Analysis and Transformation of Textual Specifications: A Compiler Infrastructure

Cristiano Carvalho Lacerda
Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo
São Carlos, Brazil
cristiano.lacerda@usp.br

Simone do Rocio Senger de Souza
Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo
São Carlos, Brazil
srocio@icmc.usp.br

José Carlos Maldonado
Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo
São Carlos, Brazil
jcmaldon@icmc.usp.br

Abstract

In regulated software projects, requirements, design elements, and verification cases are engineering evidence with stable identifiers, attribute domains and traceability obligations on which formal review and certification depend. This paper presents an open-source tool that lowers textual specifications into a typed relational intermediate representation called Spec-IR. Authors write CommonSpec, an extension of CommonMark, and SpecCompiler persists compiled facts in SQLite before emitting DOCX, HTML, or TeX through Pandoc. Structural verification is expressed as SQL queries, where non-empty result sets from policies configured as errors become blocking diagnostics with source-file and line information. The infrastructure evolved from an industrial pilot within a defense and aerospace software program, where a predecessor tool was developed, deployed, and evaluated in the production of formal software documentation. The tool demonstrates how a specification compiler can make it practical to enforce the traceability and structural integrity of long-lived technical specifications as they evolve. Demo video: https://doi.org/10.6084/m9.figshare.32400411.

Keywords

Compiled Specifications, Docs-as-Code, Requirements Engineering, Traceability, Safety-Critical Software, Software Documentation Tools

1 Introduction

In regulated software projects, documents are typed engineering artifacts. A requirement has an identifier, a status, a rationale and trace obligations. A verification case has a method, an expected result and links to the requirements it covers. A design description allocates behavior to components and units. These structures are inspected, baselined, exchanged and audited [7, 17]. If a trace link dangles, an expected field is missing, or an enumerated value is invalid, the document may render correctly while remaining structurally wrong.

Industry has well-established tools for managing such artifacts. Requirements management systems (RMS) such as IBM DOORS and Siemens Polarion provide typed objects, attributes, baselines and trace links [10, 18], but they typically involve per-seat licensing and central-server infrastructure and remain separate from the developer's everyday Git, CI and editor environment. The friction grows further across organizational boundaries, where collaborators on the other side may not have access at all. The ReqIF standard exists to interchange the underlying structures across such boundaries [5, 14].

These constraints shape how the field actually operates. A recent comparative study finds Microsoft Office still the single most-used tool for requirements work, with agile and ALM platforms such as Jira and Trello among the most popular solutions in current practice [2]. Each family solves part of the problem: enterprise RMS centralize structure away from developer workflows, agile and ALM tools track work without formal document deliverables, and lightweight writing tools (Office, Wikis) cannot verify traceability and integrity.

Docs-as-Code is an attractive starting point. Plain-text documents in Git can be reviewed, diffed and built with the same infrastructure used for source code [21]. However, a normal Markdown [8] or RST [6] pipeline primarily *renders* documents. It does not know that a heading represents a high-level requirement, that a link inside a `traceability:` field must resolve to another requirement, or that an attribute value must come from a declared domain. Rendering success is a much weaker guarantee than engineering correctness. A small but growing family of Docs-as-Code tools — Doorstop, StrictDoc, Sphinx-Needs — applies this idea to requirements specifically [1, 15, 20]. None of them, however, appears among the solutions reported in the comparative study above [2], an indicator that industrial adoption of this family remains limited.

This paper presents an open-source compiler that brings RMS-style structural rigor to Docs-as-Code workflows: the author edits Markdown with a small convention layer, and the build emits formal deliverables (DOCX, HTML, TeX, PDF) only after configured blocking checks pass. The infrastructure generalizes PORTAS, a predecessor tool developed, deployed and evaluated in a defense-aerospace software development program (Section 7). The contributions are:

1. **SpecCompiler**, an Apache-2.0 compiler infrastructure for the analysis and transformation of textual specifications, distributed as a command-line tool that produces DOCX, HTML, TeX and PDF deliverables;
2. **CommonSpec**, a small CommonMark [12] compatible convention layer for typed requirements, design elements, verification cases, attributes, trace links, floats and generated views;
3. **Spec-IR**, a persistent SQLite-backed relational intermediate representation (IR) that exposes compiled specification facts to direct queries and downstream tools.

2 Tool Overview

SpecCompiler targets engineers, technical writers and researchers who need long-lived, traceability-intensive technical documents to be authored in a developer-friendly format while still enforcing the structural checks expected of formal engineering deliverables. Two roles share this workflow: a model developer configures the project vocabulary — object and relation types, attributes, generated views, output styles and SQL verification policies — while document authors write CommonSpec against that configured model. In a typical build, an author edits Markdown in Git, runs the compiler locally or in CI, fixes source-located diagnostics, and publishes DOCX, HTML, TeX or PDF only after configured blocking checks pass.

The tool provides five core capabilities. **Typed Markdown authoring:** CommonSpec maps headings, blockquote fields, links, inline code and code blocks to typed objects, attributes, relations, floats and views. **Relational compilation:** compiled facts are persisted in SQLite as Spec-IR [9], open to direct inspection and downstream tools. **Executable verification:** structural policies are SQL views; non-empty sets produce source-located diagnostics. **Generated evidence:** traceability matrices, allocation tables and coverage summaries are generated from the same relation rows used by verification. **Model-driven extension:** object types, datatypes, relation kinds, output styles and verification policies live in model files, so new document families extend a model, not the compiler.

SpecCompiler ships with a default model intended as the base layer for all projects. It provides the technical-writing facilities CommonMark lacks: cross-referenced typed floats, automatic numbering and captions, math notation, glossary terms and generated tables of contents. Projects that need engineering structure add an overlay model, such as `sw_docs`, on top of this base. The HTML backend ships with Spec-IR and `sqlite3.js` so generated documents double as interactive databases of the compiled facts; the DOCX backend drives corporate templates through CSS-like styles and a Lua OOXML post-processor; and the TeX backend adapts to LaTeX classes with minimal per-project post-processing. Because content is typed, presentation is rule-driven: a model attaches type-dependent layout and typography to each element through the same type environment that drives verification.

The reference implementation is the third iteration of the generic compiler: about 12 KLOC of Lua for the core infrastructure. Reusable model layers, documentation compiled by SpecCompiler itself and an automated test suite bring the codebase to roughly 40 KLOC. The implementation effort was not tracked; a conservative bound based on calendar time puts the functional release under six person-months, aided by agentic coding tools (Claude Code and OpenAI Codex) and excluding the earlier pilot and research that delimited the domain and scope.

Intended users are any team with demanding documentation, where traceability matters: software-engineering teams in industries such as defense, aerospace, medical and automotive; requirements-engineering researchers who want a typed relational substrate over Markdown specifications. More broadly, teams authoring technical Markdown documents that want added typed validation and prescribed DOCX or PDF output.

3 Architecture

SpecCompiler is structured as a five-phase pipeline over a persistent typed relational store, illustrated in Figure 1. The phases run sequentially and communicate exclusively through Spec-IR, with no in-memory artifacts passed between them; intermediate states stay inspectable and the pipeline is restartable from any phase.

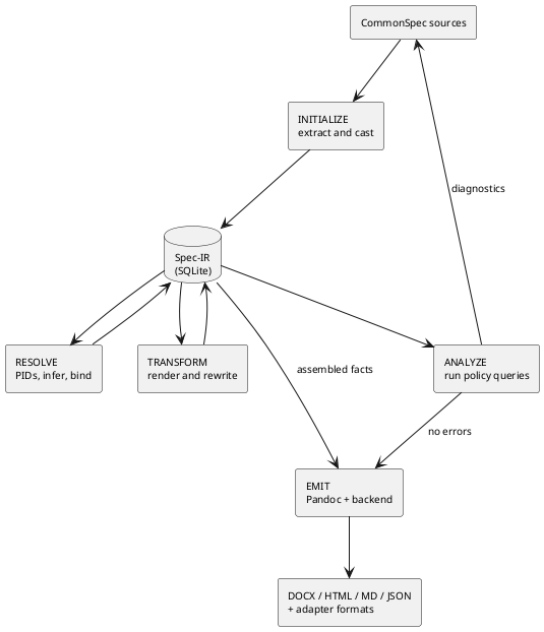

Figure 1: The five phases operating on Spec-IR.

SpecCompiler runs on top of Pandoc [11] the way a backend JavaScript application runs on top of Node.js. Pandoc provides both the runtime and the typed-AST API through which every phase reads and writes documents. Each type — object, attribute, relation, float, view — is a Lua table under the active model, discovered at startup and optionally extended with hooks invoked by the pipeline.

Initialize parses CommonSpec sources with a Pandoc Lua filter and extracts documents, typed headings, blockquote fields, link occurrences, fenced floats and source locations, casting attribute values into typed columns. **Resolve** generates missing identifiers, binds candidate relations to their targets and infers relation types from source-attribute context, recording resolution errors. **Transform** materializes derived content such as traceability matrices, renders typed floats whose backends invoke external tools (for example, compiling PlantUML diagram blocks into image assets), and rewrites the content AST through per-type rules. **Analyze** runs the policy views; non-empty error results block the next phase. **Emit** invokes Pandoc with a model-specific backend, producing DOCX, HTML, TeX or PDF.

The IR is split between content tables and a type-environment loaded from the active model, as shown in Figure 2. Content tables hold what the engineer wrote; type-environment tables hold the declared vocabulary of specification, object, attribute, relation, float and view types, with their datatypes and enumerations. Every content row points (via `type_ref`) at the type-environment row that interprets it. This separation is what allows a new document

family to be expressed as a new model rather than a new fork of the compiler — the main generalization from PORTAS, whose document model was hardcoded in project-specific tables and queries.

Links inside CommonSpec carry semantics. A Markdown link to `HLR-042`, a high-level requirement (HLR), written inside the `traceability:` field of a verification case (VC) object (as illustrated in Section 4) becomes a candidate typed relation. The Resolve phase consults the model and finds the relation type `VERIFIES` with `source_type = VC`, `target_type = HLR`, `source_attribute = 'traceability'`. The resolver then attempts to bind the target text to an object identifier, casting the result into a typed `spec_relations` row or recording a diagnostic if no matches are found. Plain Markdown links thus gain typed semantics that verification queries can reason about as constrained relations. Two relation types may also supply different transform hooks; the hook receives the IR and returns Pandoc AST, so a link can be rewritten in place — as a project-specific text notation, or as a filtered view of the target's content brought inline — rather than appearing as a plain hyperlink.

Verification policies are SQL views that return counterexamples. The canonical contract is evaluated as the violation set $\{x \mid P(x) \wedge \neg Q(x)\}$, where P selects candidates and Q names the required evidence. SQL realizes this set as an anti-join, typically `WHERE NOT EXISTS`. The view shown below reports every HLR that has no covering VC. P is the set of HLRs and Q an inbound `VERIFIES` relation. The pattern applies the closed-world assumption and negation-as-failure principles [3, 16]. Anything not stated in the IR is false. Reusing standard SQL keeps the verification language familiar and makes policies version-controllable like any other source artifact.

```
CREATE VIEW IF NOT EXISTS view_traceability_hlr_missing_vc AS
SELECT
  hlr.id        AS object_id,
  hlr.pid       AS object_pid,
  hlr.title_text AS object_title,
  hlr.from_file,
  hlr.start_line
FROM spec_objects hlr
WHERE hlr.type_ref = 'HLR'
  AND NOT EXISTS (
    SELECT 1
    FROM spec_relations r
    WHERE r.target_object_id = hlr.id
      AND r.type_ref = 'VERIFIES'
  );
```

The complete command-line interface is a single `specc build project.yaml` command. The project file names the active model, the root source files — which may transitively `include` other CommonSpec files — and the selected output formats. The compiler runs the full pipeline and emits the outputs together with `specir.db`.

4 Usage Scenario

Consider a small safety-related requirements/verification pair. The active model defines the project vocabulary — object types such as HLR and VC, their attributes, relation rules, generated views and verification policies — and the document author writes ordinary Markdown against it. The engineer adds two high-level requirements, but initially writes a verification case for only one of them.

```
<!-- srs.md -->
# Software Requirements

## HLR: Detect overheat @HLR-042

The controller shall enter safe mode when temperature
exceeds the limit.

> status: Draft
> rationale: Prevent thermal damage.
```

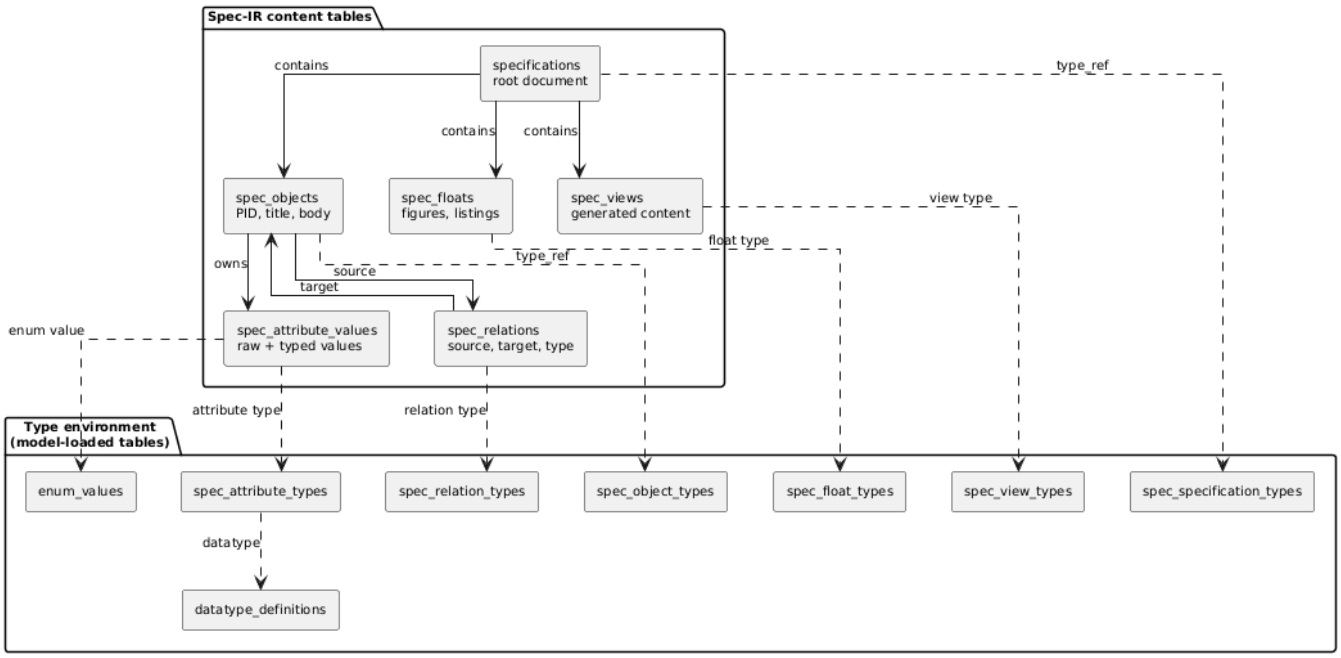

Figure 2: Spec-IR separates content tables from a model-loaded type environment.

Table 1: Lowering from CommonSpec source to compiled facts.

Source construct	Spec-IR table	Compiled representation
Typed heading	`spec_objects`	Object row with type, identifier, title, source location and body interval.
Blockquote field	`spec_attribute_values`	Attribute-value row owned by the current object, with raw value and declared datatype.
Markdown link in a typed field	`spec_relations`	Candidate relation row with source object, source attribute, target text and selector.
Typed code block	`spec_floats`	Generated non-textual, such as a figure, listing, diagram or table.
Typed inline code	`spec_views`	Materialized content computed from the relational facts.
Verification policy	(Query against Spec-IR)	SQL query whose rows are structural violations to report or block before emission.

```
## HLR: Log overheat event @HLR-043

The controller shall record an overheat event before
entering safe mode.

> status: Pending
> rationale: Provide reviewable fault evidence.

```puml:fig-thermal{caption="Thermal states."}
@startuml
[*] --> Normal
Normal --> SafeMode : temperature > limit
SafeMode --> Normal : reset
@enduml
```

<!-- svc.md -->
Software Verification Cases

VC: Verify safe-mode transition @VC-020

Exercise the controller above the thermal threshold.

> method: Test
> result: Not Run
> traceability: [HLR-042](@)

Traceability Matrix

`traceability_matrix:`
```

During compilation, each typed heading creates an object row; the blockquote lines beneath it become typed attribute-value rows. The PlantUML fence becomes a typed float with label and caption, and the Markdown link inside the `traceability:` field becomes a candidate relation whose source is `VC-020`, whose target text is `HLR-042` and whose source-attribute context is `traceability`. The active document model declares which object types, attributes, enumerated values and relation endpoints are valid. Table 1 summarizes the lowering.

Compilation reports two kinds of problem. The HLR coverage view described in the architecture section is evaluated against the compiled IR, because `HLR-043` has no inbound VC relation, the view returns a counterexample rather than the empty set expected by the policy, and SpecCompiler turns that row into a blocking diagnostic with the policy name, object identifier and source location. Similarly the value `Pending` is not in the declared enumeration domain of `status`, so the attribute cast fails with diagnostic. Emission is aborted and no deliverable is produced from the inconsistent specification. The project may configure a severity (`error`, `warn` or `ignore`) for each verification view. Figure 3 shows a representative build log with both kinds of diagnostic.

The author fixes the scenario by correcting the status to the declared value `Draft` and adding a second traceability link from `VC-020` to `HLR-043`. The rebuild leaves no blocking error — the cast succeeds and the coverage view is empty — so the compiler proceeds to Emit and generates the traceability matrix from the same relation rows it verified. The matrix is a view type, which runs a SQL query against Spec-IR and returns Pandoc AST blocks, that are rendered into the document. The `sw_docs` traceability matrix chains each HLR to its verifying VCs and their pass/fail test results. This usage flow is the key distinction between a Markdown document that is merely rendered and one that is analyzed and transformed as a compiled specification: the source remains lightweight, missing engineering structure is observable as a compiler error, and the structure that is present returns as generated evidence such as the traceability matrix.

The same workflow runs in the continuous-integration (CI) pipeline that already builds and tests the project's source code, a CI job invokes `specc build project.yaml`, and if any view configured at `error` severity returns rows, the job fails and blocks the merge exactly as a failing test suite does. Early in a program the same policies can run at `warn` while requirements mature; at a release milestone they are raised to `error`, and a passing run publishes the HTML and DOCX deliverables and archives `specir.db` as review evidence.

5 Limitations and Roadmap

The empirical evidence reported later in this paper comes from the hardcoded predecessor in a single industrial program; the generic, model-driven infrastructure has not yet been independently evaluated in production. SpecCompiler runs as a Pandoc Lua filter and depends on a specific Pandoc, Lua and Lua modules compiled as native shared libraries, such as the SQLite and libuv bindings [4, 19]. Reproducing it natively currently requires compiling much of the dependency stack from source, which is fragile across machines and operating systems. The maintained mitigation is a pre-built Docker image plus a wrapper, which gives contributors and CI runners a reproducible toolchain but does not eliminate the barrier for ordinary users, especially in Windows-centered environments, native multi-platform packaging remains future work.

A second limitation is model availability and authoring effort, but the effort is unevenly distributed. Writing documents against an existing model is deliberately accessible, authors use Markdown, follow examples and learn a small set of project conventions. The

```
$ specc build project.yaml
Building project...
INFO  Starting SpecCompiler...
INFO  Template: sw_docs
INFO  Output directory: build
INFO  Documents to process: 2
INFO  [1/2] Processing: srs.md
INFO  [2/2] Processing: svc.md
INFO  Running phase: initialize (2 context(s))
INFO  Parsed 4 total headers across 2 specifications
INFO  Created 2 total spec objects across 2 documents
INFO  Extracted 3 total attributes across 2 documents
INFO  Running phase: analyze (2 context(s))
INFO  Auto-generated 2 specification PIDs
INFO  Auto-generated 1 PIDs across all specifications
INFO  Running phase: transform (2 context(s))
INFO  Rendered 1 spec objects with type handlers
INFO  Rendered 1 spec objects with type handlers
INFO  Running phase: verify (2 context(s))
ERROR srs.md:3 [invalid_cast] Failed to cast attribute 'status' to ENUM (value: 'Pending'); valid values: Draft, Review,
 Approved, Implemented
ERROR srs.md:3 [traceability_hlr_to_vc] High-level requirement '0013' is not covered by any VC
ERROR svc.md:3 [traceability_vc_to_hlr] Verification case 'VC-001' has no traceability link to an HLR
ERROR Pipeline aborted: verification found 3 error(s)
```

Figure 3: Compiler error view produced when verification policies reject inconsistent CommonSpec source.

heavier adoption cost is instead model creation and maintenance. A model developer must capture the document family as object types, attributes, relation rules, generated views, output styles and SQL verification policies, and then keep that model aligned with the project's review process as it evolves. In the PORTAS pilot, engineers already familiar with Markdown adopted the authoring layer during ongoing work; however, neither onboarding effort nor model-development effort has been measured for the generic release.

The paper does not predict broader industrial adoption. The (Section 7) describes one program's path, the prescribed DOCX deliverables and review gates stayed unchanged, rollout ran alongside the existing process, and a second team later adopted the workflow. SpecCompiler generalizes this path, but whether if it transfers past the distribution and modeling barriers above is for independent adoption and evaluation to show.

The default model is intended to be a stable extension base, not a domain model. It provides reusable technical-documentation features but does not define requirements objects, traceability relations or structural verification rules for a regulated process. The `sw_docs` model demonstrates such a domain overlay but should be read as a showcase and starting point rather than a production model. A roadmap item is therefore to provide canonical, documented models for common documentation families, so new projects can configure and extend a maintained baseline instead of creating a new model from scratch.

The verification is limited to what can be expressed as structural invariants over Spec-IR — required attributes or objects and their cardinalities, datatype casts and enumeration domains, identifier uniqueness and resolution, referential integrity of typed relations and links coverage — and not to properties that depend on the semantic content of the specification, such as consistency between abstraction levels or between requirements and behavior descriptions. Successful compilation is therefore a strict bounded guarantee, it establishes conformance of the compiled facts to the active model and policies, not completeness of stakeholder intent, feasibility, or the semantic adequacy of a trace link, which remain subjects of human review. A research direction opened by Spec-IR is semantic reasoning over the compiled graph. Language models and structured reasoners can be applied to the typed relations rather than raw text, extending structural toward semantic verification.

The immediate roadmap has three integrations, importing code symbols through Language Server Protocol servers for bidirectional code–specification traceability; a CommonSpec language server bringing completion, diagnostics and navigation over the compiled graph into the editor; and ReqIF interoperability so CommonSpec projects can exchange typed structures and review evidence with RMS environments. Beyond these, an independent evaluation of the generic infrastructure, including a user study with document authors and model developers remains future work.

6 Related Tools

The closest neighbors — Doorstop [1], StrictDoc [15] and Sphinx-Needs [20] — share SpecCompiler's lightweight, version-controllable approach to requirements traceability. Table 2 compares the capabilities documented for these versions and SpecCompiler 0.2 accounting architecture and outputs.

The distinctive design choice in SpecCompiler is to make a persistent typed relational IR the substrate for validation, derived content and publication, and to express engineering invariants as SQL queries over that IR rather than as tool-specific validator code. This keeps the verification language standard SQL, makes diagnostics inspectable by direct IR queries, and lets the same relation rows drive verification, trace matrices and emitted artifacts. SpecCompiler is also the only compared tool whose outputs include DOCX that can adapt to prescribed corporate templates.

7 Origin and Industrial Evaluation

SpecCompiler grew out of PORTAS, a hardcoded predecessor deployed in a defense-aerospace software program where teams produced software requirements specifications (SRS), software design

Table 2: Comparison of Doorstop, StrictDoc, Sphinx-Needs and SpecCompiler

Tool	Authoring/model	Validation extension	Build/data model	Trace views	Outputs/exports
Doorstop 3.2	YAML or Markdown with front matter; extended attributes	Built-in integrity checks; Python hook scripts	Source-backed object tree	Document hierarchy and trace matrix	Text, Markdown, HTML, LaTeX; PDF via LaTeX
StrictDoc 0.27.0	SDoc or Markdown; custom typed grammars	Grammar and graph-integrity checks	In-memory project tree; JSON export	Trace, matrix and coverage views	HTML, RST, Markdown, PDF, JSON, Excel, ReqIF
Sphinx-Needs 8.3.0	Sphinx need directives; configured types, fields and links	Typed fields; schema and constraint rules; Python extensions	Optional versioned `needs.json` export	Filterable tables, lists, diagrams and charts	Sphinx HTML and LaTeX/PDF; JSON
SpecCompiler 0.2	CommonSpec; model-loaded types, attributes, relations and views	Type and integrity checks; model-registered SQL policies	Persistent relational SQLite Spec-IR used by every phase	Model-defined IR-backed matrices and views	Model-customized DOCX, HTML, TeX or PDF via Pandoc

documents (SDD) and software verification cases (SVC) under DO-178C constraints and exchanged them as DOCX across an organizational boundary. The constraint was to keep producing the exact Word deliverables the formal review process expected; the conventions that later became CommonSpec emerged from that use.

In production, PORTAS generated the formal documents for two configuration items without changing templates, review gates or checklists. The evaluation used the program's inspection records and a questionnaire with five engineers who used the tool or reviewed its output. SRS and SVC records were compared with three traditional Word-based configuration items; SDD, though in production use, was outside the review gate. The strongest structural categories moved in the expected direction: SVC traceability remarks were 0–2% in PORTAS artifacts versus 5–12% in the baseline, and SRS ports-and-data remarks (a category involving links to project data definitions) were 11% versus 15–22%.

These checklist categories mix compiler-detectable structure with semantic judgment. A VC that traces to no requirement and a VC that traces to an existing but inadequate requirement fall under the same category. SRS traceability remarks stayed in the same range across all five configuration items. The system requirements they reference were kept outside PORTAS, so the compiler had no facts to check. This negative result sharpens rather than weakens the interpretation since improvement concentrates exactly where the referenced objects were known to the compiler, which disfavors team and process confounds. The comparisons remain descriptive — two compiled and three non-random baseline configuration items of heterogeneous scale and team composition support neither significance tests nor a formal causal claim — but the observed pattern is the one the mechanism predicts.

In the most comparable pair, mean inspection effort per remark — a proxy available in the existing review data, not a standalone productivity measure — fell from 28.6 to 14.2 minutes in SVC and from 25.0 to 12.6 in SRS; triangulated with the questionnaire, where reduced review effort appeared consistently, the data point in the same direction.

Participants said the shift "changed the job from writing documents to programming documents" and that "the tool ensures the consistency of the artifacts." These results support the compiled-specification approach and motivated the generic, model-driven infrastructure presented here; they do not transfer automatically, and the generic implementation has not been independently evaluated in production.

8 Conclusion

SpecCompiler brings compiler-style structural verification to Docs-as-Code specifications, CommonSpec sources are lowered into a persistent relational IR, SQL policies report violations as source-located diagnostics, and the same compiled facts drive traceability views and customizable deliverables. Evidence from the industrial predecessor supports the pattern; the generic release is so far only exercised internally. The project dogfoods the compiler on its own SRS, SDD and SVC, and the `sw_docs` overlay recreates the PORTAS flow on public content. A separate ABNT model renders the first author's master's dissertation as editable DOCX under the standard's layout and typography rules. The automated test suite is 98 Markdown-driven end-to-end cases span persistence to OOXML checks.

Those exercising the model's policy views pursue, operationally rather than as a theorem, the guarantee that well-typed ~~programs~~ specifications cannot go wrong [13], a mutation run probes their adequacy. Six operator families over the 25 bundled SQL policy views yielded 193 mutants, all detected by the oracle.

Artifact Availability

SpecCompiler and Spec-IR are released under the Apache License 2.0 at https://github.com/SpecIR/SpecCompiler, with the manual at https://specir.github.io/SpecCompiler/manual. The separate ABNT model is at https://github.com/SpecIR/specc-abnt; the demo video at https://doi.org/10.6084/m9.figshare.32400411.

Acknowledgments

The authors thank Jefferson Ferreira, Marcello Abreu, Eric Takada, and Christoffer Nylén, whose support made possible the development and industrial evaluation of PORTAS within the Gripen E/F Saab–Embraer partnership. This paper was authored in CommonSpec and compiled by SpecCompiler, AI assistants (Claude Code and OpenAI Codex) helped scaffold and draft it.

References

[1] Jace Browning and Robert Adams. 2014. Doorstop: Text-Based Requirements Management Using Version Control. *Journal of Software Engineering and Applications* 7, 3 (2014), 187–194. doi:10.4236/jsea.2014.73020

[2] Juan M. Carrillo-de Gea, Christof Ebert, Mohamed Hosni, Aurora Vizcaíno, Joaquín Nicolás, and José L. Fernández-Alemán. 2024. Tools for Requirements Engineering. *IEEE Software* (2024). doi:10.1109/MS.2024.3385466

[3] Keith L. Clark. 1978. Negation as Failure. *Logic and Data Bases* (1978), 293–322.

[4] Tiago Dionizio. 2024. LuaSQLite3 0.9.6: A SQLite3 Binding for Lua. Retrieved July 20, 2026 from https://lua.sqlite.org/home/timeline?r=v0.9.6

[5] Christof Ebert and Michael Jastram. 2012. ReqIF: Seamless Requirements Interchange Format between Business Partners. *IEEE Software* (2012).

[6] David Goodger. 2002. reStructuredText: Markup Syntax and Parser Component of Docutils. Retrieved July 20, 2026 from https://docutils.sourceforge.io/rst.html

[7] Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mäder. 2012. Traceability Fundamentals. In *Software and Systems Traceability*. Springer, London, 3–22.

[8] John Gruber. 2004. Markdown. Retrieved July 20, 2026 from https://daringfireball.net/projects/markdown/

[9] D. Richard Hipp. 2000. SQLite. Retrieved July 20, 2026 from https://www.sqlite.org/

[10] IBM Corporation. n.d.. IBM Engineering Requirements Management DOORS. Retrieved July 20, 2026 from https://www.ibm.com/products/requirements-management

[11] John MacFarlane. 2006. Pandoc: A Universal Document Converter. Retrieved July 20, 2026 from https://pandoc.org/

[12] John MacFarlane. 2024. CommonMark Spec, Version 0.31.2. Retrieved July 20, 2026 from https://spec.commonmark.org/0.31.2/

[13] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375.

[14] Object Management Group. 2016. Requirements Interchange Format (ReqIF). https://www.omg.org/spec/ReqIF/

[15] Stanislav Pankevich and Maryna Balioura. 2026. StrictDoc 0.27.0 Release. Retrieved July 20, 2026 from https://github.com/strictdoc-project/strictdoc/releases/tag/0.27.0

[16] Raymond Reiter. 1978. On Closed World Data Bases. *Logic and Data Bases* (1978), 55–76.

[17] RTCA. 2011. DO-178C: Software Considerations in Airborne Systems and Equipment Certification.

[18] Siemens Digital Industries Software. n.d.. Polarion ALM. Retrieved July 20, 2026 from https://polarion.plm.automation.siemens.com/

[19] The Luvit Authors. 2024. luv 1.48.0-2: libuv Bindings for Lua. Retrieved July 20, 2026 from https://github.com/luvit/luv/releases/tag/1.48.0-2

[20] Useblocks team. 2026. Sphinx-Needs 8.3.0 Documentation. Retrieved July 20, 2026 from https://sphinx-needs.readthedocs.io/en/8.3.0/

[21] Writerside. n.d.. Docs as Code. Retrieved July 20, 2026 from https://www.jetbrains.com/help/writerside/docs-as-code.html