ProtoGen/HL7 — An Implementation of HL7

Version 0.9

Gunther Schadow

Preface

This paper describes the attempt to implement HL7 at the Universitätsklinikum Steglitz in Berlin as part of a larger hospital communication project. The aim of that project is to integrate of heterogeneous computer systems which exist in the various departments of the clinic. This project requires a standardized communication protocol.

The standardization of medical communication protocols is an ongoing effort which is taken on by many different standard groups. Naturally the diverse standard groups have different scopes depending on the special field where each standard has it's origin. Since late 1990, the US standard groups have been associated in the Healthcare Informatics Standards Planing Panel (HISPP) which is coordinated by the American National Standard Institute (ANSI). It's goal is to promote the convergence of the different standards, a process which is still at the beginning. However, as early as 1987 there was one standard group whose objective was to cover a wide range of data interchange in health care, including ADT, Finance and a variety of anciliary fields, like laboratory medicin or radiology.

In Europe, the Technical Board of the European Standardization Commitee (CEN) established the Technical Committee for Medical Informatics (TC 251) in March 1990; one of it's Working Groups (WG.3) is dedicated to health care communications.

Since the implementation of HL7 as presented here is embedded into a project that is seeking the integration of different standards it is conceptualized from a point of view that may be different from those of other implementors who may concentrate on mere HL7. The difference will become evident in the methods taken to build the implementation itself as well as in the final linkage between the communications protocol and the application software. Where it seems appropriate for the implementation which is centered on HL7 to stay completely within the concepts and terminology of HL7, this project seeks to go towards further generality in order to provide concepts and methods to serve as bridges between diverging standards, which are to be released in the same field.

An implementation of a communication standard is essentially a data structure, a builder and a parser. The data structure holds the data objects and reflects their relations among each other as assumed by the standard. The builder is to produce a valid representation of the data structure that is capable of being transported via electronic data interchange media. Finally the parser transforms this representation back into the data structure.

This implementation tries to assume models and methods of todays informatics technology. It's design is object oriented and is implementeted using **C++**. The code of the implementation is produced by a compiler whose input is a database which specifies the standard. The database is in turn compiled semi-automatically by scanning the text of the HL7 standard as is released by the HL7 Working Group. This method of generating the implementation directly from the textual description has the following advantages: correctness and customizability of both, the parser/builder methods and the definition of the standard itself.

Since the specifying database as well as the program code which implements the standard is not written manually but produced by a compiler, the output is assured to be correct, unless the compiler itself is incorrect. An incorrect compiler, however, will result in erroneous code but the errors tend to occur systematically rather than accidentially and are thus discovered more easily. Thus the method of ProtoGen/HL7 can be used to generate a HL7 reference implementation. On the other hand, it does reveal problems and errors in the standard definition itself, since a compiler will not guess the intended meaning of an amiguous or obviously incorrect passage as a human programmer would probably do.

The input database for the ProtoGen/HL7 compiler is customizable and allows the user to add new data elements quite easily. For example, a special Z-segment that is used by a site can be added to the standard just by adding a single line to the database. The implementation for this special segment is generated fully automatically as is the implementation of the whole standard.

On the other hand, the methods of the parser/builder may be customized in order to provide for different encoding rules or to speed up the implementation or tune it for size. These customizations are, however, not done very often and can currently only be done with a deeper understanding of the compiler itself since this requires the compiler program to be modified. Nevertheless, there are various applications that can be worth the effort ranging from modifications to HL7 coding or processing rules up to the implementation of a completely different standard. Since the UN EDIFACT encoding rules are quite similar to those of HL7 there could be an EDIFACT module added to ProtoGen with only a little to moderate effort.

Finally, the object oriented model of the HL7 standard that this implementation assumes helps to stay compatible with the new developement in the HL7 Working Group (version 3.0) in particular as well as health care data interchange standards of the future in general, such as MEDIX.

Berlin, 20. Dezember 1996, Gunther Schadow

Address of the Author:

Gunther Schadow
Windsteiner Weg 54a,
D-14165 Berlin
Phone, FAX: +49 30 815 33 55
E-Mail: gusw@fub46.zedat.fu-berlin.de

1 Introduction

A communication protocol for medical applications tends to be rather complex, since medical communication includes a variety of diverse issues, such as ADT (admission discard transfer), accounting, reporting and documentation of findings, querying and ordering etc. In order to cope with this kind of complexity it is necessary to find methods of deriving the implementation from the specification of the standard automatically.

Besides the problem of complexity, modern software engineering has pointed out the need to automate the production and management of software. These concerns are reflected in the modern technologies by which communication protocols are implemented today, and which are formulated by the OSI standard.

In this Introduction we want to take a first look on HL7 and this implementation of it, concerning as to how we can cope efficiently with the complexity mentioned above.

1.1 A view on HL7

HL7 is a communication protocol for medical applications that tries to conform to the OSI concepts of networking. Particularly, HL7 even attaches it's name to the OSI reference model (Health Level 7) therefore it should be expected, that HL7 will easyly comply to the concepts and methods that are suggested by OSI.

Indeed, the distinctions between abstract syntax and encoding rules, are reflected in the HL7 document: For each message, an abstract message definition is given, consisting of 3 parts, which are logically related to the elements which are considered by HL7 to constitute transactions: (1) messages, (2) segments and (3) fields, 'data elements' or types. The transactions (messages) are conceptually embedded into their environment in which they occur in terms of 'trigger events'. On the other hand, there are encoding rules defined in order to have an interim standard, until the OSI suite of protocols is sufficiently supported.

Finally there are several lower layer protocols (LLP) specified, which belong to the session and transport layers, and which include or use in turn resources, which are associated with again lower layers. These various protocols all meet different needs, which arise from the use of different transport media and their properties such as reliability.

However in the proceeding of this implementation attempt it turned out, that the HL7 specification doesn't make the distinctions above as strict as it claims to. This poses problems to the implementor, who tries to go with OSI. We will point out these problems in this paper at the place where they become evident (s. section 3.5 [On the abstractness of abstract syntax in HL7], p. 30)

The unique notation used in the HL7 document is another problem, that uneases the use of OSI standard methods. Probably much of the problems of HL7 could have been avoided, if a standard notation such as ASN.1 would have been taken over in time. This problem may be partialy come from kind of a misinterpretation of ASN.1: In the HL7 document it is repeatedly stated, that the scope of ASN.1 be the basic encoding rules, however ASN.1 is an abstract syntax notation, which is more appropriate for comparison with the 'abstract message notation' used by HL7. The encoding rules stay completely hidden in ASN.1 (in opposite of HL7 message definition). It seems rather likely, that ASN.1 was primarily avoided because of the difficulty "to explain to application programmers, who are not schooled in recursive languages in general, or ASN.1 in particular". This is certainly an anachronism, since tools like lex and yacc are widespread and extensively used since many years. However, to be fair, we have to note here, that HL7 was first documented back in 1987 when the ASN.1 standard was just released.

1.2 A view on this implementation

Today, there exist some tools which are dedicated to the OSI concept, most notably the ISODE, which is an almost complete development environment for OSI applications. Thus it seemed at the first glance appropriate to transform the HL7 notation to ASN.1 and feed the latter code to the various tools of the ISODE including the structure generator (pepsy) and the remote operation stub generator (rosy). However there is one single point which inhibits this rather convenient way to go:

ISODE supports only the OSI basic encoding rules (BER). Furthermore there is currently no other ASN.1 compiler, which supports the HL7 encoding rules. Even though, the HL7 encoding rules were meant as an 'interim standard' for use until OSI standards are available, it is now a major point that makes it hard for the HL7 world to migrate towards OSI. Most existing HL7 applications use the HL7 encoding rules, and for compatibility, new HL7 applications will keep doing so.

Because HL7 encoding rules have several disadvantages (Kuperman (1991), p. 179) which will cause them not to survive for many years, adding support for HL7 encoding rules into existing ASN.1 compilers (ISODE or SNACC), though possible, seems not to be worth the effort.

This implementation has developed (and is still about to develop) its own means by which the program objects are automaitcally derived from the documenting text of the standard. Since the author of this paper happened to have the text of the HL7 specification in ASCII files, there was a reasonable chance to go this way from the very beginning. Before we start going along that way step by step, we'll first take a look on it from above. Figure 1 shows the process of generation of the implementation.

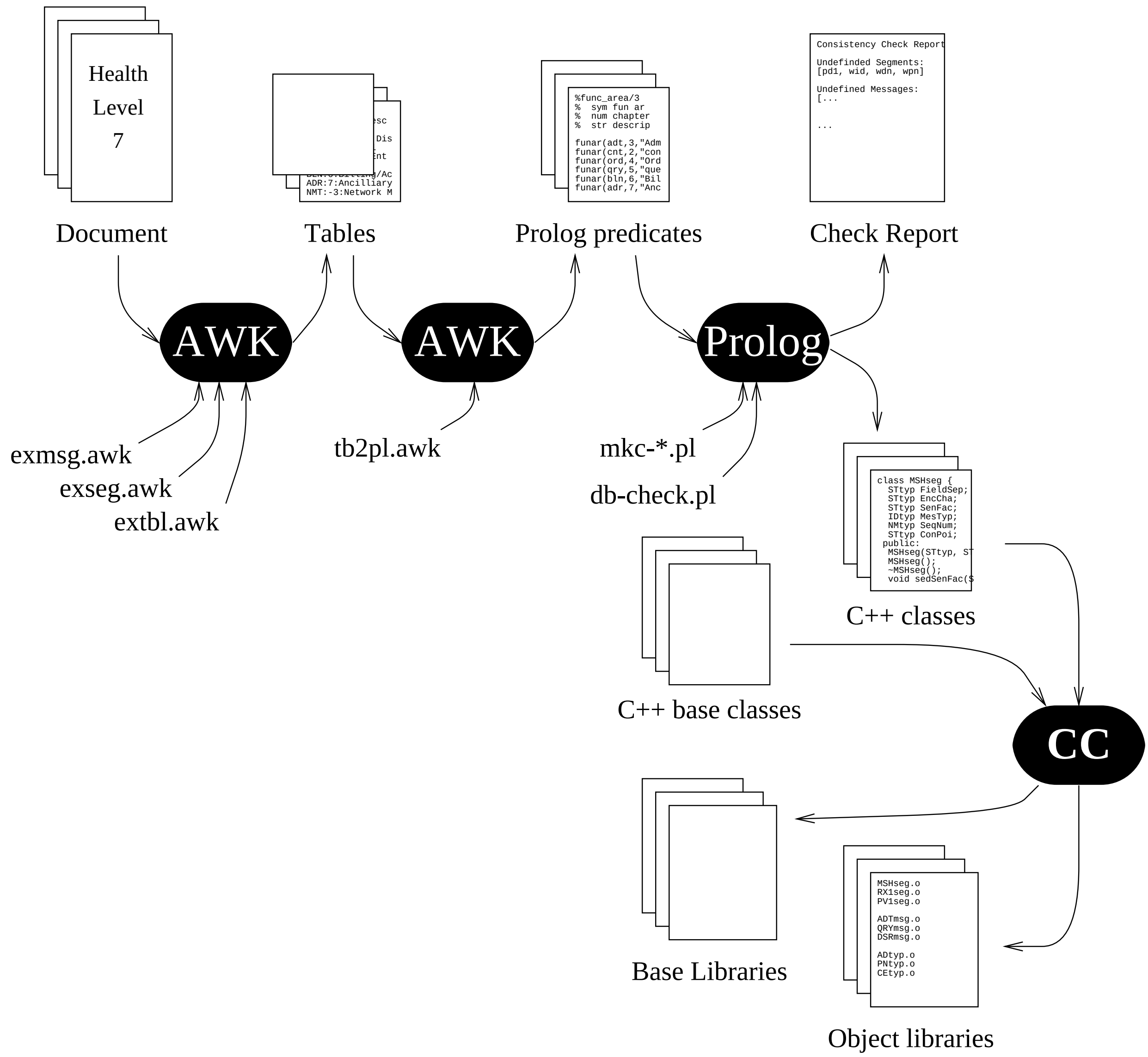

Figure 1: Generation process of the HL7 implementation

Let us start from the end, and answer the question where we want to go: We want to end with a set of definitions which are expressed in a common programming language plus an object library, which provides the actions corresponding to the definitions (the implementation in the closer meaning). This shall be further usable by applications to be built or to be extended with the HL7 interface. There is a need of compatibility in terms of platforms on which the programs are to be running if this implementation shall once pay off its effort.

The programming language that was cosen is C++. The important advantages of C are its widespread propagation on small computers like PCs as well as on workstations and minicomputers along with the naturally embedded operating system interfaces especially on UNIX platforms provide a base for compatible as well as efficient code. These advantages are shared by C++ since C++ is derived from C. Moreover, C++ provides new and powerful technologies of software design which we want to make use of here.

The steps towards the C++ code are taken by various means and methods. First we extract meaningful information from the HL7 document, this is done with AWK. The since most of this information is presented either in tables or in a simple formalism capable of expressing the message syntax, it is appropriate to process these information further by a method which is capable of handling both, relations and grammars. Prolog on the one hand expresses information in terms of relations, on the other hand it is powerful (and often used) for handling grammatical data. Moreover, analysis and generation of corresponding data can be mutually achieved by the same code. The tasks that are taken over by Prolog are the check for consistency of the information gathered from the document, and the final generation of the C++ code.

The C++ code is then compiled and packed into an object archive (a library). The C++ definitions in the header (`*.h`) files provide the Interface for application programs, that will make use of HL7. It should be possible to provide availability to C programs (and programs of other programming languages also).

Even though, we try to provide an automatical tool of code production from the standard's document which is reasonably general to tolerate future changes of HL7, we however do not intend to make a general purpose compiler for the notation in which HL7 is defined. Remember, that this notation is mainly an unstructured text, which although contains machine extractable structured information, which we try to capture. Furthermore the way this structured information is presented, is unique to HL7 and not backed by any standard. So there is little use of a general purpose compiler for that notation. We rather keep the work as specific to HL7 as possible, to minimize work overhead for the sake of unused generality. This includes, that a considerably amount of the code has to be written manually. Again here come the advantages of C++ (over C) for help to minimize software management overhead.

2 From Text to the Database

This chapter describes, how we actually scan the HL7 document to capture useful information. First we have to take a look on the HL7 document itself, its format and what structured information we may find therein. Then we describe the interim format, in which all information that can be expressed in tables is stored. Finally we describe the methods used to generate Prolog predicates from these tables. However as noted above, tables are not the only information style in which information is presented in the HL7 document, the message syntax is expressed in a simple formal language, that can be translated purely lexically to Prolog expressions.

To accomplish the tasks of this step, we use AWK, which is a common tool for text file processing, available on virtually any UNIX system. So is sed, which we sometimes use to post process our output in order to get rid of nasty junk characters. Where AWK or sed is not available, the C sources of the GNU version of these (strongly recommendable) tools can be freely copied from any major FTP server next to you.

2.1 The HL7 document

This chapter may be quite specific for the kind of text files that was available at the time, the work on this project started. Namely there were 11 Microsoft Word 5.0 files, one for each chapter (files '`kap[1-7].txt`') or appendix (files '`anh_[a-d].txt`').

The chapter files can be processed all the same way, while the appendices require a unique handling of each file. Appendix C can be treated like a chapter file. Since appendix B describes the lower level protocols and is completely different in both format and purpose, we will ignore it here until we come to the point, where we implement the lower layer protocols. Appendix D is an automatically (?) generated index, which we don't need at all. Before we start discussing the treatment of the chapters, only appendix A needs to be looked at more closely.

Appendix A is a summary full of tables, which presents a most welcome source of various information for us. This appendix was first treated manually. It was easy to convert all the tables into the interim table format described in the next section. Actually the latter format was developed while manually working on the appendix A. Since this is so, the appendix A was broken up into well formatted tables even before any AWK script was written which would do the work of extracting tables, there still exists no automatic extraction script, which could be applied to appendix A. Once our general method has proved to be useful and appropriate for the next version of HL7 as well, this gap will be filled up.

Fortunately, the format of Microsoft Word 5.0 files is merely an ASCII text file, with only a few special characters written directly into the text. The big bunch of information on the print format is appended binary to each text file. Even though, this information is completely obscure for us, it can savely be ignored. What remains open to us and is sufficient, is the ASCII text. Thus we don't have to bother with all the layout information like indentations, fonts and character styles etc.

Now, what information do we have to expect, and how is it presented in the ASCII text of our files? Each chapter consists of sections, which describe messages, segments or fields. The sections are recognized by their specific layout (e.g. tables) and by keywords, which have proved to be very helpful for us here. It can even be stated, that the strict usage of keywords (e.g. '`FIELD NOTES:`') in the text made this whole work at all possible. We'll describe the extraction of of the items in the appropriate section below.

As you can easily see from the above filenames, these were not the original files, which would for sure have english acronyms. In deed, these files were being worked on by the German section of the HL7 consortium.

This (visible) work consists merely of annotations mainly by German translations of the terminology, sometimes there have been some remarks added. However, this would have been only of little interest for us, if this efforts had not caused the format of the files to mess up. This in turn has been the cause of severe problems, some of which could not been solved satisfactorily by a general mechanism, with the consequence of having to manually edit the files that were produced defective. There would have been little sense of waisting time trying to solve a problem which is hoped to disappear in the next version of HL7 or even in the next set of textfiles we may have available.

It became quite obvious that a kind of a WYSIWYG text processor like Microsoft Word for Windows — though easy to use — is prone to cause severe problems, especially if more than one author is working on the same file. Very much dicipline and a common method (a standard) is required about the usage of resources to format the text (e.g. Print Formats vs. arbitrary formatting of marked blocks). It is not easy to always keep this kind of discipline if one can achieve ones immediate ends rather simple. Moreover the format of a text in a WYSIWYG editor tends to become obscure, even though the writer feels to have complete control over it, this gives rise to surprises. After all, it seems that text processing methods which appear rather unfriendly or outdated on the first view (like *roff and TEX), do pay off the extra effort in the long run, particularly on long texts and when used in a work group.

2.2 The Intermediate Table Format

This section describes how we store most of the information drawn from the HL7 text files. Since there are different items to extract (i.e segments and tables) which however are essentially tables, but differ in their embedding and appearance in the text, it seems appropriate to have an easy to generate interim table format, which the different extraction procedures write their information into, rather than generating the final representation directly. From these tables we can get our final representation easily by applying a common AWK script to all of them regardless of what they are derived from. This has the advantage that we are free now to decide to translate them into a different programming language than Prolog or import them into a database management system etc.

Note that if we are talking about 'tables' in this section, we mean this interim format, do not confuse this with the tables found in the HL7 text. The following is a set of rules telling us how these tables are built:

1. The first line of the table is it's name.
2. The table is ended by at least one empty line.
3. Each row of the table makes up exactly one line, the length of the line is however not limited to a specific number of characters. The line is terminated by the system's native `<EOL>` character, i.e. the one that the AWK's printf escape sequence '`\n`' expands into.
4. Columns are colon separated. A colon appears *between* two columns, not starting the first and not ending the last one. Where there are two consecutive colons, a colon at the beginning or one at the end of the row, the corresponding field is treated as empty.
5. The second line of the table names the titles of the columns.
6. The third line of the table specifies the data type to expect on each column. This can be one of the following keywords:

   '`sym`' a symbol, that will become an identifier in the target language (Prolog).

   '`num`' a number

   '`str`' a string, i.e. a sequence of characters, that will appear enclosed by string delimiters '"'.
7. The fourth line of the table is the first row of data of the table, this and any immediately following line will be treated as table data.

However, there are more complex tables, which contain subtables, all of the latter have the same format (e.g. number and types of columns). These complex tables are generated e.g. from table of '`TABLE VALUES`'[1], which appears in the appendix A. The complex tables are basically the same as

[1] yet another meaning of 'table'

described above. Notably the rules 1–6 of the definition above do still apply. Here are the other rules which apply to the complex tables:

7. the forth line must start with '-' and defines the titles of the columns of every subtable.
8. the fifth line must also start with '-' and declares the data type of the columns of every subtable.
9. any other line that is not preceded by a '-' starts a new subtable.
10. any other line that is preceded by a '-' is a row of the subtable that started recently.

The meaning of the rows is slightly different or extended from those of the simple table. The idea is, that we generate two relations, from the complex table. One is the main table (i.e. the table that results if we delete any line that starts with an '-'), while the other is a relation, that is constructed from the main table and the subtables.

Let $R = \{t_1, .., t_c\}$, be a relation of cardinality c, where each tuple is $t_i =< r_{i,1}, ..., r_{i,n} > for 1 \leq i \leq c$. R corresponds to the main table. For each t_i there is a relation $S_i(s_{i,1}, ..., s_{i,m})$ which corresponds to a subtable. If $r_{i,1}$ is a key to R then $T(r_{i,1}, s_{i,1}, ..., s_{i,m})$ is a relation which is equivalent to S, and which corresponds to the second relation we produce from the complex table.

To give examples rather than exhaust the reader with definitions, first we present a simple table:

```
DATA TYPE
DATA TYPE:DESCRIPTION:LENGTH
sym:str:num
AD:ADDRESS:
DT:DATE:8
...
PN:PERSON NAME:48
TX:TEXT:
```

Here is an example for a complex table:

```
TABLE
TABLE#:DESCRIPTION
num:str
-VALUE:DESCRIPTION:VALUE#
-sym:str:num
0001:SEX
-F:Female:000345
-M:Male:000344
0002:MARITAL STATUS
-D:Divorced:000350
-M:Married:000348
-S:Single:000349
```

From the latter table we'll produce two relations, as if they had been defined as follows, first the main table and then the constructed table:

```
TABLE
TABLE#:DESCRIPTION
num:str
0001:SEX
0002:MARITAL STATUS

VALUE
TABLE#:VALUE:DESCRIPTION:VALUE#
num:sym:str:num
0001:F:Female:000345
0001:M:Male:000344
0002:D:Divorced:000350
0002:M:Married:000348
0002:S:Single:000349
```

Note from the examples, that the title of the derived table is the title of the first column of it.

The filename conventions for these interim tables are not uniform. A simple table ends with '.tb'. However, a table which was derived from a segment definition is named with a trailing '.stb'.

2.3 Extracting tables

Most tables which are scattered throughout the chapters are compiled into the appendix A, so they don't have to be rescanned here. Actually there is no table in the chapters, which is not found again in appendix A, but we cannot be sure here. What we do is just to scan for the headings of the tables, in order to merely catch the number of the table. Afterwards, we can check, if there is any more information to get, which we do not already have taken from appendix A.

For this kind of extraction we do not even need AWK. We let sed(1) run once over each chapter file with the following command, which appears as the only command in the 'bin/chptbl' shell script.

```
sed -n -e "s/^TABLE \([0-9][0-9]*\).*/chapterTable(\1)./p" *.txt
```

This causes any line like the following example

```
TABLE 0002      MARITAL STATUS
```

to be output as

```
chapterTable(0002).
```

The latter is a Prolog predicate, which is then used to check for tables not yet known from appendix A.

2.4 Extracting segment definitions

Segment definitions are tables as well which describe one field on each row. On the header of the table there is the sequence of keywords '`SEQ`', '`LEN`' etc. The tables are again surrounded by empty lines. Following each table, there is a section which starts with the keyword '`FIELD NOTES:`', followed on the same line with the id of the segment. Even though, we take the segment id from the subsection headline, which precedes the tables, we reassure us from the correctness of our assumption with the help of the '`FIELD NOTES:`' construct, which turned out to be very reliable.

Please note, that we can not define a completely secure method at the outset, by which we find any piece of information without fail, but we rather refine our method as far as seems reasonable by trial and error, trying to catch most misinterpretations by taking advantage of redundancies found in any human readable text.

The extraction of segment definitions is done by '`bin/exseg`', which in turn runs AWK with '`bin/exseg.awk`' on its first parameter (which typically is a file name), pipes the output through several sed processes and prints its output to the standard output stream (which is typically redirected into a file).

We build the interim table with the name '`SEGMENT`' to be defined and followed by the column specification lines, which are the same for any segment. Then we convert the body of the table into the canonical format, replacing VTs by colons. We are in some trouble here, because some VTs have been mutated to consecutive spaces, thus we cannot savely discriminate columns. The only reasonable way to cope with these kind of problems seemed to manually edit the defect output. It is hoped, that any next version will be free of these problems, such that there is no use wasting time extending generality to cover faulty conditions.

There is reason for extracting the field notes as well. We will find, that there are fields declared as being of ID type, which actually are a kind of a composite types. This applies for fields, that deal with patient location. Actually we don't use the field notes a lot, but it might be useful, if we'd store the whole information in a database, or hypertext file. Field notes are extracted with '`bin/exfld`', which is itself an AWK script.

2.5 Generating Prolog predicates from tables

From the interim tables we can finally build Prolog predicates. Since Prolog predicates are essentially relations, and relations can be regarded as tables, this conversion is rather straightforward.

It is funny, that any formal information, identifiers as well as descriptive strings are printed in all uppercase in the HL7 document. However we turn everything to lower case, because of following reasons:

- text in all-uppercase is hard to read.
- text in all-uppercase is hard to type.
- Prolog likes symbols to start in lower case.

The functor symbol is derived from the name of the table, by deleting any special character and replacing consecutive white spaces by one single underscore ('_') character. Even though we could as well surround the whole title of the table by '' characters, thus marking the sequence as a symbol, we make the conversion for convenience of the working on it to come. Note that we have to refer to these symbols and we don't want to bother with the special characters and number of consecutive spaces, we rather convert the names in a canonical form.

It happens quite often, that one or more attributes of a tuple stay undefined, i.e. when nothing is found between two colons. In this case, we set the value to undefined, which we can do in Prolog by using the anonymous variable '_'. We could as well use a specific symbol which will assume the meaning of nil, like '`null`', '`nil`', '''' or even '`[]`'. However, the anonymous variable will be bound to anything during a unification, thus anything will be allowed at this attribute to cause the predicate to suceed.

Finally we begin each group of predicates with a few command lines of descriptive information about functor name and arity, column types and column headings.

This job is done by the '`bin/tb2pl`' shell script, which calls '`bin/tb2pl.awk`', and collects temporary output of '`bin/tb2pl.awk`'. These temporary files are created during the procession of subtables.

2.6 Extracting message definitions

Messages are syntactically defined in the HL7 document using the formal language to be described below in this section. We can recognize message definitions by their unique layout, which comprises

3 columns, normally separated by ASCII VT characters, except for the cases where the format was damaged or originally inconsistent. The first column contains the code in the formal language, the second column contains remarks and the third column contains the Chapter. The first row contains the message id and on it's last column the word "Chapter" or "Appendix" which we use as a keyword. Finally these tables are separated from the rest by one empty line, both at the beginning and at the end.

The following is an example of message definition as we find it in the file ‘`kap2.txt`’, all literal VTs have been replaced by ‘ - ’.

```
WRP - Widget Report - Chapter
MSH - Message Header - II
MSA - Message Acknowledgement - II
{ WDN - Widget Description - XX
  WPN - Widget Portion - XX
  { [WPD] } - Widget Portion Detail - XX
}
```

The syntax of the formal language is as follows

<message> ::= <message id> <group>
<group> ::= <item> | <item> <item>
<item> ::= <segment id> | ‘[’ <group> ‘]’ | ‘{’ <group> ‘}’
<message id> ::= <id>
<segment id> ::= <id>
<id> ::= <uppercase> <upper or digit> <upper or digit>
<upper or digit> ::= <uppercase> | <digit>
<uppercase> ::= ‘A’ . . . ‘Z’
<digit> ::= ‘0’ . . . ‘9’

At this point we can however ignore the syntax,[2] we rather make the following textual changes:

1. remove any white space
2. change all upper case to lower case
3. any opening bracket (‘[’) is replaced by ‘`opt(`’
4. any opening curly brace (‘{’) is replaced by ‘`rep(`’
5. any closing bracket (‘]’) or curly brace (‘}’) is replaced by a closing parenthesis (‘)’)
6. append a comma ‘, ’ unless
   - there was recently no token at all
   - there was recently an opening parenthesis

[2] though for other reasons, than not to disappoint application programmers who are not schooled in recursive languages (sorry, couldn't resist)

7. remove any new line character (i.e. print anything on a single line)

The first id that was read, i.e. the one that happens to be on the header line of the table becomes the message id. The rest of the first line up to the keyword '`Chapter`' or '`Appendix`' will be recognized as the description. However, for the body of the table, we recognize but the first column, which we handle as said above. Things would have been easier, if there would not be some message definition tables breaking these rules. Some tables are formatted without the VT between the columns, which made it very hard to get rid of the other columns, while keeping the integrity of the first column.

The definition of each message is then stored into a single Prolog predicate message/4:

```
message(wrp,'',"widget report",[msh, msa, rep(wdn, wpn, rep(opt(wpd)))]).
```

The second argument of message/4 is the event type code, which further qualifies a message type. However, an event type code is not always specified (which we have to discuss later). This code — if given at all — can be found in the most recent heading of a subsection. One of the keywords '`Event Code`' or '`Trigger Event`' precedes the three character id of the event type code. That one is written as the second argument to the message predicate.

We should rather have scanned the recent heading of a subsection for a description of a segment, since this may uniquely describe one message referenced by a pair of message id and event type code. For now we have redundant descriptions that poorly specify the messages, which certainly has to be fixed soon.

3 The HL7 database

After being done with the previous work, we are arrived at a point, where we can take some kind of a rest, reflecting about what information we had gained and how it is structured. What we did was to extract sections of the files and to translate them into a different syntactical presentation. However, what we did *not* do up to here is to make any considerations about what is meant by all the tables, and grammars and how they are mutually related. This will be done in this chapter, we try to clarify all these properties of the data model and concepts of transactions.

First we take a look at the the relationship of HL7 conceptual entities that explicitly appear in the specification. We will optimize it by removing redundancies, and — perhaps most important — we gain an oversight of what HL7 defines. Then we check the database that we compiled in the last step for errors and inconsistencies, and we prove an important assumption we made in the preceding section. Finally we focus on the contents of the relations, we ask what in particular is defined, and how is it defined.

3.1 Dependencies

In the following table, we give a synopsis of all relations, that we have got so far. We tried to somehow compress the names of the items here, to be specific and descriptive as well as short, in order to not loose oversight. Primary keys to the relations are are marked by preceding asterisks (‘`*`’), while other candidate keys are marked with a plus (‘`+`’). If a primary key is made up of more than one attributes, they are set in parentheses with one preceding asterisk. The name of the relation is followed by an ‘`[A]`’ which means, we have drawn this relation from appendix A or a ‘`[C]`’ telling that it was drawn from the chapters (including appendix C[1]).

Functional Area [A]

*FunArId, +Chptr, +FunArDscrptn

Message Type [A][2]

*MsgId, +MsgDscrptn, FunArId

Message [C]

*(MsgId, EvntTpCd), +MsgDscrptn, +MsgDef

Segment [A]

*SgmntId, +SgmntDscrptn, FunArId

[1] see above (s. section 2.1 [The HL7 document], p. 9)

Segment [C]

*SgmntId, +SgmntDscrptn

Data Element [A]

*DatElNum, +DatElDescrptn, SgmntID, FunArId, MaxLn, DtTypId, Opt, Rep, TblNum

Field [C] *(SgmntId, FldNum), MaxLn, DtTypId, Opt, Rep, TblNum, +DatElNum, +FldDscrptn

Data Type [A]

*DtTypId, +DtTypDscrptn, Ln

Table [A] *TblNum, +TblDscrptn, TblClss

Table [A] *TblNum, +TblDscrptn

Table Value [A]

*(TblNum, ValId), ValDscrptn, +ValNum

Field Notes [C]

*(SgmntId, FldNum), +FldDscrptn, FldNteTxt

The rows of the above table are sorted to ease orientation of the reader. Therefore, one thing becomes immediately obvious: There are sometimes more than one relation with the same name. Even though, they both are titled '`Segment`', they are not the same relation because they don't have the same cardinality.[3] This notwithstanding, it is still obvious that these relations have some domains in common. We can simplify our set of relations by rewriting it such, that any two relations which correspond this way are replaced by a third one which is defined over any domain, which is part of either the first or the second relation.

However, different names of two tables do not guarantee that they do not correspond the same way, as was just said. Consider '`Data Element`' and '`Field`'. Both are defined over the same domains by different order, except from '`FunArId`' (i.e. the column titled '`owner`' in appendix A, and lists the functional area, to where it belongs) and '`FldNum`'. The question is now, if tuples of both relations can be mapped one to one. We will see below (s. section 3.3 [Consistency check], p. 26), that they can.

The name '`Field`' was given in '`exseg.awk`', since the word 'field' is used throughout the HL7 specification to designate the parts of which the segments are built (more than 500 occurences). However 'data element' is used sometimes (42 occurences) as well. Why are there two names for the same thing? One answer might be, that 'data element' is used where we refer to an atom of data

[3] This holds unless we do not have two relations with the same cardinality but different domains, we attach our concept of equality of relations to the concept of Prolog.

regardless of the context in which it occurs, while 'field' is used, for such an atom in the context at a certain place of a certain segment. Thus a data element is the *contents* of a field. In deed, the relation '`Data Element`' doesn't have a domain, which could designate a certain place in a segment.

However, why is there an attribute for repeatability and optionality then? We wouldn't expect an object to be optional per se, whereas a certain field in a segment may well be empty sometimes. Also repeatability is no property of a data element from this point of view, even though it depends on how we think of an repetition: Does the field repeat or does it's contents repeat in the field? If the first was true, then a segment would not necessaryly be of a fixed number of fields,[4] if the second is true, then there must be something in between a field and an atom of data, we could say, that an 'occurence' is not identical to a data element. This resembles LISP's point of view: LISP would regard a field of a segment like one half of a pair, which can be a list (i.e. another pair), or an atomic data item. If we have a look at the encoding rules,[5] we notice, that repetition is realized with a special delimiter, this reconfirms us in our view of repetition as happening on a level inbetween a data element and a field, which we might call the level of 'occurence'.[6]

In order not to digress too much we decide not to consider data element and field as different things, if we can proof the one to one relation of both. We perform a rewrite on both of them, which is similar to the one we made for '`Segment`' or '`Table`'. We'll make this proof when we check the consistency of the database, that we acquired. For now assume, that this proof will succeed.

---

[4] notwithstanding the fact, that trailing missing fields are regarded as 'not present'

[5] which should be irrelevant if we discuss about conceptual issues. In fact as we'll see below (s. section 3.5 [On the abstractness of abstract syntax in HL7], p. 30), there is only a weak distinction between abstract concepts and representation issues in HL7

[6] which is somehow tautologic

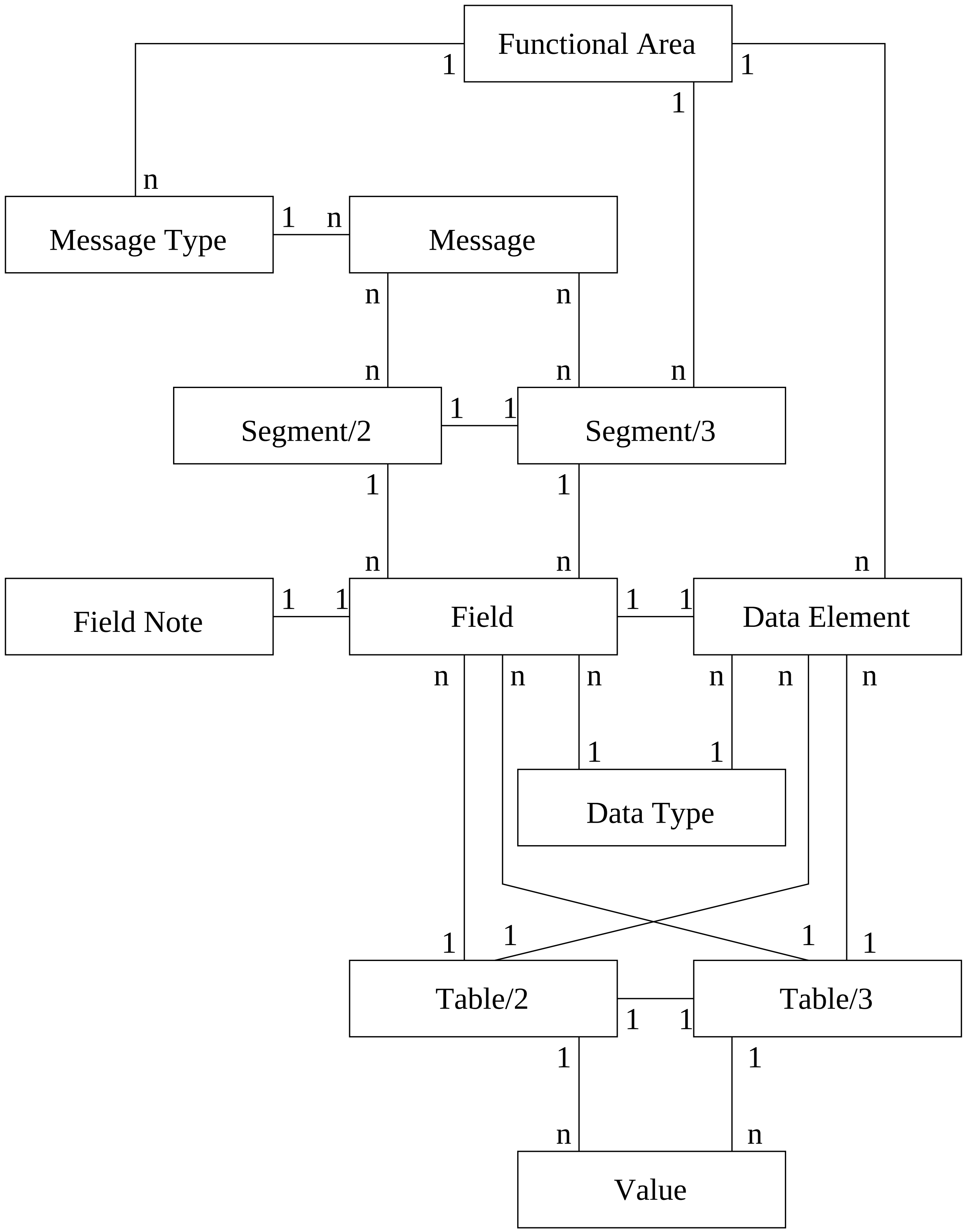

Figure 2: Model of the HL7 data base before optimization

Figure 2 shows a sort of entity relationship model of the database before we removed multiple occurences. Each relation of the database is graphed as an entity (a name in a box) which has a relation (a line linking it) to an other entity. Note the different notions of 'relation', to avoid confusion, we will speak of a 'link' if we mean relationships or dependencies between relations. At each contact between a line and a box, there is a number '1' or 'n'. This graph can be "read" by following each line with the words: "<number> <name> is linked to <number> <name>" where <number> is the number, which is written at the box of <name>.

Let's have a look if there is more to refine. Were there is a one-to-one link, as between 'Table/2' and 'Table/3', we can merge the two relations into one, that's what we have already planned to do. However there is more: There is a pair of parallel one-to-many links, one going from 'Functional Area' via 'Segment/3' and 'Field' to 'Data Element' and the other going directly from 'Functional Area' to 'Data Element'. We notice, from the table above, that this parallel link is caused only by the 'FunArId' domain. Thus, we can consider removing the domain from the relation at the many-end of the link to remove this indirect redundancy, unless it is not part of a key there, which it isn't. Note that it depends on the one-to-one link, between 'Field' and 'Data Element' whether we may commit this simplification. If it is a many-to-one link, i.e. if one data element could appear in several fields, we must not do this.

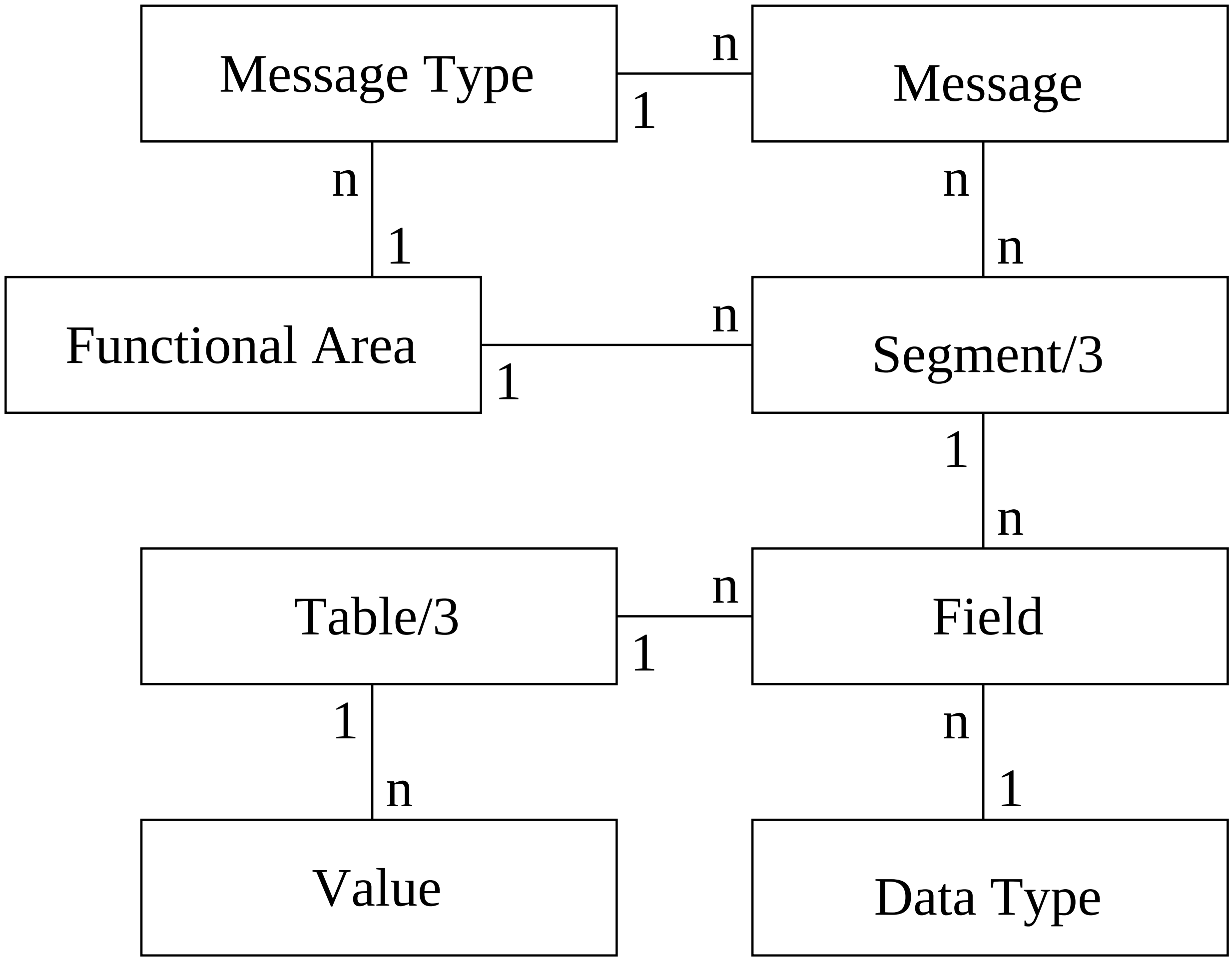

Figure 3: Model of the optimized HL7 data base

Our simplified database looks as sketched in figure 3. The table below will show it in detail:

Functional Area	
	*FunArId, +Chptr, +FunArDscrptn
Message Type	
	*MsgId, +MsgDscrptn, FunArId
Message	*(MsgId, EvntTpCd), +MsgDscrptn, +MsgDef
Segment	*SgmntId, +SgmntDscrptn, FunArId
Field	*(SgmntId, FldNum), MaxLn, DtTypId, Opt, Rep, TblNum, +DatElNum, +FldDscrptn
Data Type	
	*DtTypId, +DtTypDscrptn, Ln
Table	*TblNum, +TblDscrptn, TblClss

Table Value

*(TblNum, ValId), ValDscrptn, +ValNum

Field Notes

*(SgmntId, FldNum), +FldDscrptn, FldNteTxt

3.2 Errors

Now we will finally feed our database into Prolog just to see if what we generated was syntactically correct. We shouldn't bother the reader with the results of our typos here, these have been corrected beforehand. Rather, this section reveals the first severe errors in the HL7 specification. Here is what Prolog complains:

```
[WARNING: (/usr/share/doc/HL-7/kap4.msg:5)
        Syntax error: Operator expected]

[WARNING: (/usr/share/doc/HL-7/kap7.msg:11)
        Syntax error: Operator expected]
```

Further investigations point us directly into the documents: At the definition of the order message we find what is extracted below. Note the matching of brackets and braces. The opening bracket before PID and below ORC are never closed properly. So where do we have to assume the closing brackets to be placed? The author of this document wasn't able to do correct this until he could see Version 2.2 (ballot 1) for the answer. In the table below, corrected brackets are set between asterisks.

```
ORM                     ORDER MESSAGE           Chapter
MSH                     Message Header          II
[ { NTE} ]              Notes and Comments      II
[ PID                   Patient Identification  III
   [{NTE}]              Notes and Comments      II
   [ PV1 ] *]*          Patient Visit           III
{
   ORC                  Common Order            IV
   [
   Any Order Segment    E.g., ORO, OBR, RX1 IV
   [ { NTE } ]
       [ { OBX }        Results Segment         VII
           [ { NTE} ]]  Notes and Comments      II
  *]*
       [ BLG ]          Billing segment         IV
}
```

The second error is due to the same kind of grouping problem, however, this time we experience one closing brace too much. It's again hard to guess the correct grouping except from a minor mismatch in the PID group, it can be concluded from version 2.2 that what is set between asterisks is wrong, even though there is at least one mismatch in v2.2 too.

```
ORF                        Observational Report       Chapter
MSH                        Message Header             II
MSA                        Message Acknowledgement    II
{ QRD                      Query Definition           V
    [ QRF ]                Query Filter               V
    [ PID ]                PATIENT ID                 III
        [{NTE*}]*}         was: [{NTE]}}
    { [ ORC ]              Order common
         OBR               Observation request        VII
         {[NTE]}           Notes and comments         II
         {[OBX]            Result                     VII
            {[NTE]}        Notes and comments         II
         }
    }
* *                        was: }
[DSC]                      Continuation Pointer       V
```

While we cannot be sure who caused this error, whether it was the original editor or some people of the German section of HL7 there is more evidence for the need to edit this document with different methods. E.g. emacs is an editor, which shows paranthesis matching and warns on mismatched paranthesis while they are typed. This is more convenient than WYSIWYG, since while typing a document control of correctness and consistency should take precedence over immediate control of layout[7].

3.3 Consistency check

Being mounted with the insight of the last section, we can now start to check our database. We will have to check for

- tuples with undefined keys
- whether one relation is included in another
- consistency of conceptually redundant data
- missing data
- syntax errors in message definitions

---

[7] not to repeat here, what was stated about control of layout above (s. section 2.1 [The HL7 document], p. 9)

During the rest of this section, we will merely see the output that was generated by the Prolog check clauses:

```
Removing Facts with undefined key value:
retracting segment(G198, "")
```

There was a completely void segment definition caused while chapter 1 was processed. Something has screwed up, we can safely ignore this.

Now we're checking for tables which had not been defined in appendix A, but were used in the chapters. '`chptbl.pl`' was loaded and compared against table/3, result is '`none`', meaning that no undefined tables have been found. Next, we check for one to one relationship between table/2 and table/3, particularly if table/3 contains all of table/2, which is true. The descriptions of corresponding tables are consistent, which means, that we can savely exclude table/2 from the further process without loss of information.

```
Performing checks on:
*** Tables
chptbl compiled, 0.05 sec, 2,120 bytes.
  tables which are not yet defined: none
  tables from table/2 that are not in table/3: none
  tables with conflicting descriptions: none
```

The next part checks whether there are tables referred to for which there are no values defined. This is tue for a lot of tables but is not necessarily wrong. A table denoted as '`user_defined`' may well be undefined in the HL7 standard. But how about tables marked as '`hl7_standard`'? These may be references to tables defined by other standards, however we'll remember this problem in case we get into trouble but leave this alone for now. We also check if these missing tables are defined somewhere in the chapters, but they are in neither case. Finally we check for values which do not belong to a table, which would point us towards an error in our AWK scripts, but this is not the case.

```
  tables without any value definition:
    for class 'hl7_and_user': 111,112,101,30,90,63.
      in the chapters: none
    for class 'hl7_standard': 50,31,13,25,59,51,52,55,56,37,14,26,18,
                              75,17,29,35,41,15.
      in the chapters: none
    for class 'user_defined': 117,23,19,120,21,22,32,43,44,45,46,47,
                              49,114,113,57,66,60,24,64,96,68,69,42,
                              72,86,73,79,118,81,83,84,10,87,88,89,92,
                              93,94,115,110,98,99.
      in the chapters: none
  values without any defining table: none
```

We do a similar processing of segment as we did with table above, i.e. checking for containment and conflicting descriptions, and do not find any surprise.

```
*** Segments
  segments from segment/2 that are not in segment/3: none
  segments with conflicting descriptions: none
```

Here follows the proof of our assumptions we made about the one-to-one relationship of field and data element. The extra check on conflicts in length is of historical reasons and will disapear, since it is done again by the general check for conflicts

```
*** Data elements
  conflicts in length:
    for data_element/9: none
    for field/10: none
  data_element/9 not in field/10: none
  data_element/9 referenced by more than on field/10: none
  field/10 and data_element/9 conflicts: [obx,4,769],[obx,6,562].
```

We see that our proof for one-to-one relationship succeeds, although there are inconsistencies concerning the data types of two pairs of field and data element, which are marked below:

```
field(obx,4,20,st,_,_,_,00769,"observation subid").
data_element(00769,"observation subid",obx,anr,20,nm,_,_,_).

field(obx,6,20,st,_,_,_,00562,"units").
data_element(00562,"units",obx,anr,20,id,_,_,_).
```

Next, check is as self explaining as it succeeds well.

```
*** Fields
  field/10 without segment/3: none
  segment/3 without field/10: none
```

Finally there are the checks on messages. Message definition syntax is error free because Prolog has filtered out the mismatched parentheses. Nevertheless the problem was not corrected by Prolog, it was just removed. No syntax error does merely mean that the conversion scripts have their job well done. But then there are undefined message types. We don't care about the stuff from the 'Widget' unfunctional area. That the messages of appendix C do not appear in the list of message types in appendix A is a fact that makes us worry once more about the 'computerized data dictionary' which is said to be used to generate all the tables. Why are there inconsistencies? Why missing objects? In fact some people think that the computerized data dictionary is just a phantom. The ADR message type is unknown as well, but these problems could be fixed easily once we know about them.

However, undefined messages are not at all easy to fix, because how should we know about them if we are told nothing but its name and functional area? Note, that ORF and ORM are missing because of the syntax error above. While OCF has disappeared since v2.2, ARD and OSQ are still hanging around in the tables. We assume them to be fired next time as well.

Finally — again forget about the widgets — there are messages which refer to segments which are not defined. This is the worst error, that we have detected here, because it makes whole messages unimplementable. However, if we subtract the '`ms`'-typo in an ACK of which there are many other definitions as well, and once we know, that we have to treat '`ANY`' segments specially, there remains PD1, patient demographics, which is undefined. After all v2.2 has silently fired this segment as much as v2.1 silently contained this segment.

```
*** Messages
  message definition syntax errors: none
  undefined message types used: adr,nmd,nmq,nmr,wro,wrp.
  undefined message: ard,ocf,orf,orm,osq.
  undefined segments used: [wro,wid],[wrp,wdn],[wrp,wpn],[wrp,wpd],
         [wrp,wdn],[wrp,wpn],[wrp,wpd],[wrp,wdn],[wrp,wpn],[wrp,wpd],
         [ack,ms],[adt,pd1],[orr,any].
```

3.4 The data item numbers

The relations for fields, tables and values have each one domain that is made up from a kind of integer numbers, i.e. strings of a few numbers often beginning with some zeroes. These aren't really numbers because obviously, there is a significance given to leading zeroes. Thus a five digit string denotes a field, a four digit string denotes a table, and a six digit string denotes a value. If we want to regard these digit strings as numbers we have to admit, that these numbers are not a unique classification of data items, but not more than a key to the relation, where otherwise would be no simple key. In fact the only usage of these keys is for the relation of tables, while fields and values have a composite key which is sufficient. In the latter cases, we won't make use of the digit strings.

Nevertheless, there is evidence that what was intended with the data item numbers is a kind of classification of data items of HL7. How else would this meaning that was given to the leading zeroes be explicable? It wold have been sufficient just to assign numbers which can not uniquely specify an HL7 data item but which do their job within a single relation. Anyway, we don't need a HL7 data classification here.

3.5 On the abstractness of abstract syntax in HL7

The HL7 documentation claims, that it complies to the idea of the OSI reference model with it's seven distinct layers. There are terms like 'abstract syntax' vs. 'encoding rules' frequently used in the specification. However, as we already stated above (s. section 1.1 [A view on HL7], p. 5), these distinctions are not always made as strict as the document claims. Let us see here, why this is so.

There are the HL7 encoding rules, which are meant as an interim standard made available until there are implementations of OSI standards. These encoding rules are very simple: any data is represented as a string of displayable ASCII characters with a set of five delimiters defined, which terminate the data items. Since it is thus very unlikely, that any byte of data will interfere with the underlying transport mechanism, it is possible even for the simplest kind of text processor, batch file or serial line to transmit the HL7 messages. However, what seems like an advantage on the very first view, turns out to have a considerable impact on the higher levels of abstraction. These encoding rules impose the restriction to the higher levels of the protocol stack, that they may not send unprintable characters or even may not use the delimiter characters as data. A presentation layer, that forwards it's task up to the higher layers is of pretty little use. It rather should make it's mechanisms and those of the underlying layers transparent to the upper layers.

One could argue here, that the HL7 encoding rules were not meant to be perfect, and that better encoding standards that are now available wold replace them. The abstract message definition, as the heart of HL7 would allow this. However, there are parts of the encoding rules, which have taken a place even in the abstract message definition: The MSH segment defines the 'encoding characters' as being the first data field of the MSH. This is wrong. It would have been easy to let the negotiation about encoding characters be part of the LLP.

These problems notwithstanding, there is a way to overcome at least parts of the problems that the encoding rules impose on the higher levels of HL7: Since any data is converted to a string of printable ASCII characters, this problem is practically of little relevance for such data types, which represent numerics (i.e. NM, DT, TM, TS, SI). However, text data types (i.e. ST, TX, FT) are set directly by the application which should not be forced to worry about printability of ASCII characters. The encoding module must provide at least some transparency here, but HL7 defines no standard for this, even though the solution is so obviously at hand:

There is already one encoding character defined, which is called the 'escape character'. The usage of escape characters are common among applications as well as communication programs. Escape characters are commonly used for two purposes:

1. Protect data bytes from their misinterpretation as control bytes (e.g. as '\' used in shell programs, or the DEL character used in ANSI X3.28 transparent mode).

2. Mark some sequence of bytes in a stream of data bytes to be interpreted as entities of control (often referred to as a 'escape sequence') (e.g. ANSI terminal, TeX, SGML).

The HL7 escape character used to mark a sequence of control characters (as pointed out in number 2). Unfortunately, the usage of escape sequences is currently limited to TX and FT types. Escape sequences should, however, be aplicable for any type where data/control ambiguities might arise. Since the delimiter characters are readily available for redefinition this ambiguity might arise in the encoding of any data type. Consider some message that redefines the delimiters to be '`+.:-?`' instead of '`|~^&\`': Even the numerical an date/time data types must use the escape sequences to unambiguously encode their values.

But there is still a bigger problem: Some features of HL7 extremely corrupt the distinction between Abstract syntax (application layer) and the presentation layer. All these issues are concerned with length of fields or blocks. HL7 drags views which only exist on character streams far inside the abstract level, where we should rather deal with concepts than with strings.

The first issue is the definition of maximal lengths of fields which are not of string or text data type. These make illegal assumptions about representations of values. For example the length of a DT value does not belong into the description of a PID[8] segment, since this makes assumptions about how DT values are represented, which highly depends on the encoding rules used. To give a maximum length is not correct for the PN type too. PN type is a composite type which consists of 6 ST types, there is a maximum length defined to be 48 including the delimiter characters. Not only that delimiters should not be part of an abstract syntax, how can this restriction be applied? Two passes are needed for the correct encoding: the first pass had to assemble the PN encoding from the encoding of it's components, the second had to check the whole string that encodes the PN value for an exceeding length. If the length is more than 48, a crucial question arises: Which of the components is to be truncated? It is obvious that such a restriction is not implementable by a reasonable effort since this restriction is of no use at all but seems to exist merely for historical reasons. This might shed a light on the concept of data in earlier days of HL7: any data was obviously regarded as strings even numerics or composites.

While we could silently ignore the 48 characters restriction of PN, there are more assumptions being made about lengths which seem inadequate to the author. There is the method of continuation segments proposed in the HL7 standard. This is a feature which again loads burden onto the application that the lower layer protocol should carry. Thus an application would have to bother with the reassembling of continued messages which is extremely cumbersome. Segments are entities of data transmission and as such their integrity should not be touched on the application layer.

[8] there is an inconsistency in v2.2 which tells that PID field 7 be a '`TS`' value of length 8

There is hardly any need for the continuation of a segment if there is a proper lower layer protocol. Lengthy messages should be split into packets and reassembled which should all happen completely transparently to the application layer.

3.6 On trigger events

The trigger events build a link from HL7 transactions to the real world. Circumstances are described, in which a certain transaction is initiated. There is a many to one relation between trigger events and message types. Especially the ADT message is a superset of many messages which are though syntactically similar, distinct in their contents and purpose.

There is a classification of trigger events in the table of trigger event codes. However, there are more trigger events than are listed in the table 0003. From the view of the HL7 data base, the concept of event type codes is unfortunately degraded to merely a table of subselectors, which are defined only for those events, for which there is a many to one relation to message types. It is desirable to have a complete list of event codes, so that a message would be uniquely referred to by it's event code.

Until then we have to either refer to a message by specifying the message type and the event code (if there is one), or we have to merge the tables of message types and event codes into one such that any message type that has several event codes is removed from the merged table.

3.7 On the null value

HL7 makes a distinction between values that are not present and those that are null. While the meaning of 'not present' is obvious (it could be paraphrased to 'unknown'), the semantics of the null value remain somewhat confusing, even though the null value is repeatedly mentioned in the HL7 document.

The problem here is to get an idea of consistent general meaning of the null value, and the crucial question is:

> What does a null value mean in a NM field?

the HL7 document (v2.2) tells the following (Page 2-5):

> *The difference [between not present and null] appears when the contents of a message will be used to update a record in a database rather than create a new one. If no value is sent, (i.e., it is*

> *omitted) the old value should remain unchanged. If the null value is sent, the old value should be changed to null.*

But what does this notation of 'null' mean with respect to NM or DT fields? The problem fades away, if we do regard any data transmitted by HL7 as strings. However here we want to provide a mapping from diverse data types to HL7. Should we simply map '`""`' to '`0`' then? Even though, this would solve the problems wit NM types, it would cause nonsense values in other fields like TS.

There are two ways out of this paradox:

- restrict the null value to string like data only
- open up the meaning of the null value so, that it can apply to any data type.

Here we try to go the second way and assume an object, which is not present as unknown, whereas null indicates that an object is known to be non-existent or that it doesn't make sense in a certain context. This is still consistent to the interpretation of the null value concerning data bases, that is given in the HL7 standard: An unknown value will not cause the data base to be changed, but will rather be bound to what is stored in the data base.

It still remains subject to further discussion how the difference between a string of length zero and a null value would have to be represented and interpreted. For example, a null string could be encoded as '`\0`' (or as '`""`' while the null value would be '`\0`'). The string of length null updates a data base field to a string of length null, while a null value updates the data base to a value meaning 'value does not exist'. Thus the HL7 protocol would be able to handle nulls as they are proposed for the nested universal relational database model (see LEVENE (1992)) to handle incomplete information. Since incomplete information does concern medical informatics it is strongly recommended here, to go this way of opening up the null value for a general well defined usage in any data type.

However, a problem still remains as the scope of the null value is unclear in a composite data type: If a CM field is received as being '`""`', what does it mean? Does it mean that the first component of the field is null and the other components are not present? Or does it mean that the whole CM data is null? This ambiguity could be solved by further interdicting the deletion of delimiters which terminate trailing items which are not present. Thus '`1234^^`' must not be truncated to '`1234`'. However the error tolerance is a feature which is relied on in the extensions of v2.2. Therefore it should be hard to convince the HL7 committee to change this. But other ways can be found (and should be found) to overcome this ambiguity.

4 Generating C++ code

This chapter of is dedicated to the actual implementation of the HL7 standard, whose specifications was made available to machine processing by the process shown in the preceding chapters. I have already shown some points, which will become relevant in the implementation. This chapter will begin by presenting the general concept that was assumed for the implementation. We will then shed a light on the compiler, that does the job of translating the data base into program code.

4.1 General concept

The object oriented programming technique as provided by C++ allows a very natural view on the HL7 data items. Any data item is an object of HL7 (HL7 object). Individual data types, segments or messages are special objects, which share common properties. We define the properties that are common to all HL7 objects as the class "HL7Object". The class "HL7Object" is an abstract base class which still contains some method and data instances including the flag that shows if an object is present or not.

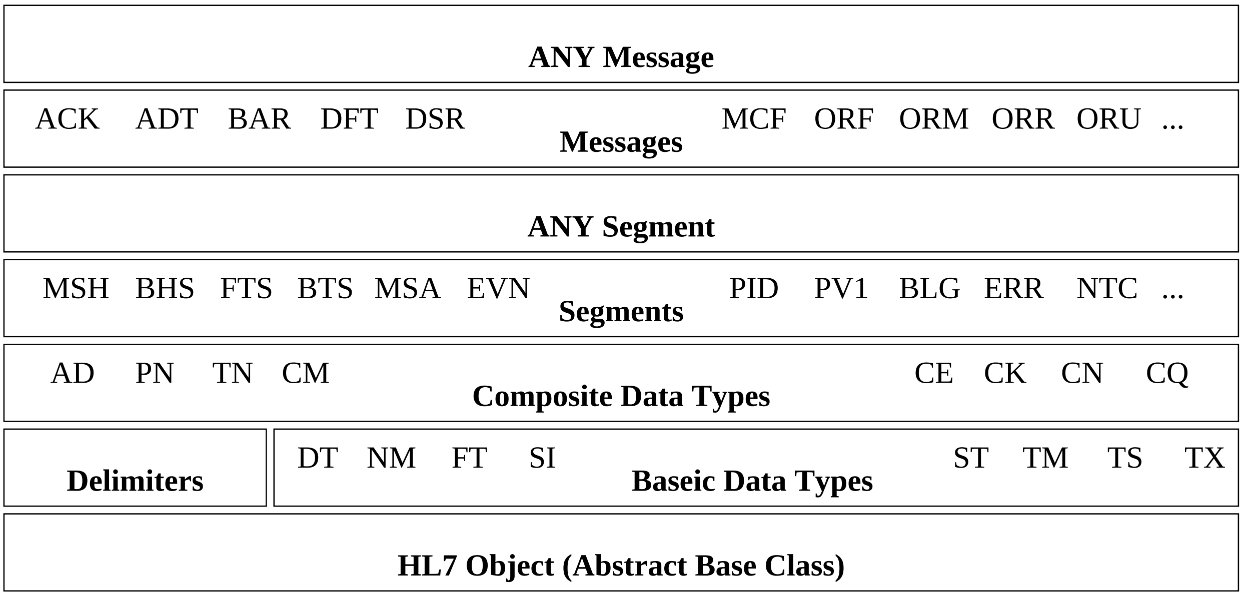

Figure 4: The hierarchy of HL7 objects

There is a hierarchy of objects as shown in figure 4. Basically there is the HL7Object, which is inherited by any other data object. Then come the basic HL7 types and the set of delimiters, which could be regarded as a special data type as well. These basic types have been implemented manually since the basic types are quite heterogeneous but only a few in all. It doesn't seem appropriate to implement them automatically from some kind of a data base. Moreover, the description of data types in the HL7 standard is presented in a narrative form which is hard to scan for specifying data.

The composite objects are however generated from a simple relation that describes their contents. Normally a component is a basic data type (with some exceptions, when a component is itself a composite, see below (s. section 4.2 [I/O methods], p. 40)), which reduces the organization in a composite data type to just collect the basic data types. The relation that describes the composites is manually edited.

Segments are described in the HL7 standard by tables, which we have brought in the form of a data base. The implementations of segments are generated from this data base. So are the implementations of messages. There is a need for an abstraction of segments to be introduced: the ANY segment. This is a segment, that can be of any type. Since segments are encoded as tagged data, i.e. data that is preceded by some identifying code, we can exactly discriminate the types of segments by the tag (the segment id). This is not possible with data types, since there are ambiguities and no tag.

What we just told about the ANY segment applies for the ANY message as well. However, to discriminate messages is more complex than it is with segments, since the information about message types is scattered throughout several segments.

There are several qualities which are common to all HL7 objects, we present these during the following subsections.

4.1.1 Names

In program generators there is the consideration of how to choose names for the various objects. The higher the complexity of the data structures the more distinct names have to be given to objects. There is always a tradeoff between highly descriptive names, which are long and thus hard to type, and short names which are easier to type but less descriptive. Even though program generators do well without descriptive names, the end user of the libraries which are to be built are human beings, programmers, for which the naming should be convenient. Because C++ allows overloading of functions names can be shorter without the risk of name conflicts.

Here we have make the following naming conventions. Classes for data types and segments are given the name of their two or three characters id in upper case, with the small qualifier '`typ`' or '`seg`' directly attached to it. Thus '`NMtyp`' is the class of numerics and '`MSHseg`' is the class of message header segments. Message classes are named a similar way by attaching '`msg`' to the uppercase letters which are taken either from the list of message type ids or from the list of event type codes[1]. Note that we try to avoid using the underscore character '`_`', which tends to merely lengthen the name, readability can be acheived by altering uppercase and lowercase letters as well.

Instantiations of classes, i.e. variables as well as symbolic names of table values are given a name which is derived from the objects description. There is a simple fuunction which produces valid C names from arbitrary text strings. The function is given three arguments. the text string, the *threshold length* and the *truncate length*. All words in the string are concatenated with each first letter in upper case and all other letters in lower case. The words are truncated to the truncate length if they are longer than the threshold length. Typically the threshold length is greater than the truncate length, thus allowing short words to be completed while longer words to be truncated to a short length. For example if the threshold is 5 and the truncate length is 3 a description like "Patient Visit - Additional Info." becomes to '`PatVisitAddInfo`'.

This produces names which are sufficiently unique in most cases. However there are ambiguities in table values, which require a refinement of the name assembling algorithm. These ambiguities occur, when there are single words which begin with a common prefix like the latin preposition "intra" or the greek quantifier "milli". Thus "intravenous" and "intradermal" both become '`Int`'. This can be overcome by splitting such composite words into two ("intra venous" giving '`IntraVen`').

4.1.2 States

Any object according to HL7 must have (at least) the following states:

1. present or not present
2. null or something other than null

The status qualities of the object can be seen by the member functions named 'is<quality>()'. However, modifying a status flag does not always make sense. Thus the 'present' quality can only be set, and the 'null' quality can only be cleared, by a class which is permitted to modify the components of the object (i.e. the object itself including derived objects). On the other hand, unset() and nullify() which makes an object not present or null respectively, may be called from any context which has an instance of an HL7 object.

[1] actually we merge these two lists into one

There are two other status flags, which are not defined in HL7, but are useful as a method to test the integrity of the Object. These qualities are

3. broken
4. zombie

A 'broken' object is one that was not initialized correctly. Normally a broken condition should end up in an exception, because it is always the result of a programming fault, however exception handling is currently not supported by some C++ compilers. Until then, it could make sense to use the broken bit.

The status of a 'zombie' means, that an object was destroyed before: either because it was explicitly deleted or that it went off the scope. Such an object is likely to contain damaged values. If an object is accessed via pointers and it happens that the zombie flag is set, there is a programming fault. These events should result in an error exit (with a core dump).

Finally there is a bit, which shows whether an object is atomic or a repeated object. This bit logically belongs to the class of the object, which is presented below (s. section 4.1.4 [Repetition and optionality], p. 39).

5. repeat

4.1.3 Classes and types of objects

An HL7 object may be of one of the following classes, which is not to be confused with a C++ class:

'`dset`'	the set of delimiters
'`message`'	a HL7 message
'`group`'	a group in a HL7 message
'`segment`'	a segment
'`datyp`'	a data type, either atomic or composite

Except for the set of delimiters, any class may be repetitive, which is reflected in the repeat bit in the status byte. An HL7 object of a certain class may be of some type. For example, if the object is a segment, then the type would be MSH or QRR etc. The appropriate values are taken from the table, which defines the message type, segment id, and data types.

The information about Class and Type is not as vital as is the status information. In fact, there are object classes which do not have a type (group, dset) or which can not repeat (dset). However, there are cases, when e.g. a segment has to be read, without knowing, in advance, what segment this will be. Therefore, we need a method to identify the type of a segment anyway, thus it seems reasonable to let the object identify itself. Classes and types can be set only by the object's setclass() and settype() members, which are not public. However the public can see the class and type of an object with theclass() and thetype().

4.1.4 Repetition and optionality

Repetition

Repetition is modeled by the class `repeated`, which is defined using the template feature and thus can be easily reused for any class of HL7 objects. The `repeated` class is a derived class from the class that is to repeat, enhancing the latter merely by a pointer to the next object which is called `cdr`, thus any repeated object is organized as a linked list. This is a more homogeneous approach than using arrays, since there is often no maximal repetition specified. There are member functions which allow the input and output of repeated object as well as an easy to use access to any member of the list, which is syntactically the same as referencing a member of an array. This could be achieved by overloading the array reference operator `[]`. The assignment operator is as well overloaded in order to make a copy of the list rather than just assigning the reference.

Optionality

Since the HL7 specifications demand a tolerance in respect to missing or unexpected objects, we do not care much about whether a segment is marked as required in the specifications. It is rather up to the application to reject incomplete messages, while unexpected objects tend to be just ignored.

4.1.5 Register

Since most C++ compilers impose a minimal size for a class, there may be some bytes left, which can be used by derived classes or are wasted otherwise. Sometimes, a derived class just needs one bit of information, which if stored individually in the space of the derived object, would require another word of minimal alignment width. In order to save memory space, the derived classes are invited to use the 'register' for the storage of their flags. sreg(), creg() and greg() give arbitrary access to the flags. To prevent conflicts along a derivation chain, the usage of the register must be documented in a register allocation table.

4.1.6 Methods

Any object has at least the following methods:

1. Two constructors, one with an empty parameter list and one with all components, which are accessible by the public. The former sets the object to non existent, the latter sets the object to existent and initializes it with the given parameters. The types of the parameters are the same as the types of the components. It is optional, to provide a constructor which takes C base types as parameters and performs the conversion.
2. A destructor, which tends to be an empty function. However, dynamically allocated memory is freed from here.
3. Selector(s) or extractor(s) one for each component of public relevance. Extractors are canonically named 'get<component>', where <component> is the name of the component that the extractor will access. The component is not returned as the value of the function but is assigned to a reference parameter. Selectors return an integer value reflecting the success of the extraction. A return Value of zero means that the extraction was performed successfully. Values less than zero mean that the object itself can not perform the extraction (often because it is not present or is null). A Value greater than zero tells that the returned object is unhealthy in any way.
4. Modifier(s) one for each component of public relevance. Modifiers always set an object to be present. nullify() is a special modifier.
5. Input method, returns 1 if reading was successful, 0 otherwise.
6. Output method. Input and output methods can handle different encoding rules depending on some properties of the stream. See S. section 4.2 [I/O methods], p. 40 for more.

4.2 I/O methods

The encoded segments finally will appear on a stream as well as the decoding will process data, that arrives on a stream. The stream may be bound to a RS232 line, a TCP/IP line, a batch file, a pipe or whatever medium is required.

There is an option to choose, whether the output methods are to produce HL7 encoding rule compliant data or human readable data, that is used in order to debug the protocol. This human readable data will be output in a LISP like style, since LISP provides a simple notation for complex objects. The style can however be modified on compile time.

The mechanisms that C++ provides with the 'iostream' library are very powerful and elegant. However we have to extend the iostream object by some variables, which allow us to set several states for the streams. There are currently the following states that a stream can assume:

'hl7er' The stream transports HL7 encoding rules

'debug' The stream transports human readable data

'level' A small integer, which tells about the delimiters currently to be used.

'hl7er' and 'debug' are boolean values which are of course mutually exclusive. They are however not represented in a single bit because there may be more coding modes to be integrated into the system.

The 'level' is normally set to 0, but there are cases, when the level of the stream must be increased. Consider a composite data type CX, which does refer in one of its components to an other composite data type CY: The components of CY thus become the subcomponents of CX. However, the i/o methods of CY need to know if CY is regarded as a composite field or as a composite component. If the latter is true, the level is increased by one and CY uses the subcomponent delimiter to terminate its components. However, it becomes clear that CY may not have in turn a composite component, since there is no sub-sub-component delimiter. In fact in v2.1 the subcomponent feature is never used. Rather such composites are flattened terminating each component by a '^' (for an example see the definition of the CN data type in chapter 2).

4.3 The code generator

The code generator is a program currently written in Prolog which produces C++ code from the HL7 data base. This compiler is currently under development and is still tentative. At this moment it generates code for composite data types, segments and tables. The last step of generating C++ classes for messages is not finished yet.

The Prolog program was first created in a monolithic approach and has then been split up into modules, which are far easier to maintain than the monolith. There are tables scattered throughout the modules, which enhance the data base which we have created by detailed information about many different things including composite types, required methods and their implementation etc.

In order to keep the complexity of the compiler as little as possible, the macro features of the C preprocessors are used extensively. The distinction made between abstract handling of objects (by macros) and concrete implementation (by the macro definitions) help to ease porting of the code to different platforms.

4.3.1 Tables

Tables are not regarded as HL7 objects as was described above (s. section 4.1 [General concept], p. 35), since tables are not exchanged during transactions. Rather tables provide means to interpret ID data fields correctly. They are merely classes with an enum type which is a mapping from symbolic names to integers, and a static array, which provides the mapping from the integers to character strings. Finally some useful member functions are provided which look up the item number of an ID in the table of character strings or translate such a number back to an ID. Even though these table objects are simple, it seems reasonable to enhance them to interface a data base management system for bigger tables as are classifications and other coding systems.

Some identifiers used by HL7 are listed in a HL7 table. Notably the message type identifiers. But there are others, such as segment and data type identifiers, which are not listed in such a table. It would be of some use for implementations to have these tables. Rather than keeping distict tables for the implementation of the protocol and the application using the protocol, there should be only one set of tables which are uniformly used by both. Thus we will generate entries in the '`Table`' and '`Table Value`' relation which list segment and type identifiers. To avoid conflicts with table numbers we will count down from the highest possible number (i.e. 9999 for tables and 999999 for values).

5 Integration into the system

The HL7 implementation as we have built it so far consists merely of a library which handles encoding and decoding. However, means have to be provided, by which these functions are controlled. In general, there are many possibilities how to do this. Here we only give a short list of what will be included into the system:

- a batch file interpreter
- a module that handles incoming unsolicited messages, i.e. a program that is invoced by the inter net daemon (inetd(8)) which listens at a certain TCP/IP port for incoming packets
- a module that does the same job on RS-232 lines by listening for incoming messages
- a module that helps connecting a HL7 server via TCP/IP or RS-232

All these modules will use some kind of lower layer protocol as described in the HL7 v2.1 document. While on TCP/IP links the minimal lower layer protocol would be sufficient, the X3.28 based data link protocol will have to be implemented to support RS-232 connections if SLIP or PPP is not used.

Appendix A Bibliography

KUPERMAN (1991)

Gilead J. Kuperman, Reed M. Gardener, T. Allan Pryor: *HELP: A dynamic hospital information system*, Springer-Verlag, 1991.

ROSE (1990)

Marshall T. Rose: *The open book: A practical perspective on OSI*, Prentice-Hall, 1990.

LEVENE (1992)

Mark Levene: *The nested universal relation database model*; Vol 595, G. Goos and J. Hartmanns (Eds.): *Lecture notes in computer science*, Springer-Verlag, 1992.

Contents