc'
```

The following example shows how to define a hexastring object with a variable value, and the use of the generation method to produce a value:

```
>>> from netzob.all import *
>>> h = HexaString(nbBytes=6)
>>> len(h.generate().tobytes())
6
```

1.2.2.4 String Type

This chapter covers the following requirements: 26.

In the API, the definition of an ASCII or Unicode type is done through the String class.

class String(*value*, *nbChars*, *encoding*, *eos*)

This class defines a String type, which is used to represent String or Unicode characters.

The type String is a wrapper for the Python `str` object with the capability to express more constraints on the permitted string values.

The String constructor expects some parameters:

Parameters

- **value** (`bitarray` or `str`, optional) – The current value of the type instance.
- **nbChars** (an `int` or a tuple with the min and the max size specified as `int`, optional) – The amount of permitted String characters.
- **encoding** (`str`, optional) – The encoding of the string, such as 'ascii' or 'utf-8'. Default value is 'utf-8'.
- **eos** (a `list` of `AbstractType` or a `list` of `Field`, optional) – A list defining the potential terminal characters for the string, with either specific constants or pointers to other fields containing the permitted terminal values. Default value is an empty list, meaning there is no terminal character.

Note: `value` and `nbChars` attributes are mutually exclusive. Setting both values raises an `Exception`.

The String class provides the following public variables:

Variables

- **typeName** (`str`) – The name of the implemented data type.
- **value** (`bitarray`) – The current value of the instance. This value is represented under the bitarray format.
- **size** (a tuple (`int`, `int`) or `int`) – The size in bits of the expected data type defined by a tuple (min, max). Instead of a tuple, an integer can be used to represent both min and max value.
- **encoding** (`str`) – The encoding of the current value, such as 'ascii' or 'utf-8'.

- **eos** – A list defining the potential terminal characters for the string, with either specific constants or pointers to other fields containing the permitted terminal values.

Supported encodings are available on the Python reference documentation: Python Standard Encodings

Strings can be either static, dynamic with fixed sizes or even dynamic with variable sizes.

The following examples show how to define a static string in UTF-8:

```
>>> from netzob.all import *
>>> s = String("Paris")
>>> s.generate().tobytes()
b'Paris'
>>> s = String("Paris in Euro: €")
>>> s.generate().tobytes()
b'Paris in Euro: \xe2\x82\xac'
>>> s = String("Paris in Euro: €", encoding='utf-8')
>>> s.generate().tobytes()
b'Paris in Euro: \xe2\x82\xac'
```

The following example shows the raising of an exception if input value is not valid, with the definition of a string where the associated value contains a non-String element:

```
>>> from netzob.all import *
>>> s = String("Paris in €", encoding='ascii')
Traceback (most recent call last):
...
ValueError: Input value for the following string is incorrect: 'Paris in €'...
```

The following example shows how to define a string with a fixed size and a dynamic content:

```
>>> from netzob.all import *
>>> s = String(nbChars=10)
>>> len(s.generate().tobytes())
10
```

The following example shows how to define a string with a variable size and a dynamic content:

```
>>> from netzob.all import *
>>> s = String(nbChars=(10, 32))
>>> 10 <= len(s.generate().tobytes()) <= 32
True
```

String with terminal character

Strings with a terminal delimiter are supported. The following example shows the usage of a delimiter that can either be a constant or a Field object (see *Field* for more information).

```
>>> from netzob.all import *
>>> f_eos     = Field(String('\t'))
>>> s = String(eos=[String('\n'), Raw(b'\x00'), f_eos])
```

The `eos` attribute specifies a list of values that is used as potential terminal characters. Terminal characters can either be a constant (such as `String('\n')` and `Raw('\x00')`

in the previous example) or a targeted Field (such as `f_eos` in the previous example) from which the terminal character is used.

1.2.2.5 BitArray Type

This chapter covers the following requirements: 27, 28, 29.

In the API, the definition of a bitfield type is done through the BitArray class.

class BitArray (*value*, *nbBits*)

This class defines a BitArray type.

The BitArray type describes an object that contains a sequence of bits of arbitrary sizes.

The BitArray constructor expects some parameters:

Parameters

- **value** (`bitarray`, optional) – The current value of the type instance.
- **nbBits** (an `int` or a tuple with the min and the max size specified as `int`, optional) – The size in bits that this value can take.

Note: `value` and `nbBits` attributes are mutually exclusive. Setting both values raises an `Exception`.

The BitArray class provides the following public variables:

Variables

- **typeName** (`str`) – The name of the implemented data type.
- **value** (`bitarray`) – The current value of the instance. This value is represented under the bitarray format.
- **size** (a tuple (`int`, `int`) or `int`) – The size in bits of the expected data type defined by a tuple (min, max). Instead of a tuple, an integer can be used to represent both min and max value.
- **constants** (a `list` of `str`) – A list of named constant used to access the bitarray internal elements.

The following example show how to define a BitArray containing a fixed constant.

```
>>> from netzob.all import *
>>> b = BitArray('00001111')
>>> b.generate().tobytes()
b'\x0f'
```

Bitarray of fixed and dynamic sizes

The following example shows how to define a bitarray of 1 bit, 47 bits, 64 bits and then a bitarray whith a variable size between 13 and 128 bits:

```
>>> from netzob.all import *
>>> b = BitArray(nbBits=1)
>>> len(b.generate())
1
```

```
>>> from netzob.all import *
>>> b = BitArray(nbBits=47)
>>> len(b.generate())
47
```

```
>>> from netzob.all import *
>>> b = BitArray(nbBits=64)
>>> len(b.generate())
64
```

```
>>> from netzob.all import *
>>> b = BitArray(nbBits=(13, 128))
>>> 13 <= len(b.generate()) <= 128
True
```

Accessing bitarray elements by named constant

In the following example, we define a bitarray with two elements. As this bitarray has a fixed length, element are automatically accessible by predefined named constants (‘item_0’ and ‘item_1’):

```
>>> from netzob.all import *
>>> b = BitArray('00')
>>> b.constants
['item_0', 'item_1']
```

Bitarray element names can be changed:

```
>>> from netzob.all import *
>>> b.constants[0] = 'Urgent flag'
>>> b.constants[1] = 'Data flag'
>>> b.constants
['Urgent flag', 'Data flag']
```

Bitarray element can be accessed in read or write mode:

```
>>> from netzob.all import *
>>> b['Urgent flag']
False
>>> b['Urgent flag'] = True
>>> b['Urgent flag']
True
```

Bitarray element can be used with binary operators:

```
>>> from netzob.all import *
>>> b['Urgent flag'] |= b['Data flag']
>>> b['Urgent flag']
True
```

1.2.2.6 IPv4 Type

In the API, the definition of an IPv4 type is done through the IPv4 class.

class IPv4 (*value*, *network*, *endianness*)

This class defines an IPv4 type.

The IPv4 type encodes a `bytes` object in an IPv4 representation, and conversely to decode an IPv4 into a raw object.

The IPv4 constructor expects some parameters:

Parameters

- **value** (`str` or `netaddr.IPAddress`, optional) – An IP value expressed in standard dot notation (ex: "192.168.0.10").
- **network** (`str` or `netaddr.IPNetwork`, optional) – A network address expressed in standard dot notation (ex: "192.168.0.0/24").
- **endianness** (*`Endianness`*, optional) – The endianness of the current value. Values must be Endianness.BIG or Endianness.LITTLE. The default value is Endianness.BIG.

Note: `value` and `network` attributes are mutually exclusive.

The IPv4 class provides the following public variables:

Variables

- **typeName** (`str`) – The name of the implemented data type.
- **value** (`bitarray`) – The current value of the instance. This value is represented under the bitarray format.
- **size** (a tuple (`int`, `int`) or `int`) – The size in bits of the expected data type defined by a tuple (min, max). Instead of a tuple, an integer can be used to represent both min and max value.
- **network** (`str` or `netaddr.IPNetwork`) – A constraint over the network the parsed data belongs to this network or not.

The following examples show the use of an IPv4 type:

```
>>> from netzob.all import *
>>> ip = IPv4("192.168.0.10")
>>> ip.size
(None, None)
>>> ip.value
bitarray('11000000101010000000000000001010')
```

It is also possible to specify an IPv4 type that accepts a range of IP addresses, through the *network=* parameter, as shown on the following example:

```
>>> from netzob.all import *
>>> ip = IPv4(network="10.10.10.0/24")
>>> len(ip.generate().tobytes())
4
```

1.2.2.7 Timestamp Type

In the API, the definition of a timestamp type is done through the Timestamp class.

class Timestamp(*value=None*, *epoch=<Epoch.UNIX: datetime.datetime(1970, 1, 1, 0, 0)>*, *unity=<Unity.SECOND: 1>*, *unitSize=UnitSize.SIZE_32*, *endianness=Endianness.BIG*, *sign=Sign.UNSIGNED*)

This class defines a Timestamp type.

The Timestamp type defines dates in a specific format (such as Windows, Unix or MacOSX formats).

The Timestamp constructor expects some parameters:

Parameters

- **value** (`bitarray` or `int`, optional) – The raw value of the timestamp (in seconds by default). If None, the default generated value is the current time in UTC.
- **epoch** (`Epoch <netzob.Model.Vocabulary.Types.Timestamp.Epoch`, optional) – The initial date expressed in UTC from which timestamp is measured. Default value is EPOCH_UNIX.
- **unity** (`Unity <netzob.Model.Vocabulary.Types.Timestamp.Unity`, optional) – This specifies the unity of the timestamp (seconds, milliseconds, nanoseconds). The default value is UNITY_SECOND.
- **unitSize** (*`UnitSize`*, optional) – The unitsize of the current value. Values must be one of UnitSize.SIZE_* (see below for supported unit sizes). The default value is UnitSize.SIZE_32.

Note: `value` and `unitSize` attributes are mutually exclusive. Setting both values raises an `Exception`.

The Timestamp class provides the following public variables:

Variables

- **typeName** (`str`) – The name of the implemented data type.

- **value** (bitarray) – The current value of the instance. This value is represented under the bitarray format.
- **size** (a tuple (int, int) or int) – The size in bits of the expected data type defined by a tuple (min, max). Instead of a tuple, an integer can be used to represent both min and max value.
- **unitSize** (*UnitSize*, optional) – The unitsize of the current value. Values must be one of UnitSize.SIZE_* (see below for supported unit sizes).
- **epoch** (Epoch <netzob.Model.Vocabulary.Types.Timestamp.Epoch) – The initial date expressed in UTC from which timestamp is measured.
- **unity** (Unity <netzob.Model.Vocabulary.Types.Timestamp.Unity) – This specifies the unity of the timestamp (seconds, milliseconds, nanoseconds).

Available values for *epoch* parameter are:

- Epoch.WINDOWS = datetime(1601, 1, 1)
- Epoch.MUMPS = datetime(1840, 12, 31)
- Epoch.VMS = datetime(1858, 11, 17)
- Epoch.EXCEL = datetime(1899, 12, 31)
- Epoch.NTP = datetime(1900, 1, 1)
- Epoch.MACOS_9 = datetime(1904, 1, 1)
- Epoch.PICKOS = datetime(1967, 12, 31)
- Epoch.UNIX = datetime(1970, 1, 1)
- Epoch.FAT = datetime(1980, 1, 1)
- Epoch.GPS = datetime(1980, 1, 6)
- Epoch.ZIGBEE = datetime(2000, 1, 1)
- Epoch.COCOA = datetime(2001, 1, 1)

Available values for *unity* parameter are:

- Unity.SECOND = 1
- Unity.DECISECOND = 10
- Unity.CENTISECOND = 100
- Unity.MILLISECOND = 1000
- Unity.MICROSECOND = 1000000

- Unity.NANOSECOND = 10000000000

In the following example, a Timestamp data is created with a specific value '1444492442' and represented as 32 bits:

```
>>> from netzob.all import *
>>> time = Timestamp(1444492442)
>>> time.size
(None, None)
>>> time.value
bitarray('01010110000110010011010010011010')
>>> time.sign
Sign.UNSIGNED
>>> time.endianness
Endianness.BIG
```

1.3 Modeling Data Variables

In the API, data variable modeling is done through the class Data.

class Data (*dataType*, *originalValue=None*, *name=None*, *svas=None*)

The Data class is a variable which embeds specific content.

A Data object stores at least two things: 1) the definition domain and the constraints over it, through a `Type` object, and 2) the current value of the variable.

The Data constructor expects some parameters:

Parameters

- **dataType** (*AbstractType*, required) – The type of the data (for example Integer, Raw, String, ...).
- **originalValue** (`bitarray`, optional) – The original value of the data (can be None, which is the default behavior).
- **name** (`str`, optional) – The name of the data (if None, the name will be generated).
- **svas** (*SVAS*, optional) – The SVAS strategy defining how the Data value is used during abstraction and specialization process. The default strategy is SVAS.EPHEMERAL.

The Data class provides the following public variables:

Variables

- **currentValue** (`bitarray`) – The current value of the data.
- **dataType** (`AbstractType`) – The type of the data.

The following example shows the definition of the Data *pseudo* with a type String and a default value *"hello"*. This means that this Data object accepts any string, and the current value of this object is *"hello"*.

```
>>> from netzob.all import *
>>> s = String('hello').value
>>> data = Data(dataType=String(), originalValue=s, name="pseudo")
>>> data.varType
'Data'
>>> data.currentValue.tobytes()
b'hello'
>>> print(data.dataType)
String=None ((None, None))
>>> data.name
'pseudo'
```

currentValue

Property (getter/setter). The current value of the data.

Type `bitarray`

`dataType`

Property (getter/setter). The type of the data.

Type `AbstractType`

©2017 AMOSSYS

1.4 Modeling Fields

In the API, field modeling is done through the class Field.

class Field(*domain=None*, *name='Field'*, *isPseudoField=False*)

The Field class is used in the definition of a Symbol structure.

A Field describes a chunk of a Symbol and is specified by a definition domain, representing the set of values the field accepts.

The Field constructor expects some parameters:

Parameters

- **domain** (a `list` of `Variable` or a `list` of *`Field`*, optional) – The definition domain of the field (i.e. the set of values the field accepts). If not specified, the default definition domain will be `Raw()`, meaning it accepts any values.
- **name** (`str`, optional) – The name of the field. If not specified, the default name will be "Field".
- **isPseudoField** (`bool`, optional) – A flag indicating if the field is a pseudo field, meaning it is used internally to help for the computation of the value of another field, but does not directly produce data.

The Field class provides the following public variables:

Variables

- **name** (`str`) – The name of the field.
- **description** (`str`) – The description of the field.
- **domain** (a `list` of `object`) – The definition domain of the field (i.e. the set of values the field accepts).
- **fields** (a `list` of *`Field`*) – The sorted list of sub-fields. This variable should be used only if sub-field domains have basic types (for example *`Integer`* or *`Raw`*). More generally, preferably use *`Agg`*.
- **parent** (*`Field`* or *`Symbol`*) – The parent element.
- **isPseudoField** (`bool`) – A flag indicating if the field is a pseudo field, meaning it is used internally to help for the computation of the value of another field, but does not directly produce data.

Fields hierarchy

A field can be composed of sub-fields. This is useful for example to separate a header, composed of multiple fields, from its payload. The parent field can be seen as a facility to access to a group of fields.

In the following example, the `fheader` field is a parent field for a group of sub-fields. The parent field does not contain any concrete data, in contrary to its sub-fields.

```
>>> from netzob.all import *
>>> fh0 = Field(name='fh0')
>>> fh1 = Field(name='fh1')
>>> fheader = Field([fh0, fh1], name='fheader')
>>> print(fheader.str_structure())
fheader
|--  fh0
     |--   Data (Raw=None ((0, 524288)))
|--  fh1
     |--   Data (Raw=None ((0, 524288)))
```

More generally, a field is part of a tree whose root is a symbol and whose all other nodes of the tree are fields. Hence, a field always has a parent which can be another field or a symbol if it is the root.

Field definition domain

The value that can take a field is defined by its definition domain. The definition domain of a field can take multiple forms, in order to easily express basic types (such as Integer or String) or to model complex data structures (such has alternatives, repetitions or sequences).

The following examples present the different forms that permit to express the same field content (i.e. an Integer with a constant value of 10):

```
>>> from netzob.all import *
>>> f = Field(Data(Integer(10)))
>>> f = Field(Integer(10))
>>> f = Field(10)
```

If these fields are equivalent, this is because the first parameter of the Field constructor is *domain=*, thus its name can be omitted. Besides, the domain parameter will be parsed by a factory, which accepts either the canonical form of a definition domain (such as *domain=Data(Integer(10))*) or a shortened form (such as *domain=Integer(10)*, or even *domain=10*).

Relationships between fields

A field can have its value related to the content of another field. Such relationships may be specified through specific domain objects, such as Size or Value classes.

The following example describes a size relationship with a String field:

```
>>> from netzob.all import *
>>> f0 = Field(String("test"))
>>> f1 = Field(Size(f0))
>>> fheader = Field([f0, f1])
>>> fheader.specialize()
b'test\x04'
```

Pseudo fields

Sometimes, a specific field can be needed to express a complex data structure that depends on external data. This is the purpose of the *isPseudoField* flag. This flag indicates that the

current field is only used for the computation of the value of another field, but does not produce real content during specialization. The following example shows a pseudo field that contains an external data, and a real field whose content is the size of the external data:

```
>>> from netzob.all import *
>>> f_pseudo = Field(domain="An external data", isPseudoField=True)
>>> f_real = Field(domain=Size(f_pseudo))
>>> fheader = Field([f_pseudo, f_real])
>>> fheader.specialize()
b'\x10'
```

A real example of a pseudo field is found in the UDP checksum, which relies on a pseudo IP header for its computation.

getField(*field_name*)

Retrieve a sub-field based on its name.

Parameters **field_name** (`str`, required) – the name of the *Field* object

Returns The sub-field object.

Return type `Field`

The following example shows how to retrieve a sub-field based on its name:

```
>>> from netzob.all import *
>>> f1 = Field("hello", name="f1")
>>> f2 = Field("hello", name="f3")
>>> f3 = Field("hello", name="f2")
>>> fheader = Field("hello", name="fheader")
>>> fheader.fields = [f1, f2, f3]
>>> fheader.getField('f2')
f2
>>> type(fheader.getField('f2'))
<class 'netzob.Model.Vocabulary.Field.Field'>
```

getSymbol()

Return the symbol to which this field is attached.

Returns The associated symbol if available.

Return type *Symbol*

Raises `NoSymbolException`

To retrieve the associated symbol, this method recursively call the parent of the current object until the root is found.

If the root is not a *Symbol*, this raises an Exception.

The following example shows how to retrieve the parent symbol from a field object:

```
>>> from netzob.all import *
>>> field = Field("hello", name="F0")
>>> symbol = Symbol([field], name="S0")
>>> field.getSymbol()
S0
>>> type(field.getSymbol())
<class 'netzob.Model.Vocabulary.Symbol.Symbol'>
```

str_structure(*deepness=0*)

Returns a string which denotes the current symbol/field definition using a tree display.

Parameters deepness (`int`, required) – Parameter used to specify the number of indentations.

Returns The current symbol/field represented as a string.

Return type `str`

```
>>> from netzob.all import *
>>> f1 = Field(String(), name="field1")
>>> f2 = Field(Integer(interval=(10, 100)), name="field2")
>>> f3 = Field(Raw(nbBytes=14), name="field3")
>>> symbol = Symbol([f1, f2, f3], name="symbol_name")
>>> print(symbol.str_structure())
symbol_name
|--  field1
     |--   Data (String=None ((None, None)))
|--  field2
     |--   Data (Integer=None ((10, 100)))
|--  field3
     |--   Data (Raw=None ((112, 112)))
>>> print(f1.str_structure())
field1
|--   Data (String=None ((None, None)))
```

specialize()

The method `specialize()` generates a `bytes` sequence whose content follows the symbol definition.

Returns The produced content after specializing the symbol.

Return type `bytes`

Raises `GenerationException` if an error occurs while specializing the field.

The following example shows the `specialize()` method used for a field which contains a string with a constant value.

```
>>> from netzob.all import *
>>> f = Field(String("hello"))
>>> f.specialize()
b'hello'
```

The following example shows the `specialize()` method used for a field which contains a string with a variable value.

```
>>> from netzob.all import *
>>> f = Field(String(nbChars=4))
>>> len(f.specialize())
4
```

1.5 Modeling Fields with Complex Structures

Multiple variables can be combined to form a complex and precise specification of the values that are accepted by a field. Two complex variable types are provided:

- **Aggregate node variables**, which can be used to model a concatenation of variables.
- **Alternate node variables**, which can be used to model an alternative of multiple variables.
- **Repeat node variables**, which can be used to model a repetition of a variable.

Those node variables are described in details in this chapter.

1.5.1 Aggregate Domain

This chapter covers the following requirements: 35, 36, 37.

In the API, the definition of an concatenation of variables is done through the Agg class.

class Agg(*children=None*, *last_optional=False*, *svas=None*)

The Agg class is a node variable that represents a concatenation of variables.

An aggregate node concatenates the values that are accepted by its children nodes. It can be used to specify a succession of tokens.

The Agg constructor expects some parameters:

Parameters

- **children** (a `list` of `Variable`, optional) – The sequence of variable elements contained in the aggregate.
- **last_optional** (`bool`, optional) – A flag indicating if the last element of the children is optional or not.
- **svas** (*SVAS*, optional) – The SVAS strategy defining how the Aggregate behaves during abstraction and specialization. The default strategy is SVAS.EPHEMERAL.

The Agg class supports modeling of direct recursions on the right. To do so, the flag `SELF` is available, and should only be used in the last position of the aggregate (see example below).

Aggregate examples

For example, the following code represents a field that accepts values that are made of a String of 3 to 20 random characters followed by a ".txt" extension:

```
>>> from netzob.all import *
>>> t1 = String(nbChars=(3,20))
>>> t2 = String(".txt")
>>> f = Field(Agg([t1, t2]))
```

The following example shows an aggregate between BitArray variables:

```
>>> from netzob.all import *
>>> f = Field(Agg([BitArray('01101001'), BitArray(nbBits=3), BitArray(nbBits=5)]))
>>> t = f.specialize()
>>> len(t)
2
```

Examples of Agg internal attribute access

```
>>> from netzob.all import *
>>> domain = Agg([Raw(), String()])
>>> domain.varType
'Agg'
>>> print(domain.children[0].dataType)
Raw=None ((0, 524288))
>>> print(domain.children[1].dataType)
String=None ((None, None))
>>> domain.children.append(Agg([10, 20, 30]))
>>> len(domain.children)
3
>>> domain.children.remove(domain.children[0])
>>> len(domain.children)
2
```

Abstraction of aggregate variables

This example shows the abstraction process of an Aggregate variable:

```
>>> from netzob.all import *
>>> v1 = String(nbChars=(1, 10))
>>> v2 = String(".txt")
>>> f0 = Field(Agg([v1, v2]), name="f0")
>>> f1 = Field(String("!"), name="f1")
>>> f = Field([f0, f1])
>>> data = "john.txt!"
>>> Field.abstract(data, [f])
(Field, OrderedDict([('f0', b'john.txt'), ('f1', b'!')]))
```

In the following example, an Aggregate variable is defined. A message that does not correspond to the expected model is then parsed, thus the returned field is unknown:

```
>>> from netzob.all import *
>>> v1 = String(nbChars=(1, 10))
>>> v2 = String(".txt")
>>> f0 = Field(Agg([v1, v2]), name="f0")
>>> f1 = Field(String("!"), name="f1")
>>> f = Field([f0, f1])
>>> data = "johntxt!"
>>> Field.abstract(data, [f])
(Unknown message 'johntxt!', OrderedDict())
```

Specialization of aggregate variables

This example shows the specialization process of an Aggregate variable:

```
>>> from netzob.all import *
>>> d1 = String("hello")
>>> d2 = String(" john")
>>> f = Field(Agg([d1, d2]))
>>> f.specialize()
b'hello john'
```

Optional last variable

This example shows the specialization and parsing of an aggregate with an optional last variable*:

```
>>> from netzob.all import *
>>> a = Agg([int8(2), int8(3)], last_optional=True)
>>> f = Field(a)
>>> res = f.specialize()
>>> res == b'\x02' or res == b'\x02\x03'
True
>>> d = b'\x02\x03'
>>> Field.abstract(d, [f])
(Field, OrderedDict([('Field', b'\x02\x03')]))
>>> d = b'\x02'
>>> Field.abstract(d, [f])
(Field, OrderedDict([('Field', b'\x02')]))
```

Modeling indirect imbrication

The following example shows how to specify a field with a structure (`v2`) that can contain another structure (`v0`), through a tierce structure (`v1`). The flag `last_optional` is used to indicate that the specialization or parsing of the last element of the aggregates `v1` and `v2` is optional.

```
>>> from netzob.all import *
>>> v0 = Agg(["?", int8(4)])
>>> v1 = Agg(["!", int8(3), v0], last_optional=True)
>>> v2 = Agg([int8(2), v1], last_optional=True)
>>> f = Field(v2)
>>>
>>> # Test specialization
>>> res = f.specialize()
>>> res == b'\x02' or res == b'\x02!\x03' or res == b'\x02!\x03?\x04'
True
>>>
>>> # Test parsing
>>> (res_object, res_data) = Field.abstract(res, [f])
>>> res_object == f
True
```

Warning: Important note about recursion

The library can handle both direct and indirect recursion. However, there is a limitation requiring to use a recursing variable on the **right side of a statement**. Any other behavior could lead to infinite recursion during the loading of the model. To help understanding what syntax should be preferred, here is a list of annotated BNF syntaxes.

invalid syntaxes:

```
A  ::=   (* recursion on the left side *)
         [ A ], integer
```


©2017 AMOSSYS

```
B ::=  "(", B | ".", ")"
       (* recursion on the middle *)
```

valid adaptations from above examples:

```
A  ::=  (* recursion is replaced by a repeat approach *)
        integer, [ integer ]*
B  ::=  (* split the statement ... *)
        B', ")"
B' ::=  (* to transform a direct recursion into an
        indirect recursion on the right side *)
        "(", B | "."
```

valid recursion examples:

```
C ::=  (* a string with one or more dot characters *)
       ".", C*
D ::=  (* a string with zero or more dot characters *)
       [ D | "." ]*
```

Modeling direct recursion, simple example

The following example shows how to specify a field with a structure (`v`) that can optionally contain itself. To model such recursive structure, the `SELF` flag has to be used in the last position of the aggregate.

```
>>> from netzob.all import *
>>> v = Agg([int8(interval=(1, 5)), SELF], last_optional=True)
>>> f = Field(v)
>>>
>>> # Test specialization
>>> res = f.specialize()
>>> res
b'\x02\x04\x01'
>>>
>>> # Test parsing
>>> (res_object, res_data) = Field.abstract(res, [f])
>>> res_object == f
True
```

Modeling direct recursion, more complex example

This example introduces a recursion in the middle of an expression by modelling a pair group of parentheses (`'('` and `')'`), around a single character (`'+'`). The BNF syntax of this model would be:

```
parentheses ::=  "(", parentheses | "+" , ")"
```

This syntax introduces a recursivity in the middle of the *left* statement, which **is not supported**. Instead, this syntax could be adapted to move the recursivity to the right.

```
parentheses ::=  left, right
left        ::=  "(", parentheses | "+"
right       ::=  ")"
```

The following models describe this issue and provide a workaround.

BAD way

```
>>> from netzob.all import *
>>> parentheses = Agg(["(", Alt([SELF, "+"]), ")"])
Traceback (most recent call last):
ValueError: SELF can only be set at the last position of an Agg
```

GOOD way

```
>>> from netzob.all import *
>>> parentheses = Agg([])
>>> left = Agg(["(", Alt([parentheses, "+"])])
>>> right = ")"
>>> parentheses.children += [left, right]
>>>
>>> symbol = Symbol([Field(parentheses)])
>>> symbol.specialize()
b'(((+)))'
```

Modeling indirect recursion, simple example

The following example shows how to specify a field with a structure (`v2`) that contains another structure (`v1`), which can itself contains the first structure (`v2`). The flag `last_optional` is used to indicate that the specialization or parsing of the last element of the aggregate `v2` is optional.

```
>>> from netzob.all import *
>>> v1 = Agg([])
>>> v2 = Agg([int8(interval=(1, 3)), v1], last_optional=True)
>>> v1.children = ["!", v2]
>>> f = Field(v2)
>>> res = f.specialize()
>>> res
b'\x03!\x03!\x03!\x03'
>>>
>>> # Test parsing
>>> (res_object, res_data) = Field.abstract(res, [f])
>>> res_object == f
True
```

Modeling indirect recursion, more complex example

The following syntax provides a way to parse and specialize a subset of mathematical ex-

pressions including pair group of parentheses, digits from 0 to 9 and two arithmetic operators ('+' and '*').

```
num         ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9
operator    ::=  "+" | "*"
operation   ::=  left, right
left        ::=  subop, ")"
subop       ::=  "(", operation
```

The following examples **should** be compatible with these expressions:

```
1 + 2
1 + 2 + 3
1 + (2 + 3)
(1 + 2) + 3
(1 + 2) + 3 + 4
1 + (2 * 3) + (4 * 5)
1 + (2 * (3 + 4)) + 5
1 + ((2 * 3) * 4) * 5
```

These last expressions **should not** be compatible with these expressions:

```
1
1 ** 2
1 * (2 * 3
1 *
```

This example of indirect recursion introduces a recursion of the *operation* statement, called in the *subop* statement.

```
>>> from netzob.all import *
>>> num = Alt("0123456789")
>>> operator = Alt([" + ", " * "])
>>> operation = Agg([], last_optional=True)
>>> subop1 = Agg(["(", operation])
>>> subop2 = Agg([subop1, ")"])
>>> left = Alt([num, subop2])
>>> right = Agg([operator, operation])
>>> operation.children += [left, right]
>>> sym = Symbol([Field(operation)])
>>> sym.specialize()
b'((((4 * 8 * 4) + 5 + 9 + 0) * 7 * 0 + (4 + 9 + (3 * 4 + 2) * 0) * 9) + 4 * 7)'
```

1.5.2 Alternate Domain

This chapter covers the following requirements: 32.

In the API, the definition of an alternate of variables is done through the Alt class.

class Alt (*children=None*, *svas=None*)

The Alt class is a node variable that represents an alternative of variables.

A definition domain can take the form of a combination of permitted values/types/domains. This combination is represented by an alternate node. It can be seen as an OR operator between two or more children nodes.

The Alt constructor expects some parameters:

Parameters

- **children** (a list of Variable, optional) – The set of variable elements permitted in the alternative.
- **svas** (*SVAS*, optional) – The SVAS strategy defining how the Alternate behaves during abstraction and specialization. The default strategy is SVAS.EPHEMERAL.

For example, the following code denotes an alternate object that accepts either the string “filename1.txt” or the string “filename2.txt”:

```
>>> from netzob.all import *
>>> t1 = String("filename1.txt")
>>> t2 = String("filename2.txt")
>>> domain = Alt([t1, t2])
```

Examples of Alt internal attribute access

```
>>> from netzob.all import *
>>> domain = Alt([Raw(), String()])
>>> domain.varType
'Alt'
>>> print(domain.children[0].dataType)
Raw=None ((0, 524288))
>>> print(domain.children[1].dataType)
String=None ((None, None))
```

1.5.3 Repeat Domain

This chapter covers the following requirements: 33, 34.

In the API, the definition of a repetition of variables, or sequence, is done through the Repeat class.

class Repeat (*child*, *nbRepeat*, *delimiter=None*, *svas=None*)

The Repeat class is a node variable that represents a sequence of the same variable. This denotes an n-time repetition of a variable, which can be a terminal leaf or a non-terminal node.

The Repeat constructor expects some parameters:

Parameters

- **child** (Variable, required) – The variable element that will be repeated.

- **nbRepeat** (an int or a tuple of int or a Python variable containing an int or a *Field* or a func method, optional) – The number of repetitions of the element. This value can be a fixed integer, a tuple of integers defining the minimum and maximum of permitted repetitions, a constant from the calling script, a value present in another field, or can be identified by calling a callback function. In the latter case, the callback function should return a boolean telling if the expected number of repetitions is reached. Those use cases are described below.
- **delimiter** (bitarray, optional) – The delimiter used to separate the repeated element.
- **svas** (*SVAS*, optional) – The SVAS strategy defining how the Alternate behaves during abstraction and specialization. The default strategy is SVAS.EPHEMERAL.

Callback prototype

The callback function that can be used in the `nbRepeat` parameter has the following prototype:

```
def cbk_nbRepeat(nb_repeat, data, remaining=None,
parsed_structure=None, child=None)
```

Where:

- `nb_repeat` is an `int` that corresponds to the amount of time the child element has been parsed or specialize.
- `data` is a `bitarray` that corresponds to the already parsed or specialized data.
- `remaining` is a `bitarray` that corresponds to the remaining data to be parsed. Only set in parsing mode. In specialization mode, this parameter will have a *None* value. This parameter can therefore be used to identify the current mode.
- `parsed_structure` is a data structure that allows access to the values of the parsed `Variable` elements. Only set in parsing mode. In specialization mode, this parameter will have a *None* value.
- `child` is a `Variable` that corresponds to the repeated element. Only set in parsing mode. In specialization mode, this parameter will have a *None* value.

The `child` parameter allows access to the root of a tree structure. The `child` `Variable` can have children. Access to `Variable` values is done through the `parsed_structure`, thanks to its methods `hasData` and `getData`:

- `parsed_structure.hasData(child)` will return a `bool` telling if a data has been parsed for the child `Variable`.
- `parsed_structure.getData(child)` will return a `bitarray` that corresponds to the value parsed by the child `Variable`.

The callback function is called each time the child element is seen.

The callback function should return a boolean telling if the expected number of repetitions is reached (True) or not (False).

Basic usage of Repeat

The following example shows a repeat variable where the repeated element is a String:

```
>>> from netzob.all import *
>>> f1 = Field(Repeat(String("A"), nbRepeat=16))
>>> f1.specialize()
b'AAAAAAAAAAAAAAAA'
```

Usage of a delimiter in Repeat

We can specify a delimiter between each repeated element, as depicted in the following example:

```
>>> from netzob.all import *
>>> delimiter = bitarray(endian='big')
>>> delimiter.frombytes(b"-")
>>> f = Field(Repeat(Alt([String("A"), String("B")]), nbRepeat=(2, 4),
...            delimiter=delimiter), name='f1')
>>> 1 <= len(f.specialize()) <= 7
True
```

Limiting the number of repetitions with an integer

The following example shows how to create a Repeat variable whose number of repetitions is limited by an integer:

```
>>> from netzob.all import *
>>> f1 = Field(Repeat(String("john"), nbRepeat=3))
```

Limiting the number of repetitions with an interval of integers

The following example shows how to create a Repeat variable whose number of repetitions is limited by an interval of integers:

```
>>> from netzob.all import *
>>> f1 = Field(Repeat(String("john"), nbRepeat=(2,5)))
```

Limiting the number of repetitions with a Python integer variable

The following example shows how to create a Repeat variable whose number of repetitions is limited by a Python integer variable. Such variable is typically managed by the calling script:

```
>>> from netzob.all import *
>>> var = 3
>>> f1 = Field(Repeat(String("john"), nbRepeat=var))
```

Limiting the number of repetitions with the value of another field

The following example shows how to create a Repeat variable whose number of repetitions is limited by the value of another field:

```
>>> from netzob.all import *
>>> f_end = Field(Integer(interval=(2, 5)))
>>> f1 = Field(Repeat(String("john"), nbRepeat=f_end))
```

Limiting the number of repetitions by calling a callback function

The following example shows how to create a Repeat variable whose number of repetitions is handled by calling a callback function which returns a boolean telling if the expected number of repetitions is reached. Here, in parsing mode, the repeat stops when the byte *b'B'* is encountered. In specialization mode, the repeat stops at the first iteration.

```
>>> from netzob.all import *
>>> def cbk(nb_repeat, data, remaining=None, parsed_structure=None, child=None):
...     if remaining is not None:  # This means we are in parsing mode
...         print("in cbk: nb_repeat:{} -- data:{} -- remaining:{}".format(nb_repeat, data.
↪tobytes(), remaining.tobytes()))
...
...         # We check the value of the second child of the parameter child
...         if child.isnode() and len(child.children) > 1:
...             second_subchild = child.children[1]
...             if parsed_structure.hasData(second_subchild) and parsed_structure.
↪getData(second_subchild).tobytes() == b'B':
...                 return True
...         return False
...     return True
>>> f1 = Field(Repeat(Alt([String("A"), String("B")]), nbRepeat=cbk), name="f1")
>>> f2 = Field(String("B"), name="f2")
>>> f3 = Field(String("C"), name="f3")
>>> f = Field([f1, f2, f3])
>>> d = f.specialize()
>>> d == b'ABC' or d == b'BBC'
True
>>> data = "AABC"
>>> Field.abstract(data, [f])
in cbk: nb_repeat:1 -- data:b'A' -- remaining:b'ABC'
in cbk: nb_repeat:2 -- data:b'AA' -- remaining:b'BC'
in cbk: nb_repeat:3 -- data:b'AAB' -- remaining:b'C'
(Field, OrderedDict([('f1', b'AA'), ('f2', b'B'), ('f3', b'C')]))
```

1.6 Modeling Field Relationships

The ZDL language enables the definition of constraints on variables, in order to handle relationships. Those constraints are leveraged during abstraction and specialization of messages. The API supports two kind of relationships:

- **Intra-symbol relationships**, which denote a relationship between the size or the value of a variable, and another field in the same symbol.
- **Inter-symbol relationships**, which denote a relationship with a field of another symbol.

1.6.1 Value Relationships

This chapter covers the following requirements: 45.

In the API, the definition of a relationship with the value of another field is done through the Value class. This class enables the computation of the relationship result by a basic copy of the targeted field or by calling a callback function.

class Value(*target*, *name=None*, *operation=None*)

The Value class is a variable whose content is the value of another field.

It is possible to define a field so that its value is equal to the value of another field, on which a transformation operation can be realized.

The Value constructor expects some parameters:

Parameters

- **target** (`Field`, required) – The targeted field of the relationship.
- **name** (`str`, optional) – The name of the variable. If None, the name will be generated.
- **operation** (`Callable`, optional) – An optional transformation operation to be applied on the targeted field value, through a callback.

The Value class provides the following public variables:

Variables operation (`Callable`) – Defines the operation to be performed on the found value. This operation takes the form of a python function that accepts a single parameter of BitArray type and returns a BitArray.

Callback prototype

A callback function can be used to specify a complex relationship. The callback function that can be used in the `operation` parameter has the following prototype:

```
def cbk_operation(data):
```

Where:

- `data` is a `bitarray` that contains the value of the targeted field.

The callback function is expected to implement relationship operations based on the provided data.

The callback function should return a `bitarray`.

Value examples

The following example shows how to define a field with a copy of another field value, in specialization mode:

```
>>> from netzob.all import *
>>> f0 = Field(String("abcd"))
>>> f1 = Field(Value(f0))
>>> fheader = Field([f0, f1])
>>> fheader.specialize()
b'abcdabcd'
```

Value field with a variable as a target

The following example shows the specialization process of a Value field whose target is a variable:

```
>>> from netzob.all import *
>>> d = Data(String("john"))
>>> f1 = Field(domain=d, name="f1")
>>> f2 = Field(String(";"), name="f2")
>>> f3 = Field(Value(d), name="f3")
>>> f4 = Field(String("!"), name="f4")
>>> f = Field([f1, f2, f3, f4])
>>> f.specialize()
b'john;john!'
```

Specialization of Value objects

The following examples show the specialization process of Value objects:

```
>>> from netzob.all import *
>>> f1 = Field(String("john"), name="f1")
>>> f2 = Field(String(";"), name="f2")
>>> f3 = Field(Value(f1), name="f3")
>>> f4 = Field(String("!"), name="f4")
>>> f = Field([f1, f2, f3, f4])
>>> f.specialize()
b'john;john!'
```

```
>>> from netzob.all import *
>>> f3 = Field(String("john"), name="f3")
>>> f2 = Field(String(";"), name="f2")
>>> f1 = Field(Value(f3), name="f1")
>>> f4 = Field(String("!"), name="f4")
>>> f = Field([f1, f2, f3, f4])
>>> f.specialize()
b'john;john!'
```

Transformation operation on targeted field value

A named callback function can be used to specify a more complex relationship. The fol-

lowing example shows a relationship where the computed value corresponds to the reversed bits of the targeted field value. The `data` parameter of the `cbk` function contains a bitarray object of the targeted field value. The `cbk` function returns a bitarray object.

```
>>> from netzob.all import *
>>> def cbk(data):
...     ret = data.copy()
...     ret.reverse()
...     return ret
>>> f0 = Field(Raw(b'\x01'))
>>> f1 = Field(Value(f0, operation = cbk))
>>> f = Field([f0, f1])
>>> f.specialize()
b'\x01\x80'
```

1.6.2 Size Relationships

This chapter covers the following requirements: 38.

class Size(*targets*, *dataType=None*, *factor=0.125*, *offset=0*, *name=None*)

The Size class is a variable whose content is the size of other field values.

It is possible to define a field so that its value is equal to the size of another field, or group of fields (potentially including itself).

By default, the computed size expresses an amount of bytes. It is possible to change this behavior by using the parameters `factor` and `offset`.

The Size constructor expects some parameters:

Parameters

- **targets** (a `list` of *`Field`*, required) – The targeted fields of the relationship.
- **dataType** (*`AbstractType`*, optional) – Specify that the produced value should be represented according to this dataType. If None, default value is Raw(nbBytes=1).
- **factor** (`float`, optional) – Specify that the initial size value (always expressed in bits) should be divided by this factor. The default value is `1.0/8`. For example, to express a size in bytes, the factor should be `1.0/8`, whereas to express a size in bits, the factor should be `1.0`.
- **offset** (`int`, optional) – Specify that an offset value should be added to the final size value (after applying the factor parameter). The default value is 0.
- **name** (`str`, optional) – The name of the variable. If None, the name will be generated.

The Size class provides the following public variables:

Variables

- **dataType** (*AbstractType*) – The type of the data.
- **factor** – Defines the multiplication factor to apply on the targeted length.
- **offset** – Defines the offset to apply on the computed length.

The following example shows how to define a size field with a Raw dataType:

```
>>> from netzob.all import *
>>> f0 = Field(String(nbChars=10))
>>> f1 = Field(String(";"))
>>> f2 = Field(Size([f0], dataType=Raw(nbBytes=1)))
>>> f = Field([f0, f1, f2])
>>> data = f.specialize()
>>> data[-1] == 10
True
```

The following example shows how to define a size field with a Raw dataType, along with specifying the `factor` and `offset` parameters.

```
>>> from netzob.all import *
>>> f0 = Field(String(nbChars=(4,10)))
>>> f1 = Field(String(";"))
>>> f2 = Field(Size([f0, f1], dataType=Raw(nbBytes=1), factor=1./8, offset=4))
>>> f = Field([f0, f1, f2])
>>> data = f.specialize()
>>> data[-1] > (4*8*1./8 + 4) # == 4 bytes minimum * 8 bits * a factor of 1./8 + an offset␣
↪of 4
True
```

In this example, the field *f2* is a size field where its value is equal to the size of the concatenated values of fields *f0* and *f1*. The *dataType* parameter specifies that the produced value should be represented as a `Raw`. The *factor* parameter specifies that the initial size value (always expressed in bits) should be divided by 8 (in order to retrieve the amount of bytes). The *offset* parameter specifies that the final size value should be computed by adding 4 bytes.

The following example shows how to define a size field so that its value depends on a list of non-consecutive fields:

```
>>> from netzob.all import *
>>> f1 = Field(String("="))
>>> f2 = Field(String("#"))
>>> f4 = Field(String("%"))
>>> f5 = Field(Raw(b"_"))
>>> f3 = Field(Size([f1, f2, f4, f5]))
>>> f = Field([f1, f2, f3, f4, f5])
>>> f.specialize()
b'=#\x04%_'
```

In the following example, a size field is declared after its targeted field. This shows that the field order does not impact the relationship computations.

```
>>> from netzob.all import *
>>> f0 = Field(String(nbChars=(1,4)), name='f0')
>>> f1 = Field(String(";"), name='f1')
```

```
>>> f2 = Field(Size(f0), name='f2')
>>> f = Field([f0, f1, f2])
>>> 3 <= len(f.specialize()) <= 6
True
```

In the following example, a size field is declared before the targeted field:

```
>>> from netzob.all import *
>>> f2 = Field(String(nbChars=(1,4)), name="f2")
>>> f1 = Field(String(";"), name="f1", )
>>> f0 = Field(Size(f2), name="f0")
>>> f = Field([f0, f1, f2])
>>> 3 <= len(f.specialize()) <= 6
True
```

Size field with fields and variables as target

The following examples show the specialization process of a Size field whose targets are both fields and variables:

```
>>> from netzob.all import *
>>> d = Data(String(nbChars=20))
>>> f0 = Field(domain=d)
>>> f1 = Field(String(";"))
>>> f2 = Field(Size([d, f1]))
>>> f = Field([f0, f1, f2])
>>> res = f.specialize()
>>> b'\x15' in res
True
```

```
>>> from netzob.all import *
>>> d = Data(String(nbChars=20))
>>> f2 = Field(domain=d)
>>> f1 = Field(String(";"))
>>> f0 = Field(Size([f1, d]))
>>> f = Field([f0, f1, f2])
>>> res = f.specialize()
>>> b'\x15' in res
True
```

dataType

Property (getter/setter). The datatype used to encode the result of the computed size.

Type *AbstractType*

factor

Property (getter/setter). Defines the multiplication factor to apply on the targeted length (in bits).

Type float

offset

Property (getter/setter). Defines the offset to apply on the computed length. computed size = (factor*size(targetField)+offset)

Type int

 ©2017 AMOSSYS

1.6.3 Padding Relationships

This chapter covers the following requirements: 30, 31.

In the API, it is possible to model a structure with a padding through the class Padding.

class Padding(*targets*, *data*, *modulo*, *factor=1.0*, *offset=0*, *name=None*)

The Padding class is a variable whose content permits to produce a padding value that can be used to align a structure to a fixed size.

The Padding constructor expects some parameters:

Parameters

- **targets** (a `list` of `Field`, required) – The targeted fields of the relationship.
- **data** (a *`AbstractType`* or a `callable`, required) – Specify that the produced value should be represented according to this data. A callback function, returning the padding value, can be used here.
- **modulo** (`int`, required) – Specify the expected modulo size. The padding value will be computed so that the whole structure aligns to this value. This typically corresponds to a block size in cryptography.
- **factor** (`float`, optional) – Specify that the length of the targeted structure (always expressed in bits) should be divided by this factor. The default value is `1.0`. For example, to express a length in bytes, the factor should be `1.0/8`, whereas to express a length in bits, the factor should be `1.0`.
- **offset** (`int`, optional) – Specify a value in bits that should be added to the length of the targeted structure (after applying the factor parameter). The default value is 0.
- **name** (`str`, optional) – The name of the variable. If None, the name will be generated.

Callback prototype

The callback function that can be used in the `data` parameter has the following prototype:

```
def cbk_data(current_length, modulo)
```

Where:

- `current_length` is an `int` that corresponds to the current size in bits of the targeted structure.
- `modulo` is an `int` that corresponds to the expected modulo size in bits.

The callback function should return a `bitarray`.

Padding examples

The following code illustrates a padding with an integer modulo. Here, the padding data `b'\x00'` is repeated `n` times, where `n` is computed by decrementing the modulo number, `128`, by the current length of the targeted structure. The padding length is therefore equal to `128 - (10+2)*8 = 32` bits.

```
>>> from netzob.all import *
>>> f0 = Field(Raw(nbBytes=10))
>>> f1 = Field(Raw(b"##"))
>>> f2 = Field(Padding([f0, f1], data=Raw(b'\x00'), modulo=128))
>>> f = Field([f0, f1, f2])
>>> d = f.specialize()
>>> d[12:]
b'\x00\x00\x00\x00'
>>> len(d) * 8
128
```

The following code illustrates a padding with the use of the `offset` parameter, where the targeted field sizes is decremented by 8 when computing the padding value length.

```
>>> from netzob.all import *
>>> f0 = Field(Raw(nbBytes=10))
>>> f1 = Field(Raw(b"##"))
>>> f2 = Field(Padding([f0, f1], data=Raw(b'\x00'), modulo=128, offset=8))
>>> f = Field([f0, f1, f2])
>>> d = f.specialize()
>>> d[12:]
b'\x00\x00\x00'
>>> len(d) * 8
120
```

The following code illustrates a padding with the use of the `factor` parameter, where the targeted field sizes is divided by 2 before computing the padding value length.

```
>>> from netzob.all import *
>>> f0 = Field(Raw(nbBytes=10))
>>> f1 = Field(Raw(b"##"))
>>> f2 = Field(Padding([f0, f1], data=Raw(b'\x00'), modulo=128, factor=1./2))
>>> f = Field([f0, f1, f2])
>>> d = f.specialize()
>>> d[12:]
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> len(d) * 8
176
```

The following code illustrates a padding with the use of a callback function that helps to determine the padding value. In this example, the padding value is an incrementing integer.

```
>>> from netzob.all import *
>>> f0 = Field(Raw(nbBytes=10))
>>> f1 = Field(Raw(b"##"))
>>> def cbk_data(current_length, modulo):
...     length_to_pad = modulo - (current_length % modulo)  # Length in bits
...     length_to_pad = int(length_to_pad / 8)  # Length in bytes
...     res_bytes = b"".join([t.to_bytes(1, byteorder='big') for t in list(range(length_to_
↪pad))])
...     res_bits = bitarray(endian='big')
...     res_bits.frombytes(res_bytes)
...     return res_bits
```

```
>>> f2 = Field(Padding([f0, f1], data=cbk_data, modulo=128))
>>> f = Field([f0, f1, f2])
>>> d = f.specialize()
>>> d[12:]
b'\x00\x01\x02\x03'
>>> len(d) * 8
128
```

The following code illustrates a padding with the use of a callback function that helps to determine the padding value. In this example, the padding value is a repetition of an incrementing integer, thus implementing the PKCS #7 padding.

```
>>> from netzob.all import *
>>> f0 = Field(Raw(nbBytes=10))
>>> f1 = Field(Raw(b"##"))
>>> def cbk_data(current_length, modulo):
...     length_to_pad = modulo - (current_length % modulo)  # Length in bits
...     length_to_pad = int(length_to_pad / 8)  # Length in bytes
...     res_bytes = b"".join([int(length_to_pad).to_bytes(1, byteorder='big') * length_to_
↪pad])
...     res_bits = bitarray(endian='big')
...     res_bits.frombytes(res_bytes)
...     return res_bits
>>> f2 = Field(Padding([f0, f1], data=cbk_data, modulo=128))
>>> f = Field([f0, f1, f2])
>>> d = f.specialize()
>>> d[12:]
b'\x04\x04\x04\x04'
>>> len(d) * 8
128
```

1.6.4 Checksum Relationships

This chapter covers the following requirements: 39.

The ZDL language enables the definition of checksum relationships between fields.

Checksum API

As an example, the API for the CRC16 checksum is as follows:

class CRC16(*targets*)

This class implements the CRC16 function.

The constructor expects some parameters:

Parameters **targets** (a `list` of *Field*, required) – The targeted fields of the relationship.

The following example shows how to create a checksum relationship with another field:

```
>>> from netzob.all import *
>>> import binascii
>>> f1 = Field(Raw(b'\xaa\xbb'))
>>> f2 = Field(CRC16([f1]))
>>> f = Field([f1, f2])
```

```
>>> binascii.hexlify(f.specialize())
b'aabb3ed3'
```

The following example shows how to create a checksum relationship with a group of fields:

```
>>> from netzob.all import *
>>> import binascii
>>> f1 = Field(Raw(b'\xaa\xbb'))
>>> f2 = Field(Raw(b'\xcc\xdd'))
>>> f3 = Field(Raw(b'\xee\xff'))
>>> f4 = Field(CRC16([f1, f2, f3]))
>>> f = Field([f1, f2, f3, f4])
>>> binascii.hexlify(f.specialize())
b'aabbccddeeff5e9b'
```

Available checksums

The following list shows the available checksums. The API for those checksums are similar to the CRC16 API.

- *CRC16(targets)*
- CRC16DNP(targets)
- CRC16Kermit(targets)
- CRC16SICK(targets)
- CRC32(targets)
- CRCCCITT(targets)
- InternetChecksum(targets) (used in ICMP, UDP, IP, TCP protocols, as specified in **RFC 1071**).

1.6.5 Hash Relationships

This chapter covers the following requirements: 39.

The ZDL language enables the definition of hash relationships between fields.

Hash API

As an example, the API for the MD5 hash is as follows:

class MD5(*targets*)

This class implements the MD5 relationships between fields.

The constructor expects some parameters:

Parameters targets (a list of *Field*, required) – The targeted fields of the relationship.

The following example shows how to create a hash relation with another field:

```
>>> from netzob.all import *
>>> import binascii
>>> f1 = Field(Raw(b'\xaa\xbb'))
>>> f2 = Field(MD5([f1]))
>>> f = Field([f1, f2])
>>> binascii.hexlify(f.specialize())
b'aabb58cea1f6b2b06520613e09af90dc1c47'
```

Available hashes

The following list shows the available hashes. The API for those hashes are similar to the MD5 API.

- `MD5(targets)`
- `SHA1(targets)`
- `SHA1_96(targets)`
- `SHA2_224(targets)`
- `SHA2_256(targets)`
- `SHA2_384(targets)`
- `SHA2_512(targets)`

1.6.6 HMAC Relationships

This chapter covers the following requirements: 39.

The ZDL language enables the definition of HMAC relationships between fields.

HMAC API

As an example, the API for the HMAC_MD5 is as follows:

class HMAC_MD5(*targets*, *key*)

This class implements the HMAC_MD5.

The constructor expects some parameters:

Parameters

- **targets** (a `list` of `Field`, required) – The targeted fields of the relationship.
- **key** (`bytes`, required) – The cryptographic key used in the hmac computation.

The following example shows how to create a HMAC relation with another field:

```
>>> from netzob.all import *
>>> import binascii
>>> f1 = Field(Raw(b'\xaa\xbb'))
```

```
>>> f2 = Field(HMAC_MD5([f1], key=b'1234'))
>>> f = Field([f1, f2])
>>> binascii.hexlify(f.specialize())
b'aabbb71c98baa40dc8a49361816d5dc1eb25'
```

Available HMACs

The following list shows the available HMACs. The API for those HMACs are similar to the HMAC_MD5 API.

- *HMAC_MD5(targets, key)*
- HMAC_SHA1(targets, key)
- HMAC_SHA1_96(targets, key)
- HMAC_SHA2_224(targets, key)
- HMAC_SHA2_256(targets, key)
- HMAC_SHA2_384(targets, key)
- HMAC_SHA2_512(targets, key)

1.7 Modeling Symbols

In the API, symbol modeling is done through the class Symbol.

class Symbol (*fields=None*, *messages=None*, *name='Symbol'*)

The Symbol class is a main component of the Netzob protocol model.

A symbol represents an abstraction of all messages of the same type from a protocol perspective. A symbol structure is made of fields.

The Symbol constructor expects some parameters:

Parameters

- **fields** (a list of *Field*, optional) – The fields that participate in the symbol definition, in the wire order. May be None (thus, a generic *Field* instance would be defined), especially when using Symbols for reverse engineering (i.e. fields identification).
- **messages** (a list of AbstractMessage, optional) – The messages that are associated with the symbol. May be None (thus, an empty list would be defined), especially when modeling a protocol from scratch (i.e. the fields are already known).
- **name** (str, optional) – The name of the symbol. If not specified, the default name will be "Symbol".

The Symbol class provides the following public variables:

©2017 AMOSSYS

Variables

- **name** (str) – The name of the symbol.
- **description** (str) – The description of the symbol.
- **fields** (a list of *Field*) – The sorted list of sub-fields.

Usage of Symbol for protocol modeling

The Symbol class may be used to model a protocol from scratch, by specifying its structure in terms of fields:

```
>>> from netzob.all import *
>>> f0 = Field("aaaa")
>>> f1 = Field(" # ")
>>> f2 = Field("bbbbbb")
>>> symbol = Symbol(fields=[f0, f1, f2])
>>> print(symbol.str_structure())
Symbol
|--  Field
     |--   Data (String=aaaa ((None, None)))
|--  Field
     |--   Data (String= #  ((None, None)))
|--  Field
     |--   Data (String=bbbbbb ((None, None)))
```

Usage of Symbol for traffic generation and parsing

A Symbol class may be used to generate concrete messages according to its field definition, through the `specialize()` method, and may also be used to abstract a concrete message into its associated symbol through the `abstract()` method:

```
>>> from netzob.all import *
>>> f0 = Field("aaaa")
>>> f1 = Field(" # ")
>>> f2 = Field("bbbbbb")
>>> symbol = Symbol(fields=[f0, f1, f2])
>>> concrete_message = symbol.specialize()
>>> concrete_message
b'aaaa # bbbbbb'
>>> (abstracted_symbol, structured_data) = Symbol.abstract(concrete_message, [symbol])
>>> abstracted_symbol == symbol
True
```

specialize(*presets=None*, *fuzz=None*, *memory=None*)

The method `specialize()` generates a bytes sequence whose content follows the symbol definition.

The specialize() method expects some parameters:

Parameters

- **presets** (dict, optional) – A dictionary of keys:values used to preset (parameterize) fields during symbol specialization. Values in this dictionary will override any field definition, constraints or relationship dependencies.

- **fuzz** (Fuzz, optional) – A fuzzing configuration used during the specialization process. Values in this configuration will override any field definition, constraints, relationship dependencies or parameterized fields. See Fuzz for a complete explanation of its use for fuzzing purpose.
- **memory** (*Memory*, optional) – A memory used to store variable values during specialization and abstraction of successive symbols, especially to handle inter-symbol relationships. If None, a temporary memory is created by default and used internally during the scope of the specialization process.

Returns The produced content after specializing the symbol.

Return type bytes

Raises GenerationException if an error occurs while specializing the field.

The following example shows the specialize() method used for a field which contains a String and a Size fields.

```
>>> from netzob.all import *
>>> f1 = Field(domain=String(nbChars=5))
>>> f0 = Field(domain=Size(f1))
>>> s = Symbol(fields=[f0, f1])
>>> result = s.specialize()
>>> result[0]
5
>>> len(result)
6
```

Parameterized specialization of field values (‘presets=‘ parameter)

It is possible to preset (parameterize) fields during symbol specialization, through a dict passed in the presets= parameter of the specialize() method. Values in this dictionary will override any field definition, constraints or relationship dependencies.

The presets dictionary accepts a sequence of keys and values, where keys correspond to the fields in the symbol that we want to override, and values correspond to the overriding content. Keys are either expressed as *Field* objects or strings containing field accessors when field names are used (such as in f = Field(name="udp.dport")). Values are either expressed as bitarray (as it is the internal type for variables in the Netzob library), as :class:‘bytes’ or in the type of the overridden field variable.

The following code shows the definition of a simplified UDP header that will be later used as base example. This UDP header is made of one named field containing a destination port, and a named field containing a payload:

```
>>> from netzob.all import *
>>> f_dport = Field(name="udp.dport", domain=Integer(unitSize=UnitSize.SIZE_8))
```

```
>>> f_payload = Field(name="udp.payload", domain=Raw(nbBytes=2))
>>> symbol_udp = Symbol(name="udp", fields=[f_dport, f_payload])
```

The three following codes show the same way to express the parameterized **values** during specialization of the fields `udp_dport` and `udp_payload`:

```
>>> from netzob.all import *
>>> presets = {}
>>> presets["udp.dport"] = 11              # udp.dport expects an int or an Integer
>>> presets["udp.payload"] = b"\xaa\xbb"   # udp.payload expects a bytes object or a␣
↪Raw object
>>> symbol_udp.specialize(presets=presets)
b'\x0b\xaa\xbb'
```

```
>>> from netzob.all import *
>>> presets = {}
>>> presets["udp.dport"] = Integer(11)         # udp.dport expects an int or an Integer
>>> presets["udp.payload"] = Raw(b"\xaa\xbb") # udp.payload expects a bytes object␣
↪or a Raw object
>>> symbol_udp.specialize(presets=presets)
b'\x0b\xaa\xbb'
```

```
>>> from netzob.all import *
>>> presets = {}
>>> presets["udp.dport"] = bitarray('00001011', endian='big')
>>> presets["udp.payload"] = bitarray('1010101010111011', endian='big')
>>> symbol_udp.specialize(presets=presets)
b'\x0b\xaa\xbb'
```

The previous example shows the use of BitArray as dict values. BitArray are always permitted for any parameterized field, as it is the internal type for variables in the Netzob library.

The two following codes show the same way to express the parameterized **keys** during specialization of the fields `udp_dport` and `udp_payload`:

```
>>> from netzob.all import *
>>> presets = {}
>>> presets[f_dport] = 11
>>> presets[f_payload] = b"\xaa\xbb"
>>> symbol_udp.specialize(presets=presets)
b'\x0b\xaa\xbb'
```

```
>>> from netzob.all import *
>>> presets = {}
>>> presets["udp.dport"] = 11
>>> presets["udp.payload"] = b"\xaa\xbb"
>>> symbol_udp.specialize(presets=presets)
b'\x0b\xaa\xbb'
```

A preset value bypasses all the constraint checks on your field definition. For example, in the following example it can be used to bypass a size field definition.

```
>>> from netzob.all import *
>>> f1 = Field()
>>> f2 = Field(domain=Raw(nbBytes=(10,15)))
>>> f1.domain = Size(f2)
>>> s = Symbol(fields=[f1, f2])
```

```
>>> presetValues = {f1: bitarray('11111111')}
>>> s.specialize(presets = presetValues)[0]
255
```

Fuzzing of Fields

It is possible to fuzz fields during symbol specialization, through the `fuzz=` parameter of the `specialize()` method. Values in this parameter will override any field definition, constraints, relationship dependencies or parameterized values.

For more information regarding the expected `fuzz=` parameter content, see the class `Fuzz`.

specialize_count (*presets=None*, *fuzz=None*, *timeout=None*)

The method `specialize_count()` computes the expected number of unique produced messages, considering the initial symbol model, the preset fields and the fuzzed fields.

The `specialize_count()` method expects the same parameters as the `specialize()` method:

Parameters

- **presets** (`dict`, optional) – A dictionary of keys:values used to preset (parameterize) fields during symbol specialization. Values in this dictionary will override any field definition, constraints or relationship dependencies.
- **fuzz** (`Fuzz`, optional) – A fuzzing configuration used during the specialization process. Values in this configuration will override any field definition, constraints, relationship dependencies or parameterized fields. See `Fuzz` for a complete explanation of its use for fuzzing purpose.
- **timeout** (`float` or `int` in seconds) – The computation time beyond which `-1` is returned

Returns The number of unique values the symbol specialization can produce.

Return type a `int`

Note: The theorical value returned by `specialize_count()` may be huge and hard to compute considering the number of variables involved. Beyond `timeout` the computation would return the special value `-1` indicating a too large value to compute.

```
>>> # Symbol definition
>>> from netzob.all import *
>>> f1 = Field(uint16(interval=(50, 1000)))
>>> f2 = Field(uint8())
>>> f3 = Field(uint8())
>>> symbol = Symbol(fields=[f1, f2, f3])
```

```
>>>
>>> # Specify the preset fields
>>> presetValues = {f1: bitarray('1111111111111111')}
>>>
>>> # Count the expected number of unique produced messages
>>> symbol.specialize_count(presets=presetValues)
279
```

abstract (*data*, *fields*, *memory=None*)

The method `abstract()` is used to retrieve the corresponding symbol according to a concrete `bytes` message.

The `abstract()` static method expects some parameters:

Parameters

- **data** (`bytes`, required) – The concrete message to abstract in symbol.
- **fields** (`list` of `Field`, required) – a list of fields targeted during the abstraction process
- **memory** (*Memory*, optional) – A memory used to store variable values during specialization and abstraction of sequence of symbols.

Returns a field/symbol and the structured received message

Return type a tuple (`Field`, dict)

Raises `AbstractionException` if an error occurs while abstracting the data

```
>>> from netzob.all import *
>>> messages = ["{0}, what's up in {1} ?".format(pseudo, city)
...             for pseudo in ['john', 'kurt']
...             for city in ['Paris', 'Berlin']]
```

```
>>> f1a = Field(name="name", domain="john")
>>> f2a = Field(name="question", domain=", what's up in ")
>>> f3a = Field(name="city", domain=Alt(["Paris", "Berlin"]))
>>> f4a = Field(name="mark", domain=" ?")
>>> s1 = Symbol([f1a, f2a, f3a, f4a], name="Symbol-john")
```

```
>>> f1b = Field(name="name", domain="kurt")
>>> f2b = Field(name="question", domain=", what's up in ")
>>> f3b = Field(name="city", domain=Alt(["Paris", "Berlin"]))
>>> f4b = Field(name="mark", domain=" ?")
>>> s2 = Symbol([f1b, f2b, f3b, f4b], name="Symbol-kurt")
```

```
>>> for m in messages:
...     (abstractedSymbol, structured_data) = Symbol.abstract(m, [s1, s2])
...     print(structured_data)
...     print(abstractedSymbol.name)
OrderedDict([('name', b'john'), ('question', b", what's up in "), ('city', b'Paris'),
 ↪ ('mark', b' ?')])
Symbol-john
OrderedDict([('name', b'john'), ('question', b", what's up in "), ('city', b'Berlin'),
 ↪ ('mark', b' ?')])
```

```
Symbol-john
OrderedDict([('name', b'kurt'), ('question', b", what's up in "), ('city', b'Paris'),
↪ ('mark', b' ?')])
Symbol-kurt
OrderedDict([('name', b'kurt'), ('question', b", what's up in "), ('city', b'Berlin'),
↪ ('mark', b' ?')])
Symbol-kurt
```

getField(*field_name*)

Retrieve a sub-field based on its name.

Parameters **field_name** (`str`, required) – the name of the `Field` object

Returns The sub-field object.

Return type `Field`

The following example shows how to retrieve a sub-field based on its name:

```
>>> from netzob.all import *
>>> f1 = Field("hello", name="f1")
>>> f2 = Field("hello", name="f3")
>>> f3 = Field("hello", name="f2")
>>> fheader = Field("hello", name="fheader")
>>> fheader.fields = [f1, f2, f3]
>>> fheader.getField('f2')
f2
>>> type(fheader.getField('f2'))
<class 'netzob.Model.Vocabulary.Field.Field'>
```

getSymbol()

Return the symbol to which this field is attached.

Returns The associated symbol if available.

Return type `Symbol`

Raises `NoSymbolException`

To retrieve the associated symbol, this method recursively call the parent of the current object until the root is found.

If the root is not a `Symbol`, this raises an Exception.

The following example shows how to retrieve the parent symbol from a field object:

```
>>> from netzob.all import *
>>> field = Field("hello", name="F0")
>>> symbol = Symbol([field], name="S0")
>>> field.getSymbol()
S0
>>> type(field.getSymbol())
<class 'netzob.Model.Vocabulary.Symbol.Symbol'>
```

str_structure(*deepness=0*)

Returns a string which denotes the current symbol/field definition using a tree display.

Parameters **deepness** (`int`, required) – Parameter used to specify the number of indentations.

Returns The current symbol/field represented as a string.

Return type `str`

```
>>> from netzob.all import *
>>> f1 = Field(String(), name="field1")
>>> f2 = Field(Integer(interval=(10, 100)), name="field2")
>>> f3 = Field(Raw(nbBytes=14), name="field3")
>>> symbol = Symbol([f1, f2, f3], name="symbol_name")
>>> print(symbol.str_structure())
symbol_name
|--  field1
     |--   Data (String=None ((None, None)))
|--  field2
     |--   Data (Integer=None ((10, 100)))
|--  field3
     |--   Data (Raw=None ((112, 112)))
>>> print(f1.str_structure())
field1
|--   Data (String=None ((None, None)))
```

1.8 Persistence during Specialization and Abstraction of Symbols

In the API, a memory capability is provided in order to support relationships between variables, as well as variable persistence during the specialization and abstraction processes. This capability is described in the Memory class.

class Memory

This class provides a memory, used to store variable values (in bitarray) in a persisting and independent way.

To compute or verify the constraints and relationships that participate in the definition of the fields, the Netzob library relies on a memory. This memory stores the values of previously captured or emitted fields. More precisely, the Memory contains all the field variables that are needed according to the field definition during the abstraction and specialization processes.

memorize (*variable*, *value*)

Memorizes the provided variable value.

Parameters

- **variable** (`Variable`, required) – The variable for which we want to memorize a value.
- **value** (`bitarray`, required) – The value to memorize.

```
>>> from netzob.all import *
>>> variable = Data(String(), name="var1")
>>> memory = Memory()
>>> memory.memorize(variable, String("hello").value)
>>> print(memory)
Data (String=None ((None, None))): b'hello'
```

hasValue (*variable*)

Returns true if the memory contains a value for the provided variable.

Parameters **variable** (`Variable`, required) – The variable to look for in the memory.

Returns `True` if the memory contains a value for the variable.

Return type `bool`

```
>>> from netzob.all import *
>>> variable = Data(String(), name="var1")
>>> memory = Memory()
>>> memory.memorize(variable, String("hello").value)
>>> memory.hasValue(variable)
True
>>> variable2 = Data(String(), name="var2")
>>> memory.hasValue(variable2)
False
```

getValue (*variable*)

Returns the value memorized for the provided variable.

Parameters **variable** (`Variable`, required) – The variable for which we want to retrieve the value in memory.

Returns The value in memory.

Return type `bitarray`

```
>>> from netzob.all import *
>>> variable = Data(String(), name="var1")
>>> memory = Memory()
>>> memory.memorize(variable, String("hello").value)
>>> memory.getValue(variable).tobytes()
b'hello'
```

forget (*variable*)

Forgets any memorized value of the provided variable

Parameters **variable** (`Variable`, required) – The variable for which we want to forget the value in memory.

```
>>> from netzob.all import *
>>> variable = Data(String(), name="var1")
>>> memory = Memory()
>>> memory.memorize(variable, String("hello").value)
>>> memory.hasValue(variable)
True
>>> memory.forget(variable)
>>> memory.hasValue(variable)
False
```

duplicate ()

Duplicates the current memory in a new memory.

Returns A new memory containing the same entries than the current memory.

Return type *Memory*

```
>>> from netzob.all import *
>>> d1 = Data(Integer())
>>> d2 = Data(String())
>>> m = Memory()
>>> m.memorize(d1, Integer(100).value)
>>> m.memorize(d2, String("hello").value)
>>> m.getValue(d1)
bitarray('01100100')
>>> m2 = m.duplicate()
>>> m2.getValue(d1)
bitarray('01100100')
>>> m.getValue(d1).bytereverse()
>>> m.getValue(d1)
bitarray('00100110')
>>> m2.getValue(d1)
bitarray('01100100')
```


©2017 AMOSSYS

The values of variables defined in fields can have different assignment strategies, depending on their persistence and lifecycle. Four assignment strategies are available, in order to describe:

- **Constant values**.
- **Persistent values**.
- **Ephemeral values**.
- **Volatile values**.

The SVAS class provides a description of those strategies, along with some examples.

class SVAS

This class represents the Assignment Strategy of a variable.

The State Variable Assignment Strategy (SVAS) of a variable defines how its value is used while abstracting and specializing, and therefore impacts the memorization strategy.

A *SVAS* strategy can be attached to a variable and is used both when abstracting and specializing. A SVAS strategy describes the set of memory operations that must be performed each time a variable is abstracted or specialized. These operations can be separated into two groups, those used during the abstraction and those used during the specialization.

The available SVAS strategies for a variable are:

- SVAS.CONSTANT
- SVAS.EPHEMERAL (the default strategy for variables)
- SVAS.VOLATILE
- SVAS.PERSISTENT

Those strategies are explained below. Besides some following examples are shown in order to understand how the strategies can be applied during abstraction and specialization of Field with Data variables.

- **SVAS.CONSTANT**: A constant value denotes a static content defined once and for all in the protocol. When abstracting, the concrete value is compared with the symbolic value which is a constant and succeeds only if it matches. On the other hand, the specialization of a constant value does not imply any additional operations than just using the value as is. A typical example of a constant value is a magic number in a protocol or a delimiter field.

  The following example shows the **abstraction of a constant data**, through the parsing of a message that corresponds to the expected model:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> value = String("john").value
>>> f.domain = Data(String(), originalValue=value, svas=SVAS.CONSTANT)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
```

```
>>> Symbol.abstract("john", [s], memory=m)
(S0, OrderedDict([('f1', b'john')]))
```

The following example shows that the abstraction of a data that does not correspond to the expected model returns an unknown symbol:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> value = String("john").value
>>> f.domain = Data(String(), originalValue=value, svas=SVAS.CONSTANT)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
>>> Symbol.abstract("kurt", [s], memory=m)
(Unknown message 'kurt', OrderedDict())
```

The following example shows the **specialization of a constant data**:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> value = String("john").value
>>> f.domain = Data(String(), originalValue=value, svas=SVAS.CONSTANT)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
>>> s.specialize(memory=m)
b'john'
>>> s.specialize(memory=m)
b'john'
>>> len(str(m))
0
```

The following example shows that the specialization of a constant data raises an exception when no original value is attached to the definition domain of the variable:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> f.domain = Data(String(nbChars=(5, 10)), svas=SVAS.CONSTANT)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
>>> s.specialize(memory=m)
Traceback (most recent call last):
  ...
Exception: Cannot specialize this symbol.
```

- **SVAS.PERSISTENT**: A persistent value carries a value, such as a session identifier, generated and memorized during its first specialization and reused as such in the remainder of the session. Conversely, the first time such persistent field is abstracted, its variable's value is not defined and the received value is saved. Later in the session, if this field is abstracted again, the corresponding variable is then defined and we compare the received field value against the memorized one.

The following example shows the **abstraction of a persistent data**:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> f.domain = Data(String(nbChars=(5, 10)), svas=SVAS.PERSISTENT)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
>>> Symbol.abstract("dylan", [s], memory=m)
```

```
(S0, OrderedDict([('f1', b'dylan')]))
>>> Symbol.abstract("dylan", [s], memory=m)
(S0, OrderedDict([('f1', b'dylan')]))
```

The following example shows that the abstraction of a persistent data that does not correspond to the expected model returns a unknown symbol:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> f.domain = Data(String(nbChars=(5, 10)), svas=SVAS.PERSISTENT)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
>>> Symbol.abstract("kurt", [s], memory=m)
(Unknown message 'kurt', OrderedDict())
```

The following examples show the **specialization of a persistent data**:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> value = String("john").value
>>> f.domain = Data(String(), originalValue=value, svas=SVAS.PERSISTENT)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
>>> s.specialize(memory=m)
b'john'
>>> len(str(m))
0
```

```
>>> from netzob.all import *
>>> f = Field()
>>> f.domain = Data(String(nbChars=5), svas=SVAS.PERSISTENT)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
>>> generated1 = s.specialize(memory=m)
>>> len(generated1)
5
>>> m.hasValue(f.domain)
True
>>> generated2 = s.specialize(memory=m)
>>> len(generated2)
5
>>> generated1 == generated2
True
```

- **SVAS.EPHEMERAL**: The value of an ephemeral variable is regenerated each time it is specialized. The generated value is memorized, and can then be used afterwards to abstract or specialize other fields. During abstraction, the value of this field is always learned for the same reason. For example, the IRC *nick* command includes such an ephemeral field that denotes the new nick name of the user. This nick name can afterward be used in other fields but whenever a NICK command is emitted, its value is regenerated.

  The following example shows the **abstraction of an ephemeral data**:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> f.domain = Data(String(nbChars=(4, 10)), svas=SVAS.EPHEMERAL)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
```

```
>>> Symbol.abstract("john", [s], memory=m)
(S0, OrderedDict([('f1', b'john')]))
>>> print(m)
Data (String=None ((32, 80))): b'john'
>>> Symbol.abstract("john", [s], memory=m)
(S0, OrderedDict([('f1', b'john')]))
>>> print(m)
Data (String=None ((32, 80))): b'john'
>>> Symbol.abstract("kurt", [s], memory=m)
(S0, OrderedDict([('f1', b'kurt')]))
>>> print(m)
Data (String=None ((32, 80))): b'kurt'
```

The following examples show the **specialization of an ephemeral data**:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> value = String("john").value
>>> f.domain = Data(String(), originalValue=value, svas=SVAS.EPHEMERAL)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
>>> m.hasValue(f.domain)
False
>>> generated1 = s.specialize(memory=m)
>>> m.hasValue(f.domain)
True
>>> generated2 = s.specialize(memory=m)
>>> generated1 == generated2
False
```

- **SVAS.VOLATILE**: A volatile variable denotes a value which changes whenever it is specialized and that is never memorized. It can be seen as an optimization of an ephemeral variable to reduce the memory usages. Thus, the abstraction process of such field only verifies that the received value complies with the field definition domain without memorizing it. For example, a size field or a CRC field is a volatile field.

  The following example shows the **abstraction of a volatile data**:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> f.domain = Data(String(nbChars=(4, 10)), svas=SVAS.VOLATILE)
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
>>> Symbol.abstract("john", [s], memory=m)
(S0, OrderedDict([('f1', b'john')]))
>>> len(m)
0
>>> Symbol.abstract("john", [s], memory=m)
(S0, OrderedDict([('f1', b'john')]))
>>> len(m)
0
>>> Symbol.abstract("kurt", [s], memory=m)
(S0, OrderedDict([('f1', b'kurt')]))
>>> len(m)
0
```

  The following example shows the **specialization of a volatile data**:

```
>>> from netzob.all import *
>>> f = Field(name='f1')
>>> f.domain = Data(String(nbChars=(5,10)), svas=SVAS.VOLATILE)
```

```
>>> s = Symbol(name="S0", fields=[f])
>>> m = Memory()
>>> m.hasValue(f.domain)
False
>>> generated = s.specialize(memory=m)
>>> m.hasValue(f.domain)
False
```

2 Sending and Receiving Messages

This chapter covers the following requirements: 40.

2.1 Underlying Concepts

In the Netzob library, a **communication channel** is an element allowing a connection to a remote device. Generally, if the device is connected with Ethernet network, the channel includes a socket object and all the properties used to configure it. The channel also provides the connection status and send/receive APIs.

An abstraction component, called **AbstractionLayer** (see *AbstractionLayer*), enables the specialization of a symbol in a contextualized concrete message, and the abstraction of a received message into a symbol. A **Memory** (see *Memory*) is used to keep track of a context for a specific communication. This memory can leverage variable from the protocol or even the environment. The memory is initialized at the beginning of the communication, and its internal state evolves throughout the exchanged messages.

Those elements are described in this chapter.

2.2 Communication Channel API

Each communication channel provides the following API:

class AbstractChannel

A communication channel is an element allowing to establish a connection to or from a remote device.

The AbstractChannel defines the API of a communication channel.

A communication channel provides the following public variables:

Variables

- **isOpen** (bool) – The status of the communication channel.
- **timeout** (int) – The default timeout in seconds for opening a connection and waiting for a message.
- **header** (*Symbol*) – A Symbol that permits to access to the protocol header.
- **header_presets** (dict, optional) – A dictionary of keys:values used to preset (parameterize) the header fields during symbol specialization. See *Symbol.specialize* for more information.

setSendLimit (*maxValue*)
Change the max number of writings.

When it is reached, no packet can be sent anymore until `clearSendLimit()` is called.

If maxValue is -1, the sending limit is deactivated.

Parameters **maxValue** (`int`) – the new max value

clearSendLimit ()
Reset the writing counters.

write (*data*, *rate=None*, *duration=None*)
Write to the communication channel the specified data, either one time, either in loop according to the *rate* and *duration* parameter.

Parameters

- **data** (`bytes`, required) – The data to write on the channel.
- **rate** (`int`, optional) – This specifies the bandwidth in octets to respect during traffic emission (should be used with duration= parameter).
- **duration** (`int`, optional) – This tells how much seconds the data is continuously written on the channel.

Returns The amount of written data, in bytes.

Return type `int`

checkReceived (*predicate*, **args*, ***kwargs*)
Method used to delegate the validation of the received data into a callback

Parameters

- **predicate** (`Callable[[bytes], bool]`) – the function used to validate the received data
- **args** – positional arguments passed to `predicate`
- **kwargs** – named arguments passed to `predicate`

close ()
Close the communication channel.

open (*timeout=None*)
Open the communication channel. If the channel is a server, it starts to listen for incoming data.

Parameters **timeout** (`float`, optional) – The default timeout of the channel for opening connection and waiting for a message. Default value is blocking (None).

read()

Read the next message from the communication channel.

Returns The received data.

Return type `bytes`

sendReceive(*data*)

Write on the communication channel the specified data, wait for a response and return the received data.

Parameters **data** (`bytes`) – The data to write on the channel.

Returns The received data.

Return type `bytes`

2.3 Available Communication Channels

The available communication channels are the following:

- *RawEthernetChannel*: this channel sends/receives Ethernet frames (with Ethernet header computed by this channel).
- *RawIPChannel*: this channel sends/receives IP payloads (with IP header computed by this channel).
- *IPChannel*: this channel sends/receives IP payloads (with IP header computed by the OS kernel).
- *UDPClient*: this channel provides the connection of a client to a specific IP:Port server over a UDP socket.
- *TCPClient*: this channel provides the connection of a client to a specific IP:Port server over a TCP socket.
- *UDPServer*: this channel provides a server listening to a specific IP:Port over a UDP socket.
- *TCPServer*: this channel provides a server listening to a specific IP:Port over a TCP socket.
- *SSLClient*: this channel provides the connection of a client to a specific IP:Port server over a TCP/SSL socket.
- *DebugChannel*: this channel provides a way to log I/O's into a specific stream

Each communication channel is described in the next sub-chapters.

2.3.1 RawEthernetChannel channel

This chapter covers the following requirements: 63.

class RawEthernetChannel(*interface*, *remoteMac=None*, *localMac=None*, *timeout=None*)

A RawEthernetChannel is a communication channel to send Ethernet frames. This channel is responsible for building the Ethernet layer.

The RawEthernetChannel constructor expects some parameters:

Parameters

- **interface** (str, required) – The local network interface name (such as 'eth0', 'lo').
- **remoteMac** (str, required) – The remote MAC address to connect to.
- **localMac** (str, required) – The local MAC address.
- **timeout** (float, optional) – The default timeout of the channel for global connection. Default value is blocking (None).

Adding to AbstractChannel variables, the RawEthernetChannel class provides the following public variables:

Variables

- **remoteMac** (str) – The remote MAC address to connect to.
- **localMac** (str) – The local MAC address.
- **interface** (str) – The network Interface name such as 'eth0', 'lo', determined with the local MAC address. Read only variable.

```
>>> from netzob.all import *
>>> from binascii import hexlify
>>> client = RawEthernetChannel(interface="lo",
...                              remoteMac="00:01:02:03:04:05",
...                              localMac="00:06:07:08:09:10")
>>> client.open()
>>> symbol = Symbol([Field("ABC")])
>>> client.write(symbol.specialize())
17
>>> client.close()
```

2.3.2 RawIPChannel channel

This chapter covers the following requirements: 64, 65, 66.

class RawIPChannel(*remoteIP*, *localIP=None*, *upperProtocol=6*, *timeout=None*)

A RawIPChannel is a communication channel to send IP payloads. This **channel** is responsible to build the IP header. It is similar to *IPChannel* channel, except that with

IPChannel the OS kernel builds the IP header. Therefore, with RawIPChannel, we **can** modify or fuzz the IP header fields.

The RawIPChannel constructor expects some parameters:

Parameters

- **remoteIP** (str, required) – The remote IP address to connect to.
- **localIP** (str, optional) – The local IP address. Default value is the local IP address corresponding to the network interface that will be used to send the packet.
- **upperProtocol** (int, optional) – The protocol following IP in the stack. Default value is `socket.IPPROTO_TCP` (6).
- **timeout** (float, optional) – The default timeout of the channel for global connection. Default value is blocking (None).

Adding to AbstractChannel variables, the RawIPChannel class provides the following public variables:

Variables

- **remoteIP** (str) – The remote IP address to connect to.
- **localIP** (str) – The local IP address. Default value is the local IP address corresponding to the interface that will be used to send the packet.
- **upperProtocol** (int) – The protocol following the IP header. Default value is socket.IPPROTO_TCP.

The following code shows the use of a *RawIPChannel* channel:

```
>>> from netzob.all import *
>>> client = RawIPChannel(remoteIP='127.0.0.1', timeout=1.)
>>> client.open()
>>> symbol = Symbol([Field("Hello everyone!")])
>>> client.write(symbol.specialize())
>>> client.close()
```

2.3.3 IPChannel channel

This chapter covers the following requirements: 64, 65, 66.

class IPChannel(*remoteIP*, *localIP=None*, *upperProtocol=6*, *timeout=None*)

An IPChannel is a communication channel to send IP payloads. The **kernel** is responsible to build the IP header. It is similar to RawIPChannel channel, except that with RawIPChannel the channel builds the IP header. Therefore, with *IPChannel*, we **cannot** modify or fuzz the IP header fields.

The IPChannel constructor expects some parameters:

Parameters

- **remoteIP** (str, required) – The remote IP address to connect to.
- **localIP** (str, optional) – The local IP address. Default value is the local IP address corresponding to the interface that will be used to send the packet.
- **upperProtocol** (int, optional) – The protocol following the IP header. Default value is socket.IPPROTO_TCP.
- **timeout** (float, optional) – The default timeout of the channel for global connection. Default value is blocking (None).

Adding to AbstractChannel variables, the IPChannel class provides the following public variables:

Variables

- **remoteIP** (str) – The remote IP address to connect to.
- **localIP** (str) – The local IP address. Default value is the local IP address corresponding to the interface that will be used to send the packet.
- **upperProtocol** (int) – The protocol following the IP header. Default value is socket.IPPROTO_TCP.

The following code shows the use of an IPChannel channel:

```
>>> from netzob.all import *
>>> client = IPChannel(remoteIP='127.0.0.1', timeout=1.)
>>> client.open()
>>> symbol = Symbol([Field("Hello everyone!")])
>>> client.write(symbol.specialize())
>>> client.close()
```

2.3.4 UDPClient channel

This chapter covers the following requirements: 67, 68.

class UDPClient(*remoteIP*, *remotePort*, *localIP=None*, *localPort=None*, *timeout=None*)

A UDPClient is a communication channel. It provides the connection of a client to a specific IP:Port server over a UDP socket.

When the actor executes an OpenChannelTransition, it calls the open method on the UDP client which connects to the server.

The UDPClient constructor expects some parameters:

Parameters

- **remoteIP** (str, required) – The remote IP address to connect to.

- **remotePort** (`int`, required) – The remote IP port.
- **localIP** (`str`, optional) – The local IP address. Default value is the local IP address corresponding to the network interface that will be used to send the packet.
- **localPort** (`int`, optional) – The local IP port. Default value in a random valid integer chosen by the kernel.
- **timeout** (`float`, optional) – The default timeout of the channel for global connection. Default value is blocking (None).

Adding to AbstractChannel variables, the UDPClient class provides the following public variables:

Variables

- **remoteIP** (`str`) – The remote IP address to connect to.
- **remotePort** (`int`) – The remote IP port.
- **localIP** (`str`) – The local IP address. Default value is the local IP address corresponding to the network interface that will be used to send the packet.
- **localPort** (`int`) – The local IP port. Default value in a random valid integer chosen by the kernel.

The following code shows the use of a UDPClient channel:

```
>>> from netzob.all import *
>>> client = UDPClient(remoteIP='127.0.0.1', remotePort=9999, timeout=1.)
>>> client.open()
>>> symbol = Symbol([Field("Hello everyone!")])
>>> client.write(symbol.specialize())
15
>>> client.close()
```

2.3.5 TCPClient channel

This chapter covers the following requirements: 71, 72, 73.

class TCPClient(*remoteIP*, *remotePort*, *localIP=None*, *localPort=None*, *timeout=None*)

A TCPClient is a communication channel. It provides the connection of a client to a specific IP:Port server over a TCP socket.

When the actor executes an OpenChannelTransition, it calls the open method on the TCP client which connects to the server.

The TCPClient constructor expects some parameters:

Parameters

- **remoteIP** (`str`, required) – The remote IP address to connect to.
- **remotePort** (`int`, required) – The remote IP port.
- **localIP** (`str`, optional) – The local IP address. Default value is the local IP address corresponding to the network interface that will be used to send the packet.
- **localPort** (`int`, optional) – The local IP port. Default value in a random valid integer chosen by the kernel.
- **timeout** (`float`, optional) – The default timeout of the channel for global connection. Default value is blocking (None).

Adding to AbstractChannel variables, the TCPClient class provides the following public variables:

Variables

- **remoteIP** (`str`) – The remote IP address to connect to.
- **remotePort** (`int`) – The remote IP port.
- **localIP** (`str`) – The local IP address. Default value is the local IP address corresponding to the network interface that will be used to send the packet.
- **localPort** (`int`) – The local IP port. Default value in a random valid integer chosen by the kernel.

The following code shows the creation of a TCPClient channel:

```
>>> from netzob.all import *
>>> client = TCPClient(remoteIP='127.0.0.1', remotePort=9999, timeout=1.)
```

2.3.6 UDPServer channel

This chapter covers the following requirements: 67, 68, 69.

class UDPServer(*localIP*, *localPort*, *timeout=None*)

A UDPServer is a communication channel. It provides a server listening to a specific IP:Port over a UDP socket.

When the actor executes an OpenChannelTransition, it calls the open method on the UDP server which makes it to listen for incoming messages.

The UDPServer constructor expects some parameters:

Parameters

- **localIP** (`str`, required) – The local IP address.
- **localPort** (`int`, required) – The local IP port.

- **timeout** (float, optional) – The default timeout of the channel for global connection. Default value is blocking (None).

Adding to AbstractChannel variables, the UDPClient class provides the following public variables:

Variables

- **localIP** (str) – The local IP address.
- **localPort** (int) – The local IP port.

The following code shows the use of a UDPServer channel:

```
>>> from netzob.all import *
>>> server = UDPServer(localIP='127.0.0.1', localPort=9999, timeout=1.)
>>> server.open()
>>> server.close()
```

2.3.7 TCPServer channel

This chapter covers the following requirements: 71, 72, 73.

class TCPServer(*localIP*, *localPort*, *timeout=None*)

A TCPServer is a communication channel. It provides a server listening to a specified IP:Port over a TCP socket.

When the actor executes an OpenChannelTransition, it calls the open method on the tcp server which starts the server. The objective of the server is to wait for the client to connect.

The TCPServer constructor expects some parameters:

Parameters

- **localIP** (str, required) – The local IP address.
- **localPort** (int, required) – The local IP port.
- **timeout** (float, optional) – The default timeout of the channel for global connection. Default value is blocking (None).

Adding to AbstractChannel variables, the TCPServer class provides the following public variables:

Variables

- **localIP** (str) – The local IP address.
- **localPort** (int) – The local IP port.

The following code shows the creation of a TCPServer channel:

```
>>> from netzob.all import *
>>> server = TCPServer(localIP='127.0.0.1', localPort=9999, timeout=1.)
```

2.3.8 SSLClient channel

class SSLClient(*remoteIP*, *remotePort*, *localIP=None*, *localPort=None*, *server_cert_file=None*, *alpn_protocols=None*, *timeout=None*)

An SSLClient is a communication channel that relies on SSL. It provides the connection of a client to a specific IP:Port server over a TCP/SSL socket.

When the actor executes an OpenChannelTransition, it calls the open method on the ssl client which connects to the server.

The SSLClient constructor expects some parameters:

Parameters

- **remoteIP** (`str`, required) – The remote IP address to connect to.
- **remotePort** (`int`, required) – The remote IP port.
- **localIP** (`str`, optional) – The local IP address. Default value is the local IP address corresponding to the network interface that will be used to send the packet.
- **localPort** (`int`, optional) – The local IP port. Default value in a random valid integer chosen by the kernel.
- **server_cert_file** (`str`, optional) – The path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. Default value is None, meaning that no verification is made on the certificate given by the peer.
- **alpn_protocols** (`list`, optional) – Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like ['http/1.1', 'spdy/2'], ordered by preference. Default value is None.
- **timeout** (`float`, optional) – The default timeout of the channel for global connection. Default value is blocking (None).

Adding to AbstractChannel public variables, the SSLClient class provides the following public variables:

Variables

- **remoteIP** (`str`) – The remote IP address to connect to.
- **remotePort** (`int`) – The remote IP port.

- **localIP** (str) – The local IP address. Default value is the local IP address corresponding to the network interface that will be used to send the packet.
- **localPort** (int) – The local IP port. Default value in a random valid integer chosen by the kernel.
- **server_cert_file** (str) – The path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. Default value is None, meaning that no verification is made on the certificate given by the peer.
- **alpn_protocols** (list) – Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like ['http/1.1', 'spdy/2'], ordered by preference. Default value is None.

The following code shows the creation of a SSLClient channel:

```
>>> from netzob.all import *
>>> server = SSLClient(remoteIP='127.0.0.1', remotePort=9999)
```

2.3.9 DebugChannel channel

class DebugChannel(*stream*, *timeout=None*)

An DebugChannel is a file-like channel that handle writing of output data.

The DebugChannel constructor expects some parameters:

Parameters

- **stream** (str or a file-like object, required) – The output stream
- **timeout** (float, optional) – The default timeout of the channel for global connection. Default value is blocking (None).

The following code shows the use of an DebugChannel channel:

```
>>> from netzob.all import *
>>> client = DebugChannel("/dev/null")
>>> symbol = Symbol([Field("Hello everyone!")])
>>> with client:
...     client.write(symbol.specialize())
18
```

2.4 Abstraction Layer

In the API, the component responsible for translating concrete messages into symbols, or for converting symbols into concrete messages, is the AbstractionLayer.

class AbstractionLayer(*channel*, *symbols*, *memory=None*)

An abstraction layer specializes a symbol into a message before emitting it and, on the other way, abstracts a received message into a symbol.

The AbstractionLayer constructor expects some parameters:

Parameters

- **channel** (`AbstractChannel`, required) – The underlying communication channel (such as IPChannel, UDPClient. . .).
- **symbols** (a `list` of `Symbol`, required) – The list of permitted symbols during translation from/to concrete messages.
- **memory** (`Memory`, optional) – A memory used to store variable values during specialization and abstraction of successive symbols, especially to handle inter-symbol relationships. If None, a locale memory is created by default and used internally.

The AbstractionLayer class provides the following public variables:

Variables

- **channel** (`AbstractChannel`) – The underlying communication channel (such as IPChannel, UDPClient. . .).
- **symbols** (a `list` of `Symbol`) – The list of permitted symbols during translation from/to concrete messages.
- **memory** (`Memory`) – A memory used to store variable values during specialization and abstraction of successive symbols, especially to handle inter-symbol relationships. If None, a locale memory is created by default and used internally.

Usage example of the abstraction layer

The following code shows a usage of the abstraction layer class, where two UDP channels (client and server) are built and transport just one permitted symbol:

```
>>> from netzob.all import *
>>> symbol = Symbol([Field(b"Hello Kurt !")], name = "Symbol_Hello")
>>> channelIn = UDPServer(localIP="127.0.0.1", localPort=8889, timeout=1.)
>>> abstractionLayerIn = AbstractionLayer(channelIn, [symbol])
>>> abstractionLayerIn.openChannel()
>>> channelOut = UDPClient(remoteIP="127.0.0.1", remotePort=8889, timeout=1.)
>>> abstractionLayerOut = AbstractionLayer(channelOut, [symbol])
>>> abstractionLayerOut.openChannel()
>>> abstractionLayerOut.writeSymbol(symbol)
12
>>> (received_symbol, received_data) = abstractionLayerIn.readSymbol()
>>> received_symbol.name
'Symbol_Hello'
>>> received_data
b'Hello Kurt !'
```

Handling message flow within the abstraction layer

The following example demonstrates that the abstraction layer can also handle a message flow:

```
>>> from netzob.all import *
>>> symbolflow = Symbol([Field(b"Hello Kurt !Whats up ?")], name = "Symbol Flow")
>>> symbol1 = Symbol([Field(b"Hello Kurt !")], name = "Symbol_Hello")
>>> symbol2 = Symbol([Field(b"Whats up ?")], name = "Symbol_WUP")
>>> channelIn = UDPServer(localIP="127.0.0.1", localPort=8889, timeout=1.)
>>> abstractionLayerIn = AbstractionLayer(channelIn, [symbol1, symbol2])
>>> abstractionLayerIn.openChannel()
>>> channelOut = UDPClient(remoteIP="127.0.0.1", remotePort=8889, timeout=1.)
>>> abstractionLayerOut = AbstractionLayer(channelOut, [symbolflow])
>>> abstractionLayerOut.openChannel()
>>> abstractionLayerOut.writeSymbol(symbolflow)
22
>>> (received_symbols, received_data) = abstractionLayerIn.readSymbols()
>>> received_symbols
[Symbol_Hello, Symbol_WUP]
```

Relationships between the environment and the produced messages

The following example shows how to define a relationship between a message to send and an environment variable, then how to leverage this relationship when using the abstraction layer.

```
>>> from netzob.all import *
>>> # Environment variables definition
>>> memory1 = Memory()
>>> env1 = Data(String(), name="env1")
>>> memory1.memorize(env1, String("John").value)
>>>
>>> # Symbol definition
>>> f7 = Field(domain=String("master"), name="F7")
>>> f8 = Field(domain=String(">"), name="F8")
>>> f9 = Field(domain=Value(env1), name="F9")
>>> symbol = Symbol(fields=[f7, f8, f9], name="Symbol_Hello")
>>>
>>> # Creation of channels with dedicated abstraction layer
>>> channelIn = UDPServer(localIP="127.0.0.1", localPort=8889, timeout=1.)
>>> abstractionLayerIn = AbstractionLayer(channelIn, [symbol], memory1)
>>> abstractionLayerIn.openChannel()
>>> channelOut = UDPClient(remoteIP="127.0.0.1", remotePort=8889, timeout=1.)
>>> abstractionLayerOut = AbstractionLayer(channelOut, [symbol], memory1)
>>> abstractionLayerOut.openChannel()
>>>
>>> # Sending of a symbol containing a data coming from the environment
>>> abstractionLayerOut.writeSymbol(symbol)
11
>>> (received_symbol, received_data) = abstractionLayerIn.readSymbol()
>>> received_symbol.name
'Symbol_Hello'
>>> received_data
b'master>John'
```

`writeSymbol`(*symbol*, *rate=None*, *duration=None*, *presets=None*, *fuzz=None*)

Write the specified symbol on the communication channel after specializing it into a contextualized message.

Parameters

- **`symbol`** (*Symbol*, required) – The symbol to write on the channel.

- **rate** (`int`, optional) – This specifies the bandwidth in octets to respect during traffic emission (should be used with duration= parameter). Default value is None (no rate).
- **duration** (`int`, optional) – This tells how much seconds the symbol is continuously written on the channel. Default value is None (write only once).
- **presets** (`dict`, optional) – This specifies how to parameterize the emitted symbol. The expected content of this dict is specified in *`Symbol.specialize`*.
- **fuzz** (`Fuzz`, optional) – A fuzzing configuration used during the specialization process. Values in this configuration will override any field definition, constraints, relationship dependencies or parameterized fields. See `Fuzz` for a complete explanation of its use for fuzzing purpose.

Raise `TypeError` if parameter is not valid and Exception if an exception occurs.

`readSymbols``()`
Read a flow from the abstraction layer and abstract it in one or more consecutive symbols.

The timeout attribute of the underlying channel is important as it represents the amount of time (in seconds) above which no reception of a message triggers the reception of an *`EmptySymbol`*.

`readSymbol``()`
Read a message from the abstraction layer and abstract it into a symbol.

The timeout attribute of the underlying channel is important as it represents the amount of time (in seconds) above which no reception of a message triggers the reception of an *`EmptySymbol`*.

`openChannel``()`
Open the underlying communication channel.

`closeChannel``()`
Close the underlying communication channel.

`reset``()`
Reset the abstraction layer (i.e. its internal memory as well as the internal parsers).

Specific symbols are used in the abstraction layer to represent the absence of received symbol (EmptySymbol) and the reception of an unknown symbol (UnknownSymbol). Those symbols are described below.

class EmptySymbol

An empty symbol is a special type of symbol that represents the fact of having nothing received or to have nothing to send. An EmptySymbol is only produced by the automata, and thus should not be instanciated.

class UnknownSymbol(*message=None*)

An unknown symbol is a special type of symbol that is returned to represent a message that cannot be abstracted.

The UnknownSymbol constructor expects some parameters:

Parameters **message** (`netzob.Model.Vocabulary.Messages.AbstractMessage.AbstractMessage`, optional) – The raw message that cannot be abstracted into a symbol.

The UnknownSymbol class provides the following public variable:

Variables **message** (`netzob.Model.Vocabulary.Messages.AbstractMessage.AbstractMessage`) – The raw message that cannot be abstracted into a symbol.

```
>>> from netzob.all import *
>>> u = UnknownSymbol()
>>> u.name
"Unknown message ''"
```

2.5 Relationships between Messages and the Environment

This chapter covers the following requirements: 43, 45.

In the API, the capability to specify relationships between successive messages or between messages and the environment is provided by the `Memory` class.

Relationships between fields of successive messages

The following example shows how to define a relationship between a received message and the next message to send:

```
>>> from netzob.all import *
>>> f1 = Field(domain=String("hello"), name="F1")
>>> f2 = Field(domain=String(";"), name="F2")
>>> f3 = Field(domain=String(nbChars=(5,10)), name="F3")
>>> s1 = Symbol(fields=[f1, f2, f3], name="S1")
>>>
>>> f4 = Field(domain=String("master"), name="F4")
>>> f5 = Field(domain=String(">"), name="F5")
>>> f6 = Field(domain=Value(f3), name="F6")
>>> s2 = Symbol(fields=[f4, f5, f6])
>>>
>>> memory = Memory()
>>> m1 = s1.specialize(memory=memory)
>>> m2 = s2.specialize(memory=memory)
>>>
>>> m1[6:] == m2[7:]
True
```

Relationships between a message field and the environment

The following example shows how to define a relationship between a message to send and an environment variable:

```
>>> from netzob.all import *
>>> # Environment variables definition
>>> memory = Memory()
>>> env1 = Data(String(), name="env1")
>>> memory.memorize(env1, String("John").value)
>>>
>>> # Symbol definition
>>> f7 = Field(domain=String("master"), name="F7")
>>> f8 = Field(domain=String(">"), name="F8")
>>> f9 = Field(domain=Value(env1), name="F9")
>>> s3 = Symbol(fields=[f7, f8, f9])
>>>
>>> # Symbol specialization
>>> s3.specialize(memory=memory)
b'master>John'
```