The MEDICI Electronic Data Interchange Library

S-Mart.NET

David Coles

About MEDICI

MEDICI is an Electronic Data Interchange (EDI) library. EDIFACT (ISO 9735), ANSI ASC X12 and UNGTDI (TRADACOMS) syntaxes are currently supported. MEDICI provides a parser for EDI streams and functions to query EDI guidelines for validation, reference information and design notes. MEDICI does not dictate how EDI guidelines should be implemented, but provides an API which can be used to interrogate external implementations and an example guideline implementation for internal use.

The operation of the MEDICI parser is roughly modelled after the Expat XML library.

If you would like improved X12 support please consider obtaining a copy of the specification to donate to the MEDICI project.

1. Introduction

1.1. Introduction to EDI

Electronic Data Interchange (EDI) is the generic term given to a selection of standards used for the application-to-application exchange of business information. It defines machine-readable formats for the encoding of order placement, shipping notification and invoicing, among many other business transactions.

The goal of EDI is to help to realise the dream of the paperless office by providing a direct electronic channel by which to carry out all aspects of business-to-business transactions. Amongst the many advantages that it provides, possibly the most useful are that it eliminates the need for re-keying of orders and invoices sent via telephone, fax or email, and improves the accuracy and speed of communications between companies.

The extent to which EDI lives up to these goals is debatable. There are numerous different standards which vary between countries and industry sectors. Industry bodies may add their own message types

and codes which are not part of the core standards, so these extensions must be supported and understood by all parties in such transactions. EDI standards (generally) only define the application level rules for structuring the transactions, and not the protocols by which the transactions are communicated. This has led to the proliferation of a number of so-called *Value Added Networks* (VANs) which provide communication between companies via numerous gateways such as modem dial-up connections and Internet protocols such as FTP.

Organisations such as large corporations and government agencies (eg. city councils) may insist on the use of EDI as a prerequisite for trading, or may provide preferable terms for partners who can communicate with them electronically. EDI software is frequently expensive, complex, and may require considerable customisation by consultancy services. The cost of transmitting interchanges via a VAN is usually many magnitudes that which we have become used to on the global IP network. Consequently, EDI is not often used to its full advantage by *Small and Medium-sized Enterprises* (SMEs).

1.2. Free/Open-source EDI software

The freely available EDI packages which I have managed to find seem to fall into one of three catagories:

- Java-based application suites, often accessed via a web based application server interface. This requires a significant degree of supporting infrastructure, not least a machine capable of handling a Java virtual machine. This would almost certainly require a dedicated server and administrative resources for the care and feeding of the application.
- Platform or language specific applications. Really I am refering to Microsoft Windows or COBOL based suites. The Windows based packages are presumably intended to appeal to smaller businesses which can dedicate a Wintel box to EDI tasks and can hand-feed messages or at least supervise the retrieval and processing of them. Alternately larger organisations which have mainframe installations and a team of programmers to hand may be attracted by a COBOL package.
- Task-specific scripts. These generally transform the EDI data to another format such as XML. This allows the client to understand the messages at a higher level, but it does not give it sufficient data to explore the EDI message domain. This rather clumsy, jargon-oriented sentence is basically trying to say "It doesn’t help if you want to write an integrated development environment around EDI".

Whilst looking for a package which could suit the needs of my organisation, I felt there was a gap in the range of open-source software on offer. We need to deal with a very limited range of transactions for a small number of partners. The adoption of a fully-fledged EDI suite would be overkill, and would probably result in the amount of administrative and customisation work exceeding that of the data-entry task it was meant to replace. What was really needed was a lightweight, flexible API to quickly extract the goodness from a small set of messages.

Ideally I would be able to use an API from within Perl applications. This would enable us to integrate the package with the rest of our process. The package would be lightweight enough to be run reasonably frequently from a range of containers. It would be simple to embed the functionality into a number of different processes which would use it in various environments.

Note: There seem to be a couple more projects which are closer to what I was looking for. However they are libraries written in Java, which isn't very helpful for a Perl environment.

1.3. MEDICI

The goal of MEDICI is to produce a library of code which allows EDI applications to be written with the complexities of the various standards hidden from the author. It achieves this by abstracting the details of the EDI stream behind a set of objects which can be easily manipulated by the programmer through method calls. Bindings to various scripting languages will give users the power to quickly assemble applications which understand EDI messages and integrate them into existing components.

MEDICI operates as a stream-oriented parser similar to the Expat library for XML documents. The client application creates an instance of a parser object, registers callbacks with it, and then feeds it the EDI document. As the parser decodes the stream the callbacks are triggered as structural events are encountered. MEDICI takes care of ensuring that the document meets the appropriate specifications, and can pragmatically deal with violations according to the policies prescribed by the client application.

Through callbacks, MEDICI passes references to data dictionaries and context information which allows the application to understand the structure of a message and the semantics of the component parts.

At a later date it is the intention to also provide an application framework utilising MEDICI which can provide some level of support for conducting business based on EDI interchanges, but for now this is left as an excercise for the reader.

1.4. MEDICI Features

- Supports EDIFACT (ISO 9735), UNGTDI (TRADACOMS) and ANSI ASC X12 syntaxes.
- Support for interrogating EDI guidelines.
- Automatic syntax detection in parser.
- Message validation through external transaction set guideline implementations.
- Lightweight, embeddable and extensible.
- Free/Open-source. LGPL licensing model allows you to link applications against MEDICI without having to open your application (although you are of course encouraged to do so).
- Sets of bindings (in development) allow you to use MEDICI from within your favourite language(s).
- A stream-based approach allows documents to be parsed on-line; no memory hit is taken for an application which simply routes EDI messages based on envelope data, for example.

2. Building and Installing

MEDICI is known to compile on:

- Linux 2.x (GCC & Intel C++ Compiler 7.1)
- Solaris 2.6 (GCC)
- IRIX 6.5 (MIPSpro & GCC)
- NetBSD 1.5 / OpenBSD 2.9 / FreeBSD 4.8 (GCC)
- Darwin 6.2 (GCC)
- Windows NT 4.0 (Borland C++ 4.0 & GCC/Cygwin)

The code is pure ANSI C, so it should compile with any modern OS/compiler combination.

2.1. Building

MEDICI is supplied as a GNU gzip compressed tar file from http://www.disintegration.org/~david/. Unpack the distribution, **cd** into the directory and follow the instructions for your platform below.

2.1.1. Building under Unix (or GNU)

Run the configure shell script to produce Makefiles and run make:

./configure
make

And optionally:

make test

2.1.2. Building under Windows

If anyone has any experience with Windows compilation I would be grateful for a description of the process or Makefile equivalents.

2.1.3. Building on an unsupported platform

The `src` directory contains all the source code for the core library. Simply compile and link all the code here to create the library. There are no dependencies on external libraries.

2.2. Installation

Installation is not yet supported. Simply add the `src` directory to your compiler's include and library paths to use MEDICI in your application.

2.3. Example applications

Some (very) simple example "applications" are included in the `examples` directory. These will read a file from the command line, or a stream from `stdin`.

MEDICI only knows about the "system segments" of each EDI standard - these are detailed in the syntax specifications. The body of a message is made up of segments described in a "directory" or "transaction set guidelines" which are published by industry bodies or organisations such as the UNECE (http://unece.org/). There is no way that the library could know about all of these guidelines - there are simply too many of them - so instead MEDICI contains a set of routines for interrogating an abstraction of the guidelines. The client application provides an implementation of the guidelines which MEDICI can then use to understand the structure of the message.

The example applications use MEDICI's own internal implementions (which are used for storing information about system segments) and load in the actual data for the directory from text or XML files at runtime. These files can be downloaded from http://www.disintegration.org/~david/tsg/. There are a couple of utilities which can be used for converting publically available specifiation files in the `util` directory. See the EDIFACT and X12 support sections for details of how to use these utilities.

Transaction set guidelines can be pretty big, and the internal implementation operates very slowly with such large amounts of data (it uses linear searches on C arrays). Bear in mind that these are only example applications. A real application would provide an implementation which would, say, read the guidelines from an SQL database (possibly with caching) or from a Berkley DB file.

Sample interchanges are provided in the `samples` directory. These samples are simple TeleSmart API files and various interchanges culled from publicly available EDI documentation or contributed by users. Additional sample files may be found in Michael Koehne's XML-Edifact (http://www.xml-edifact.org/) Perl module. If you can donate any example documents, please feel more than welcome to do so.

3. Using MEDICI

Please see the programs in `examples` as the interface is not fully defined yet. These examples will be kept more up-to-date that this documentation could ever be until then.

EDI standards such as EDIFACT specify the representation of message headers and trailers, but not the structure of individual messages. These are specified in application specific entities called *directories* or

transaction set guidelines which are drafted by industry bodies and organisations such as the UN/CEFACT. MEDICI provides an interface to work with these third party directories, and includes some example directory implementations to illustrate how to integrate them into an application.

The MEDICI parser allows an application to interpret an EDI stream in a similar way to an application reading an XML stream. This means that the implicit looping structures in an EDI stream are interpreted by the parser and presented to the application as explicit “start” and “end” events. The parser also ensures that these events are balanced in the same way that a well-formed XML document nests. The intention is not to force the user into having to deal with XML (though it may make conversion to an XML format easier), but embed the logic required to deal with implicit nesting into the parser/directory objects, rather than each appliction have to deal with it individually.

3.1. EDIFACT support

EDIFACT syntax versions 1 and 2 are currently supported. Explicit nesting and repetition are not yet supported as I have no sample documents. Full support up to version 4 is inteded, but sample files will be needed for implementation.

The official UN/EDIFACT standard directories can be downloaded in ZIP format from The UNECE’s Trade Data Interchange Directory (http://www.unece.org/trade/untdid/). The **untd2t** Perl script in the `util` directory can be used to generate an XML file from these directories. The directory must first be fully extracted (ZIP files are contained within ZIP files!) and the directory structure flattened (**unzipdirs** can do this for you). Eg.:

```
[david@kang:~/medici]$ util/unzipdirs d96a.zip
[david@kang:~/medici]$ util/untd2t d96a > d96a.xml
[david@kang:~/medici]$ examples/describe -x d96a.xml samples/orders.edi
```

3.2. TRADACOMS support

The documentation from E-Centre is in Microsoft Word format. This has delayed support considerably, but it is getting there. You should find most segments are implemented, but there is no support for message grammar as yet.

You can download a half script-generated, half hand-hacked TAB separated file from the MEDICI site (http://www.disintegration.org/~david/tsg/tradacom.txt.gz). If anyone has XML/XSLT skills it should be possible to convert the E-Centre Word files to OpenOffice.org XML files and write a script to transform those.

```
[david@kang:~/medici]$ examples/describe -t tradacom.txt samples/priinf.edi
```

Note: I have recently noticed that the TRADACOMS guidelines have essentially the same elements used in various segments or composite elements with no common code, but instead the name used to refer to them implies that the element is identical. This will require an extensive manual refactoring of the current TAB file. Given this upheaval, I will convert the whole thing to the new XML format which should also make adding transaction rules easier. If you have specific TRADACOMS needs please contact me, otherwise you are at the mercy of my lethargy ;)

3.3. ANSI ASC X12 support

GuidelineXML (gXML) files for a wide range of X12 transactions can be obtained from CommerceDesk.COM (http://www.commercedesk.com/) by following the link to the DISA X12 section of the repository. The **gxml2t** Perl script in the `util` directory can be used to generate a simplified XML file from gXML which is suitable for use with the example applications. Eg.:

```
[david@kang:~/medici]$ util/gxml2t V4011_837.gxml > V4011_837.xml
[david@kang:~/medici]$ examples/describe -x V4011_837.xml samples/837x12.edi
```

Note: All X12 envelope work has been done empirically due to lack of a freely available specification. If you can donate a copy of the X12 specification to the MEDICI project, please get in touch.

Apparently CommerceDesk.COM have taken down their publically available gXML files. You can probably still get them if you use their product. More details about Guideline XML can be found at coverpages.org (http://xml.coverpages.org/gxml.html). If you look carefully you might be able to find some example, or maybe some evaluation software has them.

4. Frequently Asked Questions

I lie. Mostly they are questions which I think people might ask. A fair few are genuine though.

4.1. What is the purpose of MEDICI?

MEDICI aims to provide an object-oriented abstraction of EDI protocols. Initially it will provide parser functionality, but emitter features will be added later. It attempts to support all segment based EDI syntaxes, although EDIFACT is probably the best supported format due to the easily available message specifications. MEDICI specifically does not provide message directories (except system directories which are essential for any level of operation). However it does provide an API to deal with third party message directory implementations.

4.2. Is MEDICI intended to be be used in medical billing applications?

Not specifically, no. Although "medici" is uncomfortably close to "medicine" there is no more profound link than that. When initially writing MEDICI I was not aware of the HIPAA standards, or that X12 was heavily used in them. Of course, there are no reasons why the library cannot be used in medical applications (other than code maturity), but it is no more suited to it than any other area of commerce.

4.3. Why are there no X12 guidelines on your TSG page?

Because I have no (copyright free) machine readable descriptions of X12 transactions (unlike the wonderful UN/EDIFACT directories). If you have GuidelineXML files for your transaction set then check out the **gxml2t** script in the `utils` directory. Some assembly may be required. Feel free to mail me with questions or if you feel like contributing an IP-free TSG of your own.

4.4. Is MEDICI similar to any other parsing library?

Yes. MEDICI is modelled after "expat" the XML library written by James Clark. Please read the fine documentation for expat in lieu of documentation for MEDICI to get an idea of the structure of an application using MEDICI.

4.5. So, it can generate XML. Is that its main purpose?

No. The editoxml executable is simply an example of how to write an application using MEDICI. It can also be used for regression testing of the MEDICI library.

4.6. OK, edi2xml is really neat and I use it for my messages. Why are you guys developing MEDICI?

edi2xml *is* neat - it has one specific goal and it gets down to it with little fuss. Because of this approach it does have limitations: The XML::Edifact package only deals with EDIFACT; Specific versions of EDIFACT directories need to be installed as part of the Perl installation; It’s Perl - I want C/C++/Perl/Python/etc. bindings; It only supports XML - I don’t want to have to deal with *two* crazy formats to get at my data!; XML::Edifact only deals with conversion of messages - MEDICI provides an API to query message directories, perform various levels of validity checks and dynamically support industry specific directory specifications.

4.7. How do directories work?

Directories are represented as a struct and an associated set of functions which provide an API. A directory implementor provides a set of function pointers which implement the API by accessing

whatever back-end functionality is required. In this way the implementor can create a directory back-end based on technologies of his choice.

The functions are the typical queries which will be performed on the directory; What is a segment/element called? What are the component elements of a composite element? What does this coded value represent? etc..

4.8. What are all these "Francesco" and "Giovanni" references?

These are implementations of directories (AKA "Transaction Set Guidelines" or TSGs). I couldn't think of any objective naming scheme for these implementations, so I chose first names of members of the Medici family. Francesco is a static source-code structure which is used for system segments. Giovanni is a dynamically allocated scheme using the included abstract data types and can be instantiated with any import functions you care to write (trivial text and XML interfaces are included).

4.9. What's with the inconsitent "EDI_" and "edi_" function naming?

Initially I started off using Expat as a model for the structure of MEDICI along with it's "XML_" naming scheme. Sadly EDI standards don't tend to be as elegant or regular as XML, so I've had to shoe-horn various functions in. Consequently to minimise the impact on the client programs (`examples` directory) I have a set of "external" API functions which map onto "internal" API functions.

4.10. I want a DOM based parser. Why aren't you doing it?

DOM parsers can be (reasonably trivially) built from stream-based parsers. A stream-based parser takes no memory hit when parsing a large document. Once MEDICI is sufficiently mature we may implement a DOM based API on top of it. If you are that keen you can build one yourself using MEDICI, or convert the EDI document to XML and use an XML DOM package. Either way, get in touch and let us know that you want this functionality!

4.11. Why is it implemented in C?

Portability. Portability. Portability. MEDICI is written in pure ANSI C, so it should compile on any reasonably modern system. With MEDICI written in C it should be trivial to provide bindings for C++, Perl, Python, etc..

4.12. Why is MEDICI so slow?

Directories are largely implemented as linked lists - search times are therefore quite slow. This will improve as I replace lists with hashes or trees. Can't wait? Then write me!

The **describe** and **editoxml** examples will run slowly as they consult directories. For a more realistic example of the speed of the core library pass the stream through **elements** instead; this does not reference a directory, so you should find it runs around a couple of magnitudes faster. Externally implemented directories may be represented in any way - maybe as a Berkley DB file for example.

4.13. Why did you do this?

EDI systems are generally used by large companies which can offset the high cost of implementation with economies of scale. In my day job I was compelled to use EDI for business reasons. Rather than spend thousands of dollars buying an EDI application and hiring consultants to customise it to our legacy database system I wrote the small amount of functionality which we needed from scratch in Perl. These are rather simple-minded scripts which were developed in a purely empirical manner; ie. they work on the limited range of messages that we have received so far, but they will not cope with valid messages slightly outside of the scope of which we have experienced. MEDICI is an attempt to proactively develop a system which will replace the scripts if they become inadequate.

A. Legalese

A.1. LICENSE

A.2. TELESMART API

Value Added Network - Edifact Commands And Acknowledgements By Telesmart API Ltd and TeleOrdering Ltd

Whitaker (owners of TeleOrdering) is now part of VNU. The specification document (docs/telesmart.txt) is attributed to them. The specification document and derived works are included in this project with the kind permission of the IT Director, VNU Entertainment Media Ltd.

This project has no other connection with Telesmart, TeleOrdering, Whitaker or VNU.

Glossary

EDI

See: Electronic Data Interchange

Electronic Data Interchange

Catch-all term for the EDIFACT, TRADACOMS and X12 families of standards for transmitting business transactions electronically

SME

See: Small and Medium-sized Enterprises

Small and Medium-sized Enterprises

Recommendation 96/280/EC (http://europa.eu.int/smartapi/cgi/sga_doc?smartapi!celexapi!prod!CELEXnumdoc&lg=EN&numdoc=31996H0280& (Link to Eur-Lex)