Axiomatic semantics of `docxwatermarker`

Fabio F.G. Buono

2026

Version 2

1 Introduction

A `.docx` file is a ZIP archive, and under `word/media/` the archive carries the raw bytes of every image anchored to the document. Replacing one of these images with other bytes and rewriting the archive yields a Word document in which the image appears different and every other piece of the file stays in place. The `docxwatermarker` library does this work, and it serves the developer who prepares a template in Word and wants to produce one personalized copy per recipient, for instance to stamp confidential documents.

These notes give an axiomatic specification of the library. They cover the library's side of a specification in two parts, whose other half, an operational semantics of the command-line tool, is the companion document *Operational semantics of the `docxwatermarker` command-line tool*. For each public operation they state a Hoare triple [1] characterizing the operation's behaviour, and from the triples they derive the properties that hold across several operations. Under this lens the library becomes a collection of contracts on bytes, archives, images, and ZIP metadata. The notation follows Mancarella's notes [5], with the standard references to Winskel [3] for the foundations and to Meyer [4] for the connection with the executable contracts the code already carries in its `require` and `ensure` primitives.

In the cited texts the triples specify constructs of a programming language, such as assignment or the while loop. Here each triple specifies a function call, and the host language, Python, is treated as an oracle.

2 How to read these notes

The Hoare-triple formalism usually applies to programs of some significance, such as critical systems, controllers, synchronization primitives, or, in textbook contexts, to small artificial code fragments. A document-personalization library falls in neither category. The code declares a partial contract through its `require` and `ensure` primitives, and making that contract explicit at the mathematical level closes a loop the code opens only halfway, because every runtime assertion gains a corresponding formal statement, and every property stated here should map to a `require` or `ensure` in the code. Writing the specification also forced us to ask questions about the library's behaviour that the code alone does not push us to ask, and four corrections in version 0.1 came out of this exercise.

The notation $\{P\}\, c\, \{Q\}$ has its standard reading. If the assertion P holds in the state of the client program's variables before the call to c, then the assertion Q holds afterwards, where in Q the symbol y denotes the value just returned and uppercase variables such as X denote the values their lowercase counterparts held on entry. The triples are in total correctness, and

they include termination of c. Operations that may raise exceptions are treated alongside the main triple, with the conditions that trigger each exception.

Lowercase variables such as x, y, t, p are client-program variables, and uppercase variables such as X, Y, T, P are specification variables fixing the entry values so that they can be referred to in the postcondition, following the convention of §6 of [5]. Fixed-width identifiers such as `Template.open` or `replace_image` are the names of classes and methods in the Python code, while mathematical italic identifiers such as *detect*, *img*, *inMedia* are names of mathematical functions that model the library's internal operations.

3 Preliminaries

The sets used in the following are treated in the intuitive notion of collection, outside any formal set theory.

We write Bytes for the set of finite octet sequences, $\mathsf{Bytes} = \{0, 1, \ldots, 255\}^*$. For $b \in \mathsf{Bytes}$, $|b|$ denotes the length in octets and $b_1 \cdot b_2$ the concatenation.

We write Name for the set of strings used as internal entry names in a ZIP archive, and on Name we assume no structure beyond equality and the function $\mathit{lower} : \mathsf{Name} \to \mathsf{Name}$ returning the lowercase form of a name.

We write Format for the set of image formats recognized by the library,

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

The value $\bot$ stands for "unrecognized format".

Entries and archives

An entry in a ZIP archive is a pair of content bytes and a collection of metadata (timestamp, compression method, POSIX permissions, miscellaneous attributes). The internal structure of the metadata plays no role in the triples, and we denote by *Meta* their domain treated as opaque. Entries produced by a modifying operation of the library are distinguished from original entries by a *modification flag* taking values tt or ff.

Definition 3.1. Entry is the set of quadruples $e = (n, d, m, \mu)$ with $n \in \mathsf{Name}$, $d \in \mathsf{Bytes}$, $m \in \mathit{Meta}$, $\mu \in \{\mathrm{tt}, \mathrm{ff}\}$, with components denoted $e.\mathit{name}$, $e.\mathit{data}$, $e.\mathit{meta}$, $e.\mathit{mod}$.

Definition 3.2. An archive is a partial function $A : \mathsf{Name} \rightharpoonup \mathsf{Entry}$ with finite domain, such that for every $n \in \mathit{dom}(A)$ we have $A(n).\mathit{name} = n$. We write $\mathsf{Archive}$ for the set of archives.

The order of entries, although relevant in practice (the library takes care to preserve it), never enters into the triples, so we leave it implicit in the phrase "partial function with finite domain".

On an archive we define the update operation. Given $A \in \mathsf{Archive}$, $n \in \mathit{dom}(A)$ and $e \in \mathsf{Entry}$ with $e.\mathit{name} = n$, we write $A[e/n]$ for the archive defined by

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

in direct analogy with Mancarella's function-update operator [5, §2].

Images

An image is the logical view of an entry whose content is recognized as one of the formats in $\mathsf{Format} \setminus \{\bot\}$. Beyond name and format, the data of interest are width, height, size in bytes, and a flag indicating whether the image carries the *marker* produced by `make_marker_png`, which is the mechanism through which the `by_marker` matcher identifies the right image in a template.

Definition 3.3. Image is the set of sextuples $i = (n,\, f,\, w,\, h,\, s,\, k)$ with $n \in \mathsf{Name}$, $f \in \mathsf{Format} \setminus \{\bot\}$, $w, h, s \in \mathbb{N}$, $k \in \{\mathrm{tt}, \mathrm{ff}\}$, subject to the constraint

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

with components accessible as $i.name$, $i.f$, $i.w$, $i.h$, $i.s$, $i.mark$.

Constraint (1) reflects the fact that the marker mechanism is based on textual `tEXt` chunks, a feature available only in the PNG format.

The total functions used in the following have signatures

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

The *detect* function identifies the format of a byte sequence from its leading *magic bytes* and returns $\bot$ when no signature matches. The *dim* function returns pixel dimensions, or $(0, 0)$ when the bytes are not decodable as an image. The *isMarker* predicate returns tt iff the first 512 bytes begin with the PNG signature and contain the marker tag. The *inMedia* predicate holds iff $\mathit{lower}(n)$ starts with the prefix `"word/media/"`.

Given an entry e with $\mathit{detect}(e.data) \neq \bot$, we set

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

The well-definedness of *img* with respect to constraint (1) is immediate, since *isMarker* returns tt only for byte sequences starting with the PNG signature, on which *detect* returns png.

Templates

A template is an archive, and we set $\mathsf{Template} = \mathsf{Archive}$. In practical use a template is an archive carrying a number of mandatory entries (`[Content_Types].xml`, `word/document.xml`, and so on), but these requirements do not enter into the library's triples.

On a template we define the function $\mathit{images} : \mathsf{Template} \to \mathsf{Image}^*$ enumerating the images through

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

where the notation $[\cdot \mid \cdot]$ denotes a sequence built by filtering (the concrete order is the iteration order on the archive's domain, and none of our properties depend on this order).

The matchers, exposed by the library as strategies for selecting an image inside a template, are partial functions $\mu : \mathsf{Image}^* \rightharpoonup \mathsf{Name}$. The three concrete matchers μ^{p}_{fname}, μ_{mark}, μ_{auto} will be defined in §5.5, where they serve as parameters to the triples for `replace_image`.

4 States and assertions

The client program that uses the library is written in Python and manipulates ordinary variables. To reason about the behaviour of the library's operations we abstract from the client language and consider only the effect on the state of its variables.

Let Var be a set of Python variable identifiers. We write

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

for the set of values our variables can take in these arguments. A *state* is a function

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

that maps each assigned variable to its value, and stays undefined on variables not yet assigned. We write State for the set of all states.

State update is defined by

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

as the function-update operator of Paragraph 3.

An *assertion* is a first-order predicate-calculus formula over the variables in Var. We use letters $P, Q, R, \ldots$ for assertions. The constants in the assertions are those of the domains introduced in Paragraph 3, and terms like $t.images$ (with $t \in \mathsf{Var}$) are state-dependent expressions. We extend the usual infix notation to the objects of our domain, so $\in$ stands for sequence membership, $=$ for equality, $|s|$ for the length of a sequence.

We write $\sigma \models P$ for "P is true in state σ", in the standard inductive definition on the structure of P. We also use specification variables (uppercase letters $X, Y, A, B, \ldots$) to fix generic values on which to parametrize preconditions and postconditions, following the convention of §6 of [5].

Hoare triples for library operations

A *Hoare triple* $\{P\}\, c\, \{Q\}$ is satisfied if, for every state σ such that $\sigma \models P$, executing c from σ terminates in a state σ' with $\sigma' \models Q$. In our setting, c is always a library function call of the form

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

where op is an operation, $\bar{a}$ is the tuple of arguments, and $y \in \mathsf{Var}$ receives the returned value, when the operation produces one. In the postcondition Q the symbol y denotes the value just returned. The variables denoting the arguments may be annotated through the *specification variables* convention, by writing X in Q to refer to the value the variable x held in σ before the call.

A triple may be weakened or strengthened through the classical rule

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

which we use as an auxiliary tool in the proofs of Paragraph 6.

Example 4.1. Consider the triple

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

The precondition states that p is a byte sequence forming a valid ZIP archive, where "p is a valid ZIP" is the predicate realized by the Python `zipfile` library (we do not define it here). The postcondition asserts that, after the call, the variable t holds a template and that the serialization of that template via $\mathit{toBytes}$ returns the same input bytes. The fact that the Python serializer preserves the bytes of a ZIP archive read and written back without changes (modulo details discussed in the paragraph on idempotence) is the reason why the postcondition carries the conjunct $\mathit{toBytes}(t) = p$ in addition to the typing $t \in \mathsf{Template}$.

5 Operation specifications

The triples for the library's operations share a convention on exceptions, which the library shapes around the hierarchy

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

defined in the `errors` module. The syntax we use to indicate that the call c, executed in a state satisfying P, may terminate in a state satisfying Q or raise one of the exceptions E_i is

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

with the convention that E_i is raised on the first matching condition R_i.

5.1 Template constructors

The operation `Template.from_bytes`(p) builds a template from raw bytes. The operation `Template.open`(ℓ) does the same from a filesystem path.

`Template.from_bytes`.

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

The predicate $\mathit{isValidZip} : \mathsf{Bytes} \to \{\mathrm{tt}, \mathrm{ff}\}$ is the one realized by `zipfile.ZipFile`, and the relation $\approx_{\mathrm{sem}}$ is the *semantic equivalence* between bytes of ZIP archives, defined by $b_1 \approx_{\mathrm{sem}} b_2$ iff the two serialized archives have the same set of names, the same per-entry contents and the same per-entry metadata, modulo possible differences in the Local File Header padding bytes computed by the serializer. The distinction between *semantic* equivalence ($\approx_{\mathrm{sem}}$, which we use throughout) and *physical* equivalence (byte-for-byte identity, which holds only with the `reproducible=True` flag) is discussed in the project README under the heading *byte-perfectness*. When p is not a valid ZIP, the behaviour is not defined by the specification, and the code raises `BadZipFile`, a Python standard library exception that lives outside the *DocxWatermarkerError* hierarchy.

`Template.open`. Opening a template from the filesystem is the composition of file reading and `from_bytes`, with the addition that a file-reading failure is tracked as `FileNotFoundError`.

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

$\mathit{fileAt}(\ell)$ denotes the bytes of the file at the system address ℓ, and $\mathit{exists}(\ell)$ the file-existence predicate.

5.2 Template inspection

`Template.list_images`. Returns the sequence of images contained in a template, according to the function *images* of Paragraph 3.

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

The conjunct $t = T$ in the postcondition expresses the immutability of the template, and will return in Paragraph 6 as a proved property.

5.3 Image replacement

The operation $t.\texttt{replace\_image}(x, \mu, v)$ is the library's pivot. It replaces, inside the template t, the image selected by the matcher μ with the bytes obtained from a normalization of the input x, and it returns a new template. The parameter $v \in \{\text{tt}, \text{ff}\}$ is the *validate* flag activating image decode validation.

The triple uses two auxiliary functions. The normalization $\mathit{normalize} : \mathsf{Val} \rightharpoonup \mathsf{Bytes}$ is defined on the union of Bytes, *Path* and *PILImage* (the set of Pillow images), and returns the input bytes when they are already in Bytes, the file bytes for a *Path*, the bytes of a PNG encoding for a Pillow image. The decoder $\mathit{canDecode} : \mathsf{Bytes} \to \{\text{tt}, \text{ff}\}$ returns tt when Pillow opens and verifies the bytes as a valid image, ff otherwise.

`Template.replace_image.`

$$\left\{\begin{array}{l} t \in \mathsf{Template} \wedge t = T \wedge \mu \in \mathit{Matcher} \wedge x \in \mathit{dom}(\mathit{normalize}) \\ \wedge\ \mu(\mathit{images}(t)) = n \wedge \mathit{detect}(\mathit{normalize}(x)) = T(n).f \\ \wedge\ (v = \text{tt} \Rightarrow \mathit{canDecode}(\mathit{normalize}(x))) \end{array}\right\}$$

$$t' \leftarrow t.\texttt{replace\_image}(x, \mu, v)$$

$$\left\{\begin{array}{l} t' \in \mathsf{Template} \wedge t' = T[e/n] \wedge t = T \\ \text{where } e.\mathit{data} = \mathit{normalize}(x),\ e.\mathit{name} = n,\ e.\mathit{meta} = T(n).\mathit{meta},\ e.\mathit{mod} = \text{tt} \end{array}\right\}$$

The call raises

- `ImageNotFoundError` if $\mu(\mathit{images}(t))$ is undefined because of no candidates;
- `MultipleImagesError` if $\mu(\mathit{images}(t))$ is undefined because of ambiguity (more than one candidate);
- `FormatMismatchError` if $\mathit{detect}(\mathit{normalize}(x)) \neq T(n).f$, namely the new input's format differs from the target's;
- `FormatMismatchError` if $v = \text{tt}$ and $\neg \mathit{canDecode}(\mathit{normalize}(x))$, namely the declared format is correct but the bytes are not decodable.

The precondition asks for $t = T$, with T a specification variable, because the postcondition itself asserts $t = T$ and needs a name for the entry value of t. The returned template is $T[e/n]$, the update of T on the single name n with the new entry e, and the other entries are left untouched (explicit *frame*). The entry e inherits the metadata $T(n).\mathit{meta}$, comprising compression method, permissions and other attributes, with $e.\mathit{data}$ and the flag $e.\mathit{mod} = \text{tt}$ as the only changes. The clause $v = \text{tt} \Rightarrow \mathit{canDecode}(\mathit{normalize}(x))$ is a precondition that activates only when the caller asks for validation, reflecting the code's choice of an opt-in validation with default $v = \text{tt}$.

5.4 Serialization

The operations $t.\texttt{to\_bytes}()$ and $t.\texttt{save}(\ell)$ produce the bytes of a template, respectively in memory and on disk.

`Template.to_bytes`.

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

where *fromBytes* is the abstract function that inverts *toBytes* up to semantic equivalence. Reapplying the reading to the produced bytes returns T, and the round-trip roundness $\mathit{toBytes} \circ \mathit{fromBytes} \approx \mathrm{id}$ underlies the *byte-perfectness* promise that the library makes in the README.

`Template.save`.

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

$\mathit{writable}(\ell)$ is the writability predicate on the filesystem at the address ℓ.

5.5 Matchers

The constructors of `ImageMatcher` are pure functions, and what characterizes each one is the partial function $\mu : \mathsf{Image}^* \rightharpoonup \mathsf{Name}$ that the produced matcher computes.

`ImageMatcher.by_filename`(p).

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

`ImageMatcher.by_marker`().

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

`ImageMatcher.auto`(s). A monoidal combinator of the previous two, with a second heuristic stage parametric in $s \in \mathbb{N}$, the minimum side of the candidate square.

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

where *or else* is the standard combinator "try the first, if undefined try the second", and

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

5.6 Image generation

The function `make_marker_png` generates a PNG placeholder carrying the marker tag. The function `make_text_watermark` generates a text watermark.

`make_marker_png`(s).

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

`make_text_watermark`$(\bar{\lambda}, s, \rho, \chi, \ldots)$. The function produces a square PNG of side s with the text lines $\bar{\lambda}$, rotated by ρ degrees, in colour χ. The dictionary of presets, denoted $\mathit{PRESETS}$, is defined in the code.

$$\left\{ \begin{array}{l} (\bar{\lambda} \in \mathit{String}^* \land \bar{\lambda} \neq \varepsilon \land \exists \lambda \in \bar{\lambda}\,.\, \mathit{trim}(\lambda) \neq \varepsilon) \lor \\ \quad (\mathit{preset} \in \mathit{dom}(\mathit{PRESETS})) \land s \in \mathbb{N} \land s \geq 1 \end{array} \right\}$$
$$d \leftarrow \texttt{make\_text\_watermark}(\bar{\lambda}, s, \rho, \chi, \ldots)$$
$$\{\, d \in \mathsf{Bytes} \land \mathit{detect}(d) = \mathrm{png} \land \mathit{dim}(d) = (s, s) \,\}$$
$$\mid \texttt{ValueError} \text{ if both conjuncts of the precondition fail}$$

The specification does not characterize the pictorial content of the produced image, since doing so would require a model of the rendering process. The relevant property is that d is a valid PNG of size (s, s).

5.7 PDF conversion

`find_libreoffice`(). A pure function returning the path to `soffice`, or $\bot$ when no path is found.

$$\{\, \mathit{true} \,\}$$
$$r \leftarrow \texttt{find\_libreoffice}()$$
$$\{\, (r \in \mathit{Path} \land \mathit{executable}(r)) \lor r = \bot \,\}$$

`to_pdf`$(\ell_{src}, \ell_{dst}, \tau, \beta)$. DOCX-to-PDF conversion goes through LibreOffice in headless mode. The parameters are the source path ℓ_{src}, the destination path ℓ_{dst} (possibly $\bot$ for a default derived from the source name), the timeout τ, and the binary override β.

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

The exceptions the function may raise are all subtypes of `PDFConversionError`, parametric in a *reason* field identifying the cause.

- `PDFConversionError`($\mathit{reason} = \mathrm{not\_found}$) if the precondition on β or `find_libreoffice` fails;
- `PDFConversionError`($\mathit{reason} = \mathrm{cannot\_create\_outdir}$) if the directory containing ℓ_{dst} cannot be created;

- `PDFConversionError`(*reason* = timeout) if the conversion exceeds τ seconds;
- `PDFConversionError`(*reason* = nonzero_exit) if `soffice` exits with a non-zero status;
- `PDFConversionError`(*reason* = no_output) if `soffice` exits cleanly but the PDF file is not produced.

The postcondition $\mathit{isPDF}(\mathit{fileAt}(\ell'))$ is the only one in the document that does not descend from the library's code, since it depends on the correct functioning of LibreOffice, which we treat as an oracle. The code does ensure that every subprocess failure is tracked as one of the variants of `PDFConversionError` above.

6 System invariants

The triples of the previous paragraph speak of one operation at a time. Some properties, however, cross several operations. Template immutability is one of these, and it holds for every operation taking a template on entry without building one from bytes.

Property 6.1. *For every $t \in$ Template and every operation o in*

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

after the execution of o on t the value of t is the one it held on entry.

Proof. The four triples in Paragraph 5 all contain $t = T$ in the postcondition, with T denoting the entry value of t. □

In the code the property is guaranteed by the type, because the constructor of `Template` freezes the dictionary with `types.MappingProxyType` (in `Template.__init__`), and `replace_image` builds a new dictionary from the original without mutating it. A related invariant fixes the template's domain under `replace_image`.

Property 6.2. *For every $t \in$ Template and every legitimate call to* `replace_image`*, the returned template t' satisfies $\mathit{dom}(t') = \mathit{dom}(t)$.*

Proof. From the postcondition of `replace_image`, $t' = T[e/n]$ with $n \in \mathit{dom}(T)$. By the definition of archive update, $\mathit{dom}(T[e/n]) = \mathit{dom}(T)$. From $t = T$ the thesis follows. □

An `ensure` (spec=`I2`) in `replace_image` checks the equality of the two namesets and raises `InvariantError` when the property is violated. It is the only structural `ensure` in the library, because adding or removing an entry from an OOXML archive would break the document. A similar guarantee holds on the format of the images, since the file `[Content_Types].xml` declares the MIME type of each entry, and replacing a PNG with a JPEG without updating that file would yield an unreadable document.

Property 6.3. *For every $t \in$ Template and every legitimate call to* `replace_image`*(x, μ, v) with $\mu(\mathit{images}(t)) = n$, the returned template t' satisfies $\mathit{img}(t'(n)).f = \mathit{img}(t(n)).f$.*

Proof. The precondition contains $\mathit{detect}(\mathit{normalize}(x)) = T(n).f$. The entry e in the postcondition has $e.\mathit{data} = \mathit{normalize}(x)$, so $\mathit{detect}(e.\mathit{data}) = T(n).f$. From $t'(n) = e$ follows $\mathit{img}(t'(n)).f = \mathit{img}(t(n)).f$. □

The constraint is in `replace_image`, before the substitution takes place, with `FormatMismatchError` when the precondition fails. The matcher `by_marker` has a unambiguous semantics only when `make_marker_png` has been called once on the template of interest.

Property 6.4. *If in $\mathit{images}(t)$ there is at most one image i with $i.\mathit{mark} = \mathrm{tt}$, then μ_{mark} is defined on $\mathit{images}(t)$ iff such an image exists, and it returns its name.*

Proof. The definition of $\mu_{\text{mark}}(\bar{\imath})$ yields $i.name$ when a unique $i \in \bar{\imath}$ exists with $i.mark = \text{tt}$, and remains undefined otherwise. The uniqueness hypothesis removes the "more than one" case. □

The library refrains from checking the convention, because inspecting the PNG metadata of every image of every template would have a hard-to-justify cost, and any violation surfaces as a `MultipleImagesError` raised by `replace_image`. Reading and writing of a template form an inverse pair, modulo the padding of the ZIP Local File Header.

Property 6.5. *For every* $t \in \mathsf{Template}$, $\mathit{fromBytes}(\mathit{toBytes}(t)) \approx_{\text{sem}} t$.

Proof. From the postcondition of `Template.to_bytes`, $\mathit{fromBytes}(b) = T$ for the returned bytes b. From the postcondition of `Template.from_bytes`, $\mathit{toBytes}(t') \approx_{\text{sem}} p$ for every t' obtained from bytes p. With $b = \mathit{toBytes}(T)$ and $t' = \mathit{fromBytes}(b)$ we get $\mathit{toBytes}(t') \approx_{\text{sem}} \mathit{toBytes}(T)$, hence $t' \approx_{\text{sem}} T$. □

A runtime comparison would be too expensive and the property is covered by tests on a sample of real templates in `tests/test_zipops.py`. Under the `reproducible=True` flag of `_zipops.write_zip` the relation $\approx_{\text{sem}}$ collapses to byte equality, because the timestamps are reset to the DOS minimum, and the property becomes a literal equality.

The triples of Paragraph 5 are in total correctness, so they include termination. The large majority of the library's functions contain no loops, and they terminate by the termination of the system and Pillow calls. The only non-trivial loop is the binary search for the largest s in $[s_{\min}, s_{\max}]$ such that the text rendered at font size s fits inside the box, in `_find_max_font_size` in `watermark.py`. The termination function is $s_{\max} - s_{\min}$, which halves at each iteration, and the proof follows Mancarella's while rule [5, §4.7].

7 Mapping to the code

The operation specifications do not refer to the code, and the invariants of Paragraph 6 only in passing. Each clause is tied to the code by a `spec=` identifier carried on the realizing `require` or `ensure`, not by a line number. Line numbers rot as the code shifts, whereas the identifier moves with the check. The test `test_spec_crossref.py` verifies that the identifiers used in the code, the catalogue in `_invariants.SPEC_REALIZATIONS`, and the tables below stay in agreement. The Location column names the file and the Python identifier. The Assertions column gives the `spec=` where one is present.

Formal operation	Location	Assertions
`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 (internal)	`_zipops.py` (`write_zip`)	—
fromBytes (internal)	`_zipops.py` (`read_zip_entries`)	—

Invariant	Realization	Enforced by
I1 (immutability)	`MappingProxyType` in `Template.__init__`	structure
I2 (nameset preservation)	`ensure(spec="I2")` in `replace_image`	ensure
I3 (format preservation)	preventive `FormatMismatchError`	structure
I4 (marker uniqueness)	`by_marker` uniqueness convention	convention
I5 (semantic round-trip)	property tests in `test_zipops.py`	tests

The table above reflects version 0.1. Version 0.2 will bring changes, in particular the exposure of the `reproducible` flag through `Template.save`, and this table will be updated.

References

[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. In particular Chapter 6, *Axiomatic semantics*.

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

[5] P. Mancarella. Note di semantica assiomatica. Lecture notes for the course in Logica per la Programmazione, B.Sc. in Computer Science, University of Pisa, 2008/2009. `https://pages.di.unipi.it/corradini/Didattica/LPP-17/Logica%20di%20Hoare.pdf`