Substrait validator overview

As of version 0.0.9, November 1st 2022

Topics

- Goals and non-goals
- Notable design choices
- Implementation overview
- Build & release quirks

Why have a validator? (1/2)

- Substrait specification (still) uses loose language to make it easy to add or change features
- Loose language is open to (mis)interpretation
- Inconsistent Substrait implementations across vendors = SQL dialects with extra steps

Why have a validator? (2/2)

- Need a reference implementation
  - Full consumer?
    - ideal, but a lot of work
  - Pure validation?
    - not enough: plan validity != correct interpretation of the plan
  - Validation + generated documentation (“explain” output)

Original goals (1/4)

- Validate with extreme prejudice
  - Indicate when neither validity or invalidity can be proven
  - Be as pedantic as possible by default
  - Implement the *complete* specification (i.e. website), as written, without bias
    - Including things *technically* legal (e.g. crazy function/type names like “…” or the empty string)
    - If something requires an insane implementation in the validator, the *spec* should be updated, not the validator

Original goals (2/4)

- Intelligent diagnostic system
  - Error recovery: don’t stop after the first error
  - Source information: track where diagnostics come from
  - Configurability: allow severities to be customized
- “Explain” output
  - Data type/schema annotation
  - Generated descriptions of how instances of protobuf message should be interpreted

Original goals (3/4)

- Multi-version support
  - Continue supporting deprecations until they are completely removed via a breaking change
  - Check that no features from beyond a given version are used (not really implemented yet)
- Easy-to-understand validation logic
  - Reference implementation use case
  - Maintenance should be easy, to reduce the burden of tracking the specification

Original goals (4/4)

- Easy to use
  - Batteries-included, cross-platform distribution so users can get started quickly
  - Wide variety of input formats (binary, JSON, YAML, Saul's JDOT, ...)
  - Wide variety of output formats (exit code, user-friendly HTML, machine-friendly protobuf, ...)
  - Various entry points (CLI, web browser, cross-language C API, ...)

Feature creep goals

- Conversion between plan representations via CLI (without validation)
  - The conversion logic is needed anyway
- Parsing YAML extensions into a more machine-friendly protobuf format
  - The validator needs to fully understand the extensions in order to validate them anyway

Non-goals (1/2)

- Performance & memory usage
  - Sacrificed in favor of code readability
  - Only intended to be used during testing and debugging
  - For production use, consumers will need their own validation anyway, to deal with:
    - Supported advanced extensions
    - Incomplete core Substrait support
    - Workarounds for producer bugs

Non-goals (2/2)

- Hardening against malicious actors
  - Extension URI resolution in general requires downloading and parsing arbitrary YAML
  - DoS attacks would undoubtedly be possible due to performance being a non-goal

Notable design choices (1/9)

- Core library programming language: Rust
  + Very hard to introduce bugs due to compile-time safety guarantees *[validate with extreme prejudice]*
  + Cross-platform compiled language with first-class C binding support *[library usage from any language]*
  + Fairly high abstraction level *[easy-to-understand validation logic]*
  + Procedural macro support for hiding boilerplate code *[easy-to-understand validation logic]*
  + Webassembly target *[browser-based entry point]*

Notable design choices (2/9)

- Core protobuf support: prost
  - \+ Generates idiomatic Rust types and binary serdes from protobuf specification *[easy-to-understand validation logic]*
  - − Lacks introspection
    - Need procedural macros to generate boilerplate traversal code
  - − Lacks proto3 JSON support
    - Third party efforts to add this exist, but are not yet mature

Notable design choices (3/9)

- Support for protobuf JSON: Python + official protobuf library
  - \+ Correct implementation by definition *[validate with extreme prejudice]*
  - \+ Option to fall back to pure-python implementation when libprotobuf throws a tantrum *[cross-platform support]*
  - – Requires Python

Notable design choices (4/9)

- Rust/Python interoperability: Python native extension with PyO3 + maturin
  - \+ Don’t need to embed an interpreter
  - ~ maturin automates the entire wheel generation process in CI
  - – Main entry point is now in Python rather than Rust
  - – Can’t use JSON-based formats using the Rust or C API

Notable design choices (5/9)

- CLI implementation: Python
  - \+ JSON-based formats are available from the CLI *[ease of use]*
- Test framework: custom Python + Rust logic
  - \+ Can write test cases in JSON-based format vs. binary serializations *[ease of maintenance]*
  - – Language boundary makes debugging failing test cases difficult
  - – Learning curve of a custom framework

Notable design choices (6/9)

- Distribution: pip/PyPI
  - \+ PyPI wheel format is precompiled, yet cross-platform and free of native dependencies *[ease of use, cross-platform]*
  - \+ Can fall back to sdist if no compatible wheel is available *[cross-platform]*
  - \+ Users are likely to already be familiar with pip *[ease of use]*

Notable design choices (7/9)

- Simple extension support: serde-yaml, jsonschema, serde-json
  - \+ serde-yaml: idiomatic wrapper around libyaml *[easy-to-understand validation logic, validate with extreme prejudice]*
  - \+ jsonschema: implements validation for the schema published by Substrait *[validate with extreme prejudice]*
  - – serde-json: needed only to connect the YAML deserializer to jsonschema

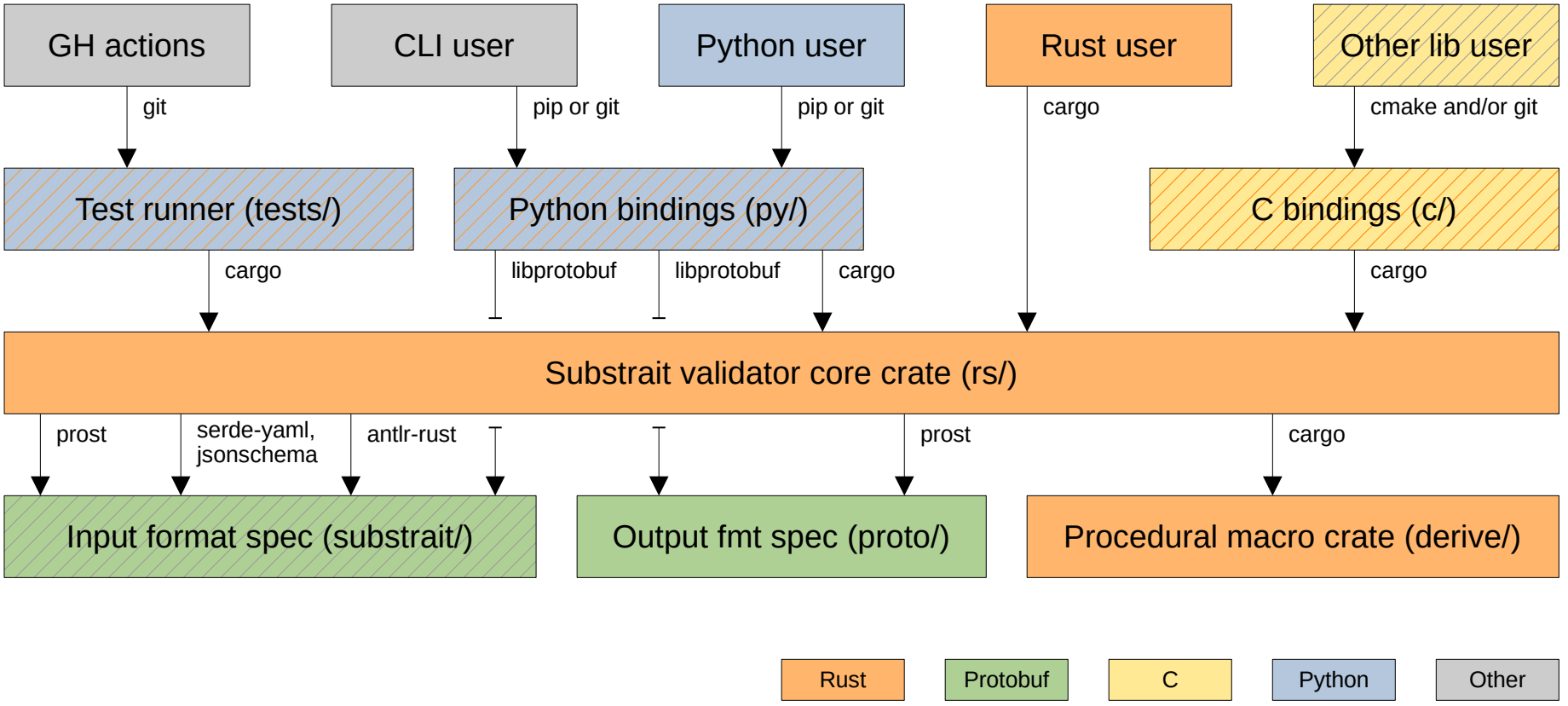
Notable design choices (8/9)

- Type expression syntax support: antlr-rust
  + substrait-java already uses ANTLR
  + ANTLR is good at auto-generating recovery and disambiguation from just EBNF input *[easy-to-understand validation logic, error recovery]*
    - No EBNF-based alternatives seem to exist for Rust; the more idiomatic parser combinator crates would duplicate the language syntax specification
  – Immature and non-idiomatic Rust

Notable design choices (9/9)

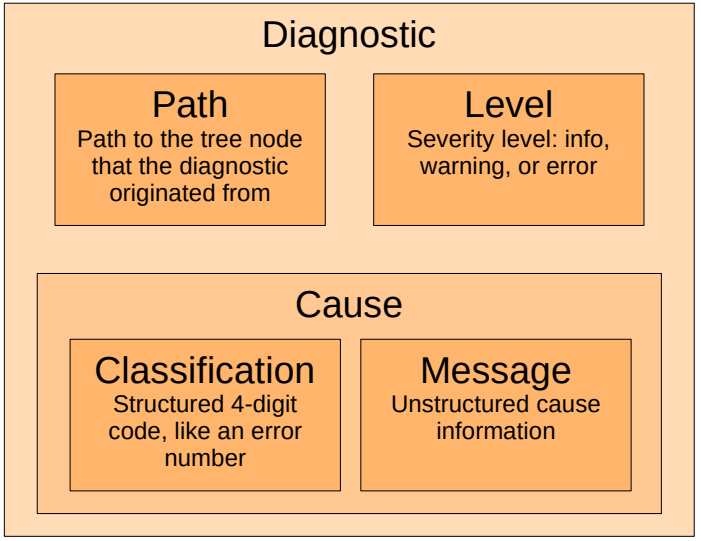
- YAML URI resolution: curl (Rust) and urllib (Python)
  - \+ curl supports all kinds of URI formats and protocols out of the box *[validate with extreme prejudice]*
  - ~ The curl crate is not self-contained, making it incompatible with wheel distribution; fall back to Python’s internal urllib library
  - ~ Allow users to specify URI overrides when expected in-plan URIs cannot be resolved with curl or urllib

Project structure



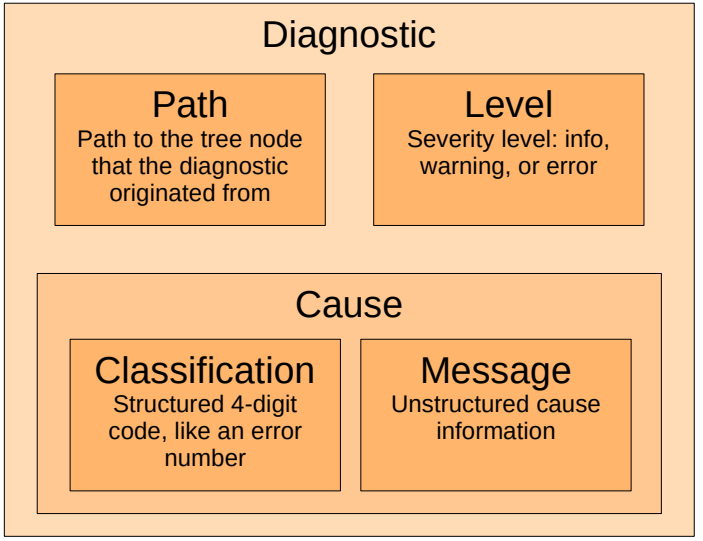
Diagnostics (1/2)

- Primary validation output
- Validation verdict is derived directly from the most severe diagnostic:
  - Info -> valid
  - Error -> invalid
  - Warning -> no proof either way



Diagnostics (2/2)

- The “cause” consists of a structured “classification” (basically an error code) and an unstructured message
- Severity levels for a particular classification can be clamped to a certain range through configuration
  - Can affect the validation verdict
  - Original level is also tracked

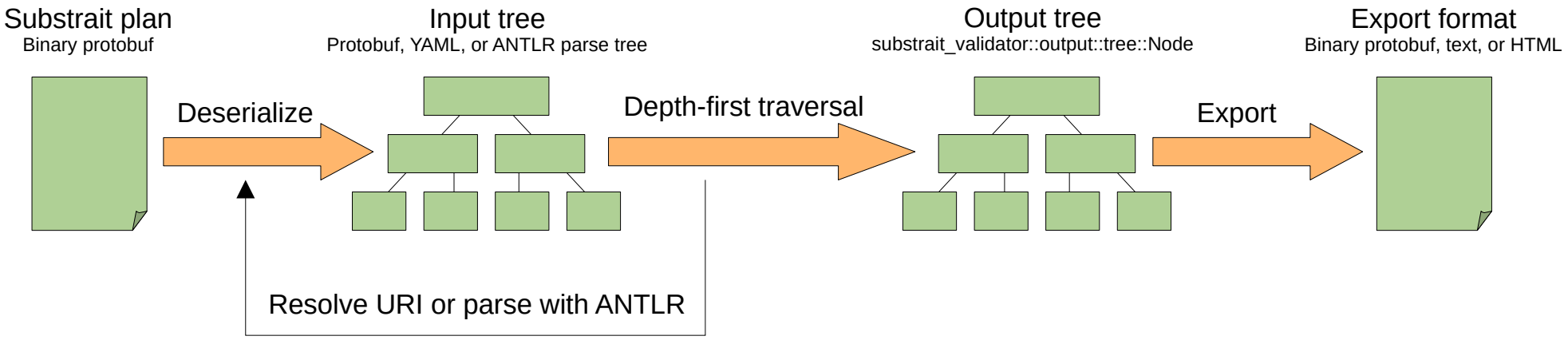


“Unresolved” values

- Special enum variant that appears all over the place
- Indicates that the value of something is unknown due to previous errors: error recovery
- Similar to null or NaN: use of unresolved *silently* yields unresolved
- If no unresolved variant exists, use whatever is least likely to result in spurious diagnostics later

Traversal (1/5)

- Validation is abstracted as a tree transformation
- Structure is generally preserved
- Input and output node types differ

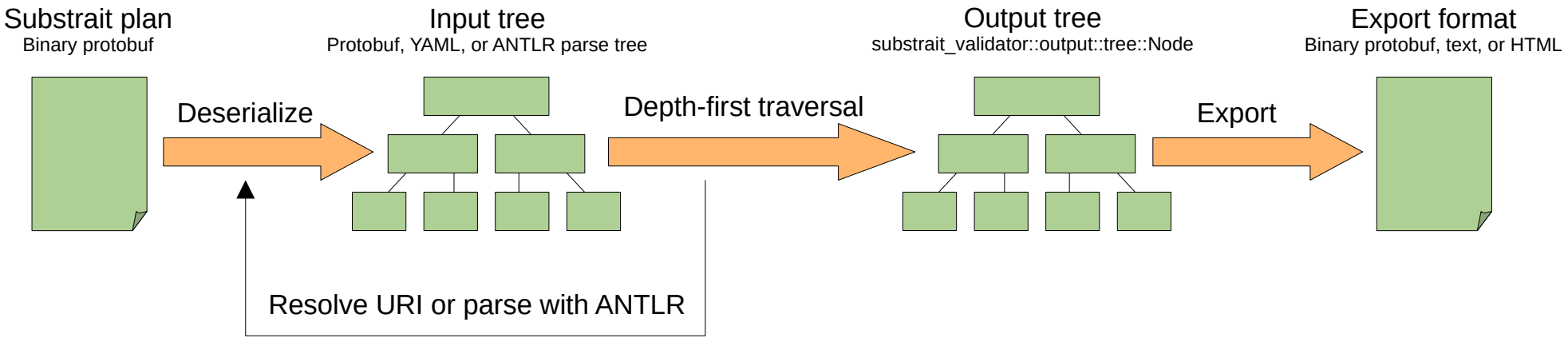


Traversal (2/5)

- Node handling is done by “parse functions” of the form:

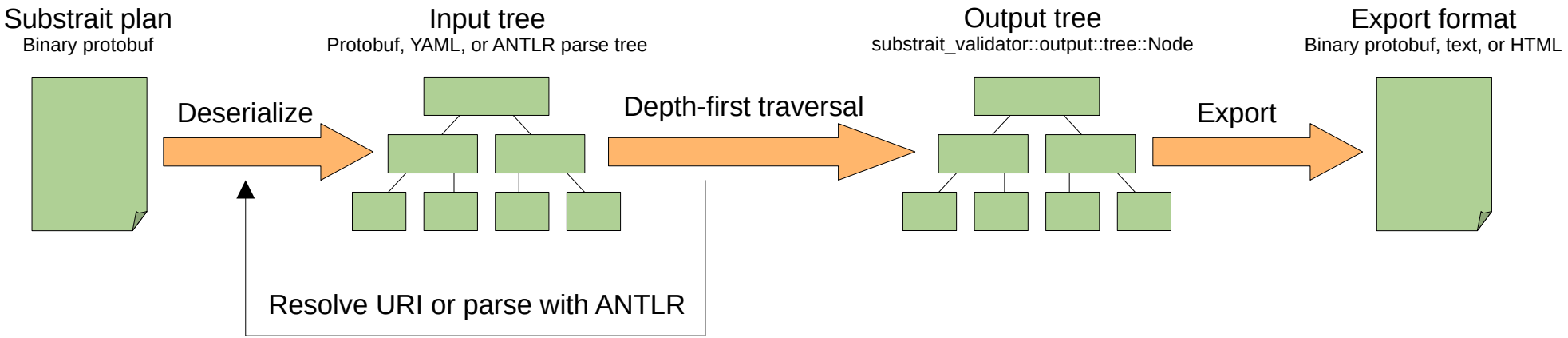
  parse(x: &Input, y: &mut Context, ...) -> Result<...>

- x is the input node, and y points to the automatically generated output node, configuration, and state tracking structures



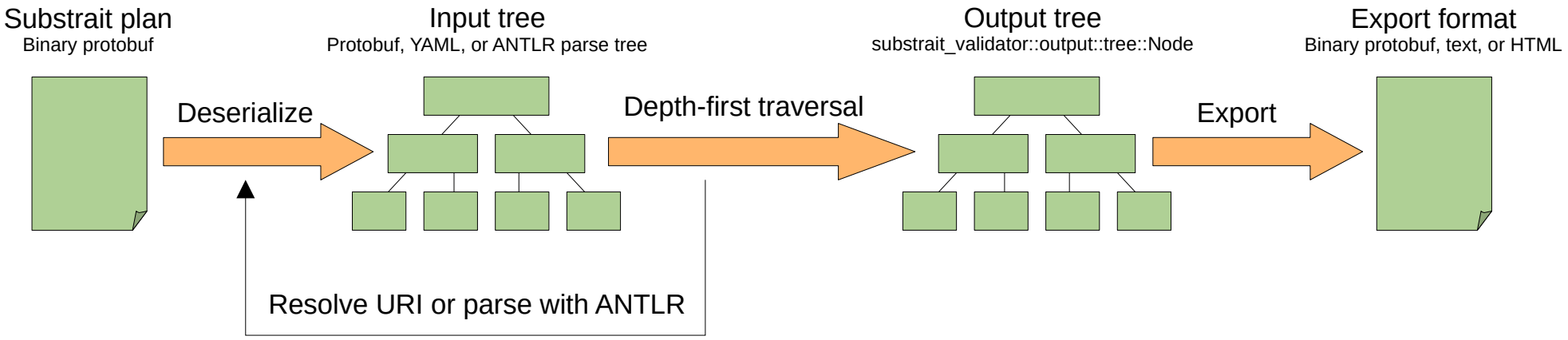
Traversal (3/5)

- Parse functions must:
  - Traverse each child node exactly once (in some order) by calling a child parser via a traversal macro
  - Annotate the output node with validation information (including diagnostics)
  - Update context/state information in Context as needed



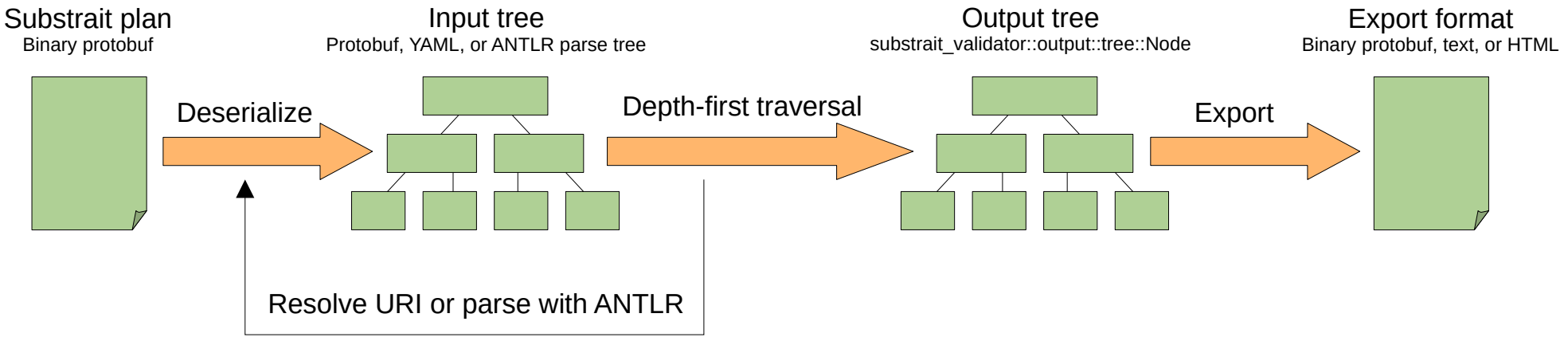
Traversal (4/5)

- Boilerplate traversal code sanity-checks the parse functions
  - Child not traversed -> emit not-yet-implemented warning, and traverse subtree with fallback logic
  - Child traversed twice -> panic (treated as a bug)



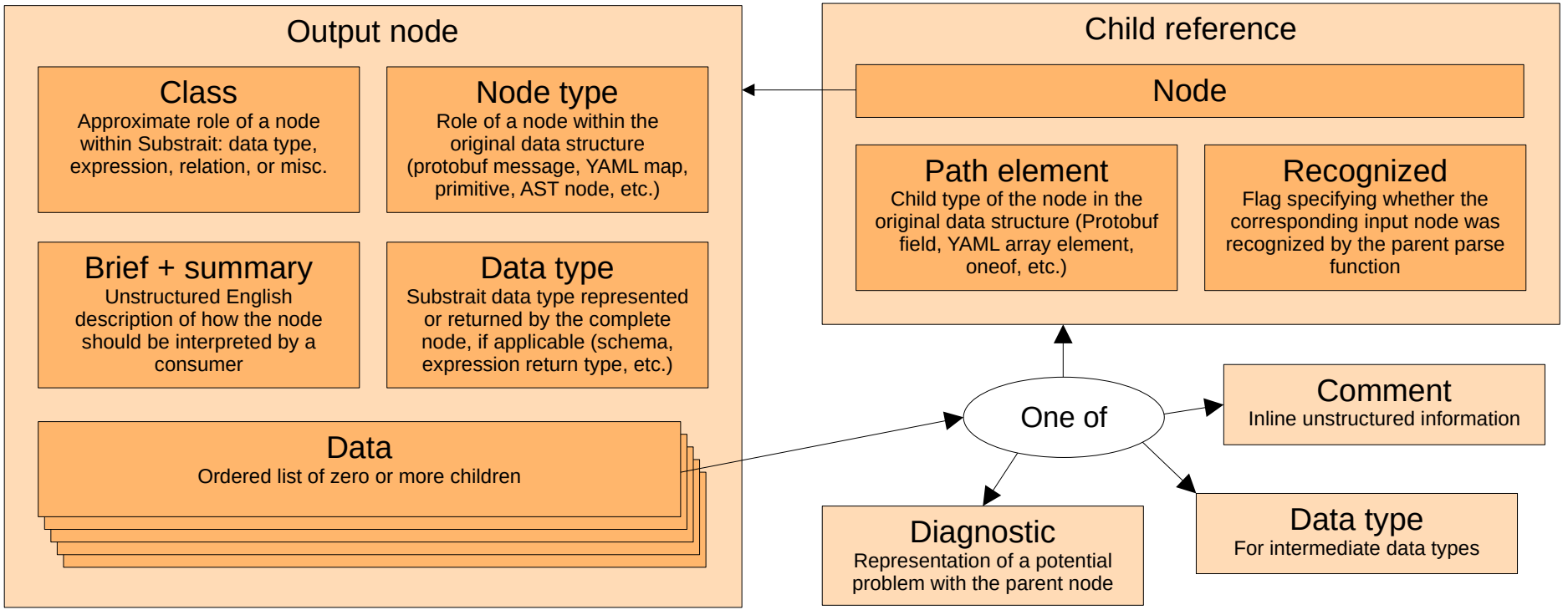
Traversal (5/5)

- Boilerplate traversal code relies on derive crate
  - One big pile of spaghetti, but…
  - Unlikely to need maintenance, unless prost or Rust itself changes significantly



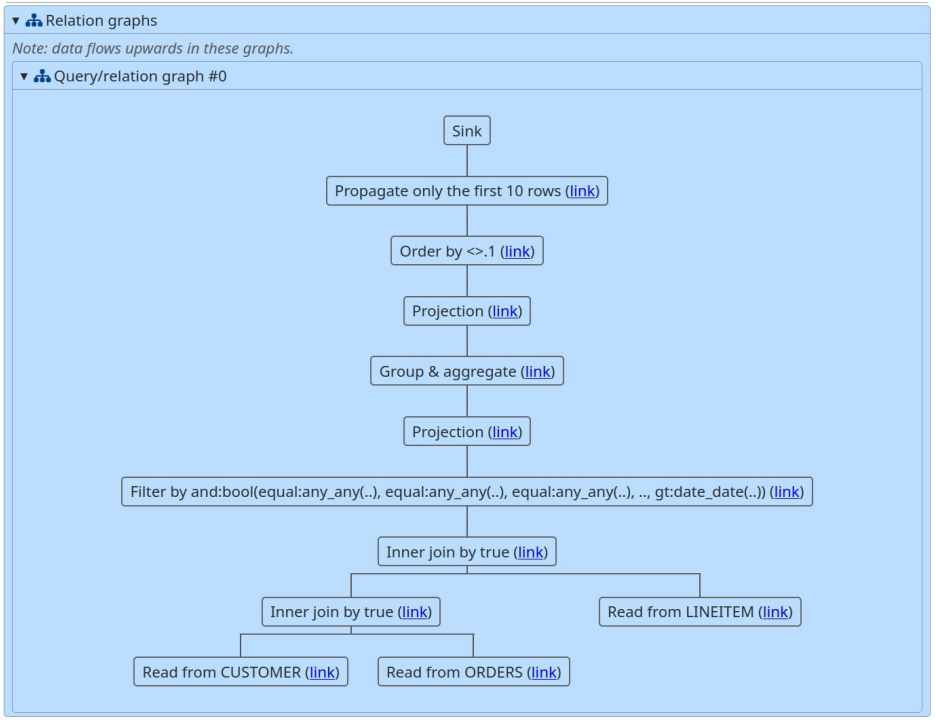
Output tree nodes

- One type to rule them all: context-free traversal by export logic

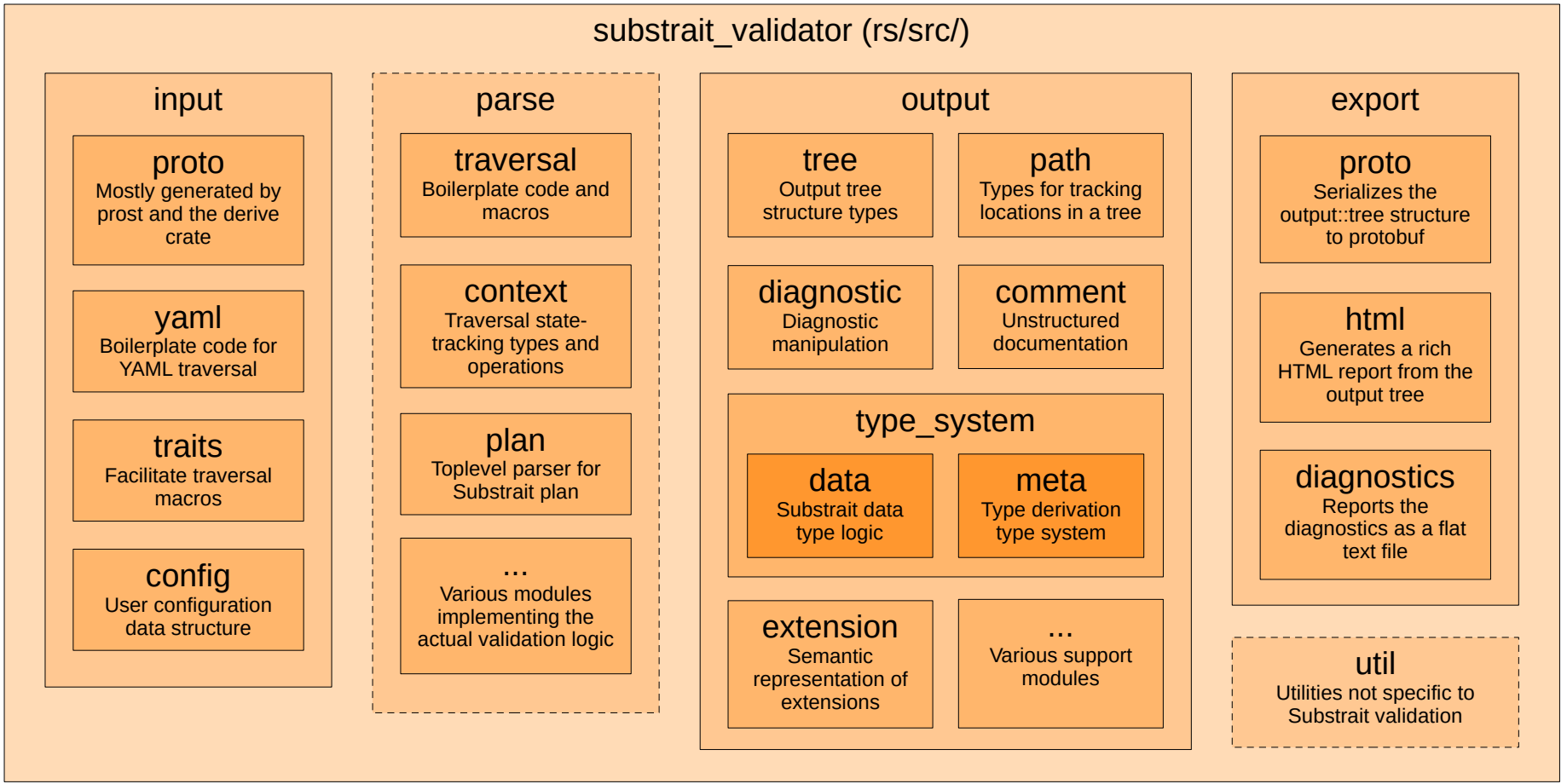


HTML export format

- Self-contained human-readable HTML representation of the output tree



Core library structure



Testing

- Unit tests are admittedly scarce
- Majority of testing is done by the integration test runner
  - Test descriptions consist of YAML files that mimic the protobuf JSON DOM, but include test commands and support inlining YAML extensions
  - Python portion of the test runner parses these files into a simpler format, which the Rust portion then executes
  - See README.md for more info

```
name: rel-root-with-names
plan:
  __test: # check that the error level is Info
  - level: i
  version: { producer: validator-test }
  relations:
  - root:
      names:
      - a
      - b
      input:
        read:
          baseSchema:
            names: [x, y]
            struct:
              nullability: NULLABILITY_REQUIRED
              types:
              - string: { nullability: NULLABILITY_REQUIRED }
              - i32: { nullability: NULLABILITY_NULLABLE }
          namedTable:
            names:
            - test
    __test: # Check that the derived schema is as expected
    - type: "NSTRUCT<a: string, b: i32?>"
```

Versioning (1/2)

- Substrait validator version is unrelated from Substrait version
  - Support matrix tracked in main README.md
- PyPI and crate versions are synchronized and released simultaneously
- Versioning scheme is semver-compliant, using cargo semantics for 0.x.y regime (x++ = breaking, y++ = feature/patch)
- Currently in 0.0.x regime

Versioning (2/2)

Change	*Release type*
Substrait breaking protobuf/spec change	Breaking release
Substrait protobuf feature addition	Feature release
Substrait breaking core extension change	Probably no release needed
Other Substrait changes	Probably no release needed
Fix of inconsistency between Substrait and validator	Patch release
Breaking validator interface change	Breaking release
Non-breaking validator interface change	Feature release
Validator-only bugfixes	Patch release
Semantic release implemented for validator	Release 0.1.0
Validator implements all of Substrait	Release 1.0.0

See also https://github.com/substrait-io/substrait-validator/issues/4

Release automation

- Currently:
  - Version increment is manual
  - Version number synchronization is semi-automatic via the ci/version.py CLI script
  - Release is automated using GH actions, but triggered manually by a tag push to main
- Idea is to mimic substrait main repo eventually for consistency (conventional commit + semantic release)

Notable build quirks (1/2)

- Pre-build code generation
  - Several files must be synchronized from the substrait/ submodule to the py/ and rs/ trees
  - Protobuf code generation is necessary for Python
  - Ugly logic needed to distinguish between builds from git and builds from sdist packages
  - Usually automatic, but no reliable hook exists
    - see README.md files in py/ and rs/ for more information
  - Sometimes a “git clean -fdx” may be needed

Notable build quirks (2/2)

- Derive crate release dependency
  - Cargo refuses to release if any dependency is not in the public index yet
  - Index updates are asynchronous to completion of the “cargo release” command
  - Blocking handled by ci/crates-io-wait.py script
  - Might actually be fixed in cargo now;

https://github.com/rust-lang/cargo/issues/9507

Where to find more information

- Important directories have dedicated README.md files
- RELEASE.md documents the release process
- CONTRIBUTING.md is the entry point for new contributors
- All Rust modules, functions, types, and fields have docstrings
  - cargo doc --all-features --document-private-items --open