HL7apy: a Python library to parse, create and handle HL7 v2.x messages

Vittorio Meloni[1], **Alessandro Sulis**[1], **Daniela Ghironi**[2], **Francesco Cabras**[1], **Mauro Del Rio**[1], **Stefano Monni**[1], **Massimo Gaggero**[1], **Francesca Frexia**[1] **Gianluigi Zanetti**[1]

[1] CRS4, Pula, Italy

[2] Inpeco SA, Lugano, Switzerland

Abstract

HL7 version 2 is the most popular messaging standard for clinical systems interoperability. Most of the tools for messaging management are Java or .NET based, while Python programming language lacks of comparable solutions. This paper describes HL7apy, an open-source HL7 v2 compliant messaging library, written in Python. The library offers means to create, parse, navigate and validate messages. As an example application, we present a full implementation of the IHE Patient Demographics Query ITI-21 transaction. The resulting module has been integrated in GNU Health, a popular open-source Hospital Information System.

Keywords

HL7, Python, API, Interoperability

Correspondence to:

Vittorio Meloni
CRS4
Address: Loc. Piscina Manna, Edificio 1 - 09010 Pula (CA)
Email: vittorio.meloni@crs4.it

EJBI 2015; 11(2):en31–en40
recieved: October 30, 2014
accepted: January 9, 2015
published: January 20, 2015

1 Introduction

HL7 (Health Level 7) is a well-known and widely used standard for the exchange, integration, sharing and retrieval of electronic health information. It supports clinical practice and the management, delivery and evaluation of health services. "Health Level Seven International" [1], founded in 1987, is the main organization responsible for HL7 development and maintenance. It defines several standards, grouped into reference categories.

In this paper we describe HL7apy, a new Python package to manage HL7 v2.x messages. It uses Python natural terseness to express and operate HL7 messages in a concise manner. We expect it to be useful for fast protyping of HL7-compliant software and potentially for the development of full applications. As an example, Listing 1 contrasts the code required to create the part of a message using HL7apy to the code required to perform the same task with Java HAPI [2], currently the most popular library for HL7 messaging.

```
# Python version

adt_a01 = Message("ADT_A01")
adt_a01.msh.sending_application.hd_1 = \
  "Sending App"
adt_a01.msh.sequence_number = "123"

adt_a01.pid.patient_name = "Doe^John"
adt_a01.pid.patient_identifier_list = "123456"

// Java version

ADT_A01 adt = new ADT_A01();
adt.initQuickstart("ADT", "A01", "P");

MSH mshSegment = adt.getMSH();
mshSegment.getSendingApplication().
 getNamespaceID().setValue("Sending App");
mshSegment.getSequenceNumber().setValue("123");

PID pid = adt.getPID();
pid.getPatientName(0).getFamilyName().
  getSurname().setValue("Doe");
pid.getPatientName(0).getGivenName().
  setValue("John");
pid.getPatientIdentifierList(0).getID().
  setValue("123456");
```

Listing 1: A comparison of the code needed to create the same message from scratch using HAPI and HL7apy. The message example and the Java code are taken from HAPI official examples.

1.1 HL7 Standards

The HL7 standard includes different versions that were developed in different periods of time and with different purposes:

- **HL7 version 2 (HL7 v2)** [3]: it is the older messaging standard; it allows the exchange of clinical data between systems and it is designed to support both central and distributed patient care systems;
- **HL7 version 3 (HL7 v3)** [4]: created with a completely different philosophy from v2; it proposes a new approach for data exchange, based on a Reference Information Model (RIM) and XML.
- **HL7 FHIR (Fast Healthcare Interoperability Resources)** [5]: it has been developed with the aim to simplify and accelerate HL7 adoption by being easily consumable but robust, and by using open Internet standards where possible [6].

HL7 v2 is used worldwide to solve interoperability problems, although the "raw structure" of its messages is less human readable and machine computable than v3, which is XML-based; furthermore, v2 is still the reference version of the IHE (Integrating Healthcare Enterprise) consortium [7]. In part this is due to the fact that HL7 v3 (and the RIM in particular), in spite of ten years of development, is still a work-in-progress undergoing intense discussions on its design [8]. FHIR has been developed to overcome these issues but it is a young standard and it will need more years of development to be an effective tool.

HL7apy focuses on HL7 v2 messaging standard.

1.2 HL7 Messaging Tools

The increasing diffusion of the HL7 v2 standard has spurred the development of several software libraries aimed at simplifying raw messages management. The most popular open-source libraries available are the following.

- **HAPI**: a Java-based HL7 v2 library providing classes for messages parsing, creation and validation. Both parser and validator strictly follow the XML message structure provided by the standard;
- **NHapi** [9]: a porting of HAPI for the Microsoft .NET framework;
- **python-hl7** [10]: a minimalistic Python HL7 v2 messages parsing library that implements basic functionality without validation. It includes the implementation of a simple MLLP[1] (Minimal Lower Layer Protocol) client for sending messages.

[1]MLLP protocol is the minimalistic OSI-session layer framing protocol used to send HL7 messages

In addition to software libraries, HL7 messaging functionality is also provided by data integration software, which are integration gateways that support multiple data formats and connectors. Two of the main tools are the following.

- **Mirth Connect** [11]: an open-source healthcare integration engine specifically designed for HL7 message integration, written in Java. It provides all tools to build integration channels able to connect a wide range of data sources. It also provides tools for HL7 message parsing and validation;
- **Interfaceware/Iguana** [12]: a commercial software for the exchange, transformation and parsing of HL7 messages, providing a mean to map message fields, transform them and validate messages.

1.3 HL7apy

The diffusion of the Python programming language has been increasing over the years [13], particularly in the scientific domain [14]. The reasons for this popularity may be attributable to the language being relatively easy to learn and offering high programmer productivity. A recent study [15] indicates that with scripting language, designing and writing the program takes no more than half as much time as writing it in C, C++ or Java and the resulting program is only half as long, making it a good choice for fast prototyping. Despite Python's popularity, it is missing a feature-complete library for HL7 messaging. The aforementioned python-hl7 only implements basic features such as message parsing and ER7-encoding, while lacking important functionality such as message validation, custom separators support and structured parsing according to HL7 messaging schemas or custom message profiles.

Our motivation for the development of HL7apy comes from all these factors. The library main functions are messages creation, parsing and validation; it supports HL7 key features like custom encoding characters, message profiles and Z-elements.

The rest of the paper is structured as follows. Section 2 describes the library architecture and its main functionality; Section 3 summarizes the current features and briefly describes a real use case HL7 module implemented with HL7apy; Section 4 presents conclusions and planned future developments.

2 Methods

This section introduces all the major components of HL7apy.

Figure 1 shows the overall architecture of the library: it is composed by two utilities scripts, that generate python modules for every HL7 v2 minor version (XSD Parser) and serialized files for message profiles usage (Message Profiles Parser), and by the inner components that create

and manage messages (Core classes), parse ER7-encoded messages (Message Parser) and validate messages (Validator).

First we will introduce the utilities that are provided with the library then, we will explain its inner components.

2.1 Utilities

HL7apy includes utility scripts that are used to create concise descriptions of HL7 messages structures, needed by the rest of the library. These utilities are:

- XSD Parser
- Message Profiles Parser.

2.1.1 XSD Parser

The XSD Parser processes all the HL7 XML schema files and generates a set of Python modules, one for each HL7 v2 minor version.

The schemas are XML documents provided by HL7 International organization itself. They contain the lists of all elements (messages, segments, fields and datatypes) and, for each one of those, they describe their children with cardinality and datatype (the latter only in case of fields and complex datatypes). These files can be used for HL7 validation by third-party libraries and applications. As an example, Listing 2 shows a snippet of the XML structure of the ADT_A01 message defined in the ADT_A01.xsd file and its representation in HL7apy.

```
<!-- XSD Schema definitions -->

...
<xsd:complexType name="ADT_A01.CONTENT">
  <xsd:sequence>
    <xsd:element ref="MSH" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element ref="SFT" minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element ref="EVN" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element ref="PID" minOccurs="1"
      maxOccurs="1"/>
    ...
    <xsd:element ref="ADT_A01.PROCEDURE"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="ADT_A01.INSURANCE"
      minOccurs="0" maxOccurs="unbounded"/>
    ...
  </xsd:sequence>
</xsd:complexType>
...
<xsd:complexType name="ADT_A01.PROCEDURE.CONTENT
    ">
  <xsd:sequence>
    <xsd:element ref="PR1" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element ref="ROL" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="ADT_A01.INSURANCE.CONTENT
    ">
  <xsd:sequence>
    <xsd:element ref="IN1" minOccurs="1"
      maxOccurs="1"/>
    <xsd:element ref="IN2" minOccurs="0"
      maxOccurs="1"/>
    <xsd:element ref="IN3" minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element ref="ROL" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

# HL7apy representation

{"ADT_A01": ("sequence",
  (("MSH", (1, 1)),
   ("SFT", (0, -1)),
   ("EVN", (1, 1)),
   ("PID", (1, 1)),
   ("ADT_A01_PROCEDURE", (0, -1)),
   ("ADT_A01_INSURANCE", (0, -1)))),
 "ADT_A01_INSURANCE": ("sequence",
  (("IN1", (1, 1)),
   ("IN2", (0, 1)),
   ("IN3", (0, -1)),
   ("ROL", (0, -1)))),
 "ADT_A01_PROCEDURE": ("sequence",
  (("PR1", (1, 1)),
   ("ROL", (0, -1)))),
}
```

Listing 2: A snippet of XSD schema for ADT_A01 message and its HL7apy representation.

The code generated by the XSD parser is used by the core classes and is the foundation of the entire library.

2.1.2 Message Profiles Parser

This utility compiles an XML message profile in a more *pythonic* format. This strategy is similar to what is done with the XSD Parser, though in this case the output are not Python modules but *cPickled*[2] serialized files that can be dynamically loaded at runtime. The Message Profiles Parser should be run every time an interoperability scenario requires a particular profile.

The concept of Message Profile was introduced for the first time in HL7 v2.5, which stated that it is "an unambiguous specification of one or more standard HL7 messages that have been analyzed for a particular use case" [16].

HL7apy can create message profiles by using the static definition [16] of the message profile in XML format. The parser takes as input a static definition in XML and produces a file containing the structure for every message it defines. The outcomes are similar to the ones produced by the XSD Parser with one main difference: the structures of the children are all included within the parent's and they are not expressed using a reference. The reason for this is that every single element in the static definition can potentially specify a different cardinality, length

[2] cPickle is a Python module that supports serialization and deserialization of Python objects

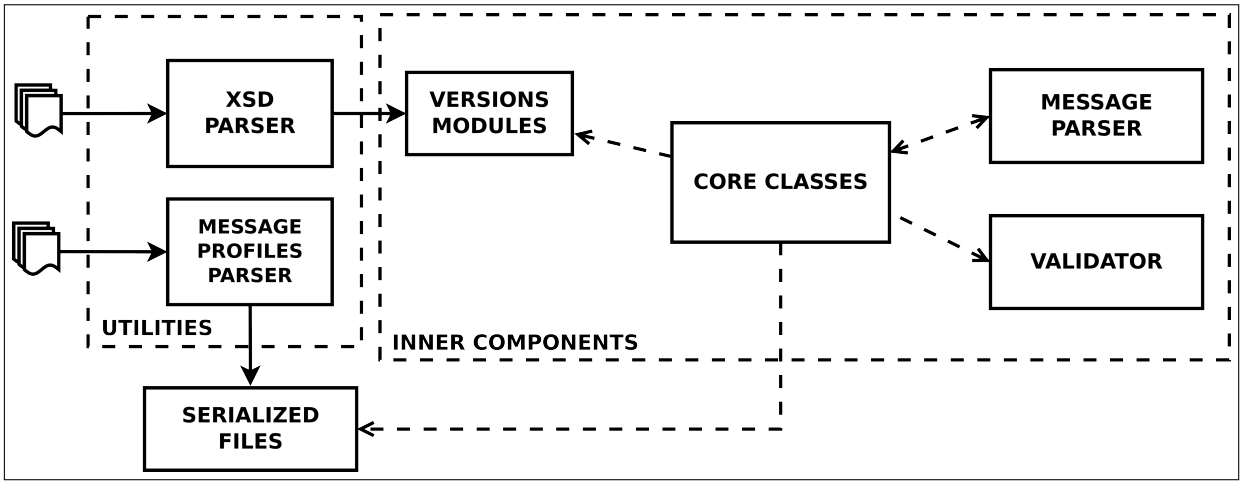

Figure 1: HL7apy overall architecture

or datatype than the same element of another message in the profile. For instance, consider the two snippets in Listing 3.

```
...
<Segment Name="PID" Usage="R" Min="1" Max="1">
  <Field Name="Patient ID" Usage="X" Min="0"
    Max="*" Datatype="CX" Length="1904">
  ...

...
<Segment Name="PID" Usage="R" Min="1" Max="1">
  <Field Name="Patient ID" Usage="X" Min="0"
    Max="*" Datatype="CX" Length="20">
  ...
```

Listing 3: A snippet of a message profile with two definitions of the Patiend ID field. The definitions specify different length for the same field.

The two *Patient ID* field versions have different lengths, so it is impossible to use one PID definition for all the messages of the profile.

Listing 4 shows an example of the IHE PDQ message profile and its HL7apy representation.

```
<!-- Message Profiles XML definition -->

...
<HL7v2xStaticDef MsgType="RSP" EventType="K22"
    MsgStructID="RSP_K21" EventDesc="RSP - Get
    person demographics response" Role="Sender">
  <MetaData Name="" OrgName="IHE" Version="2.4"
      Status="DRAFT" Topics="confsig-IHE-2.5-
      static-RSP-K22-null-RSP_K22-2.3-DRAFT-
      Sender"/>
  <Segment Name="MSH" LongName="Message Header"
      Usage="R" Min="1" Max="1">
    <Field Name="Field Separator" Usage="R" Min
        ="1" Max="1" Datatype="ST" Length="1"
        ItemNo="00001">
      <Reference>2.15.9.1</Reference>
    </Field>
    <Field Name="Encoding Characters" Usage="R"
        Min="1" Max="1" Datatype="ST" Length="4"
        ItemNo="00002">
      <Reference>2.15.9.2</Reference>
    </Field>
    <Field Name="Sending Application" Usage="R"
        Min="1" Max="1" Datatype="HD" Length
        ="180" Table="0361" ItemNo="00003">
      <Reference>2.15.9.3</Reference>
      <Component Table="0300" Name="namespace ID
          " Usage="R" Datatype="IS" Length="20"/>
      <Component Name="universal ID" Usage="C"
          Datatype="ST" Length="199"/>
      <Component Name="universal ID type" Usage
          ="C" Datatype="ID" Length="6" Table
          ="0301" />
    </Field>
...

# HL7apy representation

{"RSP_K21": ("mp",
  "sequence",
  "RSP_K21",
  (("mp",
    "sequence",
    "MSH",
    (("mp", "leaf", "MSH_1", (), (1, 1),
       "Field", "ST", 1, None),
     ("mp", "leaf", "MSH_2", (), (1, 1),
       "Field", "ST", 4, None),
     ("mp",
      "sequence",
      "MSH_3",
      (("mp", "leaf", "HD_1", (), (1, 1),
        "Component", "IS", 20, "HL70300"),
       ("mp", "leaf", "HD_2", (), (0, 1),
        "Component", "ST", 199, None),
       ("mp", "leaf", "HD_3", (), (0, 1),
        "Component", "ID", 6, "HL70301")),
      (1, 1),
      "Field",
      "HD",
      180,
      "HL70361"),

   ...
```

Listing 4: A snippet of the IHE PDQ message profile and its representation in HL7apy.

2.2 Inner Components

In this section we detail the inner components of the library, which are:

- Core Classes
- Validator
- Message Parser

2.2.1 Core Classes

The core classes offer an API to create HL7-compliant messages, navigate their structure and manipulate HL7 elements, thanks to a tree-like representation of the element relations (e.g., a Message can contain only instances of Segments or Groups, a Group can contain Segment instances only, etc.). These classes allow the developers to express operations in a very compact form, as already shown in Listing 1.

The library defines the following classes to represent all the HL7 elements.

- Message
- Group
- Segment
- Field
- Component
- SubComponent
- Base datatype classes (e.g., ST, DT, FT, etc.)

Figure 2 illustrates the main classes and their relationships. We can notice two other classes, apart from ones listed above: the `ElementFinder`, used to search element's structure in the minor version's modules, and the `ElementProxy`, used during the elements' navigation.

The next sections illustrate the main operations that can be performed using the core classes.

Elements Instantiation. The developer can instantiate HL7 elements simply by specifying their structure and/or version (Listing 5).

```
from hl7apy.core import Message, Segment,
    SubComponent

adt_a01 = Message("ADT_A01", version="2.5")
ins = adt_a01.add_group("ADT_A01_INSURANCE")

pid = Segment("PID")

s = SubComponent(datatype="FT")
s.value = FT("some information")
```

Listing 5: Examples of element instantiation.

Under the hood, the helper class `ElementFinder` is used by the core classes to retrieve the element definitions described in 2.1.1, thus enabling validation and traversal of children.

As soon as the Message is instantiated, the MSH segment is automatically created and some of its required fields are populated with default values (e.g., default separators for MSH-1 and MSH-2 fields).

Alternatively, one can specify a message profile as the reference of the `Message` at instantiation (Listing 6).

```
mp = hl7apy.load_message_profile("./pdq")
m = Message("RSP_K21", reference=mp["RSP_K21"])
```

Listing 6: Instantiation specifying a message profile.

It is also possible to create custom elements (Z-elements), as long as they follow the correct naming convention.

```
segment = Segment("ZIN")
field = Field("ZIN_1")
```

Listing 7: Instantiation of Z-elements.

To be more flexible, the library allows the creation of HL7 elements without specifying their structure. In this case, the message cannot be considered validated according to the HL7 schemas. The validation process is described in detail in Section 2.2.2. The Listing 8 shows the instantiation of a custom field that is added to a PID segment.

```
from hl7apy.core import Segment, Field

segment = Segment("PID")
unkn_field = Field()
segment.add(unkn_field)
```

Listing 8: Instantiation of custom elements.

Element Navigation. Since the library exposes a DOM-like API, the developer can easily access the children of a given element by simply using their name, description or position.

```
from hl7apy.core import Message, Segment, Field

s = Segment("PID")
s.value = "PID|||654321^^^123456||" \
  "Family^Name^^^^^"

# by name, it refers to a Field instance
print s.pid_5

# by description, it refers to a Field instance
print s.patient_name

# by position, it refers to a Component instance
print s.pid_5.pid_5_1
```

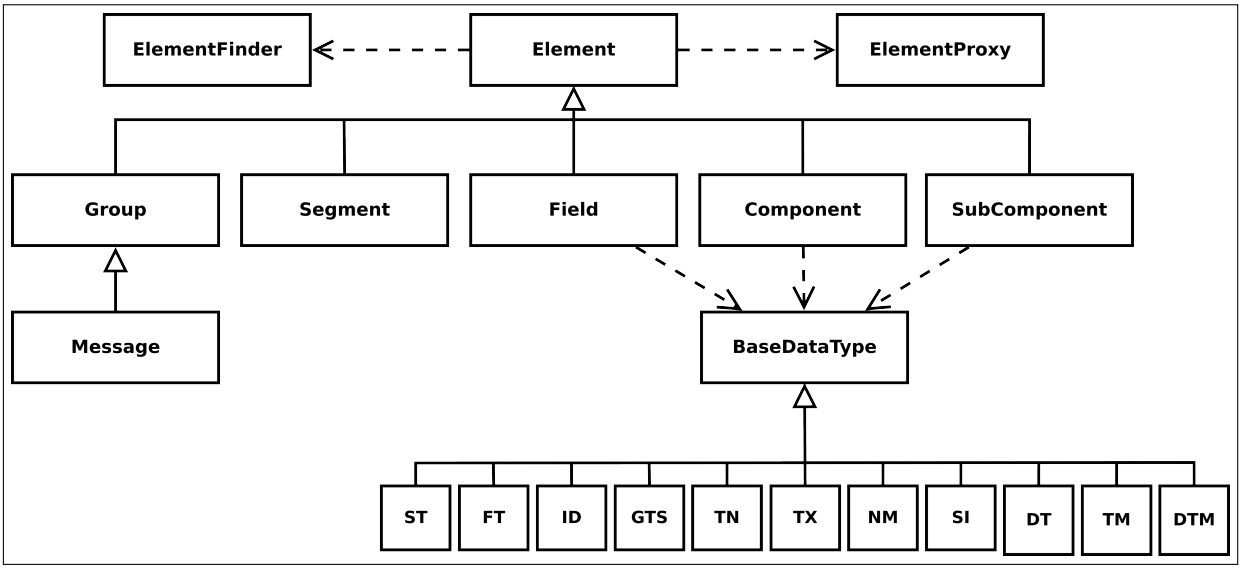

Figure 2: The architecture of HL7apy core. The main classes are shown

```
message = Message("RSP_K21")

# by description, recursively on the message
# children
print message.msh.date_time_of_message.time

# iterates over PID-5 children of the PID
# segment
for name in s.pid_5:
  print name

# iterates over all the fields of the PID
# segment
for child in s.children:
  print child

# its datatype is CX
org_5 = Field("org_5")
org_5.value = "^^^^^^^^^AG&&DEP"

# it returns the tenth component of the field,
# it is the same as org_5.cx_10
print org_5.org_5_10

# it returns the third subcomponent of the tenth
# component of the field, it is the same as
# org_5.cx_10.cwe_3
print org_5.org_5_10_3
```

Listing 9: Elements navigation. An element can be accessed by name, description or position.

When accessing a child element list without specifying an index, the library, by means of the `ElementProxy` class, automatically returns the first child. Other child elements can be accessed by specifying the appropriate index.

```
# it is the same as s.pid_13[0]
print s.pid_3.to_er7()

# if it exists, it returns the second
# instance of pid_13
print s.pid_3[1].to_er7()
```

Listing 10: Access to elements by index. If an index is not specified the library returns the first child. Other child elements can be retrieved by using the appropriate index.

Elements Population. For convenience, it is possible to populate an element or its children by:

- assigning the ER7 representation,
- calling the dedicated parsing functions,
- copying another element instance,
- assigning the base datatype value (e.g., a string, a number, etc.),
- assigning a base datatype instance.

```
m = Message("ADT_A01", version="2.5")

# base datatype value (string)
m.msh.msh_3 = "GHH_ADT"

# it will create to an instance of
# DTM base datatype
m.msh.msh_7 = "20080115153000"

# ER7 representation, MSH_9 is a complex
# datatype of 3 components
m.msh.msh_9 = "ADT^A01^ADT_A01"

# copy from another element
m.evn.evn_2 = m.msh.msh_7

# parser function
m.msh.msh_9 = hl7apy.parser.\
  parse_field("ADT^A01^ADT_A01", name="MSH_9")

s = SubComponent(datatype="IS")
```



```
# base datatype instance (IS)
s.value = IS("AAA")
```

Listing 11: Examples of elements population.

Element Encoding. The developer can generate the ER7-encoded string of a core class instance using both default or custom encoding characters (Listing 12). In the case of Message class it is also possible to generate the MLLP-encoded string.

```
from hl7apy.core import Message
from hl7apy.parser import parse_field

custom_chars = {
  "FIELD": "!",
  "COMPONENT": "@",
  "SUBCOMPONENT": "%",
  "REPETITION": "~",
  "ESCAPE": "\$"
}

msh_9 = "ADT^A01^ADT_A01"
field = parse_field(msh_9)

# it will use default encoding chars
print field.to_er7()

# it will use custom encoding chars
# defined above
print field.to_er7(encoding_chars=custom_chars)

m = Message('RSP_K21')
print repr(m.to_mllp())
```

Listing 12: Elements encoding in ER7/MLLP form. The developer can also specify custom encoding characters.

2.2.2 Validator

One of the most important features implemented in HL7apy is the validation of messages, since it ensures their compliance with the standard for the specific message type and HL7 minor version.

In an ideal world every message would adhere to the HL7 specification; however, real-life applications encounter messages that do not conform. Common issues are for example fields with more components than expected or messages with prohibited segments. For this reason HL7apy implements two levels of validation: `STRICT` and `TOLERANT`.

When a message is created using `STRICT` validation, the library verifies the exact adherence of the message to its message type. In particular, it checks that:

- all the expected elements are present;
- there are no unexpected or unknown elements;
- the cardinality of all elements is correct;
- the datatypes of the fields, components and subcomponents are correct.

On the other hand, the `TOLERANT` validation level is more permissive and allows some operations like:

- instantiating unknown elements;
- changing the default datatype of a field, component or subcomponent;
- ignoring the cardinality of the elements (e.g., inserting more identical elements than allowed or missing a required element).

Naturally, some operations are not allowed in `TOLERANT` mode either. For instance, it is not possible to insert a PID-1 field into an SPM segment.

Validation is performed in two phases. The first one checks that message creation and population do not violate the rules of the chosen mode. As soon as an error occurs, an exception is raised (e.g., when in `STRICT` mode it is inserted an unexpected segment to a message or when it is created an unknown element). The second phase must be forced by the developer using the `Validator` class.

The `Validator` class performs a `STRICT` validation of an `Element`. Its `validate()` method accepts an `Element` object and validates it against its HL7 structure or against a message profile, if specified in input. In particular it verifies element's cardinality, datatypes correctness and emits warnings, which are minor errors that don't invalidate the message (i.e., HL7apy doesn't raise an exception) but should be fixed to guarantee its complete compliance. Examples of errors in this category are:

- fields that exceed their maximum value;
- fields with a value not in their HL7 table.

Warnings and errors can be gathered together in a report file by explicitly requesting it at validation time. This feature can be especially useful to diagnose and resolve errors in the interoperability testing phase. For instance, the functionality can be used to verify the conformance of the system to an IHE profile.

It is worth noting that the `Validator` checks for the presence of issues that cannot be detected during the first phase, in particular, the absence of required elements. Thus it is wrong to consider a message completely valid without using the `Validator`.

2.2.3 Message Parser

The Message Parser is the module that provides all the functionality needed to parse an HL7 message encoded in the ER7 format. The parsing is started by the `parse_message` function which takes an ER7 string as input. The string is interpreted according to the encoding characters specified in the MSH-1 and MSH-2 fields. The parsing of sub-elements is delegated to purpose-specific functions (e.g., `parse_segment`, `parse_field` and so on). Every function generates an instance of the core classes and attaches it to the correct parent object, resulting in

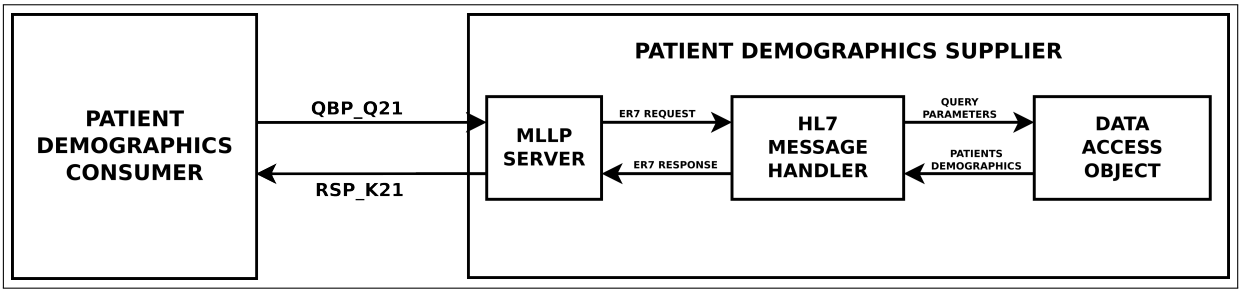

Figure 3: PDQ transaction diagram. The PDQ Supplier shows the components included in the PDQ module

.

the tree structure of the message. The reference structure of the message is obtained from the MSH-9 field.

The parser allows the caller to specify the desired validation level, the message profile to use, if necessary, and the name of the report file the `Validator` will produce in case of `STRICT` validation. It is also possible to specify a flag that makes the parser create groups and assign the segments as children of the group to which they belong, as stated in the message schema.

All the parser functions are called by the core classes in case of ER7 string assignment as shown in Listing 13.

```
m.msh.msh_9 = "ADT^A01^ADT_A01" # parse_field
m.evn = "EVN||20080115153000||AAA" \
  "|AAA|20080114003000" # parse_segment
m.evn.evn_5.xcn_1 = "AAA" # parse_component
```

Listing 13: Assigment of string as elements' value. The parser functions are called by the core classes in case of string assignment

3 Results

HL7apy supports the creation of HL7-compliant systems using the Python programming language. The library implements the following features.

- HL7 versions from 2.2 to 2.6 support
- Message Parsing
- Message Validation
- ER7 Encoding
- Custom Encoding Characters support
- Message Profile support
- Z-elements support
- Simple and Complex Datatype support
- HL7 tables support

With respect to the state of the art library (HAPI) we do not support HL7 v2.1 and XML encoding.

3.1 Testing HL7apy Message Types Coverage

In order to test HL7apy ability to parse different message types, we applied to a random sample of 1000 messages from IHE Gazelle [17]. We only used messages validated by Gazelle (marked as passed). We removed 4 messages from the set since they were using non ASCII/UTF-8 encoding characters, a feature currently not supported by HL7apy. The resulting dataset distributes messages within 12 different message types (e.g., ADT, QBP). Table 1 reports the results of message parsing using the two supported validation levels.

Table 1: HL7 messages coverage results. Abbreviations: v = valid, i = invalid, e = errors.

		tolerant			strict		
type	**tot**	**v**	**i**	**e**	**v**	**i**	**e**
QBP	74	74	0	0	31	43	0
ADT	425	420	0	5	240	183	2
SIU	4	4	0	0	4	0	0
OUL	24	24	0	0	16	8	0
ACK	47	47	0	0	28	19	0
ORU	22	22	0	0	9	13	0
ORR	16	16	0	0	3	13	0
OML	106	106	0	0	69	37	0
ORL	47	47	0	0	34	13	0
ORM	171	171	0	0	6	165	0
RSP	58	58	0	0	20	38	0
QCN	2	2	0	0	2	0	0

The errors reported on the ADT row derive from the fact that HL7apy does not currently support segment repetition outside of a group. It is interesting to note that some of these messages are rejected by `STRICT` before failing. All other messages are parsed without exception when setting the validation level to `TOLERANT`. The messages rejected by `STRICT` are correctly parsed when the parser is configured with the appropriate message profile. For instance, using the IHE ITI-21 message profile results in the acceptance of all the PDQ request messages (QBP^Q22^QBP_Q21).

3.2 Use Case Implementation

As a significant example of a real-world problem, we used HL7apy to implement the PDQ Supplier actor of the IHE ITI-21 Patient Demographics Query (PDQ) transaction [18, 19]. PDQ allows clinical systems to query a central patient demographics server with the purpose of retrieving patients' demographic information and it is one of the most used IHE transactions. As shown in Figure 3, the transaction involves two actors, PDQ Consumer and Supplier, with two exchanged messages: QPB_Q21 for requests and RSP_K21 for responses.

The PDQ Supplier receives query messages from one or more consumers and returns demographic information for all patients matching the query criteria. The module includes a network MLLP service for sending and receiving HL7 messages. All the PDQ Supplier components are shown in Figure 3. The module has been integrated in GNU Health [20], a worldwide used open-source Hospital Information System.

The module is structured as follows:

- the MLLP Server receives an HL7 PDQ request message (QBP_Q21) from a PDQ Consumer and redirects it to the Message Handler (MH);
- the MH parses and validates the message using the specific profile, extracts the query parameters and checks their compliance to the PDQ specifications. Query parameters are provided in one or more QPD_3 field repetitions. Each repetition has two components, the first indicating the parameter (e.g., name, surname, date of birth, etc) as coded by IHE, and the second specifying the value. For example, if the consumer wants to search all patients with name 'John' and surname 'Smith' the QPD_3 should be filled as '@PID.5.2^John~@PID.5.1.1^Smith';
- once the parameters are extracted, they are sent to the Data Access Object (DAO) which queries the demographic database to get the corresponding patient information and returns them to the MH;
- the MH creates the correct HL7 response message (RSP_K21) and sends it to the MLLP Server that forwards it to the PDQ Consumer.

HL7apy provides a standard MLLP server implementation through the MLLPServer class that needs to be specialized by providing the appropriate message handlers. Listing 14 shows the usage of the class in the PDQ module. In this case only one handler is necessary (PDQHandler).

```
from hl7apy.mllp import MLLPServer
from .pdq_supplier import PDQHandler

s = MLLPServer(host='localhost', port=6789,
               handlers={'QBP_Q21': PDQHandler})
s.serve_forever()
```

Listing 14: Usage of the MLLPServer class provided by HL7apy

When the server receives a QBP_Q21 message, forwards it to the PDQHandler class, whose implementation is shown in Listing 15. This class is the core of the module: it parses the request message, extracts the query parameters and gets patients information using the DAO. Finally it builds the response message and sends it back to the consumer.

```
import datetime, uuid

from hl7apy.v2_5 import DTM
from hl7apy.utils import check_date
from hl7apy.mllp import
    AbstractTransactionHandler
from hl7apy.parser import parse_message
from hl7apy.core import Message

from .dao import DAO
from .profiles import PDQ_REQ_MP, PDQ_RES_MP
from .parameters import PDQ_FIELD_NAMES

class PDQHandler(AbstractTransactionHandler):

  REQ_MP, RES_MP = PDQ_REQ_MP, PDQ_RES_MP
  FN = PDQ_FIELD_NAMES

  def __init__(self, message):
    self.dao = DAO()
    msg = parse_message(message,
                        message_profile=self.
                            REQ_MP)
    super(PDQHandler, self).__init__(msg)

  def _create_res(self, ack_code,
                  query_ack_code, patients):
    res = Message('RSP_K21',
                  reference=self.RES_MP)
    r, q = res.msh, self.msg.msh
    r.msh_5, r.msh_6 = q.msh_3, q.msh_4
    res.msh.msh_5 = self.msg.msh.msh_3
    res.msh.msh_6 = self.msg.msh.msh_4
    r.msh_7.ts_1 = DTM(datetime.datetime.now())
    r.msh_9 = 'RSP^K22^RSP_K21'
    r.msh_10 = uuid.uuid4().hex

    r, q = res.msa, self.msg.msh
    r.msa_1 = ack_code
    r.msa_2 = q.msh_10.msh_10_1

    r, q = res.qak, self.msg.qpd
    r.qak_1 = q.qpd_2
    r.qak_2 = ('OK'
               if len(patients) > 0 else 'NF')
    r.qak_4 = str(len(patients))

    res.qpd = self.msg.qpd

    g = res.add_group('rsp_k21_query_response')
    for i, p in enumerate(patients):
      g.add_segment('PID')
      g.pid[i].pid_1 = str(i)
      g.pid[i].pid_5 = "%s^%s" % (p[0], p[1])
    return res

  def _create_err(self, code, desc):
    res = self._create_res('AR', 'AR', [])
    res.ERR.ERR_1, res.ERR.ERR_2 = code, desc
    return res

  def reply(self):
```

```
params = dict(
   (self.FN[q.qip_1.value], q.qip_2.value)
   for q in self.msg.qpd.qpd_3
   if q.qip_1.value in self.FN)
if ('' in params.values() or
   (params.has_key('DOB') and
    not check_date(params.get('DOB')))):
  res = self._create_err(
       "100", "Invalid params")
else:
  p = self.dao.get_data(params)
  if len(p) > 0:
    res = self._create_res('AA', 'OK', p)
  else:
    res = self._create_res('AA', 'NF', p)
return res.to_mllp()
```

Listing 15: PDQHandler implementation

4 Conclusions and Future Work

In this paper we presented HL7apy, an HL7 v2 Python library whose main goal is to provide a pythonic way for handling HL7 messages.

The library is available at `https://github.com/crs4/hl7apy/tree/ihic2015` and it is released under the MIT License (MIT). Currently, it supports Python version 2.7.

In the near future we plan to add support for XML messages encoding, HL7 versions 2.7 and 2.8 and Python 3.

The website with the documentation can be reached at `http://hl7apy.org`.

5 Acknowldgments

The authors are grateful to the GNU Health community for their kind cooperation.

References

[1] HL7. HL7 Level Seven International. c2007-2014. Available from `http://www.hl7.org`.

[2] HAPI. University Health Network; c2001-2014. cited 2014 October 15. Available from `http://hl7api.sourceforge.net`.

[3] HL7 Version 2 Product Suite. HL7 Level Seven International. c2007-2014. Updated 2014 Oct 10; cited 2014 Oct 20. Available from `http://www.hl7.org/implement/standards/product_brief.cfm?product_id=185`.

[4] HL7 Version 3 Product Suite. HL7 Level Seven International. c2007-2014. Updated 2014 Oct 10; cited 2014 Oct 20. Available from `http://www.hl7.org/implement/standards/product_brief.cfm?product_id=186`.

[5] FHIR. HL7.org. c2011+. Updated 2014 Oct 20; cited 2014 Oct 20. Available from `http://hl7.org/implement/standards/fhir/`.

[6] D. Bender, K. Sartipi. HL7 FHIR: An Agile and RESTful Approach to Healthcare Information Exchange. Computer-Based Medical Systems (CBMS), 2013 IEEE 26th International Symposium on. 2013;326-331.

[7] IHE.net. IHE International; c2013. Available from `https//www.ihe.net`

[8] B. Smith, W. Ceusters. HL7 RIM: an incoherent standard. Studies in health technology and informatics. 2006;124:133-8.

[9] NHAPI. Available from `http://nhapi.sourceforge.net`.

[10] python-hl7. John Paulette; c2011 cited 2014 October 15. `http://python-hl7.readthedocs.org/en/latest/`.

[11] Mirth. Quality System, Inc.; c1994-2014. Available from `http://www.mirthcorp.com`.

[12] Iguana. Interfaceware Inc.; c2014. Available from `http://www.interfaceware.com/iguana.html`.

[13] TIOBE Software. TIOBE Software BV; c2014. Updated 2014 Oct 10; cited 2014 Oct 20 Available from `http://www.tiobe.com/index.php/paperinfo/tpci/Python.html`.

[14] K. Millman, M. Aivazis. Python for Scientists and Engineers. Computing in Science and Engineering. 2011;13:9–12.

[15] L. Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl. Computer. 2000;33;23-29.

[16] Health Level Seven, Inc. HL7 Messaging Standard version 2.5 - An application protocol for electronic standard exchange, 2003. Ann Arbor MI USA; 2003.

[17] Gazelle. IHE International; c2010-2014. Available from `http://gazelle.ihe.net`

[18] IHE International, Inc. IHE IT Infrastructure Technical Framework - Volume 1 $ITITF - 1$: Integration Profiles. 2014. Available from `http://ihe.net/uploadedFiles/Documents/ITI/IHE_ITI_TF_Vol1.pdf`.

[19] IHE International, Inc. IHE IT Infrastructure Technical Framework, Volume 2a $ITITF - 2a$: Transactions Part A. 2014. Available from `http://www.ihe.net/uploadedFiles/Documents/ITI/IHE_ITI_TF_Vol2a.pdf`.

[20] GNU Health. GNU Solidario; c2011. Available from `http://health.gnu.org`.

