Operational semantics of the `docxwatermarker` command-line tool

Fabio F.G. Buono

2026

Version 1

1 Introduction

These notes give an operational semantics of the `stamp` subcommand. Where the companion document *Axiomatic semantics of* `docxwatermarker` specifies the library through Hoare triples, these notes turn to the command-line tool that calls it. Here the object of study is the control flow of the command itself, the order of its phases and the exit code each failure produces. The treatment follows the notes of Barbuti, Mancarella and Turini [1], whose transition systems we use directly, with the standard reference to Plotkin [2] for the structural style and to Winskel [3] for the foundations.

The `docxwatermarker` command-line tool wraps the library of the same name behind two subcommands, `stamp` and `inspect`. Where the library exposes pure functions, the tool is a process. It reads arguments, opens a template, builds a replacement image, performs the swap, writes the result, and on request produces a PDF, and at each step it may stop and return a numeric exit code that records how far it got. A caller scripting the tool reads those codes, so their meaning is part of the tool's contract.

The axiomatic and the operational specification describe the same software from two angles. The first says what each library operation guarantees, the second says how the command that calls those operations advances and terminates. Of the two subcommands, this document concentrates on `stamp`, which carries the full chain of operations and the differentiated exit codes. The `inspect` subcommand, a read-only listing, is treated briefly at the end.

2 Command syntax and format recognition

Before the transition rules, two pieces of the tool admit a description in the classical terms of formal languages, and stating them first makes the rest more precise. A language is a set of admissible strings over an alphabet, and two standard tools describe one. A *grammar* generates the strings the language contains, and an *automaton* recognizes them. We use a grammar for the command line the user writes, and an automaton for one internal point of the library.

A word on what is, and is not, modelled this way. The library is a collection of pure functions, each characterized by the Hoare triples of the companion document. A pure function has no states to move between, so an automaton would not describe it, it would invent structure the code does not have. One internal point is genuinely a recognizer, the function that identifies an image's format from its leading bytes. That is what the automaton below models. The command's own control flow, a process moving through phases, is the subject of the transition system in the sections that follow.

2.1 Grammar of the command line

The following grammar generates the command lines the tool accepts, in the notation of context-free grammars. Terminals are set in `typewriter`, the metavariables *path*, *text*, and *name* stand for a filesystem path, an arbitrary string, and a preset name. Brackets [·] mark an optional part and the bar | marks a choice. The grammar mirrors the parser in `cli.py`, where each alternative below corresponds to an argument or mutually exclusive group declared there.

command	::=	`stamp` *stamp-args* | `inspect` *path* [`--debug`]
stamp-args	::=	*path option**
option	::=	*source* | *target* | *pdf* | *style* | *mode*
source	::=	`--image` *path* | `--watermark-text` *text* | `--preset` *name*
target	::=	`--use-marker` | `--target-filename` *path*
pdf	::=	`--pdf` | `--pdf-only`
style	::=	`--size` *int* | `--rotation` *float*
mode	::=	`--output` *path* | `--interactive` | `--verbose` | `--debug`

The long options `--output`, `--interactive`, and `--verbose` have the short aliases `-o`, `-i`, and `-v`, which the parser accepts interchangeably.

The grammar describes the shape the parser accepts, not every constraint it enforces. Three of the option groups, *source*, *target*, and *pdf*, are mutually exclusive, and the parser rejects a command line that uses two alternatives from the same group. The further rule that `stamp` needs a source, supplied either by a *source* option or interactively under `-i`, is checked after parsing, and it is the resolve phase of the transition system that enforces it.

2.2 An automaton for format recognition

When the library replaces an image, it first identifies the format of the replacement bytes, in the function `_detect_format` of `_imagedetect.py`. The function reads the leading bytes and accepts them as one of the known formats, or rejects them. This is a recognizer over a small language of magic-byte prefixes, and a deterministic finite automaton describes it.

The automaton reads the input byte by byte from the start state q_0. In the transition function below, a row is the current state, a column heading is the byte read, and the cell is the next state. States PNG, JPEG, GIF, BMP, and TIFF are accepting and name the recognized format. The state R is the one subtlety. The prefix `RIFF` is shared by several formats, so reaching R is not enough, and the automaton must read four more bytes and check for `WEBP` before accepting. Any byte not shown leads to the reject state $\perp$, from which there is no escape.

State	Input read	Next state
q_0	`0x89 50 4E 47 0D 0A 1A 0A`	PNG (accept)
q_0	`0xFF D8 FF`	JPEG (accept)
q_0	`GIF87a` *or* `GIF89a`	GIF (accept)
q_0	`BM`	BMP (accept)
q_0	`II*\0` *or* `MM\0*`	TIFF (accept)
q_0	`RIFF`	R
R	`WEBP` (4 bytes at offset 8)	WEBP (accept)
R	any other 4 bytes	$\perp$ (reject)
q_0	any other prefix	$\perp$ (reject)

The automaton recognizes the language of admissible image headers, the same set the library treats as valid input. A format mismatch in `replace_image`, exit code 6 in the transition system, is the case where the replacement bytes and the target image land in different accepting states of this automaton.

3 Transition systems

We recall the apparatus we use, in the form given in [1]. A transition system is a triple $\langle \Gamma, T, \to \rangle$ where Γ is a set of configurations, $T \subseteq \Gamma$ is the set of terminal configurations, and $\to$ is the transition relation. A configuration is a state the system can occupy, and a terminal configuration is one in which the system has finished. We write $\gamma \to \gamma'$ when the pair belongs to $\to$, and $\gamma \to^* \gamma'$ when a finite derivation $\gamma \to \gamma_1 \to \ldots \to \gamma'$ exists. A non-terminal configuration from which no transition departs is *stuck*. The semantics below is designed so that no reachable configuration is stuck, every path ends in a terminal.

The transition relation is given by conditional rules in the fraction form

$$\frac{\pi_1 \quad \pi_2 \quad \cdots \quad \pi_n}{\gamma \to \gamma'}$$

where the premises π_i are the prerequisites for the transition. A premise is an equality $x = t$, a relation $t \; rel \; t'$, or another transition $\delta \to' \delta'$. A rule with a transition premise is recursive when $\to'$ is the relation being defined. The reading is the one in [1]. A transition holds when some substitution of concrete values for the rule's variables makes every premise hold and the conclusion equal to the transition in question. We follow the convention that every variable in a rule is covered, its value determined by the configuration the rule applies to, or through the premises.

4 State of the `stamp` command

The library is the oracle for everything that happens inside an operation. What the operational semantics tracks is the data the command carries between operations, not the bytes the library moves. We collect that data into a state.

A state σ is a finite map from names to values, written as a set of bindings. The command's state binds the parsed arguments and the intermediate results computed so far. We write $\sigma(x)$ for the value bound to x, undefined when x is unbound, and $\sigma[v/x]$ for the state that agrees with σ everywhere except at x, where it takes the value v. This is the state and the update operator of [1], section 4.

The names the `stamp` command uses are these. The argument names *src*, *out*, *pdf*, *interactive* hold the parsed command line, that is the watermark source requested by the user, the output path if given, whether a PDF was asked for, and whether interactive entry was requested. The working names t, *img*, t', o hold the open template, the replacement image, the template after the swap, and the resolved output path. A name is bound only once its phase has produced it, so $\sigma(t')$ is undefined until the swap has run.

The source *src* is one of the forms the command accepts, an image path $\mathsf{image}(p)$, a list of watermark lines $\mathsf{text}(\ell)$, a preset name $\mathsf{preset}(k)$, or the absence of any of these, $\bot$. We abstract over which one it is except where a rule needs to distinguish them.

5 Configurations and exit codes

A configuration of the `stamp` command pairs a phase with a state,

$$\langle \mathsf{ph}, \sigma \rangle \in \Gamma,$$

where ph names the point the command has reached. The phases, in the order the command visits them, are $\mathsf{resolve}$ (settle the source, prompting if asked), open (open the template), build (construct the replacement image), swap (replace the image in the template), save (write the

result), and pdf (produce a PDF if requested). The initial configuration of a run on parsed arguments σ_0 is $\langle \mathsf{resolve}, \sigma_0 \rangle$.

The terminal configurations are the exit codes,

$$T = \{\, \mathsf{exit}\, n \mid n \in \{0, 1, 3, 4, 5, 6, 7\} \,\}.$$

A run ends when it reaches one of these. The codes are not arbitrary labels. Each marks a distinct way the command can stop, and a caller branches on them. Code 0 is success. Code 1 is a usage error, no source given or a malformed one. Code 3 is an unreadable template. Codes 4, 5, 6 come straight from the library exceptions that the swap can raise, image not found, ambiguous match, and format mismatch. Code 7 is a failed PDF conversion after a document was already written. The table at the end of Section 6 collects them.

6 Transition rules for `stamp`

We define $\rightarrow_{\mathrm{st}}$, the transition relation of the `stamp` command, one phase at a time. Each phase contributes a rule for the case where it proceeds and one or more rules for the cases where it stops with a code. The premises that mention the library use the same names as the axiomatic specification, and a premise of the form "*op* raises *E*" abbreviates the condition under which the library operation signals the exception *E*.

Resolve. The command first settles the watermark source. When interactive entry was requested and no source was given on the command line, the command reads lines from the terminal. The read yields a list ℓ, and on the empty list the command stops with a usage error. We write $\mathsf{prompt} \rightarrow \ell$ for the interactive read producing ℓ.

$$\frac{\sigma(\mathit{interactive}) = \mathrm{tt} \qquad \sigma(\mathit{src}) = \bot \qquad \mathsf{prompt} \rightarrow \ell \qquad \ell \neq []}{\langle \mathsf{resolve}, \sigma \rangle \rightarrow_{\mathrm{st}} \langle \mathsf{open}, \sigma[\mathsf{text}(\ell)/\mathit{src}] \rangle}$$

$$\frac{\sigma(\mathit{interactive}) = \mathrm{tt} \qquad \sigma(\mathit{src}) = \bot \qquad \mathsf{prompt} \rightarrow []}{\langle \mathsf{resolve}, \sigma \rangle \rightarrow_{\mathrm{st}} \mathsf{exit}\, 1}$$

When a source is present, the command moves on unchanged. When none is present and interactive entry was not requested, it stops with the same usage code.

$$\frac{\sigma(\mathit{src}) \neq \bot}{\langle \mathsf{resolve}, \sigma \rangle \rightarrow_{\mathrm{st}} \langle \mathsf{open}, \sigma \rangle} \qquad \frac{\sigma(\mathit{src}) = \bot \qquad \sigma(\mathit{interactive}) = \mathrm{ff}}{\langle \mathsf{resolve}, \sigma \rangle \rightarrow_{\mathrm{st}} \mathsf{exit}\, 1}$$

Open. The command opens the template at the path it was given. Success binds the open template to t and resolves the output path o, from the explicit argument when present and otherwise derived from the template path by the function *derive*. Failure to open, whether the file is missing or not a valid archive, stops the command with code 3.

$$\frac{\mathsf{open}(\sigma(\mathit{template})) \rightarrow t \qquad o = \mathit{outpath}(\sigma)}{\langle \mathsf{open}, \sigma \rangle \rightarrow_{\mathrm{st}} \langle \mathsf{build}, \sigma[t/t,\ o/o] \rangle} \qquad \frac{\mathsf{open}(\sigma(\mathit{template})) \text{ raises } \mathsf{BadZip} \vee \mathsf{OSError}}{\langle \mathsf{open}, \sigma \rangle \rightarrow_{\mathrm{st}} \mathsf{exit}\, 3}$$

where $\mathit{outpath}(\sigma) = \sigma(\mathit{out})$ when *out* is bound and $\mathit{derive}(\sigma(\mathit{template}))$ otherwise.

Build. The command constructs the replacement image from the source. A path source is read from disk, a text source is rendered to a watermark image, a preset expands to a fixed text and is then rendered. The construction can fail on a malformed source, for instance a `--image` path that does not exist, and that failure is a usage error.

$$\frac{\mathsf{build}(\sigma(\mathit{src})) \rightarrow \mathit{img}}{\langle \mathsf{build}, \sigma \rangle \rightarrow_{\mathrm{st}} \langle \mathsf{swap}, \sigma[\mathit{img}/\mathit{img}] \rangle} \qquad \frac{\mathsf{build}(\sigma(\mathit{src})) \text{ raises } \mathsf{ValueError} \vee \mathsf{FileNotFound}}{\langle \mathsf{build}, \sigma \rangle \rightarrow_{\mathrm{st}} \mathsf{exit}\, 1}$$

Swap. The command replaces the matched image in the template, producing a new template t'. This is the phase that inherits the library's differentiated failures. A successful swap moves to save.

$$\frac{\mathsf{replace}(\sigma(t), \sigma(img)) \to t'}{\langle \mathsf{swap}, \sigma \rangle \to_{\mathrm{st}} \langle \mathsf{save}, \sigma[t'/t] \rangle}$$

The three library exceptions map to three codes. No match for the requested image stops with 4, more than one candidate when the matcher requires a unique one stops with 5, and a replacement whose format differs from the image it would replace stops with 6.

$$\frac{\mathsf{replace}(\sigma(t), \sigma(img)) \text{ raises } \mathsf{ImageNotFound}}{\langle \mathsf{swap}, \sigma \rangle \to_{\mathrm{st}} \mathsf{exit}\, 4} \qquad \frac{\mathsf{replace}(\sigma(t), \sigma(img)) \text{ raises } \mathsf{MultipleImages}}{\langle \mathsf{swap}, \sigma \rangle \to_{\mathrm{st}} \mathsf{exit}\, 5}$$

$$\frac{\mathsf{replace}(\sigma(t), \sigma(img)) \text{ raises } \mathsf{FormatMismatch}}{\langle \mathsf{swap}, \sigma \rangle \to_{\mathrm{st}} \mathsf{exit}\, 6}$$

Save. The command writes the new template to the resolved output path. The write is taken to succeed, the directory being created when absent, so the phase has a single rule. When no PDF was requested this is the last step and the command succeeds. Otherwise it moves to pdf.

$$\frac{\sigma(pdf) = \mathrm{ff} \qquad \mathsf{save}(\sigma(t'), \sigma(o)) \to ok}{\langle \mathsf{save}, \sigma \rangle \to_{\mathrm{st}} \mathsf{exit}\, 0} \qquad \frac{\sigma(pdf) = \mathrm{tt} \qquad \mathsf{save}(\sigma(t'), \sigma(o)) \to ok}{\langle \mathsf{save}, \sigma \rangle \to_{\mathrm{st}} \langle \mathsf{pdf}, \sigma \rangle}$$

PDF. When a PDF was requested, the command converts the written document through LibreOffice. A successful conversion ends the run with success, a failed one with code 7. The intermediate document has already been written at this point, which is why a PDF failure has its own code rather than folding into an earlier one, the caller knows the `.docx` exists even when the `.pdf` does not.

$$\frac{\mathsf{topdf}(\sigma(o)) \to ok}{\langle \mathsf{pdf}, \sigma \rangle \to_{\mathrm{st}} \mathsf{exit}\, 0} \qquad \frac{\mathsf{topdf}(\sigma(o)) \text{ raises } \mathsf{PDFConversion}}{\langle \mathsf{pdf}, \sigma \rangle \to_{\mathrm{st}} \mathsf{exit}\, 7}$$

The exit codes together.

Code	Reached from
0	save or pdf, on success
1	resolve (no or empty source), build (malformed source)
3	open, template missing or not a valid archive
4	swap, no image matched
5	swap, more than one image matched
6	swap, replacement format differs from target
7	pdf, conversion failed after the document was written

7 Runs

A run of the command on parsed arguments σ_0 is the derivation that starts at $\langle \mathsf{resolve}, \sigma_0 \rangle$ and proceeds by $\to_{\mathrm{st}}$ until it reaches a terminal. Two properties of the relation follow by inspection of the rules in Section 6.

Property 7.1 (Progress)**.** *Every reachable non-terminal configuration has a successor. For each phase the rules cover its cases without overlap, so one rule applies and no configuration is stuck.*

The phases form a fixed order, resolve, open, build, swap, save, and then pdf only when a PDF was requested. No rule returns to an earlier phase. A run therefore visits each phase at most once and reaches a terminal in a number of steps bounded by the number of phases.

Property 7.2 (Termination)**.** *Every run is finite and ends in a terminal configuration.*

A consequence ties the operational picture back to the axiomatic one. When a run reaches exit 0, it passed through swap by the success rule, which means the library's replace operation returned a template. The postcondition of that operation, from the companion document, is what guarantees that the written document preserves the archive's nameset. The exit code 0 is, in this sense, the operational shadow of the axiomatic postcondition, the command succeeds precisely when the contract it relies on was met.

8 The `inspect` subcommand

The `inspect` subcommand reads a template and lists the images it contains, changing nothing. Its semantics needs only two phases, open and list, and three of the codes. It opens the template as `stamp` does, with the same code 3 on failure, then lists.

$$\frac{\mathsf{open}(\sigma(\mathit{template})) \to t \qquad \mathsf{images}(t) \to \mathit{imgs}}{\langle \mathsf{open}, \sigma \rangle \to_{\mathrm{st}} \mathsf{exit}\,0} \qquad \frac{\mathsf{open}(\sigma(\mathit{template})) \text{ raises } \mathsf{BadZip} \vee \mathsf{OSError}}{\langle \mathsf{open}, \sigma \rangle \to_{\mathrm{st}} \mathsf{exit}\,3}$$

The listing always succeeds once the template is open, whether or not any image is present, so the read-only command has only the success code and the open failure. It never reaches the codes 4 through 7, which belong to the swap and the PDF conversion that `inspect` does not perform.

9 Mapping to the code

The phases correspond to spans of `cmd_stamp` in `cli.py`, in the order the function executes them. The table names the function rather than a line, for the reason given in the companion document, line numbers rot while names do not.

Phase	Code location	Exit codes
resolve	`cmd_stamp` (source resolution)	1
open	`cmd_stamp` → `Template.open`	3
build	`cmd_stamp` → `_build_replacement_image`	1
swap	`cmd_stamp` → `Template.replace_image`	4, 5, 6
save	`cmd_stamp` → `Template.save`	—
pdf	`cmd_stamp` → `to_pdf`	7

The exit codes in the table are the integers `cmd_stamp` returns, and the dispatcher `main` passes them to `sys.exit` unchanged, save for two codes outside this semantics, 2 for an unparsable command line, which `argparse` returns before `cmd_stamp` runs, and 130 for an interrupt.

The set of codes is not kept in sync by hand. The test `test_spec_crossref.py` extracts every integer returned by `cmd_stamp` and `cmd_inspect` from the source and checks it against the codes tabulated here, in both language editions. A code added or removed in the code without the document, or a divergence between the two editions, fails the build. This is the operational counterpart of the `spec=` link the axiomatic specification uses.

References

[1] R. Barbuti, P. Mancarella, F. Turini. *Elementi di Semantica Operazionale.* Lecture notes for Fondamenti di Programmazione, B.Sc. in Computer Science, University of Pisa, A.A. 2004/05. `https://pages.di.unipi.it/mancarella/FP/materiale/nuovasemop.pdf`

[2] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981. Reprinted in *J. Log. Algebr. Program.*, 60–61:17–139, 2004.

[3] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, 1993. ISBN 0-262-73103-7.