Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Validazione basata su regole di documenti per il progetto Smart Publishing Management

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Tommaso Ognibene

Sessione I
2015/2016

Abstract

A partire da Febbraio 2016, tramite una serie di incontri a cadenza settimanale, si sono progressivamente poste le basi per una collaborazione tra Alstom e Università di Bologna, relativamente ad un progetto denominato Smart Publishing Management.
Il progetto concerne la definizione di un sistema informativo e, al suo interno, di un sistema informatico, orientato al miglioramento del processo di produzione documentale in seno ad Alstom.
Questa tesi ripercorre i temi teorici e gli aspetti tecnici coinvolti in tale progetto. Inoltre, si focalizza sul primo dei tre strumenti software previsti, di cui si è realizzato un prototipo.

Keywords: Document Management System, Rule-Engine, Temporal XML, Overlapping Markup, Operational Transformation, Regular Expression, XPath, XSLT, XQuery, Node.js.

Indice

Elenco delle figure

1 Introduzione

Questa tesi si inserisce nell'ambito di un progetto denominato Smart Publishing Management (SPM), finalizzato alla definizione di un sistema informativo per la gestione della documentazione tecnica di Alstom. La tesi, sviluppata durante la fase preparatoria del progetto, si compone di due parti. La prima parte, teorica, ripercorre i principali aspetti concettuali, scientifici e tecnologici che presiedono al progetto. La seconda parte, pratica, consiste nella presentazione di un prototipo di uno dei tre strumenti software previsti.

Il progetto consta di tre strumenti software, denominati rispettivamente Validation Engine, Smart Structured Editor, Meta-template Engine. Il Validation Engine presiede alla validazione dei documenti tramite applicazione di regole sintattiche intra e inter documentali. Lo Smart Structured Editor presiede alla fase di creazione e modifica dei documenti, in modo collaborativo e decentrato. Il Meta-template Engine presiede la fase di estrazione di componenti riutilizzabili dal contenuto dei documenti e di inserimento automatico di contenuti non creativi, come ad esempio la tabella degli acronimi.
A ciascuno di questi strumenti corrisponde un soggetto specifico preposto al suo utilizzo: il Process Manager impiega il Validation Engine, il Document Writer impiega lo Smart Structured Editor, il Document Editor impiega il Meta-template Engine. Questi non sono ruoli già esattamente definiti in seno ad Alstom, di conseguenza questo progetto non è meramente informatico, ma afferisce al sistema di produzione e gestione documentale nel suo complesso.

La prima parte della tesi inizia con il concetto di Document Management System, in quanto gli strumenti previsti fanno parte di questa famiglia di prodotti software. Si richiamano i concetti di versioni e varianti di un documento, che ne descrivono l'evoluzione e la ramificazione nel tempo. Si introduce il formato dati XML, impiegato per descrivere i documenti di Alstom e si sottolineano i principali percorsi di ricerca relativi alla rappresentazione del tempo in XML. Questo ambito di ricerca viene denominato complessivamente Temporal XML. Due formati, in particolare, vengono presentati: XBIT (*XML-based Bitemporal Data Model*) e MXML (*Multidimensional XML Model*).

Un altro tema teorico utile da approfondire è quello dell'*Overlapping Markup*. XML ha un struttura ad albero connotata da una forte limitazione: ogni elemento (nodo), ad eccezione della radice, ha un solo padre. Di conseguenza la capacità espressiva di XML non consente la compresenza di gerarchie logiche conflittuali, che richiedono un albero con parentela multipla.
Questa limitazione è superabile mediante due approcci: un approccio pragmatico che non vuole fare a meno di XML e un approccio teorico che invece definisce un nuovo formato dati. L'approccio pragmatico utilizza XML solo formalmente, nella sostanza l'albero XML diventa irrilevante e il vero albero che rappresenta il documento deve essere ricostruito dinamicamente dall'algoritmo che processa il file XML. Questo approccio annovera le tecniche come CLIX ed ECLIX che fanno uso dei cosidetti *milestones*. Di converso, l'approccio teorico non ha una sua implementazione in XML, in quanto XML non ha la capacità espressiva necessaria. Questo approccio propone quindi un nuovo formato dati, più potente, denominato GODDAG, acronimo di *General Ordered-Descendant Directed Acyclic Graphs*.

Oltre a definire un formato dati specifico in XML per la rappresentazione dei documenti di Alstom, il progetto SPM propone tre strumenti software, di

cui uno, lo Smart Structured Editor, è caratterizzato da particolare complessità. Si tratta di un software di editing collaborativo online, finalizzato alla produzione dei documenti Alstom, in modalità decentrata e asincrona. Strumenti del genere esistono già in commercio, il più noto è senz'altro Google Docs. Il substrato teorico di questi *group editors* prende il nome di Operational Transformation (OT). L'obbiettivo di OT è gestire sequenze di modifiche ad uno stesso documento, eseguite da parte di una pluralità di siti (utenti), dove ogni sito (utente) effettua ciascuna modifica in un determinato istante cronologico e su un determinato stato del documento. Entrano quindi in gioco due dimensioni: non solo la dimensione temporale, ma anche la dimensione dello stato del documento, dalla quale si evince l'intenzione dell'autore della modifica. OT costituisce quindi un insieme di regole finalizzate a garantire una serializzazione corretta di queste modifiche, in modo da garantire la convergenza e preservare l'intenzione originaria.

Infine si tracciano le caratteristiche principali del formato dati Office Open XML (OOXML), creato da Microsoft per i suoi prodotti Office, ma open-source tramite lo Standard ECMA-376 a partire dal Dicembre 2006. Si tratta del formato utilizzato in ambito Alstom per il salvataggio di tutti i documenti di testo. In particolare, DOCX è una collezione ordinata di file XML, quindi tutti gli aspetti precedentemente analizzati che concernono XML sono validi e applicabili ai file DOCX.

La seconda parte del progetto descrive il prototipo del Validation Engine[1]. Si tratta di un web service che consente di caricare documenti ed effettuare controlli sugli stessi tramite un insieme di regole sintattiche e strutturali. Le regole sono state scritte in XML secondo una sintassi appositamente creata, si distinguono in regole valutative (chiamate Collect-And-Check) e comparative (chiamate Collect-And-Compare). Le regole sfruttano tutti i principali

[1] `https://github.com/tomOgn/ValidationEngine`

linguaggi di interrogazione per file testuali e XML, segnatamente Regex, XPath, XSLT, XQuery.
L'utente può scegliere quali regole applicare e scaricare dei report contenenti i risultati. Non è un servizio ad accesso universale: esiste una forma di autenticazione degli utenti abilitati. Poiché il servizio è riservato ai dipendenti Alstom, la gestione delle credenziali (creazione, eliminazione) è separata. Si è previsto a tal fine un servizio intranet[2].

[2] `https://github.com/tomOgn/ValidationEngineUserManagement`

2

Il progetto Smart Publishing Management

2.1 Origine del progetto

Alstom è una società multi-nazionale francese operante sopratutto nel settore del trasporto ferroviario. Il Gruppo ha una sede a Bologna, con circa 600 dipendenti, dedicata allo sviluppo dei sistemi di segnalamento e del software e hardware relativo all'*interlocking*.
In una serie di incontri formali, tra il management di Alstom-Bologna e il gruppo di ricerca coordinato dal prof. Vitali, l'azienda ha espresso la necessità di un nuovo sistema informativo per migliorare il processo di produzione e gestione documentale interno. Il progetto Smart Publishing Management (SPM) nasce per dare una risposta a questa necessità.
Il ciclo di vita di tutta la documentazione tecnica prodotta in Alstom è particolarmente complesso. Innazitutto esistono due principali aree di attività all'interno di Alstom, alle quali corrispondono due cicli di vita documentali divergenti: l'area dei progetti e l'area dei prodotti. Per quanto concerne i progetti, ogni progetto ha due fasi distinte: la fase di tender, per l'aggiudicazione di un appalto, alla quale corrisponde la documentazione iniziale di progetto, e la fase esecutiva. La stessa fase di tender può distinguersi ulteriormente in appalto-concorso, caratterizzato da un alto livello di dettaglio documentale, e in gare dove l'unico parametro è il costo. Questa varietà si riflette in una varietà di processi e contenuti documentali. Ulteriori classifi-

cazioni sono la divisione tra documenti di piano per la parte ingegneristica e per la parte organizzativa; e tra documenti rivolti all'esterno (in particolare per i clienti) e rivolti all'interno.
La documentazione di Alstom è archiviata in una repository ufficiale, nei vari formati di Microsoft Office (docx, xlsx, pptx). Esiste un processo di verifica formale della qualità dei documenti, denominato *Gate Review*. Questo processo viene affiancato da un controllo interno di livello inferiore, chiamato *Project Review*. Ogni modifica sostanziale alla documentazione di un progetto passa attraverso una formale *Change Request*, che consente un tracciamento consistente della storia del documento.

Tre desiderata specifici espressi da Alstom sono:

1. Un meccanismo comprensivo di tracciamento dei requisiti, dalla loro formulazione alla loro attuazione in seno ai documenti finali.

2. Un meccanismo di creazione, editazione e riutilizzo di frammenti documentali.

3. Un meccanismo di creazione automatica della tabella degli acronimi, e di altri contenuti automatizzabili, attualmente creati manualmente con inutile dispendio di risorse.

2.2 Descrizione generale del progetto

Il progetto consiste nello sviluppo di un sistema unico per generare, validare, riutilizzare e gestire la documentazione tecnica in seno ad Alstom. A tal fine, sono stati identificati tre ruoli che presiedono a fasi distinte del ciclo di vita dei documenti:

Process Manager :

- Supervisiona il processo di produzione documentale.

- Verifica la corretta esecuzione del workflow documentale.
- Esegue la validazione del documento contro un insieme di regole formali e sostanziali.
- Se il documento supera la fase di validazione, rilascia il documento.

Document Editor :

- Verifica il documento contro un insieme di regole formali e sostanziali, comunicando relazioni al *Process Manager*.
- Identifica, comparando i processi documentali, le *best practices*, i *pattern*, gli *anti-pattern* e gli standard, al fine di migliorare la qualità dell'intero ciclo di vita dei documenti.

Document Writer :

- Scrive i documenti rispettando le regole formali e sostanziali.
- Sottomette i documenti scritti alla validazione e integrazione all'interno del worflow documentale.

Mentre il *Process Manager* e il *Document Writer* sono ruoli già presenti e definiti in Alstom, il *Document Editor* è una coagulazione di nuove e vecchie responsabilità, precedentemente poste in capo ad una pluralità di uffici. Lo scopo del *Document Editor* è quello di diventare un ponte tra il *Document Writer* e il *Process Manager*.
A ciascuno di questi tre ruoli corrisponderà uno specifico prodotto software:

***Validation Engine*:** esegue un insieme dinamico di regole che verificano la correttezza strutturale e sintattica dei documenti, generando relazioni contenenti in risultati. Sarà disponibile sia come Add-In di Microsoft Word, sia tramite una specifica interfaccia web.
Per la rappresentazione delle regole si definiranno due formati, chiamati rispettivamente Collect-And-Check e Collect-And-Compare.
Sarà utilizzato dal *Process Manager*.

***Smart Structured Editor*:** uno strumento web-based innovativo per l'editing collaboritivo di documenti. Si occuperà di gestire versioni, varianti, template strutturali, template di frammenti, riferimenti interni, riferimenti esterni.
Impiegherà la tecnologia nota come Operational Transformation, usata da Google Docs, adattata al formato Office Open XML.
Sarà utilizzato dal *Document Writer*.

***Meta-template Engine*:** sarà disponibile sia mediante un Add-In in Microsoft Word, sia mediante specifica interfaccia web. Attraverso il suo impiego l'utente potrà aggiungere note, selezionare frammenti, organizzare e migliorare i contenuti di ciascun documento, adoperando template precedentemente creati.
Sarà utilizzato dal *Document Editor*.

Le interazioni tra questi tre strumenti sono tratteggiate nella figura 2.1.

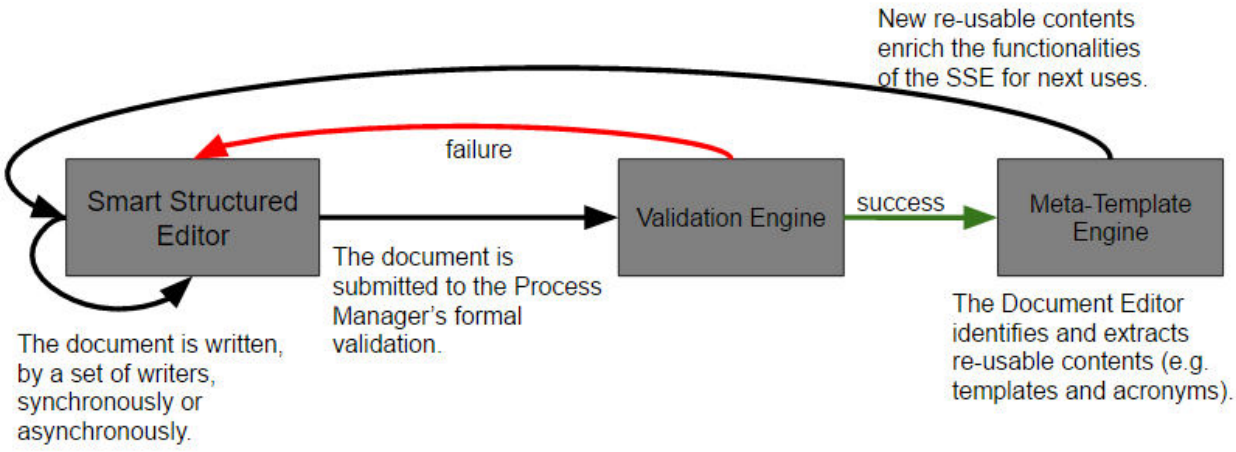

Figura 2.1: Le interazioni tra i tre componenti del progetto SPM.

3

Aspetti teorici e concettuali

3.1 Enterprise Content Management (ECM) e Document Management Systems (DMS)

Il progetto SPM si inserisce nell'ambito dei Document Management System (DMS). Un DMS è un sistema di controllo automatico della documentazione, attraverso tutto il suo ciclo di vita all'interno di un'organizzazione.
Con il termine documento si intende un contenitore di informazione - tipicamente in forma scritta e grafica - che un'organizzazione necessita di produrre e conservare. L'informazione è finalizzata allo *human understanding* ma si può trattare anche di schemi e codice. Il ciclo di vita di un documento è la sequenza di momenti che si susseguono dalla sua creazione alla sua archiviazione o distruzione.
Un DMS permette alle organizzazioni di esercitare un maggiore controllo sulla produzione, archiviazione, distribuzione dei propri documenti, massimizzando il ri-uso dell'informazione precedentemente prodotta e quindi riducendo il tempo (e il costo) complessivo del processo documentale.
Un DMS è tipicamente parte di un sistema più complesso denominato Enterprise Content Management (ECM).
I vantaggi di un DMS comprendono:

- Maggiore efficienza e produttività del processo documentale.
- Compliance con requisiti interni ed esterni, normativi e di qualità della

produzione documentale.

- Coerenza e ripetibilità delle informazioni e delle operazioni.
- Velocizzazione e minimizzazione dei costi del processo.

Esistono famosi prodotti commerciali che offrono questi servizi, ad esempio Documentum[1] e Alfresco[2].

3.2 Version Control System per documenti XML

Il progetto SPM adotta alcune tecniche affini a quelle presenti nei Version Contro System (VCS), come ad esempio Git.
Un Version Control System (VCS) è un sistema di gestione del ciclo di vita di un progetto. Effettua, *in primis*, un tracciamento delle versioni e varianti degli oggetti che compongono il progetto.
I due concetti principali sono quindi il concetto di versione e il concetto di variante. Il versionamento di un documento è il risultato del tracciamento dell'evoluzione del documento nel tempo. Una versione è quindi la forma che il documento assume in un determinato momento storico. Al contrario, la variante è il risultato di un'operazione di ramificazione (branch) che crea una nuova linea di evoluzione del documento.
In termini di teoria dei grafi, le versioni rappresentano un sentiero di sviluppo (dimensione orizzontale), le varianti invece sono dei nuovi sentiero di sviluppo paralleli (dimensione verticale). Aggiungendo altre funzionalità, quali ad esempio le funzioni di merge e di synch, si determina un grafo diretto aciclico (DAG). Il grafo è diretto aciclico in quanto i parenti di un nodo sono sempre all'indietro nel tempo, mentre la direzione degli archi è sempre in avanti nel

[1] `www.emc.com/enterprise-content-management/documentum`
[2] `www.alfresco.com`

tempo.

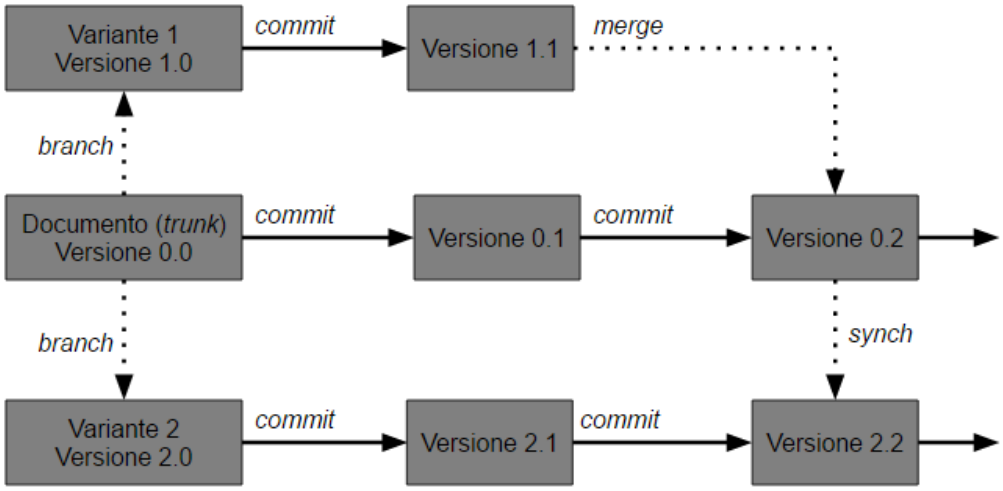

Figura 3.1: Versioni e varianti nel ciclo di vita di un documento. La terminologia è quella standard dei VCS.

Molti sistemi di versionamento documentale sono *edit-based*: tracciano la sequenza di tutte le singole modifiche al documento e ricostruiscono le versioni ripetendo tutte le modifiche intervenute in modo incrementale [1].

3.3 Temporal XML

Un documento tecnico, come quelli prodotti da Alstom, è caratterizzato tipicamente da un ciclo di vita complesso: viene creato, in modo iterativo e progressivo, da una molteplicità di persone, in una pluralità di momenti. Di conseguenza, un'efficace rappresentazione in XML deve tenere conto dell'aspetto temporale.
I documenti (XML) sono collegati alla dimensione temporale in due aspetti [15]: essi contengono informazioni temporali ed il loro contenuto evolve nel tempo. Casi rientranti nel secondo aspetto includono i testi normativi e,

come nel caso del progetto SPM, i documenti tecnici.
Per la rappresentazione di documenti connotati da dimensioni temporali, XML risulta la tecnologia al momento più idonea. Consente di creare una sintassi specifica che arrichisce quella del documento ed è compatibile con la struttura dei dati temporali. Inoltre, XML presenta il vantaggio di possedere tre linguaggi di interrogazione e manipolazione facilmente impiegabili per eseguire query di natura temporale: XQuery, XPath e XSLT.
La soluzione naive al problema del versionamento di un documento XML è il salvataggio di tutta la sequenza di versioni singolarmente. Questa soluzione è però estremamente ridondante. La soluzione ideale è salvare le versioni in modo incrementale nel medesimo file. Questa soluzione, oltre ad evitare la ridondanza, consente anche di implementare metodi efficienti di *version management*. Per realizzare questa soluzione, occorre un idoneo *data model*.
Tutti i modelli esaminati consentono di rappresentare cambiamenti nel documento XML tramite una specifica sintassi che arrichisce la semantica del documento. Due dimensioni temporali sono spesso considerate: tempo di validità (*valid time*) e tempo di transazione (*transaction time*). Altre dimensioni temporali interessano specifici sotto-insiemi di documenti, come ad esempio, i documenti a contenuto normativo. Per questi sono significativi anche la data di pubblicazione (*pubblication time*) e la data di entrata in efficacia della norma giuridica (*efficacy time*).
Di seguito, una breve panoramica dei principali modelli.

3.3.1 XBIT (XML-based Bitemporal Data Model)

In questo modello la validità temporale di un elemento è rappresentata da due attributi, segnatamente vstart e vend [14]. Il *transaction time* invece è rappresentato dagli attributi tstart e tend. Gli intervalli sono inclusivi. Un elemento temporale può essere rappresentato in XBIT nel seguente modo:

```
1 <title vstart="1998-01-01" vend="now">Sr Engineer</title>
```

Questo modello permette l'esecuzione di interrogazioni temporali in XQuery senza dovere introdurre nuovi costrutti linguistici.
Gli attributi del documento possono essere rappresentati mediante sotto-elementi, marcati da uno specifico attributo isAttr. Ad esempio:

```
<projectId isAttr="yes" vstart="2016-05-08" vend="now">
    IT730OCZZROIS0012001
</projectId>
```

Tre operazioni primitive di modifica sono considerate: cancellazione, inserimento e aggiornamento:

Cancellazione (*delete*): se un elemento è rimosso al tempo t, vend è impostanto a $t - 1$.

Inserimento (*insert*): se un nuovo elemento viene inserito al tempo t, l'attributo vstart è impostato da t, vend a now.

Aggiornamento (*update*): se un elemento e è aggiornamto al tempo t, un nuovo elemento e viene inserito subito dopo, i suoi attributi vstart e vend di questo nuovo elemento corrispondono, rispettivamente, a t e a now. L'attributo vend dell'elemento precedente e diventa $t - 1$. Ad esempio:

```
<p vstart="2015-10-01" vend="2016-01-01">
    Progettazione di impianti
</p>
<p vstart="2016-01-02" vend="now">
    Progettazione ed esecuzione di impianti
</p>
```

3.3.2 MXML (Multidimensional XML Model)

Il modello MXML [4] è un approccio generale alla stratificazione di versioni di un documento, non solo rispetto al parametro temporale, ma ad ogni tipo di paramentro (ad esempio linguistico).
Si ottiene un'istanza del documento mediante l'assegnazione dei valori ad un insieme di dimensioni. Tale istanza prende il nome di mondo (*world*).
Gli elementi multi-dimensionali sono denotati dal simbolo "@" e contengono uno o più elementi contestuali. Ad esempio:

```
<document>
    <@title>
        [language = Italian] <title>Progetto di dettaglio</title> [/]
        [language = English] <title>Detailed project</title> [/]
    </@title>
</document>
```

MXML può essere espresso mediante la sintassi XML tradizionale impiegando un insieme di elementi ed attributi specifici. Di conseguenza, l'esempio precedente può essere rappresentato anche in questo modo:

```
<mxml:group title>
    <mxml:celem>
        <mxml:context>Italian</mxml:context>
        <mxml:elem><title>Progetto di dettaglio</title></mxml:elem>
    </mxml:celem>
    <mxml:celem>
        <mxml:context>English</mxml:context>
        <mxml:elem><title>Detailed project</title></mxml:elem>
    </mxml:celem>
</mxml:group>
```

3.4 Operational transformation

Lo Smart Structured Editor è un editor online collaborativo e, come tale, si avvale della tecnologia denominata Operational Transformation (OT).
Lo scopo dell'editor è quello di consentire a un pluralità di persone di lavorare su un medesimo documento, tramite accesso a un servizio web, effettuando modifiche da qualunque luogo e in qualunque momento. Le modifiche vengono sincronizzate immeditamente con gli altri *peer*, garantendo la coerenza del documento.
Quest'ultima caratteristica differenzia OT da un sistema di controllo delle versioni come Git, dove un utente tipicamente lavora da solo su una feature e alla fine effettua un'operazione denominata *merge*, che consegna le modifiche al ramo principale del progetto. Di converso in OT, nessun *client* deve comunicare al *server* o agli altri *client* l'intenzione di effettaure una modifica. In particolare, non è necessario acquisire un *lock* dal *server*. Modifiche concorrenti possono avvenire e, dopo che tutti i cambiamenti vengono sincronizzati, ogni *client* avrà a disposizione lo stesso documento modificato [10].

In termini generali, la concorrenza è la situazione di conflitto dove più di una transazione intende accedere alla stessa risorsa nello stesso momento. Il controllo di concorrenza (*concurrency control*) è il problema di sincronizzare le transazioni concorrenti. La classificazione più comune è tra algoritmi basati su accesso mutualmente esclusivo dei dati condivisi (*locking*), e algoritmi che tentano di ordinare l'esecuzione delle transazione sulla base di un insieme di regole (*protocols*). Esistono due approcci: l'approccio pessimistico che prevede statisticamente molte transazioni confliggenti, e l'approccio ottimistico [6] che prevede pochi conflitti. Gli algoritmi pessimistici sincronizzano l'esecuzione concorrente di transazioni in maniera anticipata, mentre gli algoritmi ottimistici ritardano la sincronizzazione della transazione, aspettando che essa termini.
OT è una classe di algoritmi ottimistici per il controllo della concorrenza [9].

OT è una tecnologia originariamente sviluppata per il mantenimento della coerenza (*consistency maintenance*) e per il controllo della concorrenza (*concurrency control*) nell'editing collaborativo di documenti di testo. Successivamente si sono aggiunte nuove funzionalità, in particolare algoritmi per la risoluzione di conflitti, algoritmi per la modifica di documenti ad albero (come quelli in formato XML) e algoritmi per la modifica di documenti contenenti elementi non testuali (*rich text documents*).
Dal 2009, la tecnologia Operational Transformation è adottata dai principali software di editing collaborativo, come Apache Wave[3] e Google Docs[4].

Tipicamente, nei sistemi basati su Operational Transformation, ogni utente effettua le modifiche su una replica locale. Queste modifiche vengono, nel più breve tempo possibile, propagate agli altri utenti, tramite l'applicazione di protocolli di controllo della concorrenza, che evitano e correggono eventuali incoerenze.
Alcune delle principali caratteristiche che definiscono un sistema di editing collaborativo sono [5]:

***High Local Responsiveness*:** L'editor collaborativo deve essere tendenzialmente reattivo quanto un editor non collaborativo.

***Unconstrained Interaction*:** L'utente deve essere in grado di modificare qualunque parte del documento condiviso, in ogni istante, con le medesime modalità e potenzialità di un editor classico non collaborativo.

***Distributed*:** Ogni utente può essere separato geograficamente, usare diversi computer, avere diversi *latency delay*.

[3] `incubator.apache.org/wave`
[4] `www.google.com/docs`

***Real-Time Communication*:** Gli utenti devo essere notificati delle modifiche effettuate dagli altri utenti in modo tempestivo, per garantire un effettivo coordinamento.

***Consistency*:** Gli utenti devono essere in grado di visualizzare una versione consolidata del documento condiviso, consistente con tutte le modifiche eseguite.

3.4.1 Basi concettuali

Per capire il funzionamento di OT occorre introdurre alcune definizioni [16]:

***Causal Relation*:** Date due operazioni, O_a e O_b, generate rispettivamente dai siti i e j, O_a è causalmente antecedente a O_b, denotato $O_a \rightarrow O_b$, se e solo se: (1) $i = j$ e O_a è stata generata prima di O_b; (2) $i \neq j$ e O_a è stata eseguita prima di O_b; (3) esiste un'operazione O_c tale che $O_a \rightarrow O_c$ e $O_c \rightarrow O_b$.

***Concurrent Relation*:** Date due operazioni, O_a e O_b, esse sono concorrenti, denotato con $O_a \parallel O_b$, se e solo se non vale che $O_a \rightarrow O_b$ e nemmeno che $O_b \rightarrow O_a$.

***Total Ordering Relation*:** Date tre operazioni, O_a, O_b e O_c, tali che $O_a \neq O_b \neq O_c$, esiste una relazione di ordinamento totale se: (1) se $O_a \implies O_b$ e $O_b \implies O_c$ allora $O_a \implies O_c$ (transitività), (2) $O_a \implies O_b$ o $O_b \implies O_a$ (totalità).

***Document State Representation*:** lo stato di un documento s, denotato $C(s)$ è definito come segue: (1) lo stato iniziale s_0 corrisponde all'insieme vuoto $C(s_0) = \oslash$, (2) dopo l'esecuzione dell'operazione O sul documento s, si determina un nuovo stato $s \circ O$, rappresentato da

$C(s \circ O) = C(s) \cup org(O)$, dove $org(O)$ rappresenta la forma originale di O. Se O è un'operazione originale (generata dall'utente), allora $org(O) = O$.

***Operation Context Representation*:** il contesto di un'operazione O, denotato $C(O)$, è così definito: (1) se l'operazione è originale, $C(O) = C(s)$, con s lo stato del documento dove O è stata generata, (2) per un'operazione transformata $O' = T(O, O_x)$, $C(O') = C(O) \cup org(O_x)$, dove T è la funzione di trasformazione di O in O', tenendo conto dell'impatto di O_x.
In OT; ogni operazione è associata con un contesto, che costituisce lo stato del documento sul quale l'operazione è stata generata.

Le operazioni si distinguono in:

***Operation generation*:** un'operazione determinata dall'utente e diffusa a tutti gli altri utenti.

***Operation reception*:** una richiesta di operazione ricevuta da un altro utente.

***Operation execution*:** una richiesta di operazione eseguita, dopo che è stata ricevuta.

Le funzioni di trasformazione possono essere di due tipi [7]:

Inclusion Transformation $T_i(O_a, O_b)$**:** trasforma O_a tramite O_b in modo tale che l'impatto di O_b è incluso .

Exclusion Transformation $T_e(O_a, O_b)$**:** trasforma O_a tramite O_b in modo tale che l'impatto di O_b è escluso.

3.4.2 La coerenza

Nei sistemi di editing collaborativo real-time, le operazioni con relazione causale sono eseguite in ordine causale, ma le operazioni concorrenti possono

essere eseguite in qualunque ordine. Al contrario, OT garantisce che le operazioni concorrenti sono trasformate al fine di garantire la coerenza.

Il mantenimento della coerenza è un tema fondamentale in molte aree della scienza informatica. Ad esempio, nell'ambito delle basi di dati, dei sistemi distribuiti e dei sistemi collaborativi (anche noti come *groupware systems*). I modelli di coerenza definiscono le regole finalizzate a determinare il risultato di una sequenza di operazioni, effettuate da diversi utenti su diversi siti. Un esempio di incoerenza si verifica quando un utente modifica la sua copia locale del documento, ma non ha ancora propagato questi cambiamenti agli altri utenti. Se qualcuno di questi utenti intende leggere o scrivere sullo stesso documento, si trova ad usare una versione non aggiornata dello stesso.

La coerenza è garantita dal soddisfacimento di tre proprietà [12]:

***Convergence*:** se il medesimo insieme di operazioni è stato eseguito presso ogni utente, allora ogni utente visualizza la medesima copia del documento.

***Causality preservation*:** date due operazioni O_a e O_b, se $O_a \rightarrow O_b$ (O_a avviene prima di O_b), allora O_a è eseguito prima di O_b in ogni sito.

***Intention preservation*:** per ogni operazione O, l'intenzione originaria di O sarà identica all'esecuzione di O in tutti gli altri siti. L'intenzione di un'operazione O è definita come il documento risultante dopo che è stata applicata O al *document state* dal quale O è stata generata.

La proprietà di convergenza garantisce la correttezza del documento alla fine di ogni modifica. Invece, la proprietà di preservazione causale garantisce la correttezza del documento in ogni momento durante la sua modifica. La preservazione dell'intenzione è simile alla proprietà di convergenza, ma aggiunge anche la situazione dove le operazioni sono sottomesse da due differenti stati iniziali del documento. Questa è la proprietà più difficile da garantire, e

spesso richiede che un'operazione sottomessa da un utente venga modificata nel momento in cui viene ricevuta da un altro.

Per ottenere la convergenza sono state delineate due proprietà: la Convergence Property 1 (CP1) e la Convergence Property 2 (CP2):

***CP1*:** Date due operazioni, O_a e O_b, generate nello stato s, la funzione di trasformazione T soddisfa CP1 se e solo se $s \circ O_a \circ T(O_b, O_a) = s \circ O_b \circ T(O_a, O_b)$.

***CP2*:** Date tre operazioni, O_a, O_b e O_c, generate nello stato s, la funzione di trasformazione T soddisfa CP2 se e solo se $T(T(O_c, O_a), T(O_b, O_a)) = T(T(O_c, O_b), T(O_a, O_b))$.

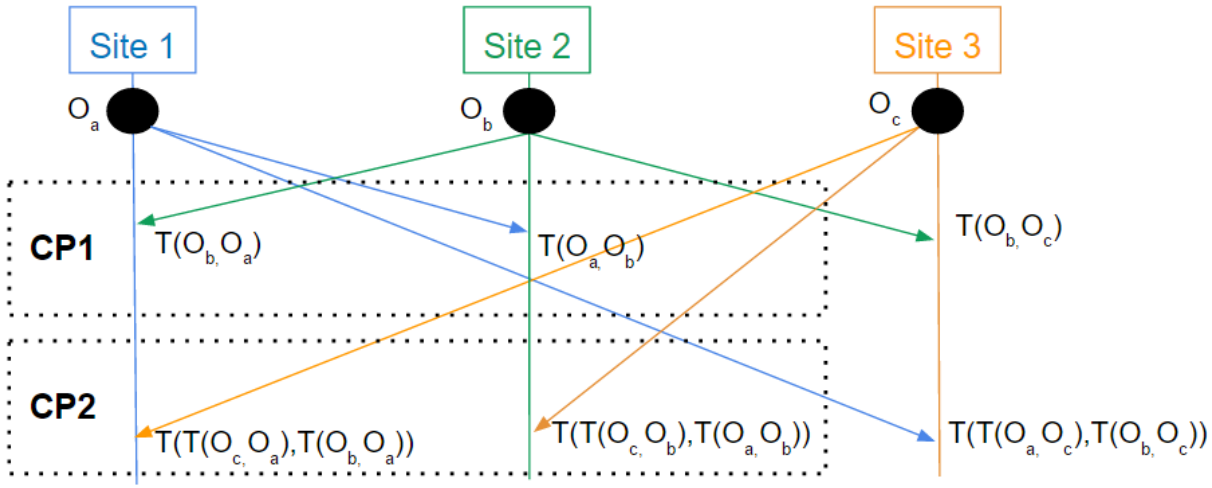

Figura 3.2: Uno scenario minimale di OT che garantisce sia CP1, sia CP2.

Come esemplificato dalla figura 3.2, la proprietà CP1 garantisce che se due operazioni (O_a, O_b) sono concorrenti, l'effetto di eseguire O_a prima di O_b o viceversa è equivalente. La proprietà CP2 garantisce che eseguire un'operazione O_c tramite sequenze di operazioni diverse ma equivalenti determina il

medesimo risultato [16].
Queste due proprietà assieme assicurano che un ordine arbitrario di comunicazione delle operazioni può condurre ad uno stato del documento finale consistente. In altre parole, non è necessario imporre un ordine totale globale delle operazioni perchè eventuali incoerenze possono essere sempre aggiustate tramite OT, se e solo se CP1 e CP2 sono soddisfatte [5].

Un sistema di OT consiste di due componenti: l'algoritmo di controllo (*control algorithm*) del processo di trasformazione; le funzioni di trasformazioni (*transformation functions*) che sono *application-specific* ed effettuano trasformazioni tra operazioni.
Ci sono due approcci che raggiungono la convergenza in un sistema OT: funzioni di trasformazinoi capaci di preservare CP1 e/o CP2; oppure disegnare un algoritmo di controllo generico capace di evitare CP1 e/o CP2.

3.4.3 L'*intention preservation*

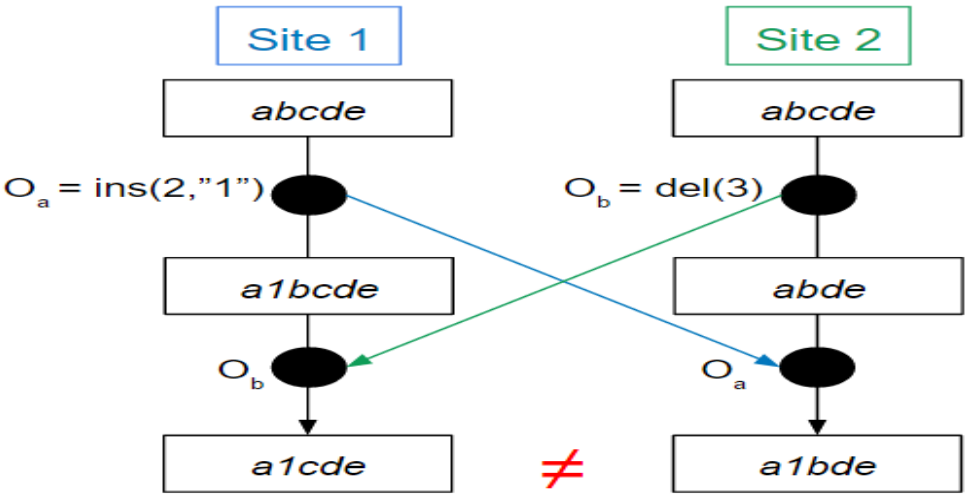

Figura 3.3: Il problema dell'*intention preservation*.

Concetto strettamente connesso alla coerenza è l'*intention preservation*. Si consideri il seguente esempio, tratteggiato nella figura 3.3. Due siti stanno modificando il medesimo documento formato dalla stringa "abcde". Il sito 1 effettua l'operazione $O_a = ins(2, "1")$ (inserisce "1" nella posizione 2), ottenendo il risultato locale "a1bcde". Il sito 1 ha quindi eseguito l'operazione con lo stato del documento "abcde", ovvero con l'intenzione di inserire "1" tra "a" e "b". Il sito 2, invece, elimina un carattere nella posizione 3 ($O_b = del(3)$), ottenendo "abde". L'intenzione del sito 2 era quindi di eliminare il carattere "c".
Se entrambe le operazioni venissero eseguite preservando l'intenzione, il risultato finale sarebbe "a1bde". Di converso, un mero protocollo di serializzazione potrebbe condurre a serializzare O_a prima di O_b, determinando come risultato finale "a1bcde". In quest'ultimo caso verrebbe garantita la convergenza ma non verrebbero rispettate le intenzioni.

3.5 Il problema dell'Overlapping Markup

3.5.1 I limiti rappresentativi della struttura dati ad albero

XML adotta una struttura dati ad albero, ideale per rappresentare una gerarchia logica tra elementi. Qualche problema sorge nel momento in cui si vuole descrivere, nel medesimo albero, più gerarchie logiche. Lo studio di gerarchie concorrenti, in ambito XML, cerca di risolvere questo problema. Tuttavia, al momento, esso è rivolto sopratutto a documenti XML contenenti dati (*data-centric XML*), più che a documenti testuali [2].

Un albero, come tutti i formalismi, ha una capacità espressiva limitata. In particolare, un documento rappresentato in XML richiede che ogni elemento figlio sia contenuto in un solo elemento padre, fino a giungere alla radice

dell'albero. Questa limitazione incide sulla possibilità di associare, ad uno stesso frammento del documento, più elementi XML intersecanti e quindi gerachicamente incompatibili. Questo problema è noto in lettaratura con il termine *overlapping problem* [13].
Una classificazione dei casi di overlap è la seguente [13]:

***Classic overlap*:** è il caso in cui due frammenti, che devono essere annotati con diversi identificatori, si intersecano tra di loro. Questo avviene tipicamente quando l'annotazione del documento concerne gerarchie multiple e concorrenti, finalizzate a descrivere il documento da punti di vista diversi. Ad esempio, un documento in cui coesistono elementi che descrivono la struttura sintattica del documento ed elementi che tracciano i cambiamenti:

```
1 <add><title>Modulo di Registrazione <add>Cronologica <del>con
      Storicizzazione </add>degli Eventi </del>e Monitoraggio
      Variabili</title></add>
```

L'esempio precedente travalica chiaramente la struttura ad albero con padre unico, richiedendo una struttura ad albero con padri multipli (vedi la figura 3.4).

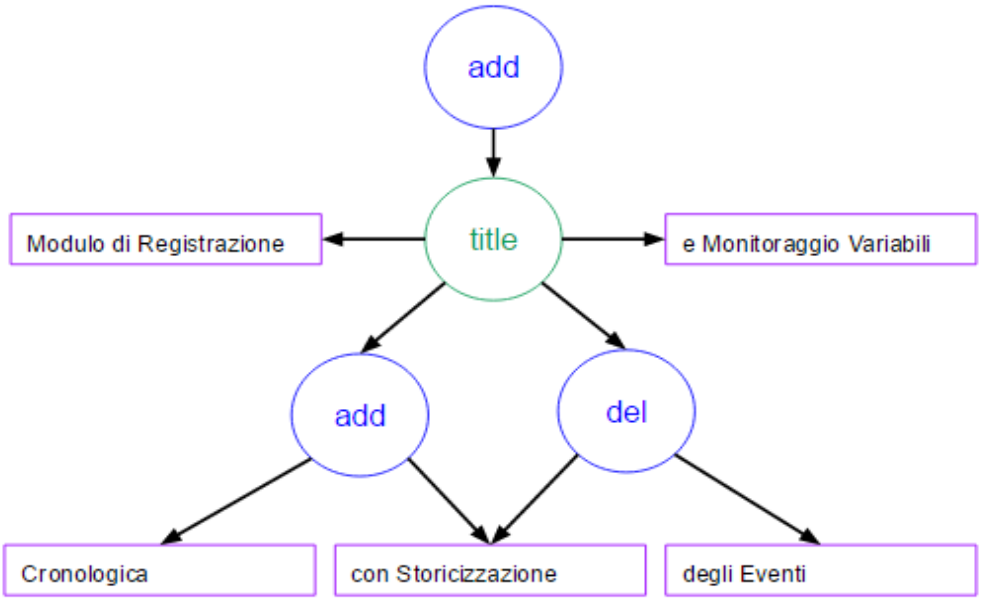

Figura 3.4: L'overlapping markup richiede una struttura dati ad albero con parentele multiple. Come descritto più avanti, questa rappresentazione prende il nome di GODDAG.

***Self overlap*:** è il caso in cui si intersecano annotazioni di 2 frammenti che impiegano lo stesso identificatore.

***Virtual elements*:** sono elementi non esplicitamente presenti nel documento, ma che vengono inferiti dall'applicazione che processa il documento. Si pensi al caso di annotare parti non continue di testo.

***Containment/dominance decoupling*:** la rappresentazione ad albero tende a collimare il concetto di contenimento con il concetto di dominazione. Invero sono concetti diversi, contenimento significa che tutto il contenuto di un frammento è racchiuso in un più grande frammento. Dominazione invece si riferisce al concetto di parentela tra elementi. Il frammento a domina il frammento b, se a è un parente superiore (*anchestor*) di b.

Per la soluzione del problema dell'overlapping, si possono distinguere due principali approcci. Il primo è volto a stressare i limiti del formato XML mediante un appiattimento dell'albero. Si tratta di un approccio pragmatico in quanto sfrutta l'esistente, ovvero le tecnologie XML, ma teoricamente "sporco" in quanto deforma l'albero XML, rendendolo irrilevante. Il secondo approccio è invece innanzitutto teorico: preso atto che un albero XML non è la struttura dati idonea, punta a definire una struttura dati più potente.

Per quanto concerne il primo approccio, le linee guida TEI (Text Encoding Initiative)[5], descrivono le seguenti tecniche:

***Milestones*:** la struttura ad albero diventa irrilevante, le gerarchie intersecanti sono espresse da elementi vuoti (chiamati *milestones*, ovvero pietre miliari). L'esempio precedente diventa:

```
1  <add id="add-1-start"/>
2  <title id="title-1-start"/>
3  Modulo di Registrazione
4  <add id="add-2-start"/>
5  Cronologica
6  <del id="del-1-start"/>
7  con Storicizzazione
8  </add id="add-2-end"/>
9  degli Eventi
10 </del id="del-1-end"/>
11 e Monitoraggio Variabili
12 </title id="title-1-end"/>
13 </add id="add-1-end"/>
```

Come risulta evidente, anche da un esempio così banale, questo approccio è pragmaticamente finalizzato a rimanere nell'alveo del formato dati

[5] http://www.tei-c.org

XML, tuttavia l'albero XML diventa irrilevante (vedi la figura 3.5). Il vero albero con parentele multiple viene ricostruito dinamicamente.

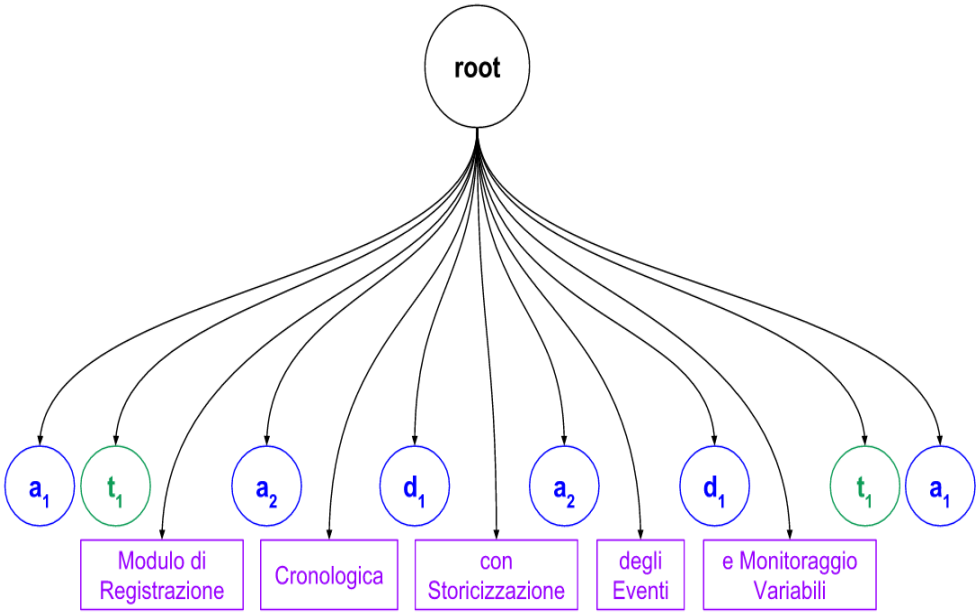

Figura 3.5: L'albero XML diventa una struttura logica irrilevante nella tecnica che impiega i *milestones* in modo totale (CLIX).

***Fragmentation*:** le strutture intersecanti sono divise in elementi parziali non intersecanti. Convenzioni sintattiche sono impiegate per distinguire tra elementi parziali (frammenti) ed elementi non parziali. Con questa tecnica l'esempio precedente può essere definito in questo modo:

```
1  <add>
2      <title>
3          Modulo di Registrazione
4          <add>
5              Cronologica
6              <del id="del-1-frag-1">
7                  con Storicizzazione
8              </del>
9          </add>
```

```
10          <del id="del-1-frag-2">
11              degli Eventi
12          </del>
13          e Monitoraggio Variabili
14      </title>
15  </add>
```

Come si vede dalla figura 3.6, la frammentazione consente di mantenere intatta la struttura dati ad albero XML, pagando un costo in termini di proliferazione del numero di elementi.

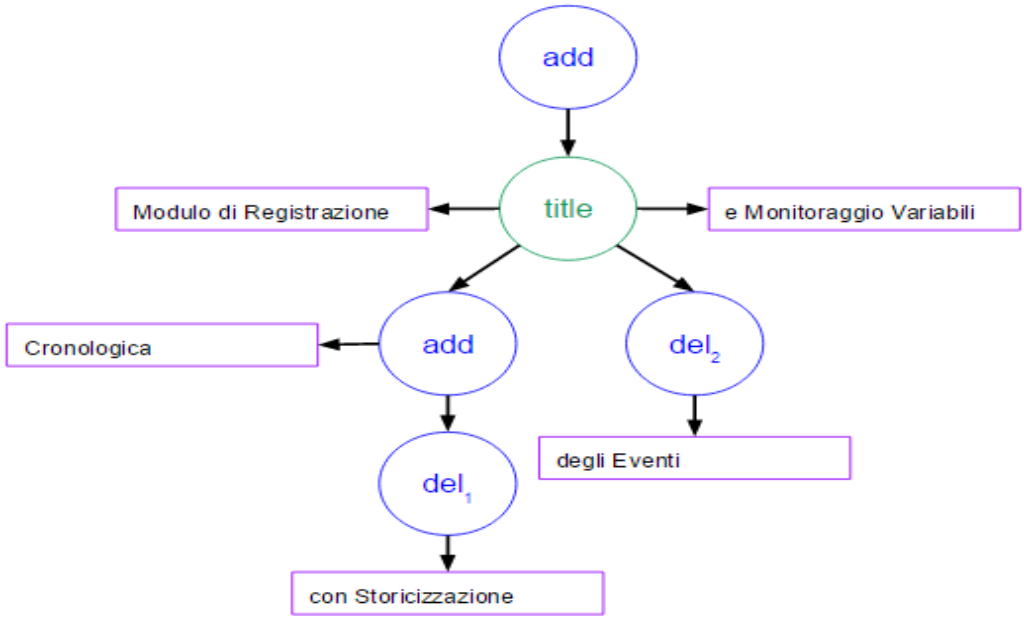

Figura 3.6: La frammentazione preserva la struttura ad albero XML tramite moltiplicazione degli elementi.

***Twin documents*:** questa è forse la soluzione più naive al problema dell'overlapping. Si tratta di creare una copia del documento per ogni gerarchia strutturale.

***Standoff markup*:** le strutture intersecanti sono collegate al di fuori del documento.

3.5.2 LMNL (Layered Markup and Annotation Language)

LMNL[6] supera le limitazioni di XML, rinunciando completamente al formato XML. Può rappresentare tutti i tipi di overlap e si basa su intervalli stratificati che possono sovrapporsi. Il suo punto di partenza è un modello dati, piuttosto che una sintassi, tuttavia, CLIX e ECLIX fanno parte di LMNL.
In breve, un documento LMNL è un insieme di strati dove ogni strato è formato da una sequenza di caratteri o da una sequenza di intervalli. Gli intervalli riprendono la nozione matematica e sono definiti in cima a uno strato. Uno strato può essere la base di più strati, ma ha come base un solo strato. Lo strato di testo non ha base [8].

3.5.3 CLIX e ECLIX Milestones

Al fine di rappresentare strutture intersecanti in XML, l'approccio che fa uso di milestone distingue tra un vocabolario primario, usato per definire la gerarchia XML del documento, e vocabolari secondari. Questi ultimi sono rappresentati medianti elementi vuoti che delimitano l'inizio e la fine delle parti da annotare. Questo è il caso ECLIX che consente di rappresentare il *classic overlap* e il *self-overlap*, ma non i *virtual element*.
Diversamente, in CLIX ogni elemento di ogni gerarchia è definitivo mediante i milestone.

[6] `http://lmnl-markup.org`

3.5.4 GODDAG

Come detto in precedenza, i documenti XML sono alberi connotati da una parentela unica (ogni elemento ha un solo padre) e una relazione di ordinamento totale dei nodi. Questa struttura garantisce la rappresentazione di un grande numero di documenti, ma non è sufficiente per l'overlapping.
Per questo motivo, un diverso di tipo di grafo è stato proposto. Il GODDAG (*General Ordered-Descendant Directed Acyclic Graphs*) è un albero che consente parentele multiple e non richiede un ordinamento totale a livello di foglie. Il GODDAG è inoltre privo di archi transitivi e rappresenta sia i contenuti testuali, sia gli elementi strutturali, come nodi.
Un GOODAG ha le seguenti caratteristiche [11]:

- Ogni nodo è terminale (foglie) o non-terminale.
- Ogni nodo terminale è connotato con una etichetta specifica.
- Ogni non-terminale è connotato con un identificatore generico.
- I nodi sono collegati mediante archi diretti.
- Se esiste un cammino dal nodo a al nodo b, chiameremo b un discendente di a.
- I nodi terminali sono quelli dai quali non parte nessun arco.
- I nodi non-terminali sono tutti quelli che non sono terminali.
- Per ogni 3 nodi a, b, c, se esistono gli archi $a \Rightarrow b$ e $b \Rightarrow c$, allora non esiste l'arco $a \Rightarrow c$.

L'albero XML è un sottoinsieme di GODDAG e, quindi, tutte le diverse tecniche basate su XML (ad esempio l'utilizzo dei milestones in CLIX e ECLIX) possono essere mappate in GODDAG.
Diverse istanze di GODDAG sono state definitve: generalizzato, ristretto e

"pulito".
Per lo scopo dell'overlapping il GODDAG ristretto è l'ideale. Le sue caratteristiche sono:

- Le foglie formano una sequenza (denominata frontiera) e sono ordinate in modo totale.
- Ogni nodo domina un sotto-sequenza contigua della frontiera.
- Non esistono due nodi che dominano la medesima parte della frontiera.

La figura 3.4 è un esempio di GODDAG.

3.6 Il formato Office Open XML

Alstom adotta e non intende rinunciare a Microsoft Office. I documenti prodotti dai suoi ingegneri sono editati in MS Word, MS Excel, MS Power Point. Il sistema di archiviazione richiede documenti nei formati Office, in particolare, nel formato Office Open XML (OOXML). Di conseguenza, per il progetto SPM, la scelta di OOXML è imprescindibile.
OOXML è un formato XML, sviluppato da Microsoft, per rappresentare fogli di calcolo (*spreadsheet*), grafici, presentazioni e documenti di testo (*word processing*). Il formato è diventato standard ECMA (ECMA-376[7]) e, successivamente, standard ISO e IEC (ISO/IEC 29500).
ECMA-376 definisce una famiglia di schemi XML, finalizzati a codificare i principali file di tipo Office (.docx, .xlsx, e .pptx), garantendo l'interoperabilità tra le applicazioni che ne fanno uso.
Tutte e tre i tipi di file sono in realtà contenitori (archivi) strutturati di file. Si basano sull'Open Packaging Conventions (OPC) che stabilisce una struttura logica per archiviare tutti i vari contenuti di un documento (contenuto

[7] `www.ecma-international.org/publications/standards/Ecma-376.htm`

testuale, immagini, metadati, ecc.) in un unico contenitore che utilizza il formato ZIP.

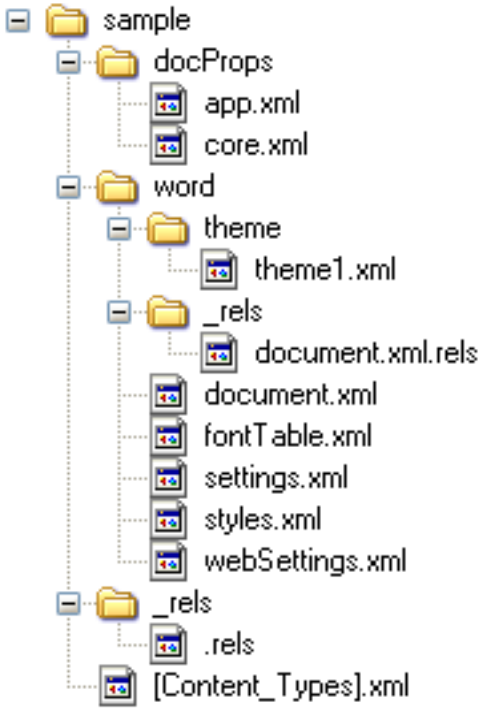

Figura 3.7: La struttura gerarchica di un file docx.

Per rappresentare markup intersecanti semplici, OOXML fa uso della frammentazione. Ad esempio, il testo "**Grassetto *Grassetto-E-Corsivo* *Corsivo***" viene definito in questo modo (il codice è stato semplificato per maggiore leggibilità):

```
1 <w:r>
2     <w:rPr>
3         <w:b />
4     </w:rPr>
5     <w:t>Grassetto  </w:t>
6 </w:r>
7 <w:r>
8     <w:rPr>
```

```
9         <w:b />
10        <w:i />
11    </w:rPr>
12    <w:t>Grassetto-E-Corsivo</w:t>
13</w:r>
14<w:r>
15    <w:rPr>
16        <w:i />
17    </w:rPr>
18    <w:t>Corsivo</w:t>
19</w:r>
```

Per quanto riguarda il *change tracking*, in particolare il tracciamento degli spostamenti, lo scenario è più complesso [3]. Le informazioni da salvare sono:

- Un insieme di frammenti spostati.
- Un contenitore (o bookmark) sorgente e uno di destinazione che specificano che tutto il loro contenuto è parte di un singolo spostamento.

Si valuti il seguente esempio:

```
1 <w:p>
2   <w:moveToRangeStart w:id="0" ... w:name="move1" />
3   <w:moveTo w:id="1" ...>
4     <w:r>
5       <w:t>2</w:t>
6     </w:r>
7   </w:moveTo>
8   <w:moveFromRangeEnd w:id="0" />
9   <w:r>
10    <w:t>1</w:t>
11  </w:r>
12  <w:moveFromRangeStart w:id="2" ... w:name="move1" />
13  <w:moveFrom w:id="3" ...>
14    <w:r>
```

```
15        <w:t>2</w:t>
16      </w:r>
17    </w:moveFrom>
18    <w:moveFromRangeEnd w:id="2" />
19  </w:p>
```

Il testo "2" è stato spostato dalla parte finale alla parte iniziale del paragrafo. Gli elementi moveToRangeStart e moveFromRangeEnd sono milestone che racchiudono un elemento modeTo con il testo spostato. Il testo "1" è invece il testo rimasto inalterato. Il testo spostato è ripetuto per semplificare la visualizzazione delle due versioni: "12" e "21", si tratta quindi di un caso di ridondanza finalizzata all'efficienza computazionale degli editor.
L'approccio è dunque misto: milestone e fragmentation.

4

Il Validation Engine

4.1 Requisiti

Il Validation Engine è un motore di regole che verificano la correttezza formale e sostanziale di un documento. Si prevede sia un controllo intra-documentale, sia inter-documentale.
Il Validation Engine annovera la realizzazione dei seguenti *use case* (4.1):

- L'utente carica uno o più documenti. Inizialmente si prevede solo la possibilità di caricare file di tipo docx, xml e zip contenenti docx e xml, ma in futuro si aggiungerà anche il formato xlsx.
- L'utente sceglie una o più regole applicabili sui documenti.
- Le regole sono state precedentemente create e rese disponibili dal responsabile del processo di validazione documentale (il Process Manager assieme al Document Editor).
- L'utente lancia l'esecuzione delle regole.
- Ad esecuzione terminata, l'utente può visualizzare i risultati, oppure scaricarli e inglobarli al documento o ai documenti ai quali si riferiscono.

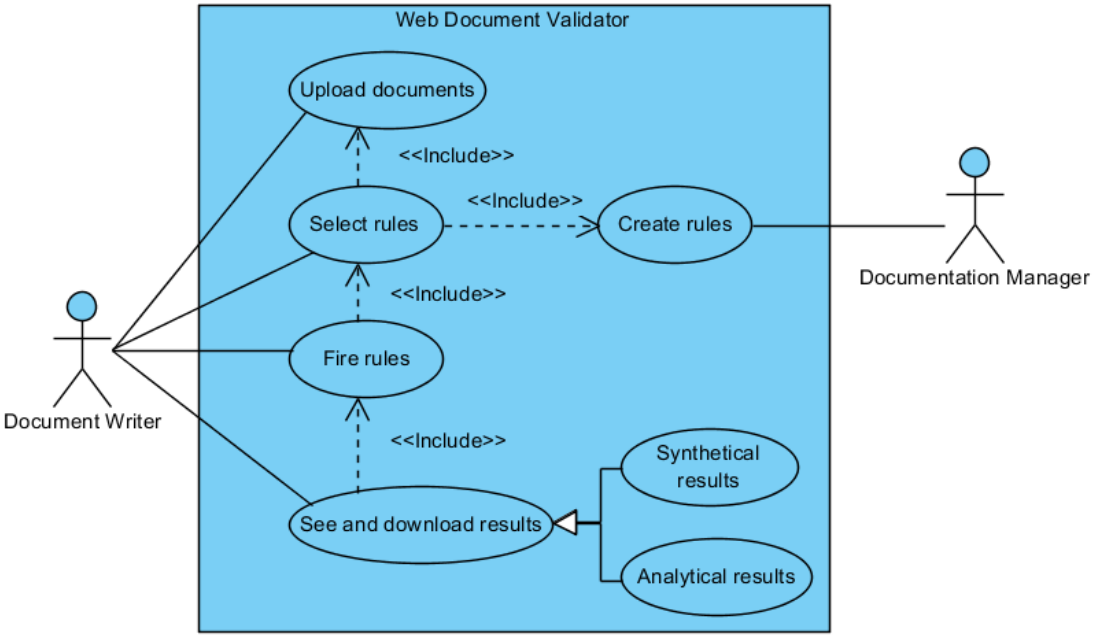

Figura 4.1: UML Use Case Diagram per il Validation Engine

4.2 Lo stack tecnologico adottato

Lo stack tecnologico adottato per lo sviluppo del Validation Engine è il seguente:

Front-end: Javascript, Jade[1] (come *template engine*), jQuery[2].

Back-end: Node.js[3], Express.js[4] (come *development framework*), Passport[5] (come *authentication middleware*), MongoDB[6] (come database per l'archiviazione delle credenziali).

[1] jade-lang.com
[2] jquery.com
[3] nodejs.org
[4] expressjs.com
[5] passportjs.org
[6] mongodb.com

Package manager: Bower[7] e NPM[8].

Revision control system: Git[9].

IDE: Visual Studio Code[10], Sublime Text[11].

Tutta la logica dell'applicazione è stata inserita, per scelta progettuale, a livello server. La parte front-end, quindi, contiene solo il codice finalizzato al funzionamento dell'interfaccia web e alla generazione di chiamate AJAX.

Come linguaggio di back-end è stato impiegato Node.js. Node.js è un ambiente di sviluppo open-source, cross-platform, per lo sviluppo di applicazioni web server-side. Si basa su Javascript e sfrutta le capacità computazionali del motore Javascript V8 di Google[12].
Una caratteristica peculiare di Node.js, sfruttata anche in questo progetto, è la sua architettura event-driven, che permette di adottare uno stile di programmazione asincrona. In particolare, questo consente di ottimizzare il *throughput* e la *scalability* di applicazioni *I/O intensive*.
Oltre al vantaggio di scrivere un'applicazione web usando Javascript sia server-side, che client-side, e oltre al vantaggio presentato da un'architettura event-driven, Node.js permette la creazione di Web server mediante specifici moduli scritti in Javascript. In particolare, Node.js opera su un singolo thread, usando chiamate I/O non-bloccanti, supportando decine di migliaia di connessioni concorrenti.
Tutti questi vantaggi stanno segnando il successo di Node.js, successo che travalica il classico emisfero open-source. Infatti, Node.js è stato adottato ufficialmente anche da Microsoft. Microsoft impiega e promuove Node.js e

[7] `bower.io`
[8] `www.npmjs.com`
[9] `git-scm.com`
[10] `code.visualstudio.com`
[11] `www.sublimetext.com`
[12] `developers.google.com/v8`

ha sviluppato Visual Studio Code, un IDE leggero ed estendibile, orientato al web development, in particolare a Node.js.
Express.js è il più popolare *application framework* per Node.js, caratterizzato da minimalità e flessibilità. Mentre Passport.js è una libreria middleware per gestire il processo di autenticazione in Node.js ed Express.js. Passport impiega MongoDB per l'archiviazione delle credenziali, ogni credenziale viene salvata in una tripla (nome utente, hash, sale). MongoDB è un database open-source NOSQL che utilizza una struttura dati documentale, denominata BSON (Binary JSON). BSON è una forma binaria per rappresentare array associativi tramite la sintassi di JSON, ma a differenza di JSON dispone di una più ampia varietà di tipi di dato (ad esempio JSON non ha un tipo *date* o *byte array*).

Per quanto riguarda la parte front-end, una scelta tipica per il *server side templating* in Node.js, è Jade. Jade è un *templating engine*, finalizzato a semplificare la sintassi per generare pagine HTML, secondo il principio di *software development* DRY (Don't Repeat Yourself). La sua sintassi è simile a quella di HAML[13] (HTML Abstraction Markup Language) che, a sua volta, è un sistema di templating per definire documenti HTML.

4.3 Schermate principali

Prima di entrare nella schermata principale occorre autenticarsi. Il servizio web è infatti riservato ad utenti qualificati: i dipendenti di Alstom. Non è previsto quindi un meccanismo di registrazione online, ma si prevede l'utilizzo di un sofware di gestione delle credenziali (creazione, cancellazione) interno all'organizzazione.
La schermata principale è, per il momento, minimale. Scinde le tre fasi principali di interazione con l'utente in tre blocchi precisi. Di sotto sono

[13] `http://haml.info`

visibili le schermate con le tre fasi e i tre blocchi. Si noti come nella fase $n, n <= 3$, i blocchi $m, m <= n$ sono visibili ma disattivati. Il solo modo per tornare indietro è cliccare nella freccia in alto a destra del blocco attuale.

Figura 4.2: Fase 0: l'utente si deve autenticare.

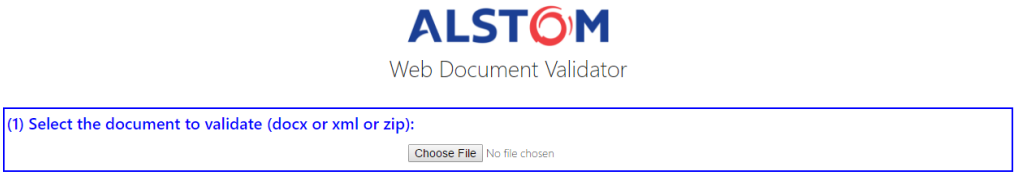

Figura 4.3: Fase 1: l'utente può caricare il documento o i documenti che intende validare.

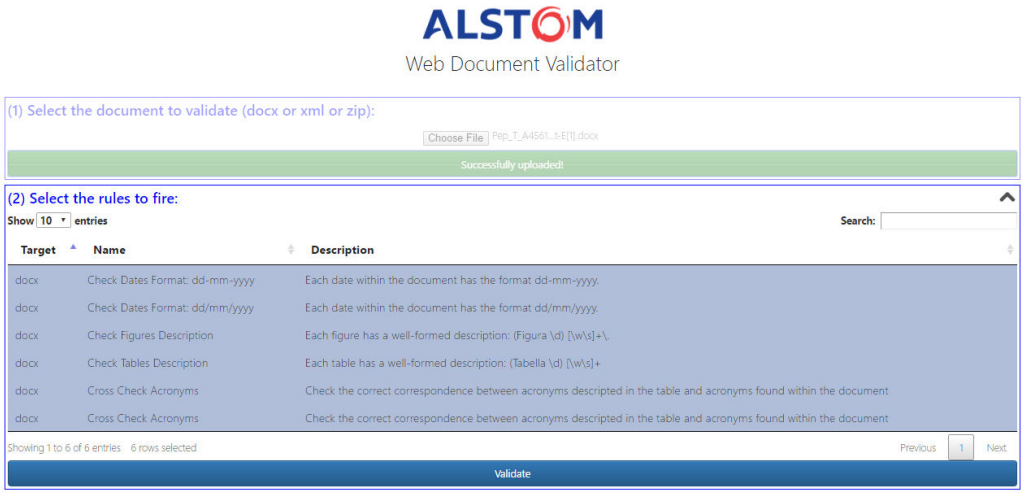

Figura 4.4: Fase 2: l'utente può selezionare le regole disponibili per la validazione dei documenti caricati ed eseguirle. Altrimenti può tornare indietro e scegliere altri documenti da caricare.

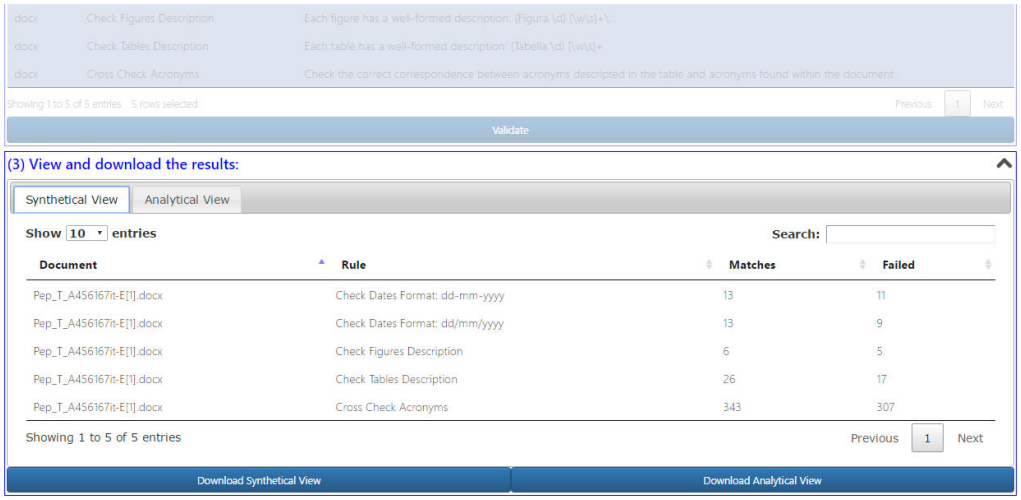

Figura 4.5: Fase 3: l'utente può visualizzare i risultati analitici e sintetici delle regole eseguite, scaricare i relativi report. Altrimenti può tornare indietro e scegliere altre regole da eseguire.

5

Aspetti implementativi

In questo capitolo verranno presentati gli aspetti implementativi principali del Validation Engine. Il codice sorgente del prototipo è ubicato, con licenza open-source, in una repository su GitHub[1].

5.1 Struttura dell'applicazione

Si possono distingure le seguenti parti di basso livello:

- Una repository di regole, in formato XML.
- Il motore delle regole, parte del codice server-side, ma concettualmente e concretamente distinto dal resto.
- Il resto del codice server-side, per l'interazione tra motore di regole e *web clients*.
- La parte client-side, per la generazione e l'operatività dell'interfaccia web.

Inoltre, per la gestione (creazione ed eliminazione) degli account è previsto un tool separato che interagisce con il database MongoDB a livello Intranet[2].

[1] `https://github.com/tomOgn/ValidationEngine`
[2] `https://github.com/tomOgn/ValidationEngineUserManagement`

5.2 Regole

Per quanto concerne le regole, ne sono state enucleate due tipologie, denominate Collect-And-Check e Collect-And-Compare. Entrambe si compongono di due parti. La prima, finalizzata a raccogliere elementi dal documento, denominata Collect. La seconda, finalizzata a valutare o confrontare gli elementi raccolti, denominata, rispettivamente, Check o Compare.

5.2.1 Collect-And-Check

La finalità di Collect-And-Check è quella di raccogliere una lista di elementi dal documento e validarli, uno ad uno, sulla base di una regola di validazione. La parte Collect, ovvero la prima fase dell'applicazione della regola, comporta la raccolta di una lista di elementi, determinati in base al contenuto del tag <collect>. La lista che si ottiene è conforme al seguente XML Schema:

```
1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2   <xs:element name="items">
3     <xs:complexType>
4       <xs:sequence>
5         <xs:element maxOccurs="unbounded" name="item"/>
6       </xs:sequence>
7     </xs:complexType>
8   </xs:element>
9 </xs:schema>
```

Si prevede che la regola possa essere di quattro tipi:

- Regex: <collect type='Regex'>,
- XPath 1.0: <collect type='XPath 1'>,
- XSLT 1.0: <collect type='XSLT 1'>,

- XSLT 2.0: <collect type='XSLT 2'>.

Il tag successivo, <check>, contiene la regola di validazione degli elementi raccolti. Si prevede che la regola possa essere di tre tipi:

- Regex: <check type='Regex'>,
- XSLT 1.0: <check type='XSLT 1'>,
- XSLT 2.0: <check type='XSLT 2'>.

5.2.2 Collect-And-Compare

La parte <collect> è simile ma non identica alla <collect> di Collect-And-Check. Simile, in quanto anche qui si tratta di raccogliere elementi da analizzare nella seconda parte. Non identica, in quanto non si raccoglie una lista di elementi, bensì due o più liste di elementi, da confrontare attraverso la regola contenuta nel tag <compare>.
Segnatamente, l'output di questa <collect> è conforme al seguente XML Schema:

```
1  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2    <xs:element name="collect">
3      <xs:complexType>
4        <xs:sequence>
5          <xs:element maxOccurs="unbounded" name="items">
6            <xs:complexType>
7              <xs:attribute name="i" use="required" type="xs:integer"/>
8              <xs:attribute name="type" use="required" type="xs:string"/>
9            </xs:complexType>
10         </xs:element>
11       </xs:sequence>
12     </xs:complexType>
13   </xs:element>
14 </xs:schema>
```

La seconda parte della regola, denominata <compare>, effettua la comparazione tra gli elementi delle varie liste. Diversamente da <check>, il contenuto può essere solo di due tipi: XSLT 1.0 o XSLT 2.0. La ragione è la complessità maggiore di eseguire un raffronto, che trascende la capacità espressiva di Regex e XPath.
Potenzialmente le liste di elementi da comparare possono essere n, con $n > 1$.

5.2.3 Esempi di Collect-And-Check

Si vuole rappresentare il seguente controllo: dato un documento in formato docx, verificare che tutte le date ivi contenute abbiamo un determinato formato.
Tale controllo è scindibile in due fasi:

Collect: Estrarre tutte le date, secondo tutti possibili formati.

Check: Verificare che tutte le date estratte corrispondano ad un unico determianto formato.

Per entrambe le fasi è sufficiente usare una regular expression. Di conseguenza, una rappresentazione possibile della regola è la seguente:

```
1  <rule type='collect-and-check' target='document.xml'>
2      <name>Check Dates Format: dd/mm/yyyy</name>
3    <description>
4        Each date within the document has the format dd/mm/yyyy.
5    </description>
6      <collect-and-check>
7          <!--
8          d-m-yy | d-m-yyyy | dd-mm-yyyy | dd-mm-yy |
9          d.m.yy | d.m.yyyy | dd.mm.yyyy | dd.mm.yy |
10         d/m/yy | d/m/yyyy | dd/mm/yyyy | dd/mm/yy
11         -->
12         <collect type='Regex'>
13             (0?[1-9]|[12][0-9]|3[01])([-/.])(0?[1-9]|1[012])\2(19|20)?\d\d
14         </collect>
```

```
15          <!-- dd/mm/yyyy -->
16          <check type='Regex'>
17              (0[1-9]|[12][0-9]|3[01])(\/)(0[1-9]|1[012])(\/)(19|20)\d\d
18          </check>
19      </collect-and-check>
20  </rule>
```

Un esempio più articolato è il seguente: dato un documento in formato docx, verificare che tutte le tabelle ivi contenute abbiano, nel paragrafo successivo, una descrizione e che questa descrizione rispecchi un determinato formato sintattico.
Tale regola è scindibile nelle seguenti due fasi:

Collect: Estrarre tutti i paragrafi successivi a tutte le tabelle contenute nel documento.

Check: Verificare che il contenuto dei paragrafi estratti rispecchi un determinato formato sintattico.

La capacità espressiva di una regular expression è insufficiente per catturare questa regola. Si utilizza quindi XSLT 1.0.
Una possibile soluzione è la seguente:

```
1   <rule type='collect-and-check' target='document.xml'>
2       <name>Check Tables Description</name>
3     <description>
4         Each table has a well-formed description: (Tabella \d) [\w\s]+
5       </description>
6       <collect-and-check>
7           <collect type='XSLT 1'>
8               <xsl:stylesheet version="1.0">
9                   <xsl:output
10                      method="xml"
11                      encoding="UTF-8"
12                      indent="yes"
```

```
13                  omit-xml-declaration="yes"/>
14              <xsl:template match="/">
15              <items>
16                  <xsl:for-each select="//w:tbl">
17                  <item>
18                      <xsl:attribute name="i">
19                          <xsl:value-of select="position()"/>
20                      </xsl:attribute>
21                      <xsl:apply-templates
22                          select="./following-sibling::w:p[1]//w:t"/>
23                  </item>
24                  </xsl:for-each>
25              </items>
26              </xsl:template>
27              <xsl:template match="w:t">
28                  <xsl:value-of select="."/>
29              </xsl:template>
30          </xsl:stylesheet>
31      </collect>
32      <check type='Regex'>(Tabella \d) [\w\s]+</check>
33  </collect-and-check>
34 </rule>
```

Come si può notare, la Collect scritta in XSLT genera direttamente la lista di <item>. Al contrario, usando Regex questo non è ovviamente possibile. Di conseguenza, è *hard-coded* nell'applicazione. Ovvero, se la Collect è di tipo Regex, la regular expression viene eseguita e per ogni istanza conforme viene creato un elemento <item>. In tutti i casi quindi l'output finale ha la medesima struttura.

5.2.4 Esempio di Collect-And-Compare

Si pensi al seguente controllo: dato un documento in formato docx, contenente una relazione tecnica, utilizzante acronimi e una relativa tabella degli

acronimi esplicativa, verificare che ogni acronimo presente nel testo è definito nella tabella, e per ogni acronimo definito nella tabella esiste almeno una sua istanza nel testo.
Si possono distinguere due fasi:

Collect: Preparare due liste di elementi. La prima deve contenere tutti gli acronimi, contenuti nel documento, ad eccezione di quelli contenuti nella tabella esplicativa. La seconda deve contenere tutti e soltanto gli acronimi contenuti nella tabella esplicativa.

Compare: Verificare che ogni elemento contenuto nella prima lista compare anche nella seconda e che ogni elemento contenuto nella seconda lista compare anche nella prima.

La complessità di questo controllo richiede XSLT versione 1.0 e 2.0. Occorre infatti usare una regular expression all'interno di XSLT, tramite XPath 2.0. Una possibile rappresentazione della fase Collect è la seguente:

- Estrarre tutti gli acronimi presenti nel testo, ad eccezione di quelli nella tabella acronimi.

```
<items i='1' type='XSLT 1'>
<xsl:stylesheet>
 <xsl:output
  method="xml"
  encoding="UTF-8"
  indent="yes"
  omit-xml-declaration="yes"/>
 <xsl:template match="/">
 <items id='1'>
  <xsl:for-each
   select="//w:tbl
   [
    w:tr[1]//w:tc[1]//w:t='Acronimo' and
    w:tr[1]//w:tc[2]//w:t='Descrizione'
   ]
```

```
16    /w:tr[position()>1]/w:tc[1]//w:t/text()">
17    <xsl:sort select="."/>
18    <item>
19     <xsl:value-of select="."/>
20    </item>
21   </xsl:for-each>
22  </items>
23  </xsl:template>
24 </xsl:stylesheet>
25 </items>
```

- Estrarre tutti gli acronimi presenti nella tabella acronimi.

```
1  <items i='2' type='XSLT 2'>
2  <xsl:stylesheet>
3   <xsl:output
4    method="xml"
5    encoding="UTF-8"
6    indent="yes"
7    omit-xml-declaration="yes"/>
8   <xsl:strip-space elements="*"/>
9
10  <xsl:variable name="allAcronyms">
11   <xsl:for-each select="//w:t
12    [not(ancestor::w:tbl
13     [
14      w:tr[1]//w:tc[1]//w:t='Acronimo' and
15      w:tr[1]//w:tc[2]//w:t='Descrizione'
16     ]
17    )]">
18    <xsl:analyze-string select="." regex="([A-Z]{{2,}})">
19     <xsl:matching-substring>
20      <xsl:value-of select="concat(regex-group(1), ' ')"/>
21     </xsl:matching-substring>
22    </xsl:analyze-string>
```

```
23    </xsl:for-each>
24   </xsl:variable>
25
26   <xsl:variable name="distinctAcronyms"
27    select="
28     distinct-values(tokenize(normalize-space(\$allAcronyms), ' '))"/>
29
30   <xsl:template match="/">
31    <items id='2'>
32     <xsl:for-each select="\$distinctAcronyms">
33      <xsl:sort select="."/>
34      <item>
35       <xsl:value-of select="."/>
36      </item>
37     </xsl:for-each>
38    </items>
39   </xsl:template>
40  </xsl:stylesheet>
41  </items>
```

La fase di comparazione richiede due controlli incrociati: ogni elemento presente nella lista 1 deve essere contenuto anche nella lista 2, e viceversa. Questa comparazione è rappresentabile mediante XSLT 1.0, come segue:

```
1   <compare type='XSLT 1'>
2     <xsl:stylesheet>
3      <xsl:template match="/">
4      <results>
5       <!--
6       Every acronym in the document must be defined in the table of
       acronyms.
7       -->
8       <xsl:for-each select="//items[@id='1']/item">
9        <xsl:call-template name="document-to-table"/>
10      </xsl:for-each>
```

```
11      <!--
12      Every acronym within the table of acronyms must be used at least once
13      in the document.
14      -->
15      <xsl:for-each select="//items[@id='2']/item">
16       <xsl:call-template name="table-to-document"/>
17      </xsl:for-each>
18     </results>
19     </xsl:template>
20     <xsl:template name="document-to-table">
21      <item>
22       <xsl:variable name="passed" select="//items[@id='2']/item = ." />
23       <xsl:attribute name="passed">
24        <xsl:value-of select="\$passed"/>
25       </xsl:attribute>
26       <description>
27        <xsl:choose>
28         <xsl:when test="\$passed"
29          The acronym is both in the document and in the table
30         </xsl:when>
31         <xsl:otherwise>
32          The acronym is only in the document
33         </xsl:otherwise>
34        </xsl:choose>
35       </description>
36       <value><xsl:value-of select="."/></value>
37      </item>
38     </xsl:template>
39     <xsl:template name="table-to-document">
40      <xsl:if test="not(//items[@id='1']/item = .)">
41       <item>
42        <xsl:attribute name="passed">
43         <xsl:value-of select="false()"/>
44        </xsl:attribute>
45        <description>The acronym is only in the table.</description>
46        <value><xsl:value-of select="."/></value>
47       </item>
```

```
48      </xsl:if>
49     </xsl:template>
50    </xsl:stylesheet>
51   </compare>
```

5.3 Sfruttare la natura asincrona di Node.js

La programmazione asincrona è un paradigma di programmazione che consente di realizzare codice non-bloccante, ovvero codice la cui esecuzione non dipende dal completamento di processi esterni (in particolare processi di I/O).

Node.js è stato sviluppato a partire dal motore V8 di Google, il quale compila Javascript in codice macchina. Come è noto, Javascript è un linguaggio che consente nativamente la programmazione asincrona. Inolte, Node.js è *single-threaded*, ovvero astrae la complessità di dovere gestire direttamente thread, con i relativi problemi di sincronizzazione.

Nell'implementazione del Validation Engine si è reso non solo opportuno, ma necessario, scrivere un codice asincrono. In particolare, per la funzione di applicazione delle regole:

```
1  // Validate the document against the selected rules.
2  self.fireRules = function(rules, response, next)
3  {
4      ValidationResults = [];
5
6      async.forEach(Documents, function(document, callbackDocument)
7      {
8          async.forEach(rules, function(rule, callbackRule)
9          {
10             self.validate(rule, document, callbackRule);
11         },
12         function(err)
```

```
13          {
14              if (err) return next(err);
15              callbackDocument();
16          });
17      },
18      function(err)
19      {
20          if (err) return next(err);
21          prepareResponse();
22          response.send(
23          {
24              'AnalyticalData' : AnalyticalData,
25              'SyntheticalData' : SyntheticalData
26          });
27      });
28  }
```

La funzione fireRules presiede all'esecuzione di tutte le regole selezionate contro tutti i documenti caricati. Richiede quindi una doppia iterazione, a livello di documento e a livello di regola.
Il problema principale riscontrato riguarda l'applicazione di regole contenenti query in XLST 2.0. Non esiste, al momento, una libreria per Node.js in grado di processare un query scritta secondo la sintassi di XLST 2.0. Di conseguenza, una delle poche, se non l'unica opzione disponibile è stata quella di utilizzare la libreria Saxon 9 Home Edition (HE), scritta in Java. A tal fine esiste una libreria Node.js amatoriale (il nome è saxon-stream, scaricabile e installabile tramite npm) che consente di eseguire Saxon tramite *stream*.
In Node.js, come in altri linguaggi, il concetto di *stream* è collegato al concetto di *pipeline*, nato in ambito Unix. Una *pipeline* è un insieme di processi incatenati in modo tale che l'output del processo precedente nutre direttamente l'input del processo successivo. Gli *stream* sono dunque disegnati per rendere più efficiente l'esecuzione di operazioni I/O in Node.js, incoraggiando

l'impiego di uno stile di programmazione asincrono. Possono essere *readable*, *writeable*, o entrambi simultaneamente. I *readable stream* emettono un evento per ogni porzione (*chunk*) di dati letti, ed emettono un evento finale quando il dato è stato letto completamente. Tutti gli *stream* emettono inoltre specifici eventi per segnalare l'occorrenza di eventuali errori.
L'impiego di *stream* per questa fattispecie di regola ha reso, quindi, necessario il *refactoring* asincrono di tutto il codice relativo alla funzione madre (fireRules). In Node.js esiste una libreria *ad hoc* per la programmazione asincrona tramite *callbacks*, chiamata async.

6

Conclusioni e prosecuzione del lavoro

Il Validation Engine richiede ancora l'implementazione di alcuni punti importanti, segnatamente il controllo inter-documentale e l'inclusione di file xlsx. A Settembre 2016 inizierà ufficialmente l'attività di sviluppo degli altri due prodotti software, in particolare dello Smart Structerd Editor, indubbiamente il prodotto di maggiore complessità. Questa tesi ha ripercorso gli aspetti fondativi di questo strumento: la definizione di un opportuno formato XML e l'editor online collaborativo. Esiste già una pianificazione del processo di produzione software che condurrà, in un anno, alla disponibilità dei tre strumenti, sebbene non ancora dotati di tutte le feature previste. Il deployment definitivo del progetto richiederà due anni di lavoro a partire dal tempo 0 di inizio, grazie ad un team di sviluppo che sarà formato da 6-7 persone.

Bibliografia

[1] Shu Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Xml document versioning. *SIGMOD Rec.*, 30(3):46–53, September 2001.

[2] Alex Dekhtyar and Ionut E. Iacob. *Conceptual Modeling for Novel Application Domains: ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM, Chicago, IL, USA, October 13, 2003. Proceedings*, chapter A Framework for Management of Concurrent XML Markup, pages 311–322. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[3] Angelo Di Iorio, Silvio Peroni, and Fabio Vitali. A semantic web approach to everyday overlapping markup. *Journal of the American Society for Information Science and Technology*, 2011.

[4] Manolis Gergatsoulis and Yannis Stavrakas. *Database and XML Technologies: First International XML Database Symposium, XSym 2003, Berlin, Germany, September 8, 2003, Proceedings*, chapter Representing Changes in XML Documents Using Dimensions, pages 208–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[5] Santosh Kumawat, M. Tech Scholar, and Ajay Khunteta. A survey on operational transformation algorithms: Challenges, issues and achievements 1, 2010.

[6] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[7] Clarence Leung. Operational transformation in cooperative software systems. *McGill Science Undergraduate Research Journal*, 8(1), 2013.

[8] Yves Marcoux, Michael Sperberg-McQueen, and Claus Huitfeldt. Modeling overlapping structures: Graphs and serializability. In *The Markup Conference 2013. Balisage Series on Markup Technologies*, 2013.

[9] Haifeng Shen and Yongyao Yan. Optimistic and efficient concurrency control for asynchronous collaborative systems. In *Proceedings of the Thirty-Fourth Australasian Computer Science Conference - Volume 113*, ACSC '11, pages 73–82, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc.

[10] Zach Smith. Overview of operational transformation. In *Proceedings of the UMM CSci Senior Seminar Conference*, 2012.

[11] C. M. Sperberg-McQueen and Claus Huitfeldt. *Digital Documents: Systems and Principles: 8th International Conference on Digital Documents and Electronic Publishing, DDEP 2000, 5th International Workshop on the Principles of Digital Document Processing, PODDP 2000, Munich, Germany, September 13-15, 2000. Revised Papers*, chapter GODDAG: A Data Structure for Overlapping Hierarchies, pages 139–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[12] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

[13] Fabio Vitali, Paolo Marinelli, and Stefano Zacchiroli. Towards the unification of formats for overlapping markup. *New Review of Hypermedia and Multimedia*, (14):57–94, 2008.

[14] Fusheng Wang and Carlo Zaniolo. *Conceptual Modeling – ER 2004: 23rd International Conference on Conceptual Modeling, Shanghai, China, November 8-12, 2004. Proceedings*, chapter XBiT: An XML-Based

Bitemporal Data Model, pages 810–824. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[15] Fusheng Wang and Carlo Zaniolo. Temporal queries and version management in xml-based document archives. *Data Knowl. Eng.*, 65(2):304–324, May 2008.

[16] Yi Xu and Chengzheng Sun. Conditions and patterns for achieving convergence in ot-based co-editors. *IEEE Trans. Parallel Distrib. Syst.*, 27(3):695–709, March 2016.