Alma Mater Studiorum · Università di Bologna

Campus di Cesena
Scuola di Scienze
CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

MASCHERE COBOL RE-INGEGNERIZZATE IN FORMATO XML

Tesi in: Programmazione ad Oggetti

Relatore:
Chiar.mo Prof. MIRKO VIROLI
Correlatori:
Dott. Ing. GIOVANNI CIATTO

Presentata da:
LUCA PASSERI

ANNO ACCADEMICO 2016-2017
II SESSIONE

Abstract

Keywords: COBOL, XML, .NET, traduttore automatico, file indicizzati

Coblo-to-Net è un progetto commissionato dall'azienda Harvard Group per realizzare un traduttore automatico di programmi COBOL in ambiente .NET.

Il lavoro discusso in questa tesi riguarda una sezione di questo progetto, inerente alla traduzione dell'interfaccia grafica. L'interfaccia è organizzata a maschere, gli elementi principali di essa, che corrispondono a schermate; ogni maschera è formata da vari elementi grafici ed espone un certo insieme di funzionalità appartenenti ad un preciso contesto. Nell'ambiente originale esse vengono memorizzate in file binari, nei quali vengono codificati tutti i dati, in maniera tuttavia sconosciuta.

L'obiettivo di questa tesi è quello di realizzare un programma in grado di estrarre automaticamente tutte le informazioni rilevanti presenti nei file e trasferirle in un formato standard che consenta di averne una rappresentazione di più alto livello.

Inizialmente è stata necessaria una fase di comprensione del contenuto dei file in seguito alla quale è stata prodotta una descrizione astratta delle informazioni. Sulla base di questa è stata poi realizzata un'implementazione vera e propria.

Dai risultati ottenuti è emerso che tale *software* è in grado di estrarre correttamente tutte le maschere presenti nell'archivio dell'azienda con una precisione quasi assoluta.

Introduzione

La qui presente tesi è articolata come segue:

Nel capitolo 1 vengono descritti il progetto Cobol-to-Net, l'ambiente all'interno del quale sono inseriti i programmi COBOL da tradurre e vengono infine definiti gli obiettivi della tesi.

Nel capitolo 2 viene effettuata l'analisi del problema, effettuando inizialmente l'analisi dei requisiti e successivamente descrivendo il contesto suddividendo il problema in più sottoproblemi.

Nel capitolo 3 viene effettuata la progettazione della soluzione, mantenendo la suddivisione individuata in fase di analisi. Viene quindi individuata una soluzione al problema ad un livello di astrazione comunque alto.

Nel capitolo 4 viene realizzata una vera e propria implementazione del *software*, descrivendo le scelte effettuate ed i particolari implementativi.

Il capitolo 5 conclude il documento presentando i risultati ottenuti e accennando a possibili futuri sviluppi.

Indice

Capitolo 1

Progetto Cobol-to-Net

Harvard Group s.r.l. è un'azienda informatica che progetta e realizza software gestionali per aziende. Uno dei suoi prodotti principali è DbWin, un *software* con interfaccia Windows. Tale sistema si basa principalmente su sorgenti scritti in COBOL, un linguaggio di programmazione nato nel 1961 per lo sviluppo di programmi gestionali. Tuttavia utilizza anche alcune tecnologie proprietarie non appartenenti allo standard del linguaggio, come ad esempio l'ESQL/COBOL, che estende il COBOL con un sintassi su misura per eseguire *query* su una base di dati IBM Informix, o il modulo VISA, che si occupa di gestire l'interfaccia grafica per l'interazione con l'utente.

Il linguaggio COBOL (*COmmon Business-Oriented Language*) era molto utilizzato per lo sviluppo di *software* gestionali ed è tuttora presente in alcune applicazioni. I punti di forza di tale linguaggio, rispetto ai concorrenti, che lo hanno portato al successo sono: una maggiore velocità e maggiore semplicità di gestione dell'input/output, l'aritmetica con il punto decimale fisso (molto utile nei programmi per la contabilità) e la sintassi simile a quella della lingua inglese. Inoltre i numeri sono trattati in maniera più vicina alla rappresentazione aritmetica umana: la virgola fissa o i numeri decimali vengono solitamente usati al posto della virgola mobile.[9]

Tuttavia si tratta di tecnologie datate, per cui il desiderio di HG (Harvard Group) è quello di trasferire questo prodotto in un ambiente moderno, Vb.NET, particolarmente indicato per la produzione di questo tipo di applicazioni. Ciò che scoraggia il processo di ri-progettazione e sviluppo nella nuova tecnologia di DbWin è la sua dimensione, che comporterebbe elevati costi, tempi di sviluppo e impiego di risorse. Pertanto una valida alternativa consiste nello sviluppare un *software* in grado di generare una versione dell'originale tradotta nella nuova tecnologia. Per fare ciò l'azienda si è rivolta all'università di Ingegneria e Scienze Informatiche di Cesena, la quale ha accolto questo progetto.

Il progetto Cobol-to-Net pertanto consiste nella realizzazione di un *software* in grado di tradurre automaticamente o semi-automaticamente l'insie-

me di programmi che costituiscono il prodotto DbWin.

1.1 Struttura dell'interfaccia grafica

Di questo progetto fa parte anche la traduzione dell'interfaccia grafica, della quale in seguito è mostrata la struttura.

La GUI di questi programmi viene realizzata mediante il modulo Visa, una libreria scritta in linguaggio C. I componenti principali di Visa sono le maschere, che corrispondono a schermate le quali permettono all'utente di interagire con il programma. Ogni maschera raggruppa un insieme di funzionalità appartenenti ad un preciso contesto e possiede vari elementi per permettere l'interazione.

Tali elementi sono: campi di testo, etichette e altri elementi grafici come bordi e linee; dove per ciascuno possono essere specificati valori agli attributi che possiedono in base alla semantica

In seguito verranno mostrati gli elementi e corrispondenti attributi fornendo una breve spiegazione per ciascuno di essi.

- **Field:** campo di testo per permettere la comunicazione di dati tra utente e programma
  - **id:** stringa alfanumerica formata da 4 caratteri tramite la quale i campi vengono referenziati nel codice
  - **fieldType:** carattere alfabetico indicante il tipo del campo, può corrispondere ad uno dei seguenti valori:

    I: *input*, l'utente può modificare il campo e inserirvi i propri dati

    O: *output*, il campo non è modificabile, ha la funzione di mostrare dati all'utente

    U: *update:* analogo al tipo *input*

    D: *display:* come *output* ma il valore non viene memorizzato nel buffer
  - **dataType:** stringa alfabetica formata da 2 caratteri alfabetici esprimenti il tipo di dato del campo. Questo campo viene utilizzato principalmente perché in COBOL non si conosce a priori il tipo di un dato, perciò tramite questo tale attributo è possibile reperirlo e capire quale codifica utilizzare. Può assumere uno dei seguenti valori:

    AN: alfanumerico

    AL: alfabetico

    NU: numerico

    NE: numerico editabile

    GR: grafico

NN: numerico negativo
DC: data controllata
DF: data free
OK: ok
IM: importo
CA: codice articolo
YY: year

- **editable:** carattere esprimente il valore booleano che indica se il campo è modificabile. Può assumere uno dei seguenti valori:[1]

  Y: true
  N: false

- **skipable:** carattere esprimente il valore booleano che indica se il campo è evitabile, un apposito pulsante può essere premuto per evitare tutti i campi con tale attributo attivo
- **visible:** carattere esprimente il valore booleano che indica se il campo è visibile
- **validable:** carattere esprimente il valore booleano che indica se deve essere passato il controllo al programma per effettuare validazione
- **explicitConfirm:** carattere esprimente il valore booleano che indica se occorre esplicitare la conferma (invio) per passare al campo successivo. Se non è attivo appena viene raggiunta la lunghezza prefissata del campo si effettua la transizione automaticamente
- **controlTable:** carattere esprimente il valore booleano che indica se è possibile specificare l'insieme di valori ammessi per questo campo
- **upperCase:** carattere esprimente il valore booleano che indica se tutti i caratteri devono essere trasformati nei relativi maiuscoli
- **inputOffset:** numero a 5 cifre indicante l'offset del campo sul buffer di input
- **outputOffset:** numero a 5 cifre indicante l'offset del campo sul buffer di output
- **filler:** carattere utilizzato per denotare l'assenza di informazioni per le stringhe. In COBOL infatti non esiste il concetto di NULL, per cui ad esempio una stringa è formata da una sequenza di caratteri più degli spazi vuoti per riempire la lunghezza prefissata. Nel caso di un numero esso viene riempito con degli zeri.

[1] Per tutti i casi in cui il carattere esprime un valore booleano si farà riferimento a questa sintassi

  - **numberOfDecimalDigits:** numero a 2 cifre che indica il numero di cifre decimali, può essere specificato un valore compreso tra 0 e 15
  - **thousandSeparator:** carattere esprimente il valore booleano che indica se si vuole visualizzare il separatore delle migliaia, il quale occupa uno spazio
  - **signed:** carattere esprimente il valore booleano che indica se si vuole visualizzare il segno
  - **fillWithZeros:** carattere esprimente il valore booleano che indica se si intende riempire gli spazi vuoti con la cifra 0
  - **packed:** carattere esprimente il valore booleano che indica la codifica per la memorizzazione nel buffer (ASCII/binaria)
- **Box:** figura geometrica corrispondente ad un rettangolo per raggruppare determinati elementi della maschera
  - **topLeftRow:** numero a 2 cifre che indica la riga del punto in alto a sinistra
  - **topLeftColumn:** numero a 2 cifre che indica la colonna del punto in alto a sinistra
  - **bottomRightRow:** numero a 2 cifre che indica la riga del punto in basso a destra
  - **bottomRightColumn:** numero a 2 cifre che indica la colonna del punto in basso a destra
- **Horizontal line:** figura geometrica corrispondente ad una linea orizzontale per evidenziare la separazione di sezioni all'interno della maschera
  - **row:** numero a 2 cifre che indica la riga
  - **topLeftColumn:** numero a 2 cifre che indica la colonna del punto in alto a sinistra
  - **bottomRightColumn:** numero a 2 cifre che indica la colonna del punto in basso a destra
- **Vertical line:** figura geometrica corrispondente ad una linea verticale per evidenziare la separazione di sezioni all'interno della maschera
  - **column:** numero a 2 cifre che indica la colonna
  - **topLeftRow:** numero a 2 cifre che indica la riga del punto in alto a sinistra
  - **bottomRightRow:** numero a 2 cifre che indica la riga del punto in basso a destra

- **Label:** testo che rappresenta l'etichetta di un campo
  - **row:** numero a 2 cifre che indica la riga di inizio
  - **column:** numero a 2 cifre che indica la colonna di inizio
  - **length:** numero che indica la lunghezza delle colonne
  - **reverse:** valore booleano che permette di visualizzare lo sfondo scuro e il colore del carattere chiaro
  - **highBrightness:** valore booleano che permette di visualizzare un colore del testo differente
  - **blinking:** valore booleano che permette di visualizzare un colore del testo differente
  - **underlined:** valore booleano che permette di visualizzare il testo sottolineato

1.2 Definizione degli obiettivi

L'obiettivo che ci si prefigge di realizzare in questa tesi è quello di sviluppare un *software* che sia in grado di tradurre in maniera automatica le maschere dell'interfaccia grafica in un formato moderno e facilmente comprensibile.

Come tecnologia di arrivo è stato scelto il formato XML, poiché consente di avere una rappresentazione dell'interfaccia ad alto livello ed è virtualmente compatibile con qualunque linguaggio di programmazione.

Capitolo 2

Analisi

Il primo passo da eseguire per realizzare tale progetto consiste nell'effettuare un'analisi del problema. Verrà quindi individuato e studiato in modo preciso il problema ed il contesto in cui esso è inserito.

2.1 Analisi dei requisiti

Il progetto consiste in un *software* in grado di estrarre maschere in formato XML a partire da file binari in maniera il più possibile automatizzata.

Per creare un nuovo linguaggio XML è necessario scegliere un linguaggio schema, vale a dire un documento attraverso cui si specificano le caratteristiche strutturali di un documento XML attraverso una serie di "regole grammaticali". In particolare definisce l'insieme degli elementi del documento XML, le relazioni gerarchiche tra essi, l'ordine di apparizione nel documento e l'opzionalità degli elementi. La scelta è ricaduta su XML Schema, poiché è una tecnica recente ed avanzata.[6][2]

Come dato di partenza si sa che le le informazioni sono codificate all'interno di un unico file. Questo file possiede quindi tutte le maschere, le quali a loro volta possiedono una serie di elementi grafici, che sono *field*, *box*, *horizontal line*, *vertical line*, *label*. È inoltre noto che tale contenitore è un particolare tipo di file COBOL, un file indicizzato, il quale rappresenta una tecnologia alternativa all'approccio classico, vale a dire l'accesso sequenziale al contenuto.

Non essendo reperibile la documentazione che spiega il procedimento di scrittura delle informazioni delle maschere occorre effettuare un analisi per *reverse-engegneering*, vale a dire studiare il prodotto per crearne una rappresentazione ad alto livello di astrazione.

Nonostante l'ingegneria inversa sia una tecnica molto potente vi sono alcune problematiche intrinsecamente legate ad essa. Trattandosi di un approccio che va dal particolare al generale si rischia di produrre un'astrazione errata per alcune situazioni.

Nel caso in oggetto inoltre è impossibile verificare la validità dell'astrazione prodotta per tutte le possibili casistiche, vista la loro vastità; per questo motivo occorre studiare un modo per verificare che il risultato del *reverse-engegneering* sia corretto.

Il problema suddetto può essere decomposto in più sottoproblemi, ciascuno indicante un passo all'interno del percorso di estrazione:

1. Comprensione della struttura interna dei file indicizzati Cobol
2. Estrazione della descrizione binaria di ogni maschera
3. Comprensione della codifica interna di ogni maschera
4. Conversione dalla codifica interna
5. Applicazione della conversione a tutte le maschere
6. Valutazione dei risultati

2.2 Comprensione della struttura interna dei file indicizzati Cobol

Come già accennato uno dei punti di forza di questo linguaggio è la gestione dell'Input/Output ed in particolare la comodità relativa all'utilizzo di file. In COBOL esistono infatti tre tipi di file: file sequenziali, file relativi e file indicizzati.

Inizialmente occorre capire come sono strutturati i file indicizzati in Cobol; questa rappresenta una fase di studio essenziale per meglio comprendere il problema della traduzione.

Va perciò effettuato uno studio su come in generale Cobol gestisce i file ed in particolare sulla struttura di quelli indicizzati. A partire dalle conoscenze acquisite si individuano poi i tratti caratteristici di essi per il file in questione.

2.3 Estrazione della descrizione binaria di ogni maschera

Come dato di partenza si ha un file indicizzato all'interno del quale sono codificate tutte le informazioni.

Occorre perciò studiare un modo separare le maschere al fine di ottenere tanti file quante sono le maschere, suddividendo semplicemente il contenuto.

Tale operazione ha come scopo quello di facilitare l'operazione di comprensione delle informazioni, permettendo di analizzare le singole maschere, e anche quella di semplificare l'estrazione vera e propria, traducendo cioè un file per volta.

2.4 Comprensione della codifica interna di ogni maschera

Una volta suddivise le maschere in più file si può iniziare quindi a studiare il contenuto esse.

Questa rappresenta la parte principale di tutto il progetto, infatti comprendere la logica secondo cui vengono memorizzate le maschere permette di avere un'idea astratta di come poter ottenere i dati da esse.

Ciò che si sa è che ogni maschere possiede degli elementi, che possono essere *field*, *box*, *horizontal line*, *vertical line* e *label*. Non è noto se esiste un ordine di questi elementi e come ognuno di questi sia codificato.

Questo passo rappresenta quello potenzialmente più complicato da realizzare, poiché occorre basarsi su pochi casi ed astrarre una logica generale valida per tutti. La realizzazione di una descrizione precisa dipende infatti dall'accuratezza dell'analisi del contenuto, avendo particolare attenzione a verificare le ipotesi tramite delle prove inconfutabili.

Se tale descrizione dovesse risultare inesatta si rischierebbe di dover rieffettuare parzialmente la progettazione sulla base dei nuovi dati, portando ad una degradazione del *software*, e questo è ciò che si intende evitare.

2.5 Conversione dalla codifica interna

Il passo successivo alla comprensione della codifica interna delle maschere è la conversione di essa.

Il problema consiste nel trovare un modo per effettuare la conversione delle informazioni comprese in un'altra tecnologia. Inoltre, nonostante sia già stata scelta tale tecnologia si intende mantenere un livello di astrazione maggiore, offrendo un meccanismo che consenta di cambiare facilmente la tecnologia di arrivo.

La difficoltà risiede nel trovare il giusto compromesso tra l'individuazione di una tecnica che permetta in maniera semplice la traduzione e il mantenimento di un alto livello di astrazione che consenta di variare tecnologia.

2.6 Applicazione della conversione a tutte le maschere

Compreso l'algoritmo per tradurre una qualunque maschera occorre applicarlo a tutte le maschere estratte dal file iniziale.

In teoria se nelle fasi precedenti è stata realizzata una descrizione completa e precisa in questa fase non vi è alcuna difficoltà. Tuttavia, considerata la grande quantità di maschere presenti e le possibili casistiche che si potrebbero verificare, si ritiene probabile che nell'atto di estendere la conversione

a tutte le maschere (che sono nell'ordine delle decine di migliaia) possano emergere delle situazioni inaspettate. In questo caso occorre modificare la descrizione definita in precedenza.

2.7 Valutazione dei risultati

Questa passo è strettamente legato a quello precedente. Ciò che si desidera è ottenere un riscontro sulla validità dei risultati prodotti.

Il problema non è affatto banale poiché non esiste un modo diretto per affermare che la conversione sia avvenuta con successo per tutte le maschere, se non quella di visualizzarne l'*output* per ciascuna confrontandolo con quello originale.

Ciò ovviamente non è fattibile vista la mole di file generati, per cui occorre studiare una tecnica che riesca a verificare in maniera automatica se le maschere generate siano analoghe a quelle originali e, nel caso contrario, che riporti le differenze con maggior precisione possibile.

Capitolo 3

Progettazione

In fase di analisi sono stati individuati i principali passi del processo di estrazione delle maschere, in questa sezione verrà riproposta la medesima suddivisione effettuando per ciascun sottoproblema individuato la progettazione.

Verrà effettuato uno studio di come risolvere, ad un livello alto di astrazione, tali problemi.

1. Comprensione della struttura interna dei file indicizzati Cobol
2. Estrazione della descrizione binaria di ogni maschera
3. Comprensione della codifica interna di ogni maschera
4. Conversione dalla codifica interna
5. Applicazione della conversione a tutte le maschere
6. Valutazione dei risultati

3.1 Comprensione della struttura interna dei file indicizzati Cobol

Nonostante COBOL possa essere utilizzato per processare numerosi tipi di *data file*, generalmente viene usato per i *record-based* file, cioè file contenenti numerosi dati inseriti all'interno di una struttura che attribuisce significato alle informazioni.

Prima di procedere con ulteriori dettagli riguardo l'utilizzo di file è opportuno introdurre alcuni termini essenziali:

- **Field:** elemento di informazione relativo ad un oggetto, un attributo di tale oggetto
- **Record:** insieme di *Field* relativi ad un oggetto

– **File:** insieme di istanze di un *record template*

Generalmente, in altri linguaggi di programmazione, i file vengono cariati interamente in memoria al fine di consentire una navigazione più veloce, tuttavia questo approccio risulta problematico in programmi COBOL dove i file arrivano a contenere anche decine di milioni di dati, richiedendo perciò una quantità di memoria spropositata (occorre inoltre tenere conto del periodo in cui COBOL nasce). L'alternativa che si utilizza è quella di utilizzare l'approccio *record-based* e caricare in memoria un *record* per volta, per poi processarlo. Occorre quindi dichiarare la struttura del *record* in modo che possa essere predisposta la memoria per salvarne un'istanza. Questa porzione di memoria viene chiamata *record buffer* ed è relativa ad un singolo file. Ogni volta che si intende leggere o scrivere su file perciò si verifica la seguente procedura:

1. Per ogni file che si intende utilizzare viene creato un *record buffer*
2. Per ogni *record buffer* occorre dichiarare la struttura del *record*
3. Viene inserito il *record* all'interno del *record buffer* relativo al file da cui si vuole leggere o scrivere
4. Viene effettuata l'operazione desiderata (lettura/scrittura).

Al fine di creare un *record buffer* sufficientemente grande da contenere un *record* occorre specificare il tipo e la dimensione di ogni campo. Viene quindi calcolata la "dimensione su *buffer*" (cioè che dipende dal tipo di dato) di ciascun campo e la somma di queste "dimensioni su *buffer*" rappresenta la dimensione del *record buffer*.

Come già accennato il linguaggio COBOL permette di creare tre tipi di file: sequenziali, relativi e indicizzati.[3] Nel primo caso rappresenta quello più semplice dovei *record* vengono letti sequenzialmente, dall'inizio alla fine. Negli altri casi si parla invece di file ad accesso diretto.

A differenza dei file sequenziali, i file ad accesso diretto permettono l'accesso in due modi:

– Accesso diretto ai record tramite l'utilizzo di chiavi

– Accesso sequenziale

Inoltre essi permettono di effettuare modifiche direttamente sul file senza doverne creare un altro contente le modifiche (come avviene per quelli sequenziali). Questa tipologia di file si divide nei due casi mostrati in precedenza, vale a dire: file relativi e file indicizzati.

I file relativi sono organizzati in *record* aventi un *relative record number* ascendente, possono quindi essere pensati come una tabella uni-dimensionale

salvata su disco dove il *relative record number* corrisponde all'indice nella tabella. Si può notare subito che la chiave che si può utilizzare deve avere un valore numerico racchiuso in un determinato intervallo, dove ogni posizione contiene un elemento o è vuota; viene perciò allocata memoria dipendentemente dalla dimensione del *range*.

Abbiamo infine i file indicizzati, il cui funzionamento è più complesso rispetto agli altri.

3.1.1 File indicizzati

I file indicizzati sono uno strumento molto utile per l'accesso ai *record*: essi permettono infatti di specificare una *primary key*, che è il valore in base al quale i dati sono effettivamente ordinati, e delle altre chiavi chiamate *alternate keys*. Il grande vantaggio di questi file è che possono essere letti sia direttamente che sequenzialmente sulla base di una qualunque delle chiavi. Viene mostrato in seguito come ciò sia possibile.[1]

I record sono disposti in sequenza in base ad una *primary key* e sopra i *record* il *file system* crea un indice, come si può osservare nella figura 3.1.1

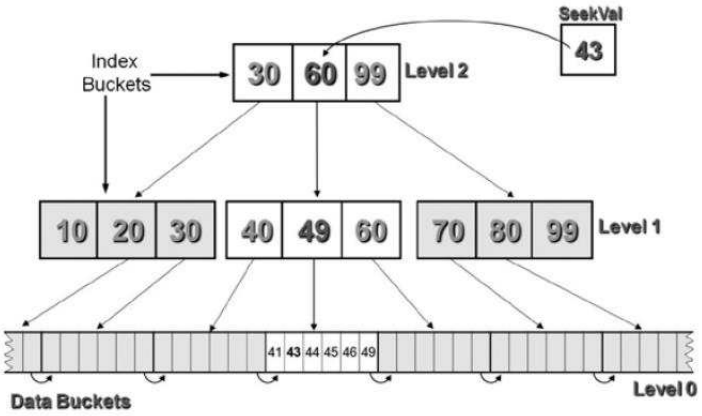

Figura 3.1: Struttura indicizzata basata sulla *primary key*. Viene mostrato il procedimento per reperire il record contenente il valore "43", che si trova nel livello 0.[3]

Al fine di comprendere l'immagine 3.1.1 è necessario spiegare il significato di alcuni termini:

- **Bucket:** il più piccolo numero di caratteri che possono essere letti/-scritti in una singola operazione di I/O

- **Index dept:** numero di livelli sopra il livello 0 (in figura, quello più in basso)

Si può osservare che, partendo dall'alto, ogni elemento di un dato livello (eccetto il livello 0) ha un puntatore al più grande elemento del livello successivo (con *index dept* minore), in questo modo si procede tra i livelli confrontando i valori e raggiungendo il livello 0. A quel punto dentro il *bucket* si trova il record desiderato. Questa struttura rende molto semplice leggere i file ordinando in base alla *primary key*, ma come si può fare per leggerli ordinando per *alternate key*?

Per fare ciò, per ognuna delle *alternate keys* si costruisce una struttura del tutto analoga a questa, dove però nel livello 0 non troviamo i *buckets* contenenti i record, ma solamente il valore cercato ed un puntatore al record nell'indice costruito sulla *primary key*. Nella figura 3.1.1 è mostrata tale struttura

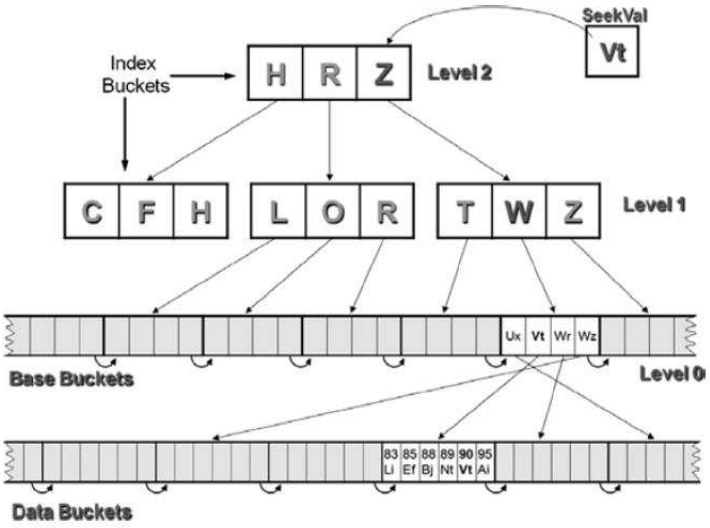

Figura 3.2: Struttura indicizzata basata su una *alternate key*. Viene mostrato il procedimento per reperire il record contenente il valore "Vt". Nel livello 0 non si trova il record, ma il valore cercato ed un puntatore al record, che si trova nell'indice costruito sulla *primary key*.[3]

In base a quanto compreso si può analizzare ora la struttura del file dal quale estrarre tutte le maschere. Si osserva che esistono in realtà due file collegati fra loro: uno con estensione "dat" ed uno con estensione "idx". In seguito ad una ricerca su come effettivamente COBOL gestisce i file indicizzati è stato scoperto che nel file "dat" risiedono i record veri e propri, mentre nel file "idx" vi sono le chiavi per effettuare l'accesso diretto con le *alternate keys*.

Per quanto riguarda la realizzazione dell'estrattore non è necessario l'utilizzo delle *alternate key*, è sufficiente infatti leggere il file "dat", all'interno

del quale sono memorizzati i record veri e propri ordinati in base alla chiave primaria.

3.2 Estrazione della descrizione binaria di ogni maschera

Il problema che si intende risolvere in questo passo è come suddividere le maschere contenute nel file di partenza.

Occorre infatti inizialmente comprendere come sono strutturati i record all'interno del file ed in seguito trovare un modo per separare le maschere.

Osservando il file si nota che ricorre con una certa distanza una sequenza di simboli alfanumerici con lunghezza fissa (8 byte); perciò presumibilmente questa indica la chiave primaria secondo cui sono ordinati i *record*. Ciò che segue tale chiave chiave è perciò il *record* vero e proprio, del quale si misura la lunghezza contando tutti i byte fino alla chiave seguente (501 byte). In realtà l'ultimo byte del *record*, essendo sempre costante, viene utilizzato come elemento per indicarne la fine, per cui il record vero e proprio è formato da 500 byte.

Tramite un confronto con le maschere presenti nell'archivio di HG è emerso che i nomi delle maschere corrispondono ai primi 6 byte delle chiavi primarie. Perciò occorre capire quale significato abbiano gli altri 2 byte. Si nota immediatamente che gli ultimi 2 byte della chiave corrispondono sempre ad un numero; osservando tale numero nel caso di più record è stato scoperto che esso indica la posizione del record legato alla chiave all'interno della maschera data dai primi 6 byte.

In pratica nel caso in cui una maschera sia formata da numerosi elementi e 500 byte non siano sufficienti a contenere tutte le informazioni vengono utilizzati più *record*. Per non perderne l'ordine perciò viene memorizzata la posizione utilizzando un numero sequenziale.

Utilizzando una grammatica formale, cioè una struttura astratta per definire formalmente un linguaggio, si ha la seguente definizione.[7]

1. $File \rightarrow Recordstruct$

2. $Recordstruct \rightarrow \epsilon \mid Maskname\ Recordnum\ Record\ Recordstop\ Recordstruct$

Dove *Maskname* è il nome della maschera di lunghezza 6 byte, *Recordnum* è il numero progressivo di lunghezza 2 byte, *Record* è il *record* vero e proprio di lunghezza 500 byte e *Recordstop* è il terminatore di lunghezza 1 byte. Perciò *Recordstruct*, che indica la struttura complessiva dei *record*, ha lunghezza 509 byte.

Sulla base di queste informazioni perciò risulta semplice suddividere il file di partenza in più file dove ognuno si riferisce ad una maschera. Occorre infatti concatenare tutti i *record* facenti riferimento alla stessa maschera nel giusto ordine. Con *record* si intendono esattamente i 500 byte di informazioni, vengono perciò escluse la chiave e il terminatore.

3.3 Comprensione della codifica interna di ogni maschera

Una volta suddiviso il file di partenza è possibile iniziare ad indagare sulla codifica interna di ogni maschera.

Come supporto all'elaborazione di teorie solide viene costruita una grammatica formale, aggiungendo produzioni man mano che vengono trovati nuovi pattern.

Analizzando attentamente più file si può notare che ogni maschera è divisa principalmente in 6 macrosezioni individuabili in ciascuno i essi, vale a dire:

1. Sezione iniziale
2. Sezione dei *field*
3. Sezione dei *border*
4. Sezione delle *label*
5. Sezione delle *control table*
6. Sezione finale

Dove si ricorda che con *border* vengono intesi *box*, *horizontal line* e *vertical line*

Sezione iniziale Si è notato che la sezione iniziale è composta costantemente da 12 byte di significato sconosciuto. In seguito sono stati decifrati alcuni elementi come per esempio la data di creazione della maschera.

Successivamente ad una più approfondita analisi è emerso che i dati riportati in questa sezione non apportano alcun contributo informativo poiché privi di significato (filler) o contenenti informazione irrilevanti.

Sezione dei *field* I *field* rappresentano la parte più complicata delle maschere, poiché possiedono numerosi attributi di carattere differente tra loro.

Appare subito evidente che i primi 4 byte (*offset* [0:3]) indicano l'id del *field*, i seguenti 4 (*offset* [4:7]) fanno riferimento alla *control table*, ed in particolare vi sono due casi:

- i byte indicano che la *control table* è assente
- i byte indicano che la *control table* è presente e permettono di identificare la posizione della *control table* nel file

Il byte con *offset* 8 indica il tipo del campo:*update*, *output*, e *display*. I successivi 2 (*offset* [9:10]) specificano la riga e la colonna del campo, mentre il byte con *offset* 11 ne determina lo stile: *normal*, *reverse*, *highbrightness*, *underlined*.

Nelle 4 posizioni successive (*offset* [12:15]) si trovano i byte che esprimono rispettivamente l'*input-offset* sul buffer (primi 2 byte) e l'*output-offset* (altri 2 byte). I byte con *offset* 17 e 19 sono legati alla lunghezza del campo nel seguente modo: il secondo indica il numero di caratteri stampati a video, perciò la lunghezza grafica, mentre il primo rappresenta la lunghezza all'interno del buffer, perciò varia in base al tipo di dato da memorizzare. All'*offset* 21 troviamo il *filler* del campo, il successivo (*offset* 22) indica se il campo è *validable* e il byte con *offset* 24 indica se il campo è *non-skipable*. Per quanto riguarda il secondo byte invece (*offset* 23) si è giunti alla conclusione che potrebbe esser riservato per modifiche future o nel corso del tempo ha perso di significato poiché con qualunque tipo di possibile variazione nella maschera il valore rimane costantemente a 0. Il byte con *offset* 26 infine è utilizzato per indicare la terminazione di un *field*. Sono stati appositamente omessi i byte con *offset* 16, 18 e 20, poiché il meccanismo secondo cui ne viene calcolato il valore è tutt'altro che semplice da intuire. Essi sono la chiave per comprendere il tipo di dato e in base ad esso possono essere specificati attributi diversi. Nei vari casi questi 3 campi possono variare in modo diverso delineando l'assenza di un pattern comune, si pensa dunque che tale situazione sia il risultato di varie modifiche ed aggiunte di elementi nel corso del tempo. Questi byte verranno chiamati rispettivamente M-Tipo, V-Tipo e PC-Edit, poiché tale è la nomenclatura utilizzata all'interno del programma che si occupa di leggere tali dati. In seguito è riportata la lista dei *data-type*:

- AN = "alphanumeric"
- AL = "alphabetic"
- NU = "numeric"
- NE = "editedNumeric"
- GR = "graphics"
- NN = "negativeNumeric"
- DC = "dataControllata"
- DF = "dataFree"

- OK = "ok"
- IM = "importo"
- CA = "codiceArticolo"
- YY = "year"

Oltre alla determinazione del dato, a partire da tali byte vengono determinati anche i restanti attributi: *modifiable*, *echo*, *explicit confirm*, *uppercase*, *number of decimal digits:*, *thousand separator:*, *packed*, *signed* e *fill with zeros:*.

Alcuni di questi attributi sono in realtà speficabili solo per certi tipi di dati, il che porta ad una complessità ancora maggiore la comprensione del pattern dei 3 byte coinvolti.

Sezione dei *border* Successivamente alla sezione dei *field* troviamo quella in cui sono contenuti i bordi. Essa contiene tutti gli elementi che hanno uno scopo puramente grafico, vale a dire: *box*, *horizontal line* e *vertical line*. Si può individuare un macro-pattern valido per entrambi gli elementi: un bordo è infatti formato da una dichiarazione più le coordinate dei due punti estremi necessari e sufficienti a costruire un elemento. I punti estremi sono le coordinate del punto in alto a sinistra e coordinate del punto in basso a destra; perciò nel caso delle linee si ha che o le righe o le colonne sono le stesse per entrambi i punti, mentre nel caso dei box sono differenti. In entrambi i casi i punti vengono utilizzati sempre 4 byte per le coordinate:

- *topLeftRow:* riga in alto a sinistra
- *topLeftColumn:* colonna in alto a sinistra
- *bottomRightRow:* riga in basso a destra
- *bottomLeftColumn:* colonna in basso a destra

Da tali dati si può già riconoscere se si tratta di una linea o di un *box*, tuttavia probabilmente per motivi di comodità al fine della costruzione della figura, questa differenza viene riportata anche nella dichiarazione, che è differente per tutti e tre gli elementi.

Non esiste un carattere speciale che indichi la fine della sezione dei *field* e l'inizio di quella dei bordi, tuttavia si può ovviare a questo problema osservando che il primo carattere di un *field* è sempre un carattere alfanumerico (poiché è il primo carattere dell'*id*), mentre il byte dichiaratore di un *border* non è mai un valore alfanumerico. Ad indicare il termine della sezione dei bordi è invece presente un carattere speciale.

Sezione delle label Le *label* possiedono una struttura piuttosto semplice: si ha infatti che tutte le informazioni (escluso il testo) sono memorizzate in 4 byte antecedenti la stringa vera e propria, che sono: riga, colonna, stile e lunghezza. In seguito a questi si trova il vero e proprio testo della *label*, il quale ha lunghezza pari al valore dell'attributo indicante la lunghezza. Tramite quest'ultimo infatti è possibile comprendere quando inizia una nuova *label*.

Sezione delle *control table* In seguito alla sezione delle *label* troviamo quella delle *control table* e ciò che viene spontaneo domandarsi è come mai nonostante esse siano strettamente legate ai *field* non siano contenute nella stessa sezione. Una plausibile risposta a tale interrogativo può essere che non si volesse interrompere la struttura di lunghezza costante dei *field*.

In questa forma di memorizzazione il legame con il relativo *field* avviene per mezzo di alcuni byte nei field che specificano dove trovare l'inizio della relativa *control table*, se presente. Il pattern che permette di identificare tali elementi è abbastanza semplice: ogni *control table* possiede una serie di valori permessi che devono necessariamente avere lunghezza pari al corrispettivo *field*, perciò tale valore viene utilizzato come "passo" per leggere i caratteri finché non viene incontrato il byte terminatore di una singola *control table*. Esiste anche una variante creata appositamente per gestire gli intervalli numerici e alfabetici, che consiste nello specificare un determinato byte; il risultato è che come valori ammessi vengono aggiunti tutti quelli compresi in questo intervallo. La terminazione della sezione delle *control table* viene indicata da un terminatore.

Sezione finale Il file termina con una sezione conclusiva che inizia con un byte che indica la fine del file. Ciò che segue tale valore sono una sequenza di byte usati per riempire l'ultimo record, che deve come gli altri avere lunghezza pari a 500.

Nonostante si abbia tentato di rendere la descrizione precedente più accurata e chiara possibile si riconosce che il linguaggio naturale è intrinsecamente poco adatto alla spiegazione di regole strutturali ricorrenti, per tale motivo si fa riferimento alla grammatica prodotta al termine del riconoscimento dei pattern, mostrata in seguito:

1. $Mask \rightarrow Init\ Fieldstruct\ Borderstruct\ Borderstop\ Labelstruct\ Labelstop\ Controltablestruct\ Maskend$
2. $Init \rightarrow AA\ MM\ GG\ Filler\ Filler\ Filler\ Filler\ Filler\ Numrecord\ Filler\ Numrighe\ Numcampi$
3. $Fieldstruct \rightarrow \epsilon \mid Field\ Fieldstruct$

4. $Field \rightarrow Id\ Controltableoffset\ Fieldtype\ Fieldrow\ Fieldcol\ Style\ Inoffs\ Outoffs\ MTipo\ Recordlength\ VTipo\ Videolength\ PCEdit\ Fieldfiller\ Validable\ PCIgno\ Nonskipable\ Fieldstop$

5. $Borderstruct \rightarrow \epsilon \mid Border\ Borderstruct$

6. $Border \rightarrow Box\ Line$

7. $Box \rightarrow Boxdecl\ Bordercoords$

8. $Line \rightarrow HLinedecl\ Bordercoords\ VLinedecl\ Bordercoords$

9. $Bordercoords \rightarrow TLrow\ TLcol\ BRrow\ BRcol$

10. $Labelstruct \rightarrow \epsilon \mid LabelLabelstruct$

11. $Label \rightarrow Labeldecl\ Labeltext$

12. $Labeldecl \rightarrow Labelrow\ Labelcol\ Style\ Labellength$

13. $Style \rightarrow Normal \mid Blinking \mid Highbrightness \mid Reverse \mid Underlined \mid (Style\ or\ Style)$

14. $Controltablestruct \rightarrow \epsilon \mid Controltable\ Controltablestruct$

15. $Controltable \rightarrow Allowedvaluestruct\ Controltablestop$

16. $Allowedvaluestruct \rightarrow \epsilon \mid Allowedvaluestruct\ Allowedvaluestruct \mid Allowedvalue \mid Allowedvalue\ Range\ Allowedvalue$

17. $Maskend \rightarrow Endoffile\ Fillerstruct$

18. $Fillerstruct \rightarrow \epsilon \mid Filler\ Fillerstruct$

In questa grammatica si vuole evidenziare la struttura generale delle informazioni presenti nei file, in base ai *pattern* identificati.

3.4 Conversione dalla codifica interna

Una volta compreso il modo in cui le maschere vengono salvate su file occorrerà effettuare l'operazione di trasposizione di queste nel linguaggio XML.

Lo *Schema* dell'XML contiene già tutti i nodi necessari alla costruzione di una maschera, per cui ciò che si intende fare è tradurre i caratteri presenti all'interno dei file nei relativi *tag* predisposti a seconda dell'elemento o attributo che si sta considerando.

Esiste una corrispondenza biunivoca tra gli attributi di ogni elemento nella tecnologia di partenza e quelli della tecnologia di arrivo. Perciò occorre

studiare il metodo migliore per la traduzione delle maschere ma prima ancora come memorizzare le informazioni comprese nella sezione precedente. La strategia che si intende adottare è quella di creare un'organizzazione ad albero in grado di mantenere oltre ai dati anche la gerarchia degli elementi. Questa idea deriva dall'osservazione che le informazioni sono organizzate su livelli differenti e ciò è ben visibile sia nella grammatica prodotta che in un esempio di file XML (dove i tag sono annidati tra loro).

Si intende dunque creare due strutture differenti, una usata per la realizzazione di un *template* adoperato per indicante gli elementi desiderati della maschera, mentre l'altra per memorizzare i dati veri e propri. Quest'ultima perciò possiede come "scheletro" un determinato *template* e manterrà tutti e soli i dati specificati da esso. Creando un'associazione tra i byte del file e gli elementi di questo *template* e tra questi e i tag dell'xml sarà possibile effettuare la conversione tramite uno strato intermedio che garantisce la corretta transizione da una struttura sequenziale ad una ad albero.

Concretamente il template corrisponde ad un'insieme di Enum che sono concettualmente organizzate in gerarchie:

- mask
    - field
        - id
        - index
        - row
        - column
        - length
        - fieldType
        - dataType
        - editable
        - skipable
        - visible
        - explicitConfirm
        - validable
        - uppercase
        - inputOffset
        - outputOffset
        - filler
        - numberOfDecimalDigits
        - packed
        - thousandSeparator
        - signed
        - fillWithZeros

      - reverse
      - highBrightness
      - blinking
      - underlined
      - controlTable
        - allowedValue
          - value
  - box
    - topLeftRow
    - topLeftColumn
    - bottomRightRow
    - bottomRightColumn
  - horizontalLine
    - orientation
    - row
    - startColumn
    - endColumn
  - verticalLine
    - orientation
    - column
    - startRow
    - endRow
  - label
    - row
    - column
    - length
    - reverse
    - highBrightness
    - blinking
    - underlined
    - text

Per ciascun nodo possedente almeno un figlio esiste una Enum; un *template* perciò è una struttura ad albero i cui nodi corrispondono agli elementi appena mostrati.

Come accennato in precedenza esiste un'altra struttura nella quale vengono memorizzati i valori relativi a tali attributi, essa ha sempre un'organizzazione ad albero ma questa volta consiste in una gerarchia di classi, una per ogni nodo avente figli. Ogni classe mantiene le informazioni necessarie, che fanno riferimento ai valori delle Enum.

L'introduzione di tale struttura comporta numerosi vantaggi:

- Viene separata la logica di estrazione da quella di conversione
- È possibile effettuare più semplicemente una forma di controllo dei dati
- È possibile variare più facilmente del formato di arrivo, per il fatto che i dati prima vengono parzialmente elaborati e memorizzati internamente

In questo caso è stato utilizzato il pattern Builder, ritenuto uno dei fondamentali della programmazione ad oggetti.[5] Il *design pattern* Builder consente di separare la costruzione di un oggetto complesso dalla sua rappresentazione cosicché possa essere data origine a diverse rappresentazioni di esso. Ciò che accade infatti è che non è la classe a doversi occupare della costruzione delle istanze ma un altro oggetto (Builder) al quale viene delegato anche il compito di controllare i valori dei campi, particolarmente utile nel caso in cui si voglia effettuare una validazione delle istanze di un oggetto. Nel caso in questione l'obiettivo è quello di evitare la proliferazione di costruttori per ogni possibile combinazione di dati specificati dal *template*, per tale motivo si ricorre a questo *pattern*; si ritiene infatti che possa apportare una maggiore qualità al codice offrendo un meccanismo elegante per creare differenti variazioni di uno stesso oggetto.

A questo punto occorre generare un file XML a partire da questo albero e la soluzione ideata è quella di inserire un metodo apposito al contratto delle classi. In questo modo una chiamata a tale metodo di un elemento provoca in cascata la chiamata di tale metodo per tutti gli elementi sottostanti, generando così l'albero XML.

3.5 Applicazione della conversione a tutte le maschere

In questo passo occorre estendere la conversione di una singola maschera a tutte quelle presenti nell'archivio di HG.

Il problema principale è che ciò che la grammatica prodotta nel passo precedente possa presentare variazioni inattese.

L'unica soluzione possibile è osservare per quali casi si verificano tali eventi ridefinire *pattern* studiati sulla base delle differenze. A questa sezione è fortemente legata quella successiva.

3.6 Valutazione dei risultati

Questa ultima fase consiste nel verificare che l'applicazione della conversione a tutte le maschere abbia un esito positivo.

Si intende infatti verificare che i file XML prodotti corrispondano in termini di informazioni a quelli originali e, nel caso non sia così, studiare un metodo per evidenziare le differenze.

Una prima semplice forma di verifica consiste nel considerare una limitata quantità di maschere di cui si conosce anticipatamente la resa grafica che devono avere e osservare se l'output delle maschere tradotte corrisponde. Tale operazione consente di avere un'immediata cognizione della correttezza dei tradotti a grandi linee, tuttavia ovviamente non è sufficiente poiché dovendo adibire questo programma all'estrazione di decine di migliaia di file potrebbero nascondersi ovunque errori o casi non contemplati. Vi è infatti anche la possibilità che vi siano variazioni rare dei pattern individuati per cui si riesca ad estrarre una maschera apparentemente funzionante che però non corrisponde in termini di aspetto o di funzionalità offerte a quella originale.

Per tale motivo si è ritenuto che la soluzione migliore fosse quella di sviluppare un estrattore inverso, che quindi generasse file.dat a partire dai file "xml" per poi confrontare il prodotto con il file "dat" originale tramite un terzo software (comparatore).

La potenza di questa tecnica risiede nel fatto che se sia l'estrattore che il traduttore inverso vengono realizzati sulla base dei pattern trovati il comparatore mette in luce tutti i casi in cui tali pattern vengono violati, permettendo quindi di effettuare nuove ipotesi sulla base di tali variazioni.

Il primo si basa fortemente sui prodotti dell'analisi e progettazione relativi all'estrattore precedente. Partendo dall'XML descrivente una maschera si memorizzano le informazioni riportate all'interno della struttura usata come strato intermedio. In questo modo vengono disassemblati i dati mantenendo comunque la struttura ad albero dell'XML; in seguito si sfruttano i *pattern* riconosciuti per effettuare l'operazione inversa, cioè scrivere un valore binario in base al valore di tale attributo. L'inverso risulta più semplice in generale, ed in particolare per quanto riguarda i valori degli attributi problematici, vale a dire quelli che sono influenzati da più attributi.

Riutilizzando il lavoro svolto per l'estrattore principale perciò non dovrebbe risultare complesso realizzare questa soluzione.

Il secondo passo è quello di creare un componente in grado di confrontare due file e comprendere se apportano lo stesso contributo informativo. In questo caso perciò non occorre nemmeno utilizzare una struttura per organizzare i dati, in quanto si tratta semplicemente di verificare che il contenuto di entrambi i file corrisponda. Un primo ipotetico approccio potrebbe essere semplicemente quello di esaminare ogni singolo byte e riportare le differenze, tuttavia ciò evidenzierebbe anche casi in cui sono presenti dissimilarità irrilevanti. Si rammenta infatti che non tutti i dati presenti nei file vengono riportati nel relativo XML, ma solo quelli strettamente necessari a costruire l'interfaccia grafica, pertanto di tutte le informazioni superflue viene persa memoria. Alla luce di tale osservazione si può applicare concettualmente un filtro al comparatore, ignorando cioè tutti i byte il cui valore è trascurabile per la realizzazione della GUI. In questo modo i restanti dati saranno gli unici da leggere e di cui confrontare il valore e, in caso di divergenza, verrà

indicato per quale attributo ciò si verifica, in modo da poter semplificare la comprensione di eventuali pattern sconosciuti o mal compresi.

Capitolo 4

Sviluppo

In questo capitolo verrà mostrato come si realizzerà nella pratica una soluzione al problema esposto, sviluppando quindi un'implementazione vera e propria. Come linguaggio di programmazione è stato scelto Python, semplice e potente e che offre supporto alla programmazione orientata agli oggetti.[10] In questo caso si ritiene che tale linguaggio ben si presti all'esigenza poiché la struttura su cui si basa l'estrattore può essere efficacemente modellata usando il paradigma ad oggetti, mentre per la parte di decodifica dei file viene in aiuto la dinamicità e semplicità di Python.

Inizialmente verrà condotto uno studio più approfondito del contesto, concentrandosi sulla comprensione del contenuto del file per ciò che riguarda i casi più complessi.

In seguito verrà mostrata e discussa l'implementazione vera e propria.

4.1 Comprensione del contenuto dei file

In seguito viene mostrata la grammatica proposta precedentemente completa dei terminali, vale a dire i byte veri e propri all'interno dei file.

1. $Mask \rightarrow Init\ Fieldstruct\ Borderstruct\ Borderstop\ Labelstruct\ Labelstop\ Controltablestruct\ Maskend$
2. $Init \rightarrow AA\ MM\ GG\ Filler\ Filler\ Filler\ Filler\ Filler\ Numrecord\ Filler\ Numrighe\ Numcampi$
3. $Fieldstruct \rightarrow \epsilon \mid Field\ Fieldstruct$
4. $Field \rightarrow Id\ Controltableoffset\ Fieldtype\ Fieldrow\ Fieldcol\ Style\ Inoffs\ Outoffs\ MTipo\ Recordlength\ VTipo\ Videolength\ PCEdit\ Fieldfiller\ Validable\ PCIgno\ Nonskipable\ Fieldstop$
5. $Borderstruct \rightarrow \epsilon \mid Border\ Borderstruct$
6. $Border \rightarrow Box\ Line$

7. $Box \rightarrow Boxdecl\ \ Bordercoords$
8. $Line \rightarrow HLinedecl\ \ Bordercoords\ \ VLinedecl\ \ Bordercoords$
9. $Bordercoords \rightarrow TLrow\ \ TLcol\ \ BRrow\ \ BRcol$
10. $Labelstruct \rightarrow \epsilon \mid LabelLabelstruct$
11. $Label \rightarrow Labeldecl\ \ Labeltext$
12. $Labeldecl \rightarrow Labelrow\ \ Labelcol\ \ Style\ \ Labellength$
13. $Style \rightarrow Normal \mid Blinking \mid Highbrightness \mid Reverse$
    $\mid Underlined \mid (Style\ or\ Style)$
14. $Controltablestruct \rightarrow \epsilon \mid Controltable\ \ Controltablestruct$
15. $Controltable \rightarrow Allowedvaluestruct\ \ Controltablestop$
16. $Allowedvaluestruct \rightarrow \epsilon \mid Allowedvaluestruct\ \ Allowedvaluestruct$
    $\mid Allowedvalue \mid Allowedvalue\ \ Range\ \ Allowedvalue$
17. $Maskend \rightarrow Endoffile\ \ Fillerstruct$
18. $Fillerstruct \rightarrow \epsilon \mid Filler\ \ Fillerstruct$
19. $Filler \rightarrow 0x00$
20. $Labelstop \rightarrow 0x00$
21. $Fieldtype \rightarrow 'U' \mid 'O' \mid 'I' \mid 'D'$
22. $Fieldstop \rightarrow 0x00$
23. $BorderDecl \rightarrow 0x03$
24. $BorderStop \rightarrow 0x00$
25. $Normal \rightarrow 0x00$
26. $Blinking \rightarrow 0x10$
27. $Highbrightness \rightarrow 0x20$
28. $Reverse \rightarrow 0x40$
29. $Underlined \rightarrow 0x80$
30. $Validable \rightarrow 0x30 \mid 0x31$
31. $PCIgno \rightarrow 0x30 \mid 0x31$
32. $Nonskipable \rightarrow 0x30 \mid 0x31$

33. $Controltablestop \rightarrow 0x0B$

34. $Endoffile \rightarrow 0xFF$

I simboli terminali mostrati in questa grammatica sono quelli facilmente rappresentabili, mentre per quelli che presentano *pattern* complessi viene riportata in seguito una descrizione. Tali sono i casi di alcuni byte dei *field*, vale a dire: *MTipo*, *VTipo* e *PC-Edit*.

Come già accennato tramite questi tre valori è possibile comprendere il tipo del dato del campo.

Inizialmente conviene guardare il valore binario di V-TIPO, se gli ultimi 2 bit sono "01" allora il tipo è NE, se vi è "11" si tratta di un NN, in tutti gli altri casi sono "00" e si considera il valore di M-TIPO. Si osserva il suo valore in esadecimale e se corrisponde a "0x0F" si può affermare che il campo è un DC, vi è poi "0x11" per OK, "0x14" per IM, "0x15" per IA, "0x13" per YY e "0X10" per DF. Nel caso non si sia riuscito ancora a determinarne il tipo se si incontra "0x01" come valore per M-Tipo allora si può stabilire che il tipo è NU, poichè tra quelli rimasti è l'unico a poter assumere tale valore, per gli altri infatti vale "0x00". In quest'ultimo caso si considera il campo PC-Edit, in particolare gli ultimi 3 bit di tale byte: se si ha "001" allora il campo è di tipo AL, con "011" il tipo è AN e con "111" GR. In seguito è riportato un automa a stati finiti (4.1) per comprendere più facilmente l'algoritmo appena esposto.[7]

Oltre alla determinazione del tipo di dato tramite questi soli tre byte è possibile comprendere altri attributi, i quali in alcuni casi esistono solo per alcuni tipi di dato. In seguito viene mostrato come ciò avviene.

MTipo: è influenzato solamente dal tipo del dato, nel modo descritto in precedenza, eccetto per i tipi NE e NN, per cui *packed* lo imposta al valore "0x06" mentre *signed* a "0x05"; nel caso entrambi siano presenti il valore dominante è quello di *packed*

VTipo: il valore base è costante, eccetto per NE e NN, in ogni caso però viene modificato da tre attributi: *modifiable*, *echo* ed *explicit confirm*.

PCEdit: è influenzato dal tipo di dato e nei casi comuni da *uppercase*; nel caso di NE e NN l'attributo *uppercase* non è presente ma il valore di tale campo viene alterato da: *number of decimal*, *thousand separator*, *signed* e *fill with zeros*

4.2 Implementazione

Il progetto che si andrà a realizzare viene scomposto in 4 software, ciascuno con una funzionalità ben precisa. Abbiamo infatti:

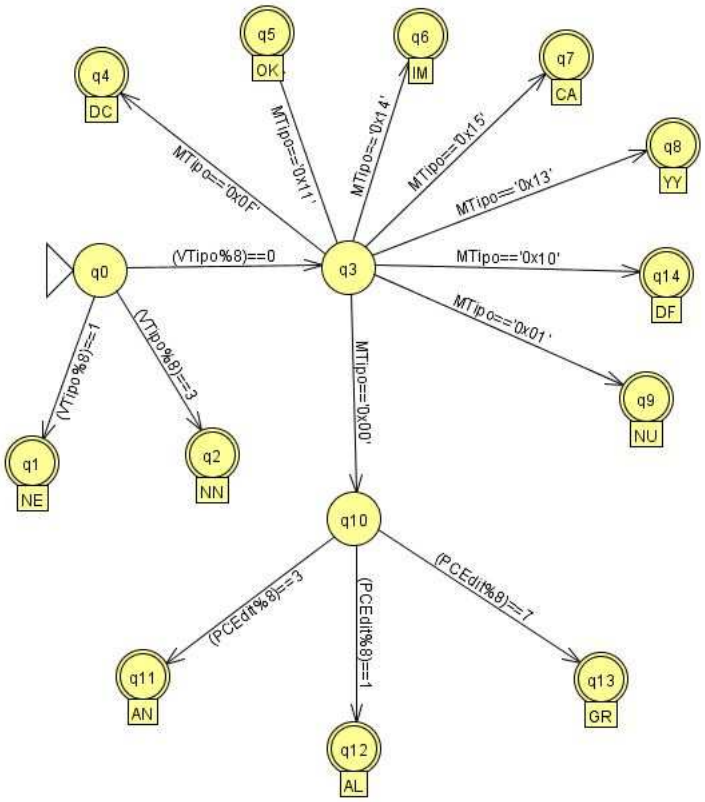

Figura 4.1: Automa a stati finiti per riconoscere il tipo di dato del *field*. Tramite l'operazione (VTipo%8) si ottiene il valore del byte meno significativo in decimale e si confronta tale valore; ciò è del tutto analogo a controllare i bit di tale byte, viene utilizzato questo approccio semplicemente per comodità

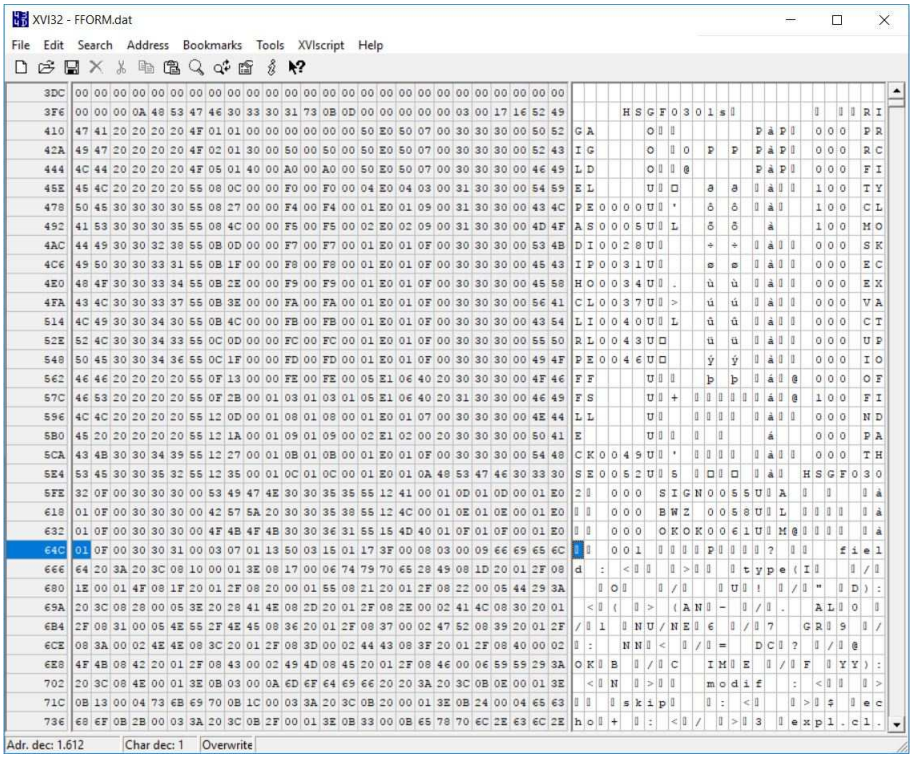

Figura 4.2: Piccola sezione del file FFORM visualizzato tramite un hex editor. A destra si notano i caratteri mostrati in codifica ASCII mentre a sinistra i valori dei byte in esadecimale

Suddivisore maschere: programma per separare le maschere incluse in un file organizzato in *record*

Estrattore: programma per generare file xml a partire dai file binari

Traduttore inverso: programma per generare file binari a partire dai file XML

Traduttore inverso: programma per comparare due file binari e riportarne le differenze

4.2.1 Suddivisore maschere

Il punto di partenza del problema consiste in un file chiamato FFORM contenente tutte le maschere da tradurre, organizzate in *record*. Tali *record* hanno lunghezza costante e sono preceduti da una chiave indicante il nome della maschera e un numero sequenziale che permette di ricostruire l'ordine dei *record* per ciascuna maschera. Occorre perciò realizzare un software in grado suddividere il contenuto del file di partenza in tanti file pari al numero di maschere concatenando per ciascuna di esse i *record* coinvolti nel giusto ordine.

Una prima semplice realizzazione consiste nel leggere il file a blocchi di dimensioni pari alla lunghezza del *record* più quella della chiave, più un ultimo byte che indica la fine del *record* (500 + 8 + 1) e per ogni maschera, il cui nome è dato dai primi 6 caratteri della chiave, appendere sequenzialmente il *record*. In seguito allo sviluppo di questa soluzione è stato scoperto che una precondizione che si riteneva essere vera non può essere considerata come tale, cioè che i *record* nei quali vengono decomposte le maschere non sono memorizzati sequenzialmente nel file. Non solo avviene ciò, ma in certi casi troviamo *record* appartenenti ad una stessa maschera distanziati di molto tra loro e in ordine sparso. Per tale motivo è stato necessario rivedere l'implementazione per garantire che ogni maschera fosse ricostruita seguendo la corretta sequenza espressa dai numeri progressivi indicati dagli ultimi 2 byte.

Inizialmente si effettua una separazione dei dati per renderne più semplice la manipolazione.

```
def get_mask_data(stream : BufferedIOBase, record_len: int):
    while True:
        key = stream.read(MASK_KEY_LEN)
        if key.decode() == '' or key[0:1] == b'\x00':
            break

        key = key.decode()
        prog_num = int(stream.read(KEY_LEN - MASK_KEY_LEN).decode())
        record = stream.read(record_len)
        #carattere terminatore '0x0A'
```

```
        stream.read(1)

        yield [key, prog_num, record]
```

Questa funzione ritorna un iteratore su una lista formata da liste di 3 elementi: la chiave della maschera, il numero progressivo del *record* corrente e il contenuto vero e proprio. A partire da questi elementi occorre poi trovare un modo per ordinare i *record* di ogni maschera ed in seguito scriverli all'interno dei file correttamente.

```
mask_dict = dict()
with inputFile:
    for mask_data in get_mask_data(inputFile, record_len):
        key = mask_data[0]
        prog_num = mask_data[1]
        record = mask_data[2]

        if mask_data[0] not in mask_dict:
            mask_dict[key] = dict()

        mask_dict[key][prog_num] = record

for key, value in mask_dict.items():
    prog_numbers = sorted(value)
    it = (x for x in prog_numbers)
    first = next(it)
    assert first==1 and all(a == b for a, b in enumerate(it, first
        + 1)), "Sequence interrupted, missing a record for mask '" +
        key +"'"

    with open(os.path.join(output_dir_name, key+".dat"), "w") as
        oFile:
        oFile.write("")

    with open(os.path.join(output_dir_name, key + ".dat"), "ab") as
        oFile:
        for prog_num in prog_numbers:
            oFile.write(value[prog_num])
```

La tecnica utilizzata sfrutta una particolare utilità di Python. Inizialmente infatti si costruisce un dizionario dove la chiave è il nome della maschera, mentre il valore è un altro dizionario, il quale ha come chiave il numero progressivo e come valore il *record*.

Una volta terminata la costruzione di tale struttura si procede nel seguente modo: per ogni coppia chiave/valore si utilizza la funzione *sorted(valore)*, la quale accetta in ingresso un qualunque oggetto *Iterable* e ritorna una lista ordinata. Nel caso corrente viene passato ad essa il valore, che è un dizionario, e ciò che fa questa funzione è ritornare una lista ordinata delle sue chiavi:

i numeri progressivi. In questo modo si ottengono tali numeri ordinati ed in seguito si controlla che la sequenza sia valida. A questo punto per ogni numero progressivo nella lista ordinata appena ottenuta si ottiene il relativo *record* accedendo al dizionario e si concatena il *record* al file. In questo modo i *record* vengono processati secondo l'ordine dettato dai numeri sequenziali.

L'utilizzo dei dizionari consente di inserire i *record* in ordine sparso senza che ciò rappresenti un problema e l'estensione della funzione *sorted* a tali strutture ha permesso di realizzare una soluzione semplice ed efficace.

4.2.2 Estrattore

Una volta generati i file per ogni maschera da tradurre è possibile procedere con la parte riguardante il cuore di questo progetto, vale a dire il software che genera codice XML a partire da file binari. Questo viene decomposto in 3 moduli: *mask_ structure*, *mask* e *extractor*

modulo *mask_ structure*

In base al risultato della fase di progettazione di questo programma verrà realizzato un albero di *Enum* contenente tutte le informazioni necessarie alla costruzione dell'interfaccia grafica. Poiché non è possibile realizzarne esplicitamente una gerarchia, dove cioè una *Enum* è un valore di un'altra *Enum*, deve essere studiato il metodo migliore per esprimere il rapporto di subordinazione tra esse.

Innanzitutto si ritiene possa essere utile tener traccia di quale sia il livello di ogni elemento, tuttavia per esprimere le relazioni di parentela occorre inserire ulteriori valori in ognuna di esse, il cui nome sia immediatamente riconducibile al nome del componente figlio. Per fare un esempio consideriamo l'elemento *Mask*, essa possiede come figlio *Field* che è anch'esso un *Enum*, pertanto si aggiunge un elemento *FIELD* a *Mask* che ha solamente il ruolo di indicare la gerarchia presente.

Il risultato ottenuto è il seguente:

```
class MaskDbElement(Enum):
    level = 0
    MASK = "mask"

class MaskElement(Enum):
    level = 1
    NAME = "name"
    FIELD = "field"
    BOX = "box"
    LABEL = "label"
    HORIZONTAL_LINE = "horizontalLine"
    VERTICAL_LINE = "verticalLine"
```

```
class FieldElement(Enum):
    level = 2
    ID = "id"
    INDEX = "index"
    ROW = "row"
    COLUMN = "column"
    LENGTH = "length"
    FIELD_TYPE = "fieldType"
    DATA_TYPE = "dataType"
    EDITABLE = "editable"
    SKIPABLE = "skipable"
    VISIBLE = "visible"
    EXPLICIT_CONFIRM = "explicitConfirm"
    VALIDABLE = "validable"
    UPPERCASE = "uppercase"
    INPUT_OFFSET = "inputOffset"
    OUTPUT_OFFSET = "outputOffset"
    FILLER = "filler"
    NUMBER_OF_DECIMAL_DIGITS = "numberOfDecimalDigits"
    PACKED = "packed"
    THOUSAND_SEPARATOR = "thousandSeparator"
    SIGNED = "signed"
    FILL_WITH_ZEROS = "fillWithZeros"
    REVERSE = "reverse"
    HIGHBRIGHTNESS = "highBrightness"
    BLINKING = "blinking"
    UNDERLINED = "underlined"
    CONTROL_TABLE = "controlTable"

class ControlTableElement(Enum):
    level = 3
    ALLOWED_VALUE = "allowedValue"

class AllowedValueElement(Enum):
    level = 4
    VALUE = "value"

class BoxElement(Enum):
    level = 2
    TL_ROW = "topLeftRow"
    TL_COLUMN = "topLeftColumn"
    BR_ROW = "bottomRightRow"
    BR_COLUMN = "bottomRightColumn"

class HorizontalLineElement(Enum):
    level = 2
    ORIENTATION = "orientation"
    ROW = "row"
    START_COLUMN = "startColumn"
```

```
    END_COLUMN = "endColumn"

class VerticalLineElement(Enum):
    level = 2
    ORIENTATION = "orientation"
    COLUMN = "column"
    START_ROW = "startRow"
    END_ROW = "endRow"

class LabelElement(Enum):
    level = 2
    ROW = "row"
    COLUMN = "column"
    LENGTH = "length"
    REVERSE = "reverse"
    HIGHBRIGHTNESS = "highBrightness"
    BLINKING = "blinking"
    UNDERLINED = "underlined"
    VALUE = "value"
```

Il valore attribuito a ciascun elemento è stato scelto in modo tale che sia univoco ma allo stesso tempo corrisponda anche al nome del *tag* XML correlato.

Tale organizzazione funge da infrastruttura per la memorizzazione dei dati delle maschere ed esprime i rapporti di subordinazione presenti nell'XML finale. I dati sono memorizzati invece in un'altra struttura la cui forma viene definita dinamicamente mediante un *template* esprimente gli elementi e attributi che si intende estrarre.

Questo *template* si basa necessariamente sulla struttura appena mostrata ma utilizza una classe organizzata ad albero, contenente gli elementi delle *Enum*.

```
class MaskTree:

    def __init__(self, builder):
        if not isinstance(builder, MaskTree.Builder):
            raise ValueError
        else:
            self.key = builder.key
            self.children = builder.children

    def has_child(self, child_key):
        for child in self.children:
            if child.key == child_key:
                return True
        return False

    def get_child(self, child_key):
```

```
        for child in self.children:
            if child.key == child_key:
                return child
        return None

    def get_tree_template_string(self, indentation=0):

        print_string = ""
        for i in range(indentation):
            print_string+="-"

        print_string += self.key.value
        print_string += "\n"

        if (self.children):
            indentation += 1
            for child in self.children:
                print_string += child.print_tree_template(indentation)

        return print_string


    class Builder:

        def __init__(self):
            self.key = None
            self.children = list()

        def set_key_node(self, key):
            self.key = key
            return self

        def add_child(self, child):
            self.children.append(child)
            return self

        def build(self):
            return MaskTree(self)
```

In particolare questa è una semplice classe che possiede due campi:

key: un elemento di una *Enum*

children: una lista di elementi figli di quello corrente

La logica è che ogni nodo possiede una chiave, che corrisponde ad un elemento di una delle *Enum* illustrate precedentemente, ed una lista di figli, vuota nel caso in cui sia una foglia. Questa classe non impone una struttura rigida in quanto rappresenta semplicemente un albero a supporto della memorizzazione degli elementi di interesse da riportare nell'interfaccia finale.

Una qualunque istanza di tale classe viene definita *template* e tramite essa viene modellata la classe in cui vengono effettivamente salvate le informazioni contenute nei file.

modulo *mask*

Questa rappresenta la parte più complessa del traduttore in quanto il componente da modellare deve svolgere diverse funzioni:

- Memorizza i dati d'interesse
- Rappresenta l'effettivo passaggio dalla struttura sequenziale del file binario a quella ad albero del file XML
- Comprende la logica di generazione del codice XML

Come detto in precedenza la realizzazione di questo componente segue le orme della struttura gerarchica studiata; più precisamente abbiamo tante classi quante sono le *Enum*, organizzate ad albero nel seguente modo.

```
class Mask:
   maskElements = dict()

   class Field:
      fieldElements = dict()

      class ControlTable:
         controlTableElements = dict()

         class AllowedValue:
            allowedValueElements = dict()

   class Box:
      boxElements = dict()

   class HorizontalLine:
      horizontalLineElements = dict()

   class VerticalLine:
      verticalLineElements = dict()

   class Label:
      labelElements = dict()
```

Ogni dizionario rappresenta il contenitore di tutte le informazioni dell'elemento a cui fa riferimento.

In questa situazione viene sfruttata la potenza di Python, il quale permette di creare dizionari che possono contenere valori di tipi diversi. L'idea è quella di utilizzare questa utilità per realizzare una variante del *pattern*

Builder. Piuttosto che creare tanti campi quante sono le informazioni da memorizzare si utilizza un unico campo: un dizionario dove la chiave è un elemento di una *Enum* mentre il valore è l'informazione vera e propria, che può anche essere una classe figlia di essa. Questa tecnica consente di avere un solo campo e un solo metodo per modificarne il contenuto ed effettuare inoltre tutti i controlli che derivano dall'utilizzo della versione comune di tale *pattern*, semplicemente ricavando l'attributo tramite la chiave passata come argomento e verificando se il valore rispetta determinate condizioni. Considerando quindi che tutti i dati sono memorizzati in un unico campo esiste un unico metodo *set* del *Builder* che accetta come argomenti una chiave ed un valore e li inserisce nel dizionario.

In seguito viene mostrata l'implementazione di quanto appena mostrato per un caso, per tutte le altre classi si ha una situazione del tutto analoga.

```
class Mask:

    def __init__(self, builder):
        if not isinstance(builder, Mask.Builder):
            raise ValueError
        else:
            self.maskElements = dict()
            for key, value in builder.maskElements.items():
                self.maskElements[key] = value


    class Builder():

        def __init__(self):
            self.maskElements = dict()

        def set(self, key, value):
            self.maskElements[key] = value
            return self

        def build(self):
            return Mask(self)
```

All'interno di queste classi inoltre è incapsulata la logica di realizzazione del codice XML. Per fare ciò è stata utilizzata la libreria *xml.etree*; essa permette di creare un albero di elementi, dove la radice è un oggetto di tipo *Element* per cui va specificato il nome, mentre i nodi figli sono oggetti di tipo *SubElement* per cui va specificato oltre al nome anche il nodo padre. Per la creazione di tale albero viene inserito un metodo *xml_ subElement_ of(self, father_ el)* all'interno di ogni classe, il quale provvede a creare un *SubElement* figlio del nodo passato come argomento e a costruire il relativo sottoalbero.

Per fare ciò, all'interno di questo metodo vengono scorsi tutti gli elementi del dizionario e per ogni chiave facente rifermento ad un nodo foglia viene

salvato il relativo valore come attributo del *SubElement* creato. Per le altre chiavi invece si ha come valore una classe, la quale a sua volta possiede il metodo *xml_ subElement_ of(self, father_ el)* che viene chiamato in cascata per realizzare il *SubElement* del *SubElement* appena creato.

In seguito sono riportati due casi significativi, cioè quello relativo alla classe *Mask*, il quale possiede tutti nodi figli che vanno rappresentati a loro volta come *SubElement*, e quello relativo alla classe *Field*, diretto figlio di *Mask* che presenta sia nodi foglia che non. Per i restanti casi si procede analogamente.

```
class Mask:

    def xml_subElement_of(self, father_el):

        mask_el = SubElement(father_el, MS.MaskDbElement.MASK.value)
        mask_el.set(MS.MaskElement.NAME.value,
            self.maskElements[MS.MaskElement.NAME])

        if (MS.MaskElement.FIELD in self.maskElements):
            [field_el.xml_subElement_of(mask_el) for field_el in
                self.maskElements[MS.MaskElement.FIELD]]

        if (MS.MaskElement.BOX in self.maskElements):
            [box_el.xml_subElement_of(mask_el) for box_el in
                self.maskElements[MS.MaskElement.BOX]]

        if (MS.MaskElement.HORIZONTAL_LINE in self.maskElements):
            [hline_el.xml_subElement_of(mask_el) for hline_el in
                self.maskElements[MS.MaskElement.HORIZONTAL_LINE]]

        if (MS.MaskElement.VERTICAL_LINE in self.maskElements):
            [vline_el.xml_subElement_of(mask_el) for vline_el in
                self.maskElements[MS.MaskElement.VERTICAL_LINE]]

        if (MS.MaskElement.LABEL in self.maskElements):
            [label_el.xml_subElement_of(mask_el) for label_el in
                self.maskElements[MS.MaskElement.LABEL]]

        return mask_el



    class Field:

        def xml_subElement_of(self, father_el):

            field_el = SubElement(father_el,
                MS.MaskElement.FIELD.value)
```

```
        [field_el.set(key.value, value) for key, value in
            self.fieldElements.items() if key is not
            MS.FieldElement.CONTROL_TABLE]

        if (MS.FieldElement.CONTROL_TABLE in self.fieldElements):
            self.fieldElements[MS.FieldElement.CONTROL_TABLE]
            .xml_subElement_of(field_el)

        return field_el
```

modulo *extractor*

A questo punto occorre realizzare la parte vera e propria dell'estrattore, cioè quel componente che legge i file binari, suddivide ed elabora il contenuto e memorizza le informazioni nella struttura mostrata nella sezione precedente.

Perciò ora si tratta di realizzare un'implementazione della logica studiata nella fase di progettazione

In generale i byte vengono trattati in modi diversi a seconda del significato che hanno, il quale può essere individuato in uno dei seguenti casi.

simboli speciali: non apportano alcun contributo informativo alla realizzazione dell'interfaccia, hanno generalmente funzione di dichiaratore, separatore o riempitore

simboli comuni: sono utilizzati per memorizzare informazioni, sono suddivisi a loro volta in tre casi:

- **simboli testuali:** riportano l'informazione direttamente in forma di testo, comprensibile utilizzando la codifica ASCII
- **simboli numerici:** riportano l'informazione tramite il valore numerico dei byte; si ottiene quindi il dato convertendo tale valore binario in notazione decimale
- **simboli generici:** riportano l'informazione tramite un concetto che viene attribuito al valore del byte. Di per sé non ha alcun significato ma al valore assunto corrisponde un'informazione precisa.

A seconda dei casi perciò potrebbe esser necessario: leggere semplicemente il valore dei byte (simboli speciali), ottenerne una rappresentazione testuale (simboli testuali), ottenerne i valori numerici decimali (simboli numerici) o ricavare a partire da essi i valori a cui si riferiscono (simboli generici).

Considerando un file per volta, la prima operazione da effettuare è quella di suddividere il contenuto nelle 6 categorie rappresentanti gli elementi grafici di una maschera: *field*, *box*, *horizontal line*, *vertical line*, *label*, *control table*. Questo compito viene affidato alla seguente funzione:

```
def read_file(stream : BufferedIOBase):

    fields = list()
    # Codice per memorizzare i field

    boxes = list()
    h_lines = list()
    v_lines = list()
    # Codice per memorizzare i box, le horizontal line e le
        vertical line

    labels = list()
    # Codice per memorizzare le label

    controlTables = dict()
    # Codice per memorizzare le control table

    return [fields, boxes, h_lines, v_lines, labels, controlTables]
```

Questa funzione pertanto prende in ingresso un file e restituisce una lista contenente 5 liste di elementi e 1 dizionario. Le *control table* vengono infatti memorizzate in maniera differente dagli altri elementi poiché si vuole mantenere il collegamento con i *field* a cui fanno riferimento, memorizzandone l'id come chiave.

In questa prima fase l'elaborazione del contenuto è minima, lo scopo infatti è solamente quello di suddividere i dati per facilitarne poi la comprensione.

Una volta effettuata questa operazione si considerano gli elementi uno per volta e si procede con l'estrazione e memorizzazione delle informazioni.

Questo comportamento viene racchiuso all'interno della seguente funzione:

```
def build_tree_list(inputDir, creation_mode):

    mask_list = list()

    for in_file in input_files:
        iFile = open(file = os.path.join(inputDir, in_file), mode =
            "rb")
        inputFile : BufferedReader = BufferedReader(raw = iFile)

        with inputFile:
            data = read_file(inputFile)

        mask_builder = Mask.Builder()
        raw_fields = data[0]
```

```
field_builders = list()
for raw_field in raw_fields:
    field_builder = Mask.Field.Builder()
    field_builders.append(field_builder)
    # Codice per settare gli attributi a raw_field

    raw_controlTables = data[5]
    for raw_controlTable in raw_controlTables:
        controlTable_builder =
            Mask.Field.ControlTable.Builder()
        # Codice per settare le control table inserendo gli
            allowed value

        field_builder.set(MS.FieldElement.CONTROL_TABLE,
            controlTable_builder.build())

mask_builder.set(MS.MaskElement.FIELD,
    [field_builder.build() for field_builder in
    field_builders])


raw_boxes = data[1]
box_builders = list()

for raw_box in raw_boxes:
    box_builder = Mask.Box.Builder()
    box_builders.append(box_builder)
    # Setto gli attributi a raw_box

mask_builder.set(MS.MaskElement.BOX, [box_builder.build()
    for box_builder in box_builders])

raw_hlines = data[2]
hline_builders = list()

for raw_hline in raw_hlines:
    hline_builder = Mask.HorizontalLine.Builder()
    hline_builders.append(hline_builder)
    # Codice per settare gli attributi a raw_hline

mask_builder.set(MS.MaskElement.HORIZONTAL_LINE,
    [hline_builder.build() for hline_builder in
    hline_builders])


raw_vlines = data[3]
vline_builders = list()

for raw_vline in raw_vlines:
```

```
            vline_builder = Mask.VerticalLine.Builder()
            vline_builders.append(vline_builder)
            # Codice per settare gli attributi a raw_vline

        mask_builder.set(MS.MaskElement.VERTICAL_LINE,
            [vline_builder.build() for vline_builder in
            vline_builders])


        raw_labels = data[4]
        label_builders = list()

        for raw_label in raw_labels:
            label_builder = Mask.Label.Builder()
            label_builders.append(label_builder)
            # Codice per settare gli attributi a label_builder

        mask_builder.set(MS.MaskElement.LABEL,
            [label_builder.build() for label_builder in
            label_builders])

        mask_list.append(mask_builder.build())

    return mask_list
```

Il codice appena mostrato rappresenta una versione privata di tutta la logica di traduzione dei dati e ha solamente l'obiettivo di illustrare la logica generale secondo cui vengono memorizzati i dati. Per ogni elemento viene creata una lista di *Builder* di esso, uno per ogni istanza. Ogni volta che sono state memorizzate tutte le informazioni di interesse per quanto riguarda una determinata classe sarà possibile costruire le istanze tramite tali *Builder*; per cui per ognuno di essi presenti nella lista viene chiamato il metodo *build*, ottenendo una lista di istanze di tale classe che viene aggiunta all'elemento padre tramite il *Builder*. Procedendo in questo modo per tutti i livelli della gerarchia si costruisce l'albero finale comprendente tutte le informazioni di rilievo. Riassumendo, la logica che si segue è quella di creare liste di *Builder* per ogni classe nella gerarchia partendo dalla radice e arrivando fino alle foglie, appena vengono memorizzati i dati relativi a tutti i figli di un determinato nodo viene creata una lista di istanze create chiamando il metodo *build* su ogni *Builder* e si aggiunge tale lista al padre tramite il metodo *set*.

La parte di pura estrazione delle informazioni rappresenta invece solamente un'implementazione dei *pattern* studiati; non essendovi particolare difficoltà nello sviluppare tale funzionalità non è necessario presentare e discutere il codice prodotto.

4.2.3 Estrattore inverso

Come spiegato in fase di analisi si vuole garantire l'accuratezza del risultato prodotto dall'estrattore per evitare che si possano verificare comportamenti indesiderati. Il modo più efficace per fare ciò consiste nel generare un file binario partendo dal file XML generato seguendo i *pattern* individuati ma applicandone la logica inversa e confrontando poi i due file binari. In questo modo viene garantito che tali *pattern* sono validi e nel caso ciò non avvenga si può partire dalla differenza rilevata per studiare l'anomalia.

Una volta terminato lo sviluppo dell'estrattore perciò si procede con la realizzazione dell'estrattore inverso. Trattandosi della realizzazione del procedimento inverso rispetto al primo problema la parte strutturale fine alla memorizzazione dei dati può essere riutilizzata senza bisogno di apportare alcuna modifica. In questo caso risulta anche più semplice la costruzione della struttura a partire dai dati noti in quanto i file XML possiedono già una struttura ad albero.

Per ottenere le informazioni viene utilizzata sempre la libreria *xml.etree*, questa volta ricavando i valori dei nodi e in seguito inserendoli all'interno della struttura messa a disposizione, nello stesso modo in cui avviene per l'estrattore.

A questo punto occorre realizzare l'algoritmo vero e proprio che trasforma i dati nei valori da inserire nei file in modo da ottenerne uno del tutto analogo a quello originale. Per fare ciò si utilizzano gli stessi *pattern* individuati per l'estrazione, è immediato notare infatti che la loro validità è bidirezionale. La differenza sta nel fatto che se precedentemente si osservavano i valori binari e da essi si derivavano le informazioni corrispondenti, ora si conoscono le informazioni e da esse si ricavano i valori binari. Il legame tra valori e informazioni pertanto rimane qualunque sia il verso di lettura.

In generale l'implementazione di tale algoritmo risulta più semplice rispetto a quella dell'algoritmo per effettuare l'estrazione normale. Ciò è dovuto al fatto che in alcuni casi più informazioni vengono compresse all'interno di un numero minore di byte, dove i valori corrispondenti alle informazioni sono messi in or tra loro generando una combinazione univoca. Si può facilmente notare che, dato un certo algoritmo, generare una combinazione è più semplice che comprendere i valori che la compongono. Per effettuare quest'ultima operazione infatti occorre prima generare tutte le possibili combinazioni tramite l'algoritmo utilizzando tutti i valori di partenza e poi trovare i valori che compongono quella uguale a quella data; per la prima operazione invece è sufficiente applicare l'algoritmo che genera la combinazione.

4.2.4 Comparatore

Terminata la parte di sviluppo dell'estrattore e dell'estrattore inverso è possibile realizzare il programma che confronta i file binari e ne riporta le

differenze.

Durante lo sviluppo di questo componente sono emersi alcuni problemi relativi a come realizzare questa funzionalità. Dalla fase di progettazione è emerso che è necessaria prima una fase di collezionamento e classificazione dei dati. Come già anticipato infatti risulta impossibile confrontare direttamente i file byte per byte poiché nel processo di estrazione numerosi dati superflui sono stati ignorati. Si potrebbe allora pensare di ignorare tali byte durante la comparazione ma seguendo questo approccio si possono creare dinamiche tali per cui risulta impossibile segnalare le differenze tra i file. Ad esempio se si verifica un *pattern* inatteso per cui in uno dei due file vi è un byte in più rispetto all'altro, da questo punto in poi i due file non hanno più alcun elemento in comune e per ogni byte verrebbe segnalato un errore. Perciò la soluzione di una prima forma di elaborazione di essi risulta essere forzata.

La modularizzazione e ripartizione dei compiti in funzioni effettuata durante tutta la fase di sviluppo si è mostrata in questo caso molto utile; per eseguire questa prima fase infatti si è ricorso alla funzione *read_file* del modulo *extractor*, il cui scopo era preparare i file per essere processati.

A questo punto si procede implementando l'algoritmo per il confronto vero e proprio.

Durante la fase di estrazione, per la maggior parte degli elementi vi è una memorizzazione in ordine di lettura all'interno della struttura e tale ordine viene mantenuto anche nella scrittura del file XML. Perciò l'approccio generale è quello di confrontare gli elementi di ogni categoria sequenzialmente e indicando il punto preciso in cui vi è dissimilarità.

```
    if (len(labels_1)!=len(labels_2)):
        mismaches.append("Different number of labels")

    i = 0
    for label_1, label_2 in zip(labels_1, labels_2):
        labelDecl_1 = label_1[:MD.LABEL_DECL_LEN]
        labelDecl_2 = label_1[:MD.LABEL_DECL_LEN]
        if (labelDecl_1[0] != labelDecl_2[0]):
            mismaches.append("Different row of label n. %d" % i)
        if (labelDecl_1[1] != labelDecl_2[1]):
            mismaches.append("Different column of label n. %d" % i)
        if (labelDecl_1[2] != labelDecl_2[2]):
            mismaches.append("Different style of label n. %d" % i)
        if (labelDecl_1[3] != labelDecl_2[3]):
            mismaches.append("Different length of label n. %d" % i)

        content_1 = label_1[MD.LABEL_DECL_LEN:]
        content_2 = label_2[MD.LABEL_DECL_LEN:]


        if (content_1 != content_2):
```

```
            mismaches.append("Different text of label n. %d" % i)

        i+=1
```

Il problema principale è che per alcuni elementi, in seguito alla fase di estrazione, viene perso l'ordine, ciò avviene per *box*, *horizontal line* e *vertical line*. Si tratta infatti di una sezione eterogenea che comprende più elementi simili tra loro ma concettualmente differenti, che è bene tenere separati nella realizzazione del codice XML e perciò anche nella struttura in cui memorizzare le informazioni. Questi tre elementi possono trovarsi in qualunque ordine, il quale viene necessariamente perso nel prodotto finale, perciò risulta impossibile scansionare tali elementi sequenzialmente. La soluzione realizzata per questo problema è la seguente: per ogni elemento di una di queste categorie si verifica se è presente anche nella lista ottenuta dal secondo file e in caso contrario si riporta come messaggio di errore che l'elemento con quei determinati valori è assente nell'altro file. Purtroppo ciò comporta che non viene indicato né l'elemento da cui differisce né il pattern violato, viene segnalato solamente che esso non appartiene all'insieme degli elementi dell'altro file. Tuttavia ciò non rappresenta un problema di grande rilevanza poiché viene comunque segnalata una forma di differenza e, se i casi sono contenuti, è possibile risalire facilmente aglii elementi differiscono.

In seguito è mostrata l'implementazione di tale tecnica.

```
    for box_1 in boxes_1:
        if box_1 not in boxes_2:
            mismaches.append("Box with values " + str(box_1) + "
                present in '" + stream_1.name + "' and not in " +
                stream_2.name + "'")

    for box_2 in boxes_2:
        if box_2 not in boxes_1:
            mismaches.append("Box with values " + str(box_2) + "
                present in '" + stream_2.name + "' and not in " +
                stream_1.name + "'")
```

Un'ultima nota riguarda il caso delle *control table*. Trattandosi di elementi molto particolari, che sono stati gestiti in maniera differente rispetto agli altri, l'approccio utilizzato è analogo a quello utilizzato per il caso precedente. Considerando inoltre che possiedono un solo elemento, vale a dire un insieme di valori che i *field* possono assumere, tale tecnica è una valida alternativa alla scansione sequenziale.

```
    if (len(control_tables_1)!=len(control_tables_2)):
        mismaches.append("Different number of control tables")

    for control_table_1 in control_tables_1:
```

```
        if control_table_1 not in control_tables_2:
            mismaches.append("Different control table of field " +
                str(control_table_1))

    for control_table_2 in control_tables_2:
        if control_table_2 not in control_tables_1:
            mismaches.append("Different control table of field " +
                str(control_table_2))
```

Capitolo 5

Conclusioni e sviluppi futuri

5.1 Risultati

In questa sezione vengono mostrati i risultati ottenuti. Inizialmente è stato eseguito un test basandosi su alcune maschere già tradotte a mano; si analizzava quindi visivamente l'interfaccia grafica generata dall'estrattore confrontandola con quella a disposizione. Come si può osservare in figura 5.1 le due maschere sono estremamente simili, fatta eccezione di alcuni elementi che sono stati appositamente modificati nella traduzione manuale per avere una resa grafica migliore. Risultando tuttavia impossibile effettuare un'analisi visiva per circa 12 mila maschere, il test per misurare l'effettiva efficacia dell'estrattore avviene utilizzando il *software* comparatore mostrato nei capitoli precedenti.

Una volta estratto l'XML comprendente tutte le maschere presenti nell'archivio è stato poi utilizzato l'estrattore inverso per generare i file binari a partire da esso. Tramite il comparatore è stato ottenuto un riscontro di quanti file coincidono in termini di contenuto informativo. È emerso che per 12029 maschere su 12333 la rappresentazione ottenuta è del tutto analoga a quella di partenza.

Per i restanti casi il comparatore ha evidenziato che l'unico *pattern* per cui si verifica tale dissimilarità è lo stile, sia per i *field* che per le *label*. Studiando le differenze riscontrate sembra che esistano più combinazioni stilistiche di quante si pensava esistessero.

La precisione dell'estrattore attualmente è perciò superiore al 97%.

Per ottenere una precisione assoluta sarà necessario studiare in quali casi si verificano le situazioni inattese, modificare la descrizione dei *pattern* e aggiungere l'implementazione di essi.

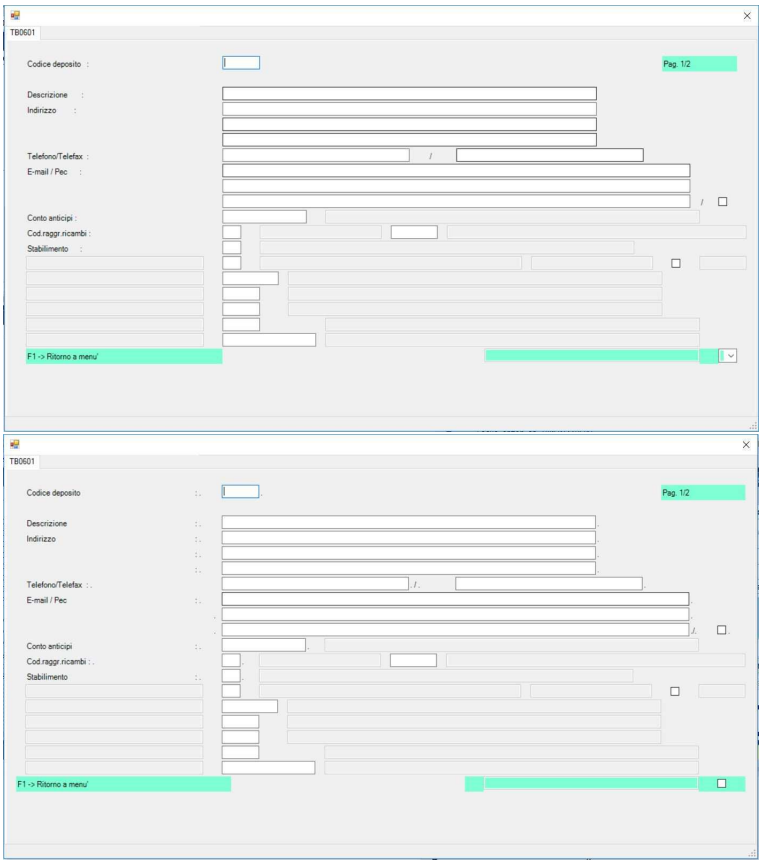

Figura 5.1: Interfaccia grafica della maschera TB0601 tradotta a mano e generata. La figura più in alto rappresenta la maschera costruita, mentre la figura in basso riporta quella generata usando l'estrattore.

5.2 Sviluppi futuri

In questa sezione verranno discussi gli eventuali sviluppi futuri resi possibili dal progetto appena descritto.

Innanzitutto osservando la figura 5.1 possiamo notare alcuni dettagli che impattano negativamente sulla resa grafica delle maschere tradotte. Un esempio immediato è la presenza dei punti "." prima e dopo ogni campo. Nella visualizzazione da terminale la loro presenza era essenziale per indicare all'utente dove inserire i dati; in ambiente .NET invece viene utilizzata la libreria *Windows Forms*, la quale autonomamente prevede una resa grafica personalizzata per i campi di testo, rendendo di fatto la presenza dei punti superflua e disarmonica.

In generale una possibile modifica consiste nel ri-elaborare l'XML ottenuto in modo da ottenere un interfaccia grafica più accattivante e che ne incrementi l'usabilità.

Altri possibili sviluppi riguardano l'utilizzo di varianti dell'XML specifici per determinate librerie grafiche. La struttura alla base dell'estrattore rende infatti possibile la variazione arbitraria del formato di arrivo, per il fatto che i dati prima vengono parzialmente elaborati e memorizzati internamente.

Una prima possibile tecnologia alternativa è lo XAML, un linguaggio di *markup* basato su XML, utilizzato per descrivere l'interfaccia grafica delle applicazioni basate sulla libreria *Windows Presentation Foundation* (WPF). Quest'ultima è una libreria più nuova e potente rispetto a *Windows Forms*, inoltre il formato XAML rende più semplice la realizzazione dell'interfaccia grafica rispetto all'utilizzo di XML poiché tale è l'utilizzo per cui è nato.[8]

Un'altra tecnologia interessante è l'FXML, un linguaggio di *markup* basato su XML creato per definire l'interfaccia grafica di applicazioni scritte con l'utilizzo di JavaFX. JavaFX è una libreria estremamente pratica per lo sviluppo di applicazioni grafiche e l'utilizzo di FXML permette di evitare di inserire la descrizione della GUI all'interno del codice, mettendo a disposizione una serie di *tag* predefiniti tramite i quali si può facilmente realizzare l'interfaccia utente.[4]

Bibliografia

[1] Ibm processing indexed files. https://www.ibm.com/support/knowledgecenter/en/SSAE4W_9.1.0/com.ibm.etools.iseries.pgmgd.doc/c0925405407.htm.

[2] Xml schema. https://www.w3.org/standards/xml/schema.

[3] Michael Coughlan. *Beginning COBOL for Programmers*. Apress, 2014.

[4] I Fedortsova and G Brown. Javafx mastering fxml, release 8 (2014).

[5] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[6] Elliotte Rusty Harold, W Scott Means, and Katharina Udemadu. *XML in a Nutshell*, volume 3. O'reilly Sebastopol, CA, 2004.

[7] John E Hopcroft, Rajeev Motwani, Jeffrey D Ullman, Luca Bernardinello, and Lisa Scarpa. *Automi, linguaggi e calcolabilità*. Addison-Wesley Italia, 2003.

[8] Lori MacVittie. *XAML in a Nutshell*. " O'Reilly Media, Inc.", 2006.

[9] James B Maginnis. *Fundamental ANSI Cobol Programming*. Prentice Hall, 1974.

[10] Dusty Phillips. *Python 3 object oriented programming*. Packt Publishing Ltd, 2010.

Ringraziamenti

In primo luogo intendo ringraziare il professor Mirko Viroli che mi ha offerto la possibilità di svolgere una tesi così formativa ed interessante, supervisionando costantemente il lavoro.

Un profondo ringraziamento è rivolto al correlatore Giovanni Ciatto che mi ha guidato durante tutto lo svolgimento. Con grandi capacità e preparazione mi ha sempre offerto saggi consigli, aiutandomi a superare problemi e difficoltà.

Ringrazio l'azienda Harvard Group che ha permesso lo svolgimento di questa tesi e si è resa sempre disponibile collaborando al fine della realizzazione del progetto.

Ringrazio infinitamente la mia famiglia che mi ha sempre supportato, la mia fidanzata che mi è sempre stata vicino e tutte le persone che mi hanno accompagnato fino a questo punto.