Semantica assiomatica di `docxwatermarker`

Fabio F.G. Buono

2026

Versione 2

1 Introduzione

Un file `.docx` è un archivio ZIP, e fra le entry dell'archivio compaiono, sotto `word/media/`, i byte delle immagini ancorate al documento. Sostituire una di queste immagini con altri byte e riscrivere l'archivio produce un documento Word in cui l'immagine appare diversa e il resto è rimasto al suo posto. La libreria `docxwatermarker` fa questo lavoro, e serve a chi prepara un modello in Word e vuole generare una copia per ogni destinatario, ad esempio per marchiare documenti confidenziali.

Queste note danno una specifica assiomatica della libreria. Coprono il lato della libreria di una specifica in due parti, la cui altra metà, una semantica operazionale dello strumento da riga di comando, è il documento gemello *Semantica operazionale dello strumento da riga di comando* `docxwatermarker`. Per ciascuna operazione pubblica enunciano una tripla di Hoare [1] che ne caratterizza il comportamento, e dalle triple ricavano poi le proprietà che valgono attraverso più operazioni. La libreria diventa, sotto questa lente, una collezione di contratti su byte, archivi, immagini e metadati ZIP. La notazione segue gli appunti di Mancarella [5], con i riferimenti standard a Winskel [3] per i fondamenti e a Meyer [4] per la connessione con i contratti eseguibili che il codice porta già nelle sue `require` e `ensure`.

Nei testi citati le triple specificano costrutti di un linguaggio di programmazione, come l'assegnamento o il while. Qui ogni tripla specifica una chiamata di funzione, e il linguaggio cliente, Python, viene trattato come oracolo.

2 Come leggere queste note

Il formalismo delle triple di Hoare si applica di solito a programmi di un certo rilievo, come sistemi critici, controllori, primitive di sincronizzazione, o, nei testi didattici, a frammenti di codice piccoli e artificiali. Una libreria di personalizzazione di documenti Word sta fuori da entrambe le categorie. Il codice dichiara già un contratto parziale tramite le sue `require` e `ensure` interne, e renderlo esplicito al livello matematico chiude un cerchio che il codice apre soltanto a metà, perché ogni asserzione runtime trova qui il suo enunciato formale, e ogni proprietà enunciata qui ha (o dovrebbe avere) la sua `require` o `ensure` nel codice. Scrivere la specifica obbliga inoltre a porre domande sul comportamento della libreria che leggere il codice non costringe a porsi, e quattro correzioni nella versione 0.1 sono uscite da questo esercizio.

La notazione $\{P\}\, c\, \{Q\}$ va letta in modo canonico. Se prima della chiamata c vale l'asserzione P sullo stato delle variabili del programma cliente, dopo la chiamata vale l'asserzione Q, nella quale y denota il valore appena restituito e le variabili in maiuscolo come X denotano i valori che le rispettive variabili minuscole avevano in entrata. Le triple sono in correttezza

totale, e includono la terminazione di c. Le eccezioni delle operazioni vengono trattate accanto alla tripla principale, indicando le condizioni che provocano ciascuna eccezione.

Le variabili in minuscolo come x, y, t, p sono variabili del programma cliente, e quelle in maiuscolo come X, Y, T, P sono variabili di specifica, che fissano i valori in entrata per poterli riferire nella postcondizione, secondo la convenzione del Paragrafo 6 di [5]. Gli identificatori in carattere a spaziatura fissa come `Template.open` o `replace_image` sono i nomi delle classi e dei metodi nel codice Python, mentre gli identificatori in *matematico* come *detect*, *img*, *inMedia* sono nomi delle funzioni matematiche che modellano le operazioni interne della libreria.

3 Preliminari

Gli insiemi che useremo sono trattati nella nozione intuitiva di collezione, fuori da qualsiasi teoria formale degli insiemi.

Indichiamo con Bytes l'insieme delle sequenze finite di ottetti, ovvero $\mathsf{Bytes} = \{0, 1, \ldots, 255\}^*$. Per $b \in \mathsf{Bytes}$, $|b|$ denota la lunghezza in ottetti e $b_1 \cdot b_2$ la concatenazione.

Indichiamo con Name l'insieme delle stringhe usate come nomi di entry interne in un archivio ZIP, e sull'insieme Name non assumiamo struttura aggiuntiva oltre all'uguaglianza e alla funzione $\mathit{lower} : \mathsf{Name} \to \mathsf{Name}$ che restituisce la forma minuscola di un nome.

Indichiamo con Format l'insieme dei formati di immagine riconosciuti dalla libreria, ovvero

$$\mathsf{Format} = \{\mathrm{png}, \mathrm{jpeg}, \mathrm{gif}, \mathrm{bmp}, \mathrm{tiff}, \mathrm{webp}, \bot\}.$$

Il valore $\bot$ rappresenta "formato non riconosciuto".

Entry e archivi

Una entry di un archivio ZIP è una coppia composta dai byte del contenuto e da una collezione di metadati (timestamp, metodo di compressione, permessi POSIX, attributi vari). La struttura interna dei metadati non interviene nelle triple, e denotiamo con *Meta* il loro dominio trattato come opaco. Le entry prodotte da una operazione di modifica della libreria sono distinte dalle entry originali tramite un *flag di modifica* che assume i valori tt o ff.

Definizione 3.1. Entry è l'insieme delle quadruple $e = (n, d, m, \mu)$ con $n \in \mathsf{Name}$, $d \in \mathsf{Bytes}$, $m \in \mathit{Meta}$, $\mu \in \{\mathrm{tt}, \mathrm{ff}\}$, e indichiamo le componenti rispettivamente con $e.\mathit{name}$, $e.\mathit{data}$, $e.\mathit{meta}$, $e.\mathit{mod}$.

Definizione 3.2. Un archivio è una funzione parziale $A : \mathsf{Name} \rightharpoonup \mathsf{Entry}$, con dominio finito, tale che per ogni $n \in \mathit{dom}(A)$ valga $A(n).\mathit{name} = n$, e indichiamo con $\mathsf{Archive}$ l'insieme degli archivi.

L'ordine delle entry, pur essendo rilevante in pratica (la libreria si impegna a preservarlo), non interviene mai nelle triple, sicché lo lasciamo implicito nel termine "funzione parziale a dominio finito".

Sull'archivio definiamo l'operazione di aggiornamento. Dati $A \in \mathsf{Archive}$, $n \in \mathit{dom}(A)$ ed $e \in \mathsf{Entry}$ con $e.\mathit{name} = n$, scriviamo $A[e/n]$ per l'archivio definito da

$$A[e/n](n') = \begin{cases} e & \text{se } n' = n; \\ A(n') & \text{altrimenti,} \end{cases}$$

in totale analogia con l'aggiornamento di funzione di Mancarella [5, §2].

Immagini

Una immagine è la vista logica di una entry il cui contenuto è riconosciuto come uno dei formati di $\mathsf{Format} \setminus \{\bot\}$. Oltre al nome e al formato, di una immagine ci interessano larghezza, altezza, dimensione in byte e un flag che indica se l'immagine porta il *marker* prodotto da `make_marker_png` (il meccanismo che permette al matcher `by_marker` di individuare l'immagine giusta in un template).

Definizione 3.3. Image è l'insieme delle sestuple $i = (n,\, f,\, w,\, h,\, s,\, k)$ con $n \in \mathsf{Name}$, $f \in \mathsf{Format} \setminus \{\bot\}$, $w, h, s \in \mathbb{N}$, $k \in \{\mathrm{tt}, \mathrm{ff}\}$, soggette al vincolo

$$k = \mathrm{tt} \implies f = \mathrm{png}, \tag{1}$$

con le componenti accessibili come $i.name$, $i.f$, $i.w$, $i.h$, $i.s$, $i.mark$.

Il vincolo (1) riflette il fatto che il meccanismo di marker è basato su *chunk* testuali `tEXt`, che esistono solo nel formato PNG.

Le funzioni totali che useremo nel seguito hanno la signature

$$\begin{aligned} \mathit{detect} &: \mathsf{Bytes} \to \mathsf{Format}, \\ \mathit{dim} &: \mathsf{Bytes} \to \mathbb{N} \times \mathbb{N}, \\ \mathit{isMarker} &: \mathsf{Bytes} \to \{\mathrm{tt}, \mathrm{ff}\}, \\ \mathit{inMedia} &: \mathsf{Name} \to \{\mathrm{tt}, \mathrm{ff}\}. \end{aligned}$$

La *detect* identifica il formato di una sequenza di byte dai *magic bytes* iniziali e ritorna $\bot$ se nessuna firma è riconosciuta. La *dim* restituisce le dimensioni in pixel, o $(0, 0)$ se i byte non sono decodificabili come immagine. La *isMarker* ritorna tt se e solo se i primi 512 byte cominciano con la firma PNG e contengono il tag di marker. La *inMedia* è il predicato che vale tt sse $\mathit{lower}(n)$ inizia con `"word/media/"`.

Data una entry e con $\mathit{detect}(e.data) \neq \bot$, poniamo

$$\mathit{img}(e) = (\, e.name,\ \mathit{detect}(e.data),\ \mathit{dim}(e.data)_1,\ \mathit{dim}(e.data)_2,\ |e.data|,\ \mathit{isMarker}(e.data)\,).$$

La buona definizione di *img* rispetto al vincolo (1) è immediata, dal momento che *isMarker* ritorna tt solo per byte che cominciano con la firma PNG, sui quali *detect* ritorna png.

Template

Un template è un archivio, e poniamo $\mathsf{Template} = \mathsf{Archive}$. Nell'uso concreto un template è un archivio dotato di alcune entry obbligatorie (`[Content_Types].xml`, `word/document.xml`, e così via), ma tali requisiti non intervengono nelle triple della libreria.

Su un template è definita la funzione $\mathit{images} : \mathsf{Template} \to \mathsf{Image}^*$ che enumera le immagini secondo la formula

$$\mathit{images}(t) = \big[\, \mathit{img}(t(n)) \;\big|\; n \in \mathit{dom}(t),\ \mathit{inMedia}(n),\ \mathit{detect}(t(n).data) \neq \bot \,\big],$$

dove la notazione $[\cdot \mid \cdot]$ denota una sequenza costruita per filtro (l'ordine concreto è quello di iterazione del dominio dell'archivio, e nessuna delle nostre proprietà dipende da questo ordine).

I matcher, che la libreria espone come strategie per selezionare una immagine all'interno di un template, sono funzioni parziali $\mu : \mathsf{Image}^* \rightharpoonup \mathsf{Name}$. I tre matcher concreti μ^{p}_{fname}, μ_{mark}, μ_{auto} saranno definiti in §5.5, dove servono come parametri delle triple di `replace_image`.

4 Stati e asserzioni

Il programma cliente che usa la libreria è scritto in Python e manipola variabili ordinarie. Per ragionare sul comportamento delle operazioni della libreria astraiamo dal linguaggio cliente e ne consideriamo solo l'effetto sullo stato delle sue variabili.

Sia Var un insieme di identificatori di variabili Python. Indichiamo con

$$\mathsf{Val} = \mathsf{Template} \cup \mathsf{Image}^* \cup \mathsf{Bytes} \cup \mathsf{Name} \cup \mathbb{Z} \cup \{\mathrm{tt}, \mathrm{ff}\}$$

l'insieme dei valori che le variabili possono assumere nei nostri ragionamenti. Uno *stato* è una funzione

$$\sigma : \mathsf{Var} \rightharpoonup \mathsf{Val}$$

che associa a ciascuna variabile assegnata il proprio valore, e resta indefinita sulle variabili non ancora assegnate. Indichiamo con State l'insieme di tutti gli stati.

L'aggiornamento di stato è definito da

$$\sigma[v/x](y) = \begin{cases} v & \text{se } y = x, \\ \sigma(y) & \text{altrimenti,} \end{cases}$$

ovvero come l'operatore di aggiornamento di funzione del Paragrafo 3.

Una *asserzione* è una formula del calcolo dei predicati del prim'ordine sulle variabili in Var. Useremo le lettere $P, Q, R, \ldots$ per asserzioni. Le costanti utilizzate nelle asserzioni sono quelle dei domini introdotti al Paragrafo 3, e termini come $t.images$ (dove $t \in \mathsf{Var}$) sono espressioni il cui valore dipende dallo stato. Estendiamo agli oggetti del nostro dominio le notazioni infisse usuali, ossia $\in$ per appartenenza a sequenza, $=$ per uguaglianza, $|s|$ per la lunghezza di una sequenza.

Indichiamo con $\sigma \models P$ il fatto che P è vera nello stato σ, nella definizione standard induttiva sulla struttura di P. Useremo anche le variabili di specifica (lettere maiuscole $X, Y, A, B, \ldots$) per fissare valori generici su cui parametrizzare pre- e postcondizioni, secondo la convenzione del Paragrafo 6 di [5].

Triple di Hoare per le operazioni della libreria

Una *tripla di Hoare* $\{P\}\, c\, \{Q\}$ è soddisfatta se, per ogni stato σ tale che $\sigma \models P$, l'esecuzione di c a partire da σ termina in uno stato σ' tale che $\sigma' \models Q$. Nel nostro contesto, c è sempre una chiamata di funzione della libreria, della forma

$$y \;\leftarrow\; op(\bar{a}),$$

dove op è una operazione, $\bar{a}$ è la tupla degli argomenti e $y \in \mathsf{Var}$ riceve il valore di ritorno, se l'operazione lo produce. Nella postcondizione Q il simbolo y denota il valore appena restituito. Le variabili che denotano gli argomenti possono essere annotate con la convenzione delle *variabili di specifica*, scrivendo X in Q per indicare il valore che la variabile x aveva in σ, prima della chiamata.

Una tripla può essere indebolita o rafforzata secondo la regola classica

$$\frac{P \Rightarrow P' \qquad \{P'\}\, c\, \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\, c\, \{Q\}} \quad (\textit{pre-post})$$

che useremo come strumento ausiliario nelle dimostrazioni del Paragrafo 6.

Esempio 4.1. Consideriamo la tripla

$$\begin{gathered}\{\, p \in \mathsf{Bytes} \wedge p \text{ è uno ZIP valido} \,\} \\ t \leftarrow \texttt{Template.from\_bytes}(p) \\ \{\, t \in \mathsf{Template} \wedge \mathit{toBytes}(t) = p \,\}.\end{gathered}$$

La precondizione enuncia che p sono byte che costituiscono un archivio ZIP valido, dove il predicato "p è uno ZIP valido" è quello concretizzato dalla libreria `zipfile` di Python (non lo definiamo qui). La postcondizione afferma che la variabile t, dopo la chiamata, contiene un template e che la serializzazione di tale template attraverso *toBytes* restituisce gli stessi byte di input. Il fatto che il serializzatore di Python preservi i byte di un archivio ZIP letto e riscritto senza modifiche (a meno di dettagli che discuteremo nel paragrafo dedicato all'idempotenza) è la ragione per cui la postcondizione include la congiunzione $\mathit{toBytes}(t) = p$ oltre alla tipizzazione $t \in \mathsf{Template}$.

5 Specifiche delle operazioni

Le triple delle operazioni della libreria condividono una convenzione sulle eccezioni, che la libreria modula intorno alla gerarchia

$$\begin{aligned}\mathit{DocxWatermarkerError} \supseteq \{&\mathit{ImageNotFoundError}, \mathit{MultipleImagesError}, \\ &\mathit{FormatMismatchError}, \mathit{PDFConversionError}, \mathit{InvariantError}\},\end{aligned}$$

definita nel modulo `errors`. La sintassi che useremo per indicare che la chiamata c, eseguita in uno stato che soddisfa P, può terminare in uno stato che soddisfa Q oppure sollevare una delle eccezioni E_i è

$$\{P\}\ c\ \{Q\} \ \mid\ E_1 \text{ se } R_1,\ \ldots,\ E_n \text{ se } R_n,$$

con la convenzione che E_i viene sollevata in corrispondenza della prima condizione R_i verificata.

5.1 Costruttori di template

L'operazione `Template.from_bytes`(p) costruisce un template a partire da byte raw, mentre `Template.open`(ℓ) fa lo stesso a partire da un percorso del filesystem.

`Template.from_bytes`.

$$\begin{gathered}\{\, p \in \mathsf{Bytes} \wedge \mathit{isValidZip}(p) \,\} \\ t \leftarrow \texttt{Template.from\_bytes}(p) \\ \{\, t \in \mathsf{Template} \wedge \mathit{toBytes}(t) \approx_{\text{sem}} p \,\}\end{gathered}$$

Il predicato $\mathit{isValidZip} : \mathsf{Bytes} \to \{\text{tt}, \text{ff}\}$ è quello concretizzato da `zipfile.ZipFile`, e la relazione $\approx_{\text{sem}}$ è l'*equivalenza semantica* fra byte di archivi ZIP, definita scrivendo $b_1 \approx_{\text{sem}} b_2$ se i due archivi serializzati hanno lo stesso insieme di nomi, gli stessi contenuti e gli stessi metadati su ciascuna entry, modulo eventuali differenze nei byte di padding del Local File Header calcolati dal serializzatore. La distinzione fra equivalenza *semantica* ($\approx_{\text{sem}}$, che useremo nel resto) ed equivalenza *fisica* (identità byte-per-byte, che vale solo con il flag `reproducible=True`) è discussa nel README del progetto sotto la voce *byte-perfectness*. Quando p non è uno ZIP valido il comportamento non è definito dalla specifica, e il codice solleva `BadZipFile`, una eccezione della libreria standard Python che vive fuori dalla gerarchia *DocxWatermarkerError*.

`Template.open`. L'apertura di un template dal filesystem è una composizione di lettura del file e `from_bytes`, con la sola differenza che il fallimento nella lettura del file viene tracciato come `FileNotFoundError`.

$$\{\, \ell \in \mathit{Path} \wedge \mathit{fileAt}(\ell) = p \wedge \mathit{isValidZip}(p) \,\}$$
$$t \leftarrow \texttt{Template.open}(\ell)$$
$$\{\, t \in \mathsf{Template} \wedge \mathit{toBytes}(t) \approx_{\text{sem}} p \,\} \;\mid\; \texttt{FileNotFoundError} \text{ se } \neg \mathit{exists}(\ell)$$

$\mathit{fileAt}(\ell)$ denota i byte del file di sistema all'indirizzo ℓ, e $\mathit{exists}(\ell)$ il predicato di esistenza del file.

5.2 Ispezione di un template

`Template.list_images`. Restituisce la sequenza delle immagini contenute in un template, secondo la funzione *images* del Paragrafo 3.

$$\{\, t \in \mathsf{Template} \,\}$$
$$\bar{\imath} \leftarrow t.\texttt{list\_images}()$$
$$\{\, \bar{\imath} = \mathit{images}(T) \wedge t = T \,\}$$

La congiunzione $t = T$ in postcondizione verbalizza l'immutabilità del template, e tornerà nel Paragrafo 6 come proprietà dimostrata.

5.3 Sostituzione di una immagine

L'operazione $t.\texttt{replace\_image}(x, \mu, v)$ è il fulcro della libreria. Sostituisce, all'interno del template t, l'immagine selezionata dal matcher μ con i byte ottenuti da una normalizzazione dell'input x, e ritorna un nuovo template. Il parametro $v \in \{\mathrm{tt}, \mathrm{ff}\}$ è il flag *validate* che attiva la validazione del decode dell'immagine.

La tripla impiega due funzioni ausiliarie. La normalizzazione $\mathit{normalize} : \mathsf{Val} \rightharpoonup \mathsf{Bytes}$ è definita sull'unione di Bytes, *Path* e *PILImage* (l'insieme delle immagini Pillow), e restituisce i byte di input se questi sono già Bytes, i byte del file se è un *Path*, i byte di una codifica PNG se è una immagine Pillow. La decodifica $\mathit{canDecode} : \mathsf{Bytes} \to \{\mathrm{tt}, \mathrm{ff}\}$ ritorna tt se Pillow apre e verifica i byte come immagine valida, e ff altrimenti.

`Template.replace_image`.

$$\left\{ \begin{array}{l} t \in \mathsf{Template} \wedge t = T \wedge \mu \in \mathit{Matcher} \wedge x \in \mathit{dom}(\mathit{normalize}) \\ \wedge\ \mu(\mathit{images}(t)) = n \wedge \mathit{detect}(\mathit{normalize}(x)) = T(n).f \\ \wedge\ (v = \mathrm{tt} \Rightarrow \mathit{canDecode}(\mathit{normalize}(x))) \end{array} \right\}$$
$$t' \leftarrow t.\texttt{replace\_image}(x, \mu, v)$$
$$\left\{ \begin{array}{l} t' \in \mathsf{Template} \wedge t' = T[e/n] \wedge t = T \\ \text{dove } e.\mathit{data} = \mathit{normalize}(x),\ e.\mathit{name} = n,\ e.\mathit{meta} = T(n).\mathit{meta},\ e.\mathit{mod} = \mathrm{tt} \end{array} \right\}$$

La chiamata solleva

- `ImageNotFoundError` se $\mu(\mathit{images}(t))$ è indefinita per assenza di candidati;
- `MultipleImagesError` se $\mu(\mathit{images}(t))$ è indefinita per ambiguità (più di un candidato);
- `FormatMismatchError` se $\mathit{detect}(\mathit{normalize}(x)) \neq T(n).f$, ovvero formato del nuovo input diverso da quello del target;

- `FormatMismatchError` se $v = \mathrm{tt}$ e $\neg \mathit{canDecode}(\mathit{normalize}(x))$, ovvero formato dichiarato corretto ma byte non decodificabili.

La precondizione richiede $t = T$, con T variabile di specifica, perché la postcondizione asserisce a sua volta $t = T$ e ha bisogno di un nome per il valore di t in entrata. Il template restituito è $T[e/n]$, ovvero l'aggiornamento di T nel solo nome n con la nuova entry e, e le altre entry restano invariate (la *frame* esplicita). L'entry e eredita il metadato $T(n).\mathit{meta}$, che comprende metodo di compressione, permessi e attributi vari, e cambiano solo $e.\mathit{data}$ e il flag $e.\mathit{mod} = \mathrm{tt}$. La condizione $v = \mathrm{tt} \Rightarrow \mathit{canDecode}(\mathit{normalize}(x))$ è una precondizione che si attiva solo se il chiamante richiede la validazione, e riflette la scelta del codice di rendere la validazione opt-in con default $v = \mathrm{tt}$.

5.4 Serializzazione

Le operazioni $t.\texttt{to\_bytes}()$ e $t.\texttt{save}(\ell)$ producono i byte di un template, rispettivamente in memoria e su disco.

`Template.to_bytes`.

$$\{\, t \in \mathsf{Template} \wedge t = T \,\}$$
$$b \leftarrow t.\texttt{to\_bytes}()$$
$$\{\, b \in \mathsf{Bytes} \wedge \mathit{isValidZip}(b) \wedge \mathit{fromBytes}(b) = T \wedge t = T \,\}$$

dove *fromBytes* è la funzione astratta che inverte *toBytes* a meno di equivalenza semantica. Riapplicare la lettura ai byte prodotti restituisce T, e questa rotondità del round-trip $\mathit{toBytes} \circ \mathit{fromBytes} \approx \mathrm{id}$ sta dietro alla promessa di *byte-perfectness* che la libreria fa nel README.

`Template.save`.

$$\{\, t \in \mathsf{Template} \wedge t = T \wedge \ell \in \mathit{Path} \wedge \mathit{writable}(\ell) \,\}$$
$$\ell' \leftarrow t.\texttt{save}(\ell)$$
$$\{\, \ell' \in \mathit{Path} \wedge \mathit{fileAt}(\ell') = t.\texttt{to\_bytes}() \wedge t = T \,\} \mid \texttt{OSError} \text{ se } \neg \mathit{writable}(\ell)$$

$\mathit{writable}(\ell)$ è il predicato di scrivibilità sul filesystem all'indirizzo ℓ.

5.5 Matcher

I costruttori di `ImageMatcher` sono funzioni pure, e ciò che caratterizza ciascuno è la funzione parziale $\mu : \mathsf{Image}^* \rightharpoonup \mathsf{Name}$ che il matcher prodotto calcola.

`ImageMatcher.by_filename`(p).

$$\{\, p \in \mathsf{Name} \,\}$$
$$\mu \leftarrow \texttt{ImageMatcher.by\_filename}(p)$$
$$\{\, \mu \in \mathit{Matcher} \wedge \forall \bar{\imath} \in \mathsf{Image}^* \,.\, \mu(\bar{\imath}) = \begin{cases} i.\mathit{name} & \text{se } \exists!\, i \in \bar{\imath} \,.\, i.\mathit{name} = p, \\ \text{indef.} & \text{altrimenti} \end{cases} \,\}$$

`ImageMatcher.by_marker`().

$$\{\, \mathit{true} \,\}$$
$$\mu \leftarrow \texttt{ImageMatcher.by\_marker}()$$
$$\{\, \mu \in \mathit{Matcher} \wedge \forall \bar{\imath} \in \mathsf{Image}^* \,.\, \mu(\bar{\imath}) = \begin{cases} i.\mathit{name} & \text{se } \exists!\, i \in \bar{\imath} \,.\, i.\mathit{mark} = \mathrm{tt}, \\ \text{indef.} & \text{altrimenti} \end{cases} \,\}$$

`ImageMatcher.auto`(s). Combinator monoidico dei due precedenti, con un secondo stadio euristico parametrico in $s \in \mathbb{N}$, lato minimo del quadrato candidato.

$$\{\, s \in \mathbb{N} \,\}$$
$$\mu \leftarrow \texttt{ImageMatcher.auto}(s)$$
$$\{\, \mu \in \mathit{Matcher} \;\wedge\; \forall \bar{\imath}\,.\, \mu(\bar{\imath}) = \mu_{\mathrm{mark}}(\bar{\imath}) \;\text{ or else }\; \mu^{s}_{\mathrm{heur}}(\bar{\imath}) \,\}$$

dove *or else* è il combinator standard "prova il primo, e se indefinito prova il secondo", e

$$\mu^{s}_{\mathrm{heur}}(\bar{\imath}) = \begin{cases} i.\mathit{name} & \text{se } \exists!\, i \in \bar{\imath}\,.\, i.f = \mathrm{png} \wedge i.w = i.h \wedge i.w \geq s, \\ \text{indef.} & \text{altrimenti.} \end{cases}$$

5.6 Generazione di immagini

`make_marker_png` genera un placeholder PNG col tag di marker, e `make_text_watermark` una watermark di testo.

`make_marker_png`(s).

$$\{\, s \in \mathbb{N} \wedge s \geq 1 \,\}$$
$$d \leftarrow \texttt{make\_marker\_png}(s)$$
$$\{\, d \in \mathsf{Bytes} \wedge \mathit{detect}(d) = \mathrm{png} \wedge \mathit{isMarker}(d) = \mathrm{tt} \wedge \mathit{dim}(d) = (s, s) \,\}$$

`make_text_watermark`$(\bar{\lambda}, s, \rho, \chi, \ldots)$. La funzione produce una immagine PNG quadrata di lato s con le linee di testo $\bar{\lambda}$, ruotata di ρ gradi, di colore χ. Il dizionario dei preset, denotato $\mathit{PRESETS}$, è definito nel codice.

$$\left\{ \begin{array}{c} (\bar{\lambda} \in \mathit{String}^{*} \wedge \bar{\lambda} \neq \varepsilon \wedge \exists \lambda \in \bar{\lambda}\,.\, \mathit{trim}(\lambda) \neq \varepsilon) \vee \\ (\mathit{preset} \in \mathit{dom}(\mathit{PRESETS})) \wedge s \in \mathbb{N} \wedge s \geq 1 \end{array} \right\}$$
$$d \leftarrow \texttt{make\_text\_watermark}(\bar{\lambda}, s, \rho, \chi, \ldots)$$
$$\{\, d \in \mathsf{Bytes} \wedge \mathit{detect}(d) = \mathrm{png} \wedge \mathit{dim}(d) = (s, s) \,\}$$

| `ValueError` se entrambe le congiunzioni della precondizione falliscono

La specifica non caratterizza il contenuto pittorico dell'immagine prodotta, perché farlo richiederebbe un modello del rendering. La proprietà rilevante è che d sia un PNG valido di dimensione (s, s).

5.7 Conversione PDF

`find_libreoffice`(). Funzione pura che restituisce il percorso a `soffice` o $\perp$ se non lo trova.

$$\{\, \mathit{true} \,\}$$
$$r \leftarrow \texttt{find\_libreoffice}()$$
$$\{\, (r \in \mathit{Path} \wedge \mathit{executable}(r)) \;\vee\; r = \perp \,\}$$

`to_pdf`$(\ell_{src}, \ell_{dst}, \tau, \beta)$. La conversione DOCX→PDF avviene tramite LibreOffice in modalità headless. I parametri sono il path sorgente ℓ_{src}, il path destinazione ℓ_{dst} (eventualmente $\perp$ per derivare il nome dal sorgente), il timeout τ, e l'eventuale binary override β.

$$\left\{ \begin{array}{l} \ell_{src} \in \mathit{Path} \wedge \mathit{exists}(\ell_{src}) \wedge \tau > 0 \\ \wedge\; (\beta \neq \perp \Rightarrow \mathit{executable}(\beta)) \wedge (\beta = \perp \Rightarrow \texttt{find\_libreoffice}() \neq \perp) \end{array} \right\}$$
$$\ell' \leftarrow \texttt{to\_pdf}(\ell_{src}, \ell_{dst}, \tau, \beta)$$
$$\{\, \ell' \in \mathit{Path} \wedge \mathit{exists}(\ell') \wedge \mathit{isPDF}(\mathit{fileAt}(\ell')) \,\}$$

Le eccezioni che la funzione può sollevare sono tutte sottotipi di `PDFConversionError`, parametriche in un campo *reason* che ne identifica la causa.

- `PDFConversionError`(*reason* = not_found) se la precondizione su β o `find_libreoffice` fallisce;
- `PDFConversionError`(*reason* = cannot_create_outdir) se la directory contenente ℓ_{dst} non è creabile;
- `PDFConversionError`(*reason* = timeout) se la conversione supera τ secondi;
- `PDFConversionError`(*reason* = nonzero_exit) se `soffice` termina con stato diverso da zero;
- `PDFConversionError`(*reason* = no_output) se `soffice` termina correttamente ma il file PDF non viene prodotto.

La postcondizione $\mathit{isPDF}(\mathit{fileAt}(\ell'))$ è l'unica del documento che non discende dal codice della libreria, perché dipende dal corretto funzionamento di LibreOffice, che qui trattiamo come oracolo. Il codice si limita a garantire che ogni fallimento del sottoprocesso venga tracciato come una delle varianti di `PDFConversionError` sopra.

6 Invarianti di sistema

Le triple del paragrafo precedente parlano di una operazione per volta. Alcune proprietà però attraversano più operazioni. L'immutabilità del template è una di queste, e vale per tutte le operazioni che prendono in ingresso un template senza costruirlo da byte.

Proprietà 6.1. *Per ogni $t \in$ Template e per ogni operazione o in*

$$\{\texttt{list\_images}, \texttt{replace\_image}, \texttt{to\_bytes}, \texttt{save}\},$$

dopo l'esecuzione di o su t il valore di t è quello che aveva in entrata.

Dimostrazione. Le quattro triple del Paragrafo 5 contengono tutte $t = T$ in postcondizione, e T denota il valore di t in entrata. □

Nel codice la proprietà è garantita dal tipo, perché il costruttore di `Template` congela il dizionario con `types.MappingProxyType` (in `Template.__init__`), e `replace_image` costruisce un dizionario nuovo a partire dall'originale senza mutarlo. Un invariante correlato fissa il dominio del template sotto `replace_image`.

Proprietà 6.2. *Per ogni $t \in$ Template e per ogni invocazione legittima di* `replace_image`*, il template t' restituito soddisfa $\mathit{dom}(t') = \mathit{dom}(t)$.*

Dimostrazione. Dalla postcondizione di `replace_image`, $t' = T[e/n]$ con $n \in \mathit{dom}(T)$. Per definizione dell'aggiornamento di archivio, $\mathit{dom}(T[e/n]) = \mathit{dom}(T)$. Da $t = T$ segue la tesi. □

Una `ensure` (spec=`I2`) in `replace_image` controlla l'uguaglianza dei due nameset e solleva `InvariantError` se la proprietà viene violata. È l'unica `ensure` di struttura nella libreria, perché aggiungere o togliere una entry da un archivio OOXML romperebbe il documento. Una garanzia analoga vale sul formato delle immagini, dato che il file `[Content_Types].xml` dichiara il tipo MIME di ciascuna entry, e sostituire una PNG con una JPEG senza aggiornare quel file produrrebbe un documento illeggibile.

Proprietà 6.3. *Per ogni $t \in$ Template e per ogni invocazione legittima di* `replace_image`*(x, μ, v) con $\mu(\mathit{images}(t)) = n$, il template t' restituito soddisfa $\mathit{img}(t'(n)).f = \mathit{img}(t(n)).f$.*

Dimostrazione. La precondizione contiene $\mathit{detect}(\mathit{normalize}(x)) = T(n).f$. L'entry e in postcondizione ha $e.\mathit{data} = \mathit{normalize}(x)$, quindi $\mathit{detect}(e.\mathit{data}) = T(n).f$. Da $t'(n) = e$ segue $\mathit{img}(t'(n)).f = \mathit{img}(t(n)).f$. □

Il vincolo è in `replace_image`, prima che la sostituzione abbia luogo, con `FormatMismatchError` quando la precondizione fallisce. Il matcher `by_marker` ha una semantica univoca solo a patto che `make_marker_png` sia stata chiamata una volta sola sul template di interesse.

Proprietà 6.4. *Se in $\mathit{images}(t)$ c'è al più una immagine i con $i.\mathit{mark} = \mathrm{tt}$, allora μ_{mark} è definita su $\mathit{images}(t)$ se e solo se tale immagine esiste, e ne restituisce il nome.*

Dimostrazione. La definizione di $\mu_{\mathrm{mark}}(\bar{\imath})$ dà $i.\mathit{name}$ se esiste un'unica $i \in \bar{\imath}$ con $i.\mathit{mark} = \mathrm{tt}$, ed è indefinita altrimenti. L'ipotesi di unicità rimuove il caso "più di una". □

La libreria si astiene dal verificare la convenzione, perché ispezionare i metadati PNG di ogni immagine di ogni template avrebbe un costo difficile da giustificare, e l'eventuale violazione si manifesta come `MultipleImagesError` sollevato da `replace_image`. Lettura e scrittura di un template formano poi una coppia inversa, a meno del padding del Local File Header dello ZIP.

Proprietà 6.5. *Per ogni $t \in \mathsf{Template}$, $\mathit{fromBytes}(\mathit{toBytes}(t)) \approx_{\mathrm{sem}} t$.*

Dimostrazione. Dalla postcondizione di `Template.to_bytes`, $\mathit{fromBytes}(b) = T$ per i byte b restituiti. Dalla postcondizione di `Template.from_bytes`, $\mathit{toBytes}(t') \approx_{\mathrm{sem}} p$ per ogni t' ottenuto da byte p. Con $b = \mathit{toBytes}(T)$ e $t' = \mathit{fromBytes}(b)$ si ha $\mathit{toBytes}(t') \approx_{\mathrm{sem}} \mathit{toBytes}(T)$, e quindi $t' \approx_{\mathrm{sem}} T$. □

Il confronto a runtime sarebbe troppo costoso e la proprietà è coperta da test su un campione di template reali in `tests/test_zipops.py`. Con il flag `reproducible=True` di `_zipops.write_zip` la relazione $\approx_{\mathrm{sem}}$ collassa in uguaglianza di byte, perché i timestamp vengono azzerati al minimo DOS, e la proprietà diventa una uguaglianza letterale.

Le triple del Paragrafo 5 sono in correttezza totale, e includono la terminazione. La grande maggioranza delle funzioni della libreria non contiene cicli, e termina per la terminazione delle chiamate al sistema e a Pillow. L'unico ciclo non triviale è la ricerca binaria del maggior s in $[s_{\min}, s_{\max}]$ tale che il testo renderizzato col font di dimensione s entri nel riquadro, in `_find_max_font_size` in `watermark.py`. La funzione di terminazione è $s_{\max} - s_{\min}$, che si dimezza ad ogni iterazione, e la dimostrazione segue la regola del while di Mancarella [5, §4.7].

7 Mappatura al codice

Le specifiche delle operazioni non si riferiscono al codice, e gli invarianti del Paragrafo 6 solo di sfuggita. Ogni clausola è legata al codice da un identificatore `spec=` portato dalla `require` o `ensure` che la realizza, e non da un numero di riga. I numeri di riga si corrompono quando il codice si sposta, mentre l'identificatore si muove con il controllo. Il test `test_spec_crossref.py` verifica che gli identificatori usati nel codice, il catalogo in `_invariants.SPEC_REALIZATIONS` e le tabelle seguenti restino in accordo. La colonna Locazione indica il file e l'identificatore Python. La colonna Asserzioni dà il `spec=` dove presente.

Operazione formale	Locazione	Asserzioni
`Template.open`	`core.py`	—
`Template.from_bytes`	`core.py`	`require` spec=`P:from_bytes`
`Template.list_images`	`core.py`	—
`Template.replace_image`	`core.py`	`ensure` spec=`I2`
`Template.save`	`core.py`	—
`Template.to_bytes`	`core.py`	—
μ_{fname}	`core.py` (`by_filename`)	`require` spec=`P:by_filename`
μ_{mark}	`core.py` (`by_marker`)	—
μ_{auto}	`core.py` (`auto`)	—
normalize	`core.py` (`_normalize_image_input`)	—
detect	`_imagedetect.py` (`_detect_format`)	—
images	`_imagedetect.py` (`list_images`)	—
dim	`_imagedetect.py` (`_read_dimensions`)	—
`make_marker_png`	`_imagedetect.py`	—
isMarker	`_imagedetect.py` (`is_marker_image`)	—
`make_text_watermark`	`watermark.py`	—
`find_libreoffice`	`pdf.py`	—
`to_pdf`	`pdf.py`	—
toBytes (interno)	`_zipops.py` (`write_zip`)	—
fromBytes (interno)	`_zipops.py` (`read_zip_entries`)	—

Invariante	Realizzazione	Garantito da
I1 (immutabilità)	`MappingProxyType` in `Template.__init__`	struttura
I2 (preservazione nameset)	`ensure(spec="I2")` in `replace_image`	ensure
I3 (preservazione formato)	`FormatMismatchError` preventivo	struttura
I4 (unicità marker)	convenzione di `by_marker`	convenzione
I5 (round-trip semantico)	test di proprietà in `test_zipops.py`	test

La tabella qui sopra riflette la versione 0.1. La 0.2 introdurrà modifiche, in particolare l'esposizione del flag `reproducible` attraverso `Template.save`, e questa tabella verrà aggiornata.

Riferimenti bibliografici

[1] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[2] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[3] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993. ISBN 0-262-73103-7. `https://direct.mit.edu/books/monograph/4338/The-Formal-Semantics-of-Programming-LanguagesAn` In particolare il capitolo 6, *Axiomatic semantics*.

[4] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[5] F. Corradini. Logica di Hoare. Appunti per il corso di Logica per la Programmazione, Università di Pisa, 2017. `https://pages.di.unipi.it/corradini/Didattica/LPP-17/Logica%20di%20Hoare.pdf`