Text2System

T2S Specification

Text Defined System (T2S) Language Specification

Author Gabor Soos

Version v2026.09.06-1040
Language en
Status RC5

Contents

0. Introduction

Text2System (T2S) is a Text-Defined Systems Engineering language for describing software and software-intensive systems using a single human-readable, machine-readable plain-text source.

A T2S System Model completely describes a software system from architectural definition through packaging and intended deployment while remaining independent of implementation language, runtime technology, and deployment technology.

T2S defines what a system is.

Programming languages define how the system behaves.

Deployment technologies define how the system is realized and operated.

Unlike traditional Model-Based Systems Engineering, T2S treats text as the canonical definition of the system. Documentation, architecture views, implementation skeletons, interface definitions, validation reports, tests, deployment artifacts, diagrams, and other engineering artifacts are not part of the language itself; they are projections of the Canonical System Model.

This document is the normative specification of T2S Language 1.0.

T2S combines:

- T2D narrative for human-readable documentation.
- T2S semantic declarations for machine-readable system definition.

A conforming T2S processor performs the following normative stages:

- Region scanning.
- Narrative parsing into a Narrative AST.
- Semantic parsing into a Semantic AST.
- Semantic validation.
- Construction of the Canonical System Model.

The Canonical System Model is the only canonical semantic representation defined by the language.

Projectors transform the Canonical System Model into implementation languages, documentation, interface definitions, package descriptions, deployment descriptions, tests, diagrams, and other engineering artifacts.

T2S 1.0 defines three reference projectors:

- Rust
- Zig
- Python

Reference projectors are part of the reference toolchain and are not part of the language semantics. The portable meaning of a T2S source shall never depend on a particular projector.

The normative processing pipeline is:

T2S Source
↓
Region Scan
↓
Narrative AST + Semantic AST
↓
Canonical System Model Construction
↓
Semantic Validation
↓
Validated Canonical System Model
↓
Projector
↓
Projection Model
↓
Renderer
↓
Artifacts

Additional projectors may be developed independently without modifying the T2S language specification.

T2S defines the system from design through distribution and intended deployment. Implementation languages, build systems, package formats, deployment technologies, and operational tooling realize that definition but do not redefine it.

T2S requirement declaration

```
requirement:
  id: T2S-GEN-001
  statement: A T2S source shall be readable as a human-oriented system specification.
```

T2S requirement declaration

```
requirement:
  id: T2S-GEN-002
  statement: A T2S source shall define deterministic machine-readable system semantics.
```

T2S requirement declaration

```
requirement:
  id: T2S-GEN-003
  statement: T2S shall define the system independently of algorithm implementation.
```

T2S requirement declaration

```
requirement:
  id: T2S-GEN-004
  statement: A core T2S construct shall describe a portable property of a system rather than a feature of a particular implementatio
```

T2S requirement declaration

```
requirement:
  id: T2S-GEN-005
  statement: Package and deployment descriptions are part of the System Model and shall remain independent of deployment technologie
```

T2S requirement declaration

```
requirement:
  id: T2S-GEN-006
  statement: Projectors shall preserve the portable semantics of the Canonical System Model.
```

T2S requirement declaration

```
requirement:
  id: T2S-GEN-007
  statement: A Realization Profile may refine realization but shall not alter the portable semantics of the Canonical System Model.
```

1. Scope

T2S 1.0 defines a portable system description and implementation-contract language.

T2S 1.0 is intended to support:

- Canonical implementation-independent system definitions.
- Software and software-intensive system architecture.
- Architectural decomposition into systems and modules.
- Technology-independent interfaces and callable operations.
- Portable data contracts.
- Explicit error contracts.
- Synchronous, asynchronous, and one-way invocation semantics.
- Requirement and test traceability.
- Packaging and deployment descriptions.
- Projection to Rust, Zig, Python, C ABI, documentation, and diagrams through projectors.

T2S 1.0 does not define:

- Algorithm bodies.
- General-purpose expressions.
- Language-specific generics.
- Rust lifetimes or traits as core semantics.
- Zig comptime constructs as core semantics.
- C preprocessor macros as core semantics.
- Concrete thread, lock, allocator, or async-runtime implementations.

2. Language Composition

Text2System is composed of two independently parseable source forms.

T2D narrative source describes the system for human readers.

T2S semantic source defines the formal system model.

A mixed T2S document contains both forms in one source file.

T2S relationship declaration

```
relationship:
  subject: Text2System
  predicate: contains
  object: Text2Doc
```

T2S relationship declaration

```
relationship:
  subject: Text2System
  predicate: contains
  object: T2SSemanticSource
```

T2S requirement declaration

```
requirement:
  id: T2S-LANG-001
  statement: The T2D parser shall parse narrative regions independently of the T2S semantic parser.
```

T2S requirement declaration

```
requirement:
  id: T2S-LANG-002
  statement: The T2S semantic parser shall parse standalone semantic source independently of a T2D document.
```

3. Source Regions

The T2S cover follows the existing T2D cover rules.

Within the document body, an unindented non-empty line belongs to T2D narrative source.

Within the document body, a non-empty line beginning with two SPACE (U+0020) characters belongs to T2S semantic source.

The two leading spaces form the semantic envelope. The envelope is not part of the semantic source and shall be removed before semantic parsing.

A semantic region continues while subsequent non-empty lines begin with at least two spaces. Blank lines may occur within a semantic region. The region ends before the next non-empty body line beginning at column one.

T2S requirement declaration

```
requirement:
  id: T2S-SRC-001
  statement: In the T2D body, a non-empty line beginning at column one shall be interpreted as T2D narrative source.
```

T2S requirement declaration

```
requirement:
  id: T2S-SRC-002
  statement: In the T2D body, a non-empty line beginning with two SPACE (U+0020) characters shall be interpreted as T2S semantic sou
```

T2S requirement declaration

```
requirement:
  id: T2S-SRC-003
  statement: The region scanner shall remove exactly two leading two SPACE (U+0020) characters from every semantic source line befor
```

T2S requirement declaration

```
requirement:
  id: T2S-SRC-004
  statement: Leading spaces in the T2D cover shall retain their existing T2D meaning and shall not introduce T2S semantics.
```

4. Semantic Notation

T2S semantic source uses the T2S YAML Profile.

The T2S YAML Profile is a deterministic subset of YAML used exclusively as the concrete notation of T2S semantic declarations.

The T2S YAML Profile defines the concrete syntax of semantic source. YAML itself does not define T2S semantics.

Allowed notation:

- mappings
- sequences
- string scalars
- decimal integer scalars
- decimal floating-point scalars
- boolean scalars
- null scalars
- folded block strings (>)
- literal block strings (|)
- comments

Mapping keys shall be strings.

Duplicate mapping keys are forbidden.

Indentation shall use SPACE (U+0020) characters only. Tab characters shall not be used for indentation.

The only boolean scalar spellings are:

- true
- false

The only null scalar spelling is:

- null

Integer scalars may use:

- decimal notation
- hexadecimal notation
- binary notation
- octal notation

Floating-point scalars may use:

- decimal notation
- scientific notation

NaN and infinite floating-point values are forbidden.

Single-line scalar values may be represented using ordinary YAML scalars.

Scalar values spanning multiple physical lines shall use either:

- folded block strings (>) for ordinary prose
- literal block strings (|) for verbatim text

Single-line scalar values may be represented using ordinary YAML scalars.

Multi-line plain scalars are forbidden.

Implementation-dependent scalar forms, including timestamps, alternative boolean or null spellings, and implementation-defined numeric representations, are forbidden.

Forbidden notation:

- anchors
- aliases
- merge keys
- custom tags
- executable constructors
- complex mapping keys
- duplicate mapping keys
- multiple YAML documents in one semantic compilation unit
- implementation-dependent implicit scalar conversions
- multi-line plain scalars

Comments and presentation details shall not alter T2S semantics.

Mapping order shall be preserved in the Semantic AST for deterministic processing, diagnostics, formatting, and projection.

YAML is notation only. YAML structure shall not independently define T2S semantics.

T2S requirement declaration

```
requirement:
  id: T2S-YAML-001
  statement: T2S semantic source shall conform to the T2S YAML Profile.
```

T2S requirement declaration

```
requirement:
  id: T2S-YAML-002
  statement: A conforming T2S implementation shall reject YAML features forbidden by the T2S YAML Profile.
```

T2S requirement declaration

```
requirement:
  id: T2S-YAML-003
  statement: Scalar interpretation shall be deterministic across conforming implementations.
```

T2S requirement declaration

```
requirement:
  id: T2S-YAML-004
  statement: Duplicate mapping keys shall be rejected.
```

T2S requirement declaration

```
requirement:
  id: T2S-YAML-005
  statement: Mapping order shall be preserved in the Semantic AST.
```

T2S requirement declaration

```
requirement:
  id: T2S-YAML-006
  statement: Scalar values spanning multiple physical lines shall use folded block strings (>) or literal block strings (|).
```

5. Processing and Semantic Model

A conforming T2S processor transforms a source document through a sequence of well-defined representations.

The Semantic AST records the parsed structure of the source document together with its source locations.

The Canonical System Model records implementation-independent semantics constructed from the Semantic AST. A Canonical System Model becomes valid for projection only after successful semantic validation.

A Projection Model records the target-specific semantic representation produced by a Projector. Projection Models are target dependent and are outside the core language.

T2S type declaration

```
type:
  name: T2SModelLayer
  kind: enum
  values:
  - source_ast
  - canonical_system_model
  - projection_model
```

T2S requirement declaration

```
requirement:
  id: T2S-MODEL-001
  statement: Every conforming processor shall construct a Canonical System Model from a valid Semantic AST.
```

T2S requirement declaration

```
requirement:
  id: T2S-MODEL-002
  statement: Validators shall operate exclusively on the Canonical System Model.
```

T2S requirement declaration

```
requirement:
  id: T2S-MODEL-003
  statement: A Projector shall derive one Projection Model from one Canonical System Model and one Realization Profile.
```

T2S requirement declaration

```
requirement:
  id: T2S-MODEL-004
  statement: A Projection Model shall preserve the portable semantics of the Canonical System Model.
```

T2S requirement declaration

```
requirement:
  id: T2S-MODEL-005
  statement: A Projector shall report unsupported semantics explicitly and shall not silently modify or discard portable semantics.
```

6. Core Ontology

The Canonical System Model is composed of a small, orthogonal set of fundamental semantic element kinds. These element kinds define the portable structure of a system and are independent of implementation language, runtime environment, and deployment platform.

The fundamental architectural organization of a Canonical System Model is:

System
├── Subsystem(s)
├── Module(s)
├── Package(s)
└── Deployment(s)

Module
├── Interface(s)
│ └── Function(s)
├── Type(s)
├── Constant(s)
├── Configuration(s)
├── Requirement(s)
├── Constraint(s)
├── Implementation(s)
└── Test(s)

Relationship is a first-class semantic element. Relationships connect semantic elements throughout the Canonical System Model and are not constrained by the hierarchical organization shown above.

The responsibilities of the core semantic element kinds are:

- System defines the architectural boundary of a software system.
- Module groups related semantic elements into cohesive architectural units.
- Interface defines externally visible contracts.
- Function defines callable behavior belonging to an interface.
- Type defines structured or primitive data.
- Constant defines immutable values.
- Configuration defines configurable properties.
- Requirement defines normative system requirements.
- Constraint defines semantic or architectural rules.
- Implementation defines language-specific realization bindings.
- Test defines verification artifacts.
- Package defines distribution units.
- Deployment defines runtime realization.
- Relationship defines explicit semantic connections between elements.

Documentation is represented exclusively by the Narrative AST and is not part of the Canonical System Model.

Projection concerns, including Projectors, Projection Models, and Realization Profile, are defined separately and are not part of the Core Ontology.

T2S type declaration

```
type:
  name: T2SElementKind
  kind: enum
  values:
  - system
  - module
  - interface
  - function
  - type
  - constant
  - configuration
  - requirement
  - constraint
  - implementation
  - test
  - package
  - deployment
  - relationship
```

T2S requirement declaration

```
requirement:
  id: T2S-ONTOLOGY-001
  statement: Every semantic element shall have exactly one element kind.
```

T2S requirement declaration

```
requirement:
  id: T2S-ONTOLOGY-002
  statement: Every semantic element except relationship shall define exactly one stable identifier. A relationship may omit its iden
```

T2S requirement declaration

```
requirement:
  id: T2S-ONTOLOGY-003
  statement: Every semantic element shall belong to exactly one Canonical System Model.
```

T2S requirement declaration

```
requirement:
  id: T2S-ONTOLOGY-004
  statement: The order of T2SElementKind values is descriptive only and shall not define declaration order, processing order, owners
```

7. Relationships

Relationships define explicit semantic connections between elements of the Canonical System Model.

A relationship connects one subject element to one object element through one predicate.

Relationships are first-class semantic elements. They may be validated, queried, documented, and projected independently of the elements they connect.

A relationship identifier is optional. When an id is present, it shall be a stable identifier and may be used to address the relationship explicitly. When id is omitted, the relationship is defined by its subject, predicate, and object triple and is not addressable through an identifier reference.

Example

T2S type declaration

```
type:
  name: T2SRelationshipKind
  kind: enum
  values:
  - contains
  - depends_on
  - uses
  - provides
  - implements
  - realizes
  - extends
  - specializes
  - references
  - includes
  - hosts
  - deploys
  - verifies
  - configures
  - communicates_with
  - exposes
  - binds
```

T2S requirement declaration

```
requirement:
  id: T2S-REL-001
  statement: Every relationship shall reference exactly one existing subject element.
```

T2S requirement declaration

```
requirement:
  id: T2S-REL-002
  statement: Every relationship shall reference exactly one existing object element.
```

T2S requirement declaration

```
requirement:
  id: T2S-REL-003
  statement: Every relationship shall specify exactly one predicate.
```

T2S requirement declaration

```
requirement:
  id: T2S-REL-004
  statement: A relationship predicate shall be compatible with the kinds of its subject and object elements.
```

T2S requirement declaration

```
requirement:
  id: T2S-REL-005
  statement: A relationship may omit id. When id is defined, it shall be a non-empty stable identifier unique within the Canonical S
```

T2S relationship declaration

```
relationship:
  subject: org.text2system.example.s07.system
  predicate: contains
  object: org.text2system.example.s07.module
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s07.rel.module-provides-interface
  subject: org.text2system.example.s07.module
  predicate: provides
  object: org.text2system.example.s07.interface
```

8. Semantic AST

The Semantic AST is the parsed representation of T2S semantic source.

A Semantic AST preserves the syntactic structure of semantic source before semantic interpretation and validation.

The Semantic AST represents semantic declarations, compilation-unit metadata, imports, scalar values, mapping order, sequence order, and source locations.

The Semantic AST is an implementation artifact. It is not part of the Canonical System Model and does not independently define portable system semantics.

Semantic references may remain unresolved in the Semantic AST.

A Semantic AST may contain declarations that are structurally valid but semantically invalid.

The internal representation of the Semantic AST is implementation-defined.

Processing pipeline:

T2S Semantic Source
↓
Semantic Parser
↓
Semantic AST
↓
Semantic Interpretation
↓
Canonical System Model

T2S requirement declaration

```
requirement:
  id: T2S-SOURCE-AST-001
  statement: A conforming parser shall construct a Semantic AST from valid T2S semantic source.
```

T2S requirement declaration

```
requirement:
  id: T2S-SOURCE-AST-002
  statement: Every Semantic AST node shall preserve its originating source location.
```

T2S requirement declaration

```
requirement:
  id: T2S-SOURCE-AST-003
  statement: Mapping and sequence order shall be preserved in the Semantic AST.
```

T2S requirement declaration

```
requirement:
  id: T2S-SOURCE-AST-004
  statement: The Semantic AST shall preserve compilation-unit metadata and semantic declarations.
```

T2S requirement declaration

```
requirement:
  id: T2S-SOURCE-AST-005
  statement: Semantic AST structure shall not independently define portable semantics.
```

T2S requirement declaration

```
requirement:
  id: T2S-SOURCE-AST-006
  statement: Semantic validation shall consume a Semantic AST and construct a Canonical System Model.
```

9. Canonical System Model

The Canonical System Model is the normative semantic representation defined by T2S.

The Canonical System Model defines the portable meaning of a T2S source independently of source-file organization, implementation language, runtime environment, realization profile, projector, and generated artifacts.

The Canonical System Model contains only the semantic element kinds defined by the Core Ontology.

Documentation, Semantic AST nodes, source locations, projection models, generated artifacts, and implementation-specific metadata are not part of the Canonical System Model.

Implementations may retain source locations and traceability information provided they do not alter portable semantics.

A Canonical System Model that has successfully completed semantic validation is a Validated Canonical System Model.

The semantic element kinds of the Canonical System Model are defined by the Core Ontology.

Conceptual model:

Semantic AST -> Canonical System Model -> Validated Canonical System Model

10. Compilation Units

A T2S source file is processed as a compilation unit.

A compilation unit is a source-level container for language metadata, imports, narrative, and semantic declarations.

A compilation unit exists only during source processing and is not part of the Canonical System Model.

Every semantic compilation unit shall define exactly one root schematic.

The root schematic is the primary semantic subject of the compilation unit.

Any T2S semantic element kind may serve as the root schematic.

The root schematic does not imply architectural ownership, containment, composition, or semantic precedence.

If a compilation unit contains exactly one top-level semantic declaration, that declaration is the implicit root semantic.

If a compilation unit contains multiple top-level semantic declarations, the compilation-unit metadata shall explicitly identify the root schematic.

Compilation-unit membership shall not imply semantic ownership.

Example

T2S requirement declaration

```
requirement:
  id: T2S-UNIT-001
  statement: Every semantic compilation unit shall define exactly one root semantic.
```

T2S requirement declaration

```
requirement:
  id: T2S-UNIT-002
  statement: Any T2S semantic element kind may serve as the root schematic.
```

T2S requirement declaration

```
requirement:
  id: T2S-UNIT-003
  statement: If exactly one top-level semantic declaration exists, that declaration shall be the implicit root semantic.
```

T2S requirement declaration

```
requirement:
  id: T2S-UNIT-004
  statement: If multiple top-level semantic declarations exist, compilation-unit metadata shall explicitly identify the root schemat
```

T2S requirement declaration

```
requirement:
  id: T2S-UNIT-005
  statement: Compilation-unit membership shall not imply semantic ownership.
```

T2S requirement declaration

```
requirement:
  id: T2S-UNIT-006
  statement: The declared root semantic shall resolve to a semantic element contained in or imported by the compilation unit.
```

T2S t2s declaration

```
t2s:
  language: '1.0'
  namespace: org.text2system.example.storage
  root: org.text2system.example.storage
```

T2S system declaration

```
system:
  id: org.text2system.example.storage
```

T2S module declaration

```
module:
  id: org.text2system.example.storage.core
```

T2S interface declaration

```
interface:
  id: org.text2system.example.storage.api
```

10.1 Schematic Scopes

A schematic scope is a semantic element that may contain or compose other semantic elements.

T2S 1.0 defines the following schematic scopes:

- system
- module

A schematic scope provides architectural organization independent of source files.

A schematic scope may define canonical members and composed inclusions.

members identifies elements owned by the scope.

includes identifies independently defined schematic scopes or semantic elements participating in the composed system.

Example

T2S requirement declaration

```
requirement:
  id: T2S-SCOPE-001
  statement: System and module shall be schematic scopes.
```

T2S requirement declaration

```
requirement:
  id: T2S-SCOPE-002
  statement: A schematic scope shall not be inferred from the compilation-unit structure.
```

T2S requirement declaration

```
requirement:
  id: T2S-SCOPE-003
  statement: The members relation shall define canonical ownership.
```

T2S requirement declaration

```
requirement:
  id: T2S-SCOPE-004
  statement: The includes relation shall define semantic composition.
```

T2S module declaration

```
module:
  id: org.text2system.example.s10.1.module
  members:
  - org.text2system.example.s10.1.input_record
  - org.text2system.example.s10.1.interface
```

T2S system declaration

```
system:
  id: org.text2system.example.s10.1.system
  members:
  - org.text2system.example.s10.1.subsystem
  includes:
  - org.text2system.example.s10.1.module
```

11. Identifiers and References

Every semantic element except relationship shall have exactly one stable identifier. A relationship may omit id.

A semantic element may additionally define a human-readable name.

Identifiers uniquely identify addressable semantic elements within the Canonical System Model and shall remain stable across revisions whenever practical. An anonymous relationship is identified structurally by its subject, predicate, and object triple and cannot be referenced by identifier.

Names are intended for presentation and documentation and need not be unique outside their enclosing semantic scope.

Identifiers and names are case-sensitive.

The lexical grammar of an identifier is:

[A-Za-z_][A-Za-z0-9_.-]*

References shall resolve by identifier.

Qualified names are a presentation form and shall not participate in semantic resolution.

T2S requirement declaration

```
requirement:
  id: T2S-ID-001
  statement: Every semantic element except relationship shall define exactly one identifier. A relationship may omit id.
```

T2S requirement declaration

```
requirement:
  id: T2S-ID-002
  statement: No two semantic elements within a Canonical System Model shall share the same identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-ID-003
  statement: Duplicate names within the same semantic scope shall be rejected.
```

T2S requirement declaration

```
requirement:
  id: T2S-ID-004
  statement: Every non-external semantic reference shall resolve to exactly one compatible semantic element.
```

T2S requirement declaration

```
requirement:
  id: T2S-ID-005
  statement: Identifier matching shall be case-sensitive.
```

12. System

A system is the highest-level architectural entity defined by T2S.

A system represents a coherent software or software-intensive system boundary.

A system may contain systems and other semantic elements through explicit relationships.

A system contained by another system has the contextual role of subsystem. Subsystem is not a separate semantic element kind.

A system may serve as the root declaration of a compilation unit.

Mandatory properties

Optional properties

Permitted nested elements

Permitted outgoing relationships

Permitted incoming relationships

Semantic requirements

Example

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
```

T2S optional declaration

```
optional:
  version:
    type: version
  description:
    type: string
```

T2S nested declaration

```
nested:
- system
- module
- interface
- function
- type
- constant
- configuration
- requirement
- constraint
- implementation
- test
- package
- deployment
```

T2S outgoing declaration

```
outgoing:
- contains
- uses
- depends_on
- realizes
- references
```

T2S incoming declaration

```
incoming:
- contains
- depends_on
- realizes
- references
```

T2S requirement declaration

```
requirement:
  id: T2S-SYS-001
  statement: A system shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-SYS-002
  statement: A system shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-SYS-003
  statement: A system may contain zero or more systems as subsystems through explicit contains relationships.
```

T2S requirement declaration

```
requirement:
  id: T2S-SYS-004
  statement: A system shall not contain itself directly or transitively.
```

T2S requirement declaration

```
requirement:
  id: T2S-SYS-005
  statement: Compilation-unit structure shall not imply system containment.
```

T2S requirement declaration

```
requirement:
  id: T2S-SYS-006
  statement: A contained system shall remain an independently identifiable semantic element.
```

T2S system declaration

```
system:
  id: org.text2system.example.s12.system
  name: RootSystem
  version: 1.0.0
  description: Example system used by this specification.
```

T2S system declaration

```
system:
  id: org.text2system.example.s12.subsystem
  name: ChildSystem
  version: 1.0.0
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s12.rel.system-contains-subsystem
  subject: org.text2system.example.s12.system
  predicate: contains
  object: org.text2system.example.s12.subsystem
```

13. Module

Description

A module is a logical architectural subdivision of a system.

A module groups related semantic elements into a cohesive unit and establishes a stable architectural boundary within a system.

A module is independent of implementation language, compilation strategy, and deployment topology.

A module may contain functions, interfaces, types, constants, configurations, requirements, constraints, implementations, and tests.

Mandatory properties

Optional properties

Permitted nested elements

Permitted outgoing relationships

Permitted incoming relationships

Semantic requirements

Example

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
```

T2S optional declaration

```
optional:
  version:
    type: version
  description:
    type: string
  status:
    type: status
  tags:
    type: sequence<string>
```

T2S nested declaration

```
nested:
- interface
- function
- type
- constant
- configuration
- requirement
- constraint
- implementation
- test
```

T2S outgoing declaration

```
outgoing:
- contains
- provides
- uses
- implements
- depends_on
- references
- configures
```

T2S incoming declaration

```
incoming:
- contains
- depends_on
- realizes
- references
- deploys
```

T2S requirement declaration

```
requirement:
  id: T2S-MOD-001
  statement: Every module shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-MOD-002
  statement: Every module shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-MOD-003
  statement: In a complete Canonical System Model, every module shall belong to exactly one system.
```

T2S requirement declaration

```
requirement:
  id: T2S-MOD-004
  statement: Module membership shall be expressed through an explicit contains relationship.
```

T2S requirement declaration

```
requirement:
  id: T2S-MOD-005
  statement: A module shall not contain itself directly or transitively.
```

T2S requirement declaration

```
requirement:
  id: T2S-MOD-006
  statement: A module shall remain independent of implementation language, compilation strategy, and deployment topology.
```

T2S requirement declaration

```
requirement:
  id: T2S-MOD-007
  statement: Module nesting is not defined by this revision; a module shall not contain another module.
```

T2S module declaration

```
module:
  id: org.text2system.example.s13.module
  name: CoreModule
```

T2S interface declaration

```
interface:
  id: org.text2system.example.s13.interface
  name: MainInterface
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s13.rel.module-contains-interface
  subject: org.text2system.example.s13.module
  predicate: contains
  object: org.text2system.example.s13.interface
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s13.rel.module-provides-interface
  subject: org.text2system.example.s13.module
  predicate: provides
  object: org.text2system.example.s13.interface
```

14. Interface

An interface defines an explicit interaction contract between a system and an external or internal entity.

An interface specifies the functions, data types, constants, requirements, and constraints that constitute the interaction contract.

An interface defines the portable semantics of an interaction. Communication technologies, programming languages, binary interfaces, transport mechanisms, and deployment topology are realization concerns outside the core language.

An interface may be contained by a system or module. Containment establishes architectural ownership but does not determine realization.

Mandatory properties

Optional properties

Permitted nested elements

Permitted outgoing relationships

Permitted incoming relationships

Semantic requirements

Example

The following realization examples describe possible projections of the same portable interface:

- Rust library API
- C ABI
- REST API
- Command-line interface (CLI)
- gRPC service
- HTTP service
- Message-based protocol

The choice of interface realization is defined by projectors and realization profiled and does not alter the portable semantics of the interface.

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
```

T2S optional declaration

```
optional:
  version:
    type: version
  description:
    type: string
```

T2S nested declaration

```
nested:
- function
- type
- constant
- requirement
- constraint
- test
```

T2S outgoing declaration

```
outgoing:
- contains
- exposes
- uses
- depends_on
- references
```

T2S incoming declaration

```
incoming:
- contains
- provides
- implements
- realizes
- uses
- depends_on
- references
```

T2S requirement declaration

```
requirement:
  id: T2S-IF-001
  statement: Every interface shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-IF-002
  statement: Every interface shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-IF-003
  statement: An interface may expose zero or more functions.
```

T2S requirement declaration

```
requirement:
  id: T2S-IF-004
  statement: Every function exposed by an interface shall remain an independently identifiable semantic element.
```

T2S requirement declaration

```
requirement:
  id: T2S-IF-005
  statement: Interface ownership shall be expressed through an explicit contains relationship.
```

T2S requirement declaration

```
requirement:
  id: T2S-IF-006
  statement: An interface shall remain independent of communication technology, programming language, binary interface, transport pr
```

T2S interface declaration

```
interface:
  id: org.text2system.example.s14.interface
  name: MainInterface
  version: 1.0.0
  description: Example interface used by this specification.
```

T2S function declaration

```
function:
  id: org.text2system.example.s14.function.start
  name: start
```

T2S function declaration

```
function:
  id: org.text2system.example.s14.function.stop
  name: stop
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s14.rel.interface-contains-start
  subject: org.text2system.example.s14.interface
  predicate: contains
  object: org.text2system.example.s14.function.start
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s14.rel.interface-exposes-start
  subject: org.text2system.example.s14.interface
  predicate: exposes
  object: org.text2system.example.s14.function.start
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s14.rel.interface-contains-stop
  subject: org.text2system.example.s14.interface
  predicate: contains
  object: org.text2system.example.s14.function.stop
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s14.rel.interface-exposes-stop
  subject: org.text2system.example.s14.interface
  predicate: exposes
  object: org.text2system.example.s14.function.stop
```

15. Function

A function defines a callable interaction contract.

A function belongs to exactly one containing system, module, or interface.

A function contained directly by a system or module is a free function of that architectural scope. A function contained by an interface defines one operation of that interface and shall be exposed by the same interface.

A function may declare:

- Parameters.
- Return type.
- Error type.
- Invocation semantics.
- Requirement traceability.

Mandatory properties

Optional properties

Permitted nested elements

Permitted outgoing relationships

Permitted incoming relationships

Semantic requirements

Example

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
```

T2S optional declaration

```
optional:
  description:
    type: string
  parameters:
    type: sequence<parameter>
  returns:
    type: return
  errors:
    type: error
  invocation:
    type: invocation_kind
```

T2S nested declaration

```
nested:
- requirement
- constraint
```

T2S outgoing declaration

```
outgoing:
- references
- depends_on
- verifies
```

T2S incoming declaration

```
incoming:
- contains
- exposes
- realizes
- references
```

T2S requirement declaration

```
requirement:
  id: T2S-FUNC-001
  statement: Every function shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-FUNC-002
  statement: Every function shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-FUNC-003
  statement: Every function shall belong to exactly one containing system, module, or interface.
```

T2S requirement declaration

```
requirement:
  id: T2S-FUNC-004
  statement: Function ownership shall be expressed through exactly one explicit contains relationship.
```

T2S requirement declaration

```
requirement:
  id: T2S-FUNC-010
  statement: A function contained by an interface shall be exposed by that same interface; system-owned and module-owned free functi
```

T2S requirement declaration

```
requirement:
  id: T2S-FUNC-005
  statement: Parameter order shall be significant.
```

T2S requirement declaration

```
requirement:
  id: T2S-FUNC-006
  statement: Parameter names shall be unique within a function.
```

T2S requirement declaration

```
requirement:
  id: T2S-FUNC-007
  statement: A function may define at most one return type.
```

T2S requirement declaration

```
requirement:
  id: T2S-FUNC-008
  statement: A function may define at most one error type.
```

T2S requirement declaration

```
requirement:
  id: T2S-FUNC-009
  statement: Invocation semantics shall be independent of programming language and implementation technology.
```

T2S function declaration

```
function:
  id: org.text2system.example.s15.function
  name: execute
  parameters:
  - name: request
    direction: input
    type: org.text2system.example.s15.input_record
  returns:
    type: org.text2system.example.s15.output_record
  errors:
    type: org.text2system.example.s15.error_enum
  invocation: synchronous
```

16. Type System

The T2S Type System defines portable data contracts that are independent of programming language, runtime environment, processor architecture, binary representation, and deployment technology.

The semantic meaning of a type shall remain identical across all conforming implementations and projectors.

T2S defines the following primitive types:

- bool
- i8
- i16
- i32
- i64
- u8
- u16
- u32
- u64
- f32
- f64
- string
- bytes

T2S 1.0 intentionally does not define implementation-dependent integer types, including usize, isize, long, unsigned long, intptr_t, or pointer-sized integers.

T2S defines the following composite type kinds:

- record
- enum
- sequence
- array
- option
- union
- variant
- alias
- opaque

A record defines an ordered collection of named fields.

An enum defines a closed collection of named values.

A sequence defines a runtime-length ordered collection of values of one declared element type.

An array defines a fixed-length ordered collection of values of one declared element type.

An option represents either the absence of a value or the presence of exactly one value of its declared element type.

A union represents one value selected from several possible member types. The active member is determined by the surrounding semantic contract and is not intrinsically represented by the union itself.

A variant represents one value selected from several named alternatives. A variant intrinsically identifies its active alternative.

An alias defines an alternative name for another type while preserving its portable semantics.

An opaque type defines a portable semantic identity whose internal representation is intentionally hidden.

Mandatory properties

Optional properties

Permitted nested elements

Permitted outgoing relationships

Permitted incoming relationships

Semantic requirements

Example

Primitive Types
├── bool
├── integers
├── floating-point
├── string
└── bytes

Composite Types
├── record
├── enum
├── sequence
├── array
├── option
├── union
├── variant
├── alias
└── opaque

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
  kind:
    type: type_kind
```

T2S optional declaration

```
optional:
  description:
    type: string
  fields:
    type: map<identifier, field>
  underlying_type:
    type: integer_type
  values:
    type: map<identifier, integer>
  element:
    type: type_reference
  length:
    type: u64
  members:
    type: sequence<type_reference>
  alternatives:
    type: map<identifier, alternative>
  target:
    type: type_reference
```

T2S nested declaration

```
nested:
- requirement
- constraint
- test
```

T2S outgoing declaration

```
outgoing:
- references
- depends_on
```

T2S incoming declaration

```
incoming:
- contains
- references
- uses
- depends_on
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-001
  statement: Primitive types shall have identical semantic meaning across all conforming implementations.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-002
  statement: Every user-defined type shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-003
  statement: Every user-defined type shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-004
  statement: Every user-defined type shall declare exactly one type kind.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-005
  statement: Every referenced type shall resolve to a declared primitive type or user-defined type.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-006
  statement: A projector shall preserve the semantic meaning of every declared type.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-007
  statement: A record shall define zero or more uniquely named fields.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-008
  statement: Record field order shall be significant.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-009
  statement: An enum shall define zero or more uniquely named values.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-010
  statement: Every enum value shall conform to the declared underlying integer type.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-011
  statement: A sequence shall declare exactly one element type.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-012
  statement: An array shall declare exactly one element type and exactly one fixed length.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-013
  statement: An option shall contain either no value or exactly one value of its declared element type.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-014
  statement: A union shall declare two or more member types.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-015
  statement: A union value shall conform to exactly one declared member type.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-016
  statement: A variant shall declare one or more uniquely named alternatives.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-017
  statement: A variant value shall contain exactly one declared alternative and shall identify that alternative.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-018
  statement: An alias shall reference exactly one primitive type or user-defined type.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-019
  statement: An alias shall preserve the semantics of its referenced type.
```

T2S requirement declaration

```
requirement:
  id: T2S-TYPE-020
  statement: The internal representation of an opaque type shall not be required for portable semantic validation.
```

T2S type declaration

```
type:
  id: org.text2system.example.s16.input_record
  name: InputRecord
  kind: record
  fields:
    data:
      type: bytes
    label:
      type:
        option: string
```

T2S type declaration

```
type:
  id: org.text2system.example.s16.output_record
  name: OutputRecord
  kind: record
  fields:
    value:
      type: string
    score:
      type: f32
```

T2S type declaration

```
type:
  id: org.text2system.example.s16.error_enum
  name: Error
  kind: enum
  underlying_type: u32
  values:
    invalid_input: 1
    unavailable: 2
    internal_failure: 3
```

T2S type declaration

```
type:
  id: org.text2system.example.s16.byte_sequence
  name: ByteSequence
  kind: sequence
  element:
    type: bytes
```

T2S type declaration

```
type:
  id: org.text2system.example.s16.fixed_array
  name: TransformationMatrix
  kind: array
  element:
    type: f64
  length: 9
```

T2S type declaration

```
type:
  id: org.text2system.example.s16.optional_text
  name: OptionalText
  kind: option
  element:
    type: string
```

T2S type declaration

```
type:
  id: org.text2system.example.s16.number_union
  name: Number
  kind: union
  members:
  - type: i32
  - type: i64
  - type: f32
  - type: f64
```

T2S type declaration

```
type:
  id: org.text2system.example.s16.result_variant
  name: Result
  kind: variant
  alternatives:
    value:
      type: org.text2system.example.s16.output_record
    error:
      type: org.text2system.example.s16.error_enum
```

T2S type declaration

```
type:
  id: org.text2system.example.s16.text_alias
  name: Text
  kind: alias
  target:
    type: string
```

T2S type declaration

```
type:
  id: org.text2system.example.s16.handle
  name: Handle
  kind: opaque
```

17. Record Type

This chapter defines the additional semantics of `type.kind = record`.

All properties and requirements defined by Chapter 16 (Type System) remain applicable unless explicitly extended or constrained by this chapter.

A record defines a composite type consisting of an ordered set of uniquely named fields.

Each field shall declare exactly one type.

Field order is preserved by the Canonical System Model to ensure deterministic processing and projection. Field order alone does not define binary layout, memory representation, serialization, or calling convention.

A field may additionally declare:

- option.
- default.
- documentation.

A field whose type is an option may be absent according to the semantics of the selected projection.

Binary layout, alignment, padding, serialization, and calling conventions are realization concerns defined by implementations, realization profiles, or projectors and are outside the core language.

Additional properties

Semantic requirements

Example

T2S optional declaration

```
optional:
  fields:
    type: ordered_map<identifier, field>
```

T2S requirement declaration

```
requirement:
  id: T2S-RECORD-001
  statement: Every record field shall define a unique name within its enclosing record.
```

T2S requirement declaration

```
requirement:
  id: T2S-RECORD-002
  statement: Every record field shall declare exactly one type.
```

T2S requirement declaration

```
requirement:
  id: T2S-RECORD-003
  statement: Record field order shall be preserved by the Canonical System Model.
```

T2S requirement declaration

```
requirement:
  id: T2S-RECORD-004
  statement: A projector shall preserve declared field order unless an explicitly selected realization defines an alternative repres
```

T2S requirement declaration

```
requirement:
  id: T2S-RECORD-005
  statement: A field may declare at most one default value.
```

T2S requirement declaration

```
requirement:
  id: T2S-RECORD-006
  statement: A default value shall conform to the declared field type.
```

T2S requirement declaration

```
requirement:
  id: T2S-RECORD-007
  statement: A field whose declared type is option may be absent according to the semantics of the selected realization.
```

T2S type declaration

```
type:
  id: org.text2system.example.s17.record
  name: RecognitionRequest
  kind: record
  fields:
    data:
      type: bytes
    label:
      type:
        option: string
    timeout:
      type: u32
      default: 1000
    enabled:
      type: bool
      default: false
```

T2S type declaration

```
type:
  id: org.text2system.example.s17.settings
  name: CalibrationSettings
  kind: record
  fields:
    label:
      type: string
    scale:
      type: f64
    offset_x:
      type: f64
    offset_y:
      type: f64
    values:
      type:
        sequence: f64
```

18. Enum Type

This chapter defines the additional semantics of `type.kind = enum`.

All properties and requirements defined by Chapter 16 (Type System) remain applicable unless explicitly extended or constrained by this chapter.

An enum defines a closed set of named values.

Each enum value represents a distinct semantic alternative.

An enum may declare an underlying integer type to support deterministic projection into implementation languages and binary interface definitions.

The permitted underlying integer types are:

- u8
- u16
- u32
- u64

If an underlying integer type is declared, every enum value shall define an explicit, unique integer value.

The choice of underlying integer type does not alter the portable semantics of an enum.

Binary representation, alignment, and ABI compatibility are realization concerns defined by implementations, realization profiles, or projectors.

Additional properties

Semantic requirements

Example

T2S optional declaration

```
optional:
  underlying_type:
    type: integer_type
  values:
    type: ordered_map<identifier, integer>
```

T2S requirement declaration

```
requirement:
  id: T2S-ENUM-001
  statement: Every enum value shall define a unique name.
```

T2S requirement declaration

```
requirement:
  id: T2S-ENUM-002
  statement: If an underlying integer type is declared, every enum value shall define a unique explicit integer value.
```

T2S requirement declaration

```
requirement:
  id: T2S-ENUM-003
  statement: An enum shall not contain duplicate value names or duplicate integer values.
```

T2S requirement declaration

```
requirement:
  id: T2S-ENUM-004
  statement: An enum shall declare only one of the permitted underlying integer types.
```

T2S requirement declaration

```
requirement:
  id: T2S-ENUM-005
  statement: A projector shall preserve the semantic meaning of every enum value.
```

T2S type declaration

```
type:
  id: org.text2system.example.s18.status
  name: Status
  kind: enum
  underlying_type: u32
  values:
    invalid_input: 1
    unavailable: 2
    internal_failure: 3
```

19. Sequence, Array, String, and Bytes

This chapter defines the additional semantics of `type.kind = sequence` and `type.kind = array`.

The primitive types `string` and `bytes` are defined by Chapter 16 (Type System). This chapter defines their portable semantic interpretation.

All properties and requirements defined by Chapter 16 (Type System) remain applicable unless explicitly extended or constrained by this chapter.

A sequence defines an ordered collection of zero or more values of the same element type. The length of a sequence is determined at runtime.

An array defines an ordered collection of a fixed number of values of the same element type. The length of an array is part of its type definition.

A string defines an ordered sequence of Unicode scalar values.

A bytes value defines an ordered sequence of uninterpreted octets.

The portable type system defines the semantic meaning of these types and does not prescribe:

- memory layout
- ownership
- allocation strategy
- binary representation
- character encoding
- null termination

These realization properties are defined by implementations, realization profiles, and projectors.

Additional properties

Semantic requirements

Example

T2S optional declaration

```
optional:
  element:
    type: type_reference
  length:
    type: u64
```

T2S requirement declaration

```
requirement:
  id: T2S-COLLECTION-001
  statement: Every sequence shall declare exactly one element type.
```

T2S requirement declaration

```
requirement:
  id: T2S-COLLECTION-002
  statement: Every array shall declare exactly one element type and a fixed length.
```

T2S requirement declaration

```
requirement:
  id: T2S-COLLECTION-003
  statement: The declared length of an array shall be greater than zero.
```

T2S requirement declaration

```
requirement:
  id: T2S-COLLECTION-004
  statement: A string shall represent Unicode text independently of its projected encoding.
```

T2S requirement declaration

```
requirement:
  id: T2S-COLLECTION-005
  statement: A bytes value shall not imply any character encoding or interpretation.
```

T2S type declaration

```
type:
  id: org.text2system.example.s19.byte_sequence
  name: ByteSequence
  kind: sequence
  element:
    type: u8
```

T2S type declaration

```
type:
  id: org.text2system.example.s19.fixed_array
  name: TransformationMatrix
  kind: array
  element:
    type: f64
  length: 9
```

T2S type declaration

```
type:
  id: org.text2system.example.s19.text_alias
  name: Text
  kind: alias
  target:
    type: string
```

T2S type declaration

```
type:
  id: org.text2system.example.s19.bytes_alias
  name: BinaryData
  kind: alias
  target:
    type: bytes
```

20. Option Type

This chapter defines the additional semantics of `type.kind = option`.

All properties and requirements defined by Chapter 16 (Type System) remain applicable unless explicitly extended or constrained by this chapter.

An option type represents either the absence of a value or exactly one value of its declared element type.

An option defines portable semantic presence or absence. It does not prescribe a nullable pointer, sentinel value, tagged union, or any other implementation representation.

The realization of an option type is defined by implementations, realization profiles, and projectors.

Kind-specific properties

Semantic requirements

Example

T2S optional declaration

```
optional:
  element:
    type: type_reference
```

T2S requirement declaration

```
requirement:
  id: T2S-OPTION-001
  statement: An option shall declare exactly one element type.
```

T2S requirement declaration

```
requirement:
  id: T2S-OPTION-002
  statement: An option shall contain either no value or exactly one value of its declared element type.
```

T2S requirement declaration

```
requirement:
  id: T2S-OPTION-003
  statement: The semantic meaning of an option shall be preserved by every conforming projector.
```

T2S type declaration

```
type:
  id: org.text2system.example.s20.optional_text
  name: OptionalText
  kind: option
  element:
    type: string
```

21. Union Type

This chapter defines the additional semantics of `type.kind = union`.

All properties and requirements defined by Chapter 16 (Type System) remain applicable unless explicitly extended or constrained by this chapter.

A union defines a value whose storage or semantic position may contain a value of exactly one of several declared member types.

Unlike a variant, a union does not intrinsically identify its active member. The active member shall therefore be determined by an enclosing contract, discriminator, protocol rule, or other explicitly defined semantic context.

A union does not prescribe a C union, overlapping memory layout, tagged representation, or any other implementation representation.

The realization of a union is defined by implementations, realization profiles, and projectors.

Kind-specific properties

Semantic requirements

Example

T2S optional declaration

```
optional:
  members:
    type: ordered_list<type_reference>
```

T2S requirement declaration

```
requirement:
  id: T2S-UNION-001
  statement: A union shall declare two or more member types.
```

T2S requirement declaration

```
requirement:
  id: T2S-UNION-002
  statement: A union value shall contain a value of exactly one declared member type at a time.
```

T2S requirement declaration

```
requirement:
  id: T2S-UNION-003
  statement: The semantic context of a union shall define how its active member is determined.
```

T2S requirement declaration

```
requirement:
  id: T2S-UNION-004
  statement: The semantic meaning of a union shall be preserved by every conforming projector.
```

T2S type declaration

```
type:
  id: org.text2system.example.s21.number
  name: Number
  kind: union
  members:
  - type: i32
  - type: i64
  - type: f32
  - type: f64
```

22. Variant Type

This chapter defines the additional semantics of `type.kind = variant`.

All properties and requirements defined by Chapter 16 (Type System) remain applicable unless explicitly extended or constrained by this chapter.

A variant defines a value that may assume exactly one of several named alternatives.

Unlike a union, a variant intrinsically identifies its active alternative.

The realization of a variant is defined by implementations, realization profiles, and projectors.

Kind-specific properties

Semantic requirements

Example

T2S optional declaration

```
optional:
  alternatives:
    type: ordered_map<identifier, alternative>
```

T2S requirement declaration

```
requirement:
  id: T2S-VARIANT-001
  statement: A variant shall declare one or more named alternatives.
```

T2S requirement declaration

```
requirement:
  id: T2S-VARIANT-002
  statement: Every variant alternative shall define a unique name.
```

T2S requirement declaration

```
requirement:
  id: T2S-VARIANT-003
  statement: Every variant alternative shall declare exactly one type.
```

T2S requirement declaration

```
requirement:
  id: T2S-VARIANT-004
  statement: A variant value shall identify exactly one active alternative.
```

T2S requirement declaration

```
requirement:
  id: T2S-VARIANT-005
  statement: A projector shall preserve the semantic identity of every variant alternative.
```

T2S type declaration

```
type:
  id: org.text2system.example.s22.result_variant
  name: Result
  kind: variant
  alternatives:
    success:
      type: org.text2system.example.s22.output_record
    failure:
      type: org.text2system.example.s22.error_enum
```

23. Alias Type

This chapter defines the additional semantics of `type.kind = alias`.

All properties and requirements defined by Chapter 16 (Type System) remain applicable unless explicitly extended or constrained by this chapter.

An alias defines an alternative semantic name for an existing type.

An alias introduces no new portable data semantics. The aliased type and its alias are semantically equivalent.

The realization of an alias is defined by implementations, realization profiles, and projectors.

Kind-specific properties

Semantic requirements

Example

T2S optional declaration

```
optional:
  target:
    type: type_reference
```

T2S requirement declaration

```
requirement:
  id: T2S-ALIAS-001
  statement: An alias shall reference exactly one existing primitive type or user-defined type.
```

T2S requirement declaration

```
requirement:
  id: T2S-ALIAS-002
  statement: An alias shall preserve the complete semantics of its referenced type.
```

T2S requirement declaration

```
requirement:
  id: T2S-ALIAS-003
  statement: A projector shall preserve the semantic equivalence between an alias and its referenced type.
```

T2S type declaration

```
type:
  id: org.text2system.example.s23.text_alias
  name: Text
  kind: alias
  target:
    type: string
```

24. Opaque Type

This chapter defines the additional semantics of `type.kind = opaque`.

All properties and requirements defined by Chapter 16 (Type System) remain applicable unless explicitly extended or constrained by this chapter.

An opaque type defines a semantic identity whose internal representation is intentionally hidden from consumers of the type.

An opaque type represents a resource whose internal structure is outside the Canonical System Model.

Opaque types are suitable for representing:

- runtime resources
- implementation-defined objects
- external library objects
- operating system resources
- device resources
- implementation handles

Consumers of an opaque type may store, compare, and transfer opaque values but shall not depend upon their internal representation.

The realization of an opaque type is defined by implementations, realization profiles, and projectors.

Kind-specific properties

Semantic requirements

Example

T2S optional declaration

```
optional: null
```

T2S requirement declaration

```
requirement:
  id: T2S-OPAQUE-001
  statement: The internal representation of an opaque type shall not be required for portable semantic validation.
```

T2S requirement declaration

```
requirement:
  id: T2S-OPAQUE-002
  statement: An opaque type shall be treated as an indivisible semantic value.
```

T2S requirement declaration

```
requirement:
  id: T2S-OPAQUE-003
  statement: Consumers of an opaque type shall not depend upon its internal representation.
```

T2S requirement declaration

```
requirement:
  id: T2S-OPAQUE-004
  statement: A projector shall preserve the semantic identity of an opaque type.
```

T2S type declaration

```
type:
  id: org.text2system.example.s24.handle
  name: Handle
  kind: opaque
```

T2S type declaration

```
type:
  id: org.text2system.example.s24.device
  name: Device
  kind: opaque
```

T2S type declaration

```
type:
  id: org.text2system.example.s24.connection
  name: Connection
  kind: opaque
```

25. Constants

A constant defines an immutable named value with a declared type.

A constant represents a portable semantic value whose meaning shall remain identical across all conforming implementations and projectors.

The representation and storage of a constant are realization concerns defined by implementations, realization profiles, and projectors.

Mandatory properties

Optional properties

Semantic requirements

Example

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
  type:
    type: type_reference
  value:
    type: literal
```

T2S optional declaration

```
optional:
  description:
    type: string
```

T2S requirement declaration

```
requirement:
  id: T2S-CONSTANT-001
  statement: Every constant shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONSTANT-002
  statement: Every constant shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONSTANT-003
  statement: Every constant shall declare exactly one type.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONSTANT-004
  statement: Every constant shall define exactly one value compatible with its declared type.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONSTANT-005
  statement: The value of a constant shall be immutable.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONSTANT-006
  statement: A projector shall preserve the semantic value of every constant.
```

T2S constant declaration

```
constant:
  id: org.text2system.example.s25.max_count
  name: MaxCount
  type: u32
  value: 128
```

T2S constant declaration

```
constant:
  id: org.text2system.example.s25.default_ratio
  name: DefaultRatio
  type: f32
  value: 0.85
```

T2S constant declaration

```
constant:
  id: org.text2system.example.s25.empty_bytes
  name: EmptyBytes
  type: bytes
  value: []
```

T2S constant declaration

```
constant:
  id: org.text2system.example.s25.optional_text
  name: OptionalText
  type:
    option: string
  value: null
```

26. Error Contracts

This chapter defines the additional semantics of the `errors` property of a function.

All properties and requirements defined by Chapter 15 (Function) remain applicable unless explicitly extended or constrained by this chapter.

A function may declare an error type that defines the portable contract for expected operation failures.

An error type shall normally be defined as an enum, variant, or another declared type whose semantics are appropriate for the operation.

An error contract defines expected failure conditions that are part of the function's portable semantics.

Unexpected implementation failures, process termination, panic, exceptions, undefined behavior, or other abnormal execution mechanisms are not part of a function's declared error contract.

The realization of an error contract is defined by implementations, realization profiles, and projectors.

Additional properties

Semantic requirements

Example

T2S optional declaration

```
optional:
  errors:
    type: type_reference
```

T2S requirement declaration

```
requirement:
  id: T2S-ERROR-001
  statement: A function shall declare at most one error type.
```

T2S requirement declaration

```
requirement:
  id: T2S-ERROR-002
  statement: Every declared error type shall resolve to an existing type.
```

T2S requirement declaration

```
requirement:
  id: T2S-ERROR-003
  statement: A declared error type should normally be an enum, variant, or another appropriate user-defined type.
```

T2S requirement declaration

```
requirement:
  id: T2S-ERROR-004
  statement: A projector shall preserve the semantic distinction between successful completion and every declared error condition.
```

T2S type declaration

```
type:
  id: org.text2system.example.s26.error
  name: RecognitionError
  kind: enum
  values:
    invalid_input: 1
    unavailable: 2
    internal_failure: 3
```

T2S function declaration

```
function:
  id: org.text2system.example.s26.function
  name: Execute
  parameters:
    input:
      type: org.text2system.example.s26.input_record
  returns:
    type: org.text2system.example.s26.output_record
  errors:
    type: org.text2system.example.s26.error
```

27. Configuration

A configuration defines a named set of configurable properties that influence system behavior without modifying the portable semantics of the system.

A configuration defines the configuration contract. Runtime configuration values are supplied by deployments, implementations, or operational environments.

Each configuration property defines a named configurable value and shall declare exactly one type.

A configuration property may additionally declare:

- default
- documentation

Mandatory properties

Optional properties

Semantic requirements

Example

Configuration -> Configuration Schema -> Deployment

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
  properties:
    type: ordered_map<identifier, property>
```

T2S optional declaration

```
optional:
  description:
    type: string
```

T2S requirement declaration

```
requirement:
  id: T2S-CONFIG-001
  statement: Every configuration shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONFIG-002
  statement: Every configuration shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONFIG-003
  statement: Every configuration property shall declare exactly one type.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONFIG-004
  statement: Default values shall be compatible with the declared property type.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONFIG-005
  statement: Every configuration property shall define a unique name within its enclosing configuration.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONFIG-006
  statement: A projector shall preserve the semantic meaning of every configuration property.
```

T2S configuration declaration

```
configuration:
  id: org.text2system.example.s27.configuration
  name: DefaultConfiguration
  properties:
    address:
      type: string
      default: 0.0.0.0
    port:
      type: u16
      default: 8080
    threshold:
      type: f32
      default: 0.75
```

28. Requirements

A requirement defines a normative, verifiable statement about the system.

Requirements express functional, non-functional, architectural, safety, performance, interoperability, or other system obligations.

Every requirement shall define exactly one stable identifier.

A requirement may additionally declare:

- priority
- category
- rationale
- verification
- documentation

Requirements participate in traceability and may be referenced by functions, interfaces, types, tests, deployments, implementations, relationships, and other semantic elements.

Mandatory properties

Optional properties

Semantic requirements

Example

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  statement:
    type: string
```

T2S optional declaration

```
optional:
  priority:
    type: priority
  category:
    type: string
  rationale:
    type: string
  verification:
    type: sequence<identifier>
  documentation:
    type: string
```

T2S requirement declaration

```
requirement:
  id: T2S-REQ-001
  statement: Every requirement shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-REQ-002
  statement: Every requirement shall define exactly one normative statement.
```

T2S requirement declaration

```
requirement:
  id: T2S-REQ-003
  statement: Requirement identifiers shall be unique within the Canonical System Model.
```

T2S requirement declaration

```
requirement:
  id: T2S-REQ-004
  statement: A requirement may be referenced by any semantic element.
```

T2S requirement declaration

```
requirement:
  id: T2S-REQ-005
  statement: Every requirement intended for verification should identify one or more verification artifacts.
```

T2S requirement declaration

```
requirement:
  id: org.text2system.example.s28.requirement
  statement: Every accepted request shall produce either a result or a declared error.
  priority: high
  verification:
  - org.text2system.example.s28.test
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s28.rel.function-satisfies-requirement
  subject: org.text2system.example.s28.function
  predicate: satisfies
  object: org.text2system.example.s28.requirement
```

29. Test

A test defines a verification activity that demonstrates one or more semantic elements satisfy their intended contracts.

A test is a first-class semantic element of the Canonical System Model.

A test may verify:

- requirements
- systems
- modules
- interfaces
- functions
- types
- configurations
- implementations
- deployments

Requirements define the obligations of a system. Tests define the verification activities that demonstrate those obligations are satisfied. Together they establish the normative verification model of the Canonical System Model.

The execution, framework, automation, reporting, and scheduling of a test are realization concerns defined by implementations, realization profiles, and projectors.

Mandatory properties

Optional properties

Semantic requirements

Example

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
```

T2S optional declaration

```
optional:
  description:
    type: string
  level:
    type: type_reference
```

T2S requirement declaration

```
requirement:
  id: T2S-TEST-001
  statement: Every test shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-TEST-002
  statement: Every test shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-TEST-003
  statement: A test shall verify one or more semantic elements.
```

T2S requirement declaration

```
requirement:
  id: T2S-TEST-004
  statement: A test shall not prescribe implementation-specific execution semantics.
```

T2S requirement declaration

```
requirement:
  id: T2S-TEST-005
  statement: A projector shall preserve traceability between tests and the semantic elements they verify.
```

T2S type declaration

```
type:
  id: org.text2system.example.s29.test_level
  name: TestLevel
  kind: enum
  values:
    unit: 1
    integration: 2
    system: 3
    acceptance: 4
    performance: 5
    regression: 6
```

T2S test declaration

```
test:
  id: org.text2system.example.s29.test
  name: IntegrationTest
  level:
    type: org.text2system.example.s29.test_level.integration
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s29.rel.test-verifies-requirement
  subject: org.text2system.example.s29.test
  predicate: verifies
  object: org.text2system.example.s29.requirement
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s29.rel.test-verifies-module
  subject: org.text2system.example.s29.test
  predicate: verifies
  object: org.text2system.example.s29.module
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s29.rel.test-verifies-function
  subject: org.text2system.example.s29.test
  predicate: verifies
  object: org.text2system.example.s29.function
```

30. Package

A package defines a distributable unit of a system.

A package groups one or more semantic elements for distribution while preserving their portable semantics.

A package defines what is distributed. It does not prescribe a packaging technology, archive format, installer, repository format, or distribution mechanism.

Packaging technologies and installation mechanisms are realization concerns defined by implementations, realization profiles, and projectors.

Mandatory properties

Optional properties

Semantic requirements

Example

Package -> Projector -> DEB / RPM / MSI / OCI / ZIP / ...

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
```

T2S optional declaration

```
optional:
  description:
    type: string
```

T2S requirement declaration

```
requirement:
  id: T2S-PACKAGE-001
  statement: Every package shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-PACKAGE-002
  statement: Every package shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-PACKAGE-003
  statement: A package shall include one or more semantic elements.
```

T2S requirement declaration

```
requirement:
  id: T2S-PACKAGE-004
  statement: Package membership shall not imply architectural ownership.
```

T2S requirement declaration

```
requirement:
  id: T2S-PACKAGE-005
  statement: A projector shall preserve package composition.
```

T2S package declaration

```
package:
  id: org.text2system.example.s30.package
  name: ExamplePackage
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s30.rel.package-includes-system
  subject: org.text2system.example.s30.package
  predicate: includes
  object: org.text2system.example.s30.system
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s30.rel.package-includes-module
  subject: org.text2system.example.s30.package
  predicate: includes
  object: org.text2system.example.s30.module
```

31. Deployment

A deployment defines the intended execution topology of one or more packaged system elements.

A deployment specifies what is deployed and the target environment in which it is intended to execute.

A deployment does not prescribe an orchestration technology, virtualization technology, container format, installation procedure, or operational tooling.

Execution technologies are realization concerns defined by implementations, realization profiles, and projectors.

Mandatory properties

Optional properties

Semantic requirements

Example

Deployment -> Projector -> Executable System

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
```

T2S optional declaration

```
optional:
  description:
    type: string
```

T2S requirement declaration

```
requirement:
  id: T2S-DEPLOY-001
  statement: Every deployment shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-DEPLOY-002
  statement: Every deployment shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-DEPLOY-003
  statement: A deployment shall reference one or more packages.
```

T2S requirement declaration

```
requirement:
  id: T2S-DEPLOY-004
  statement: A deployment may reference one or more realization profiles.
```

T2S requirement declaration

```
requirement:
  id: T2S-DEPLOY-005
  statement: A projector shall preserve deployment intent.
```

T2S deployment declaration

```
deployment:
  id: org.text2system.example.s31.deployment
  name: ExampleDeployment
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s31.rel.deployment-deploys-package
  subject: org.text2system.example.s31.deployment
  predicate: deploys
  object: org.text2system.example.s31.package
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s31.rel.deployment-targets-profile
  subject: org.text2system.example.s31.deployment
  predicate: targets
  object: org.text2system.example.s31.native_profile
```

32. Implementation

An implementation defines a realization of one or more portable semantic elements in an implementation language or external realization technology.

An implementation connects the Canonical System Model to handwritten, generated, or externally maintained realization artifacts.

Algorithms and implementation-internal behavior remain outside the Canonical System Model and are defined in the implementation language or realization technology.

An implementation may declare:

- implementation language
- artifact references
- source locations
- generated status
- version information

Source locations and artifact references identify realization resources. They do not become part of the portable semantics of the elements being realized.

Mandatory properties

Optional properties

Semantic requirements

Example

Portable Semantic Element
│
│ realizes
▼
Implementation
│
├── language
├── generated
├── version
└── artifacts

T2S mandatory declaration

```
mandatory:
  id:
    type: identifier
  name:
    type: identifier
```

T2S optional declaration

```
optional:
  description:
    type: string
  language:
    type: identifier
  generated:
    type: bool
  version:
    type: version
  artifacts:
    type: sequence<artifact>
```

T2S requirement declaration

```
requirement:
  id: T2S-IMPLEMENTATION-001
  statement: Every implementation shall define exactly one stable identifier.
```

T2S requirement declaration

```
requirement:
  id: T2S-IMPLEMENTATION-002
  statement: Every implementation shall define exactly one name.
```

T2S requirement declaration

```
requirement:
  id: T2S-IMPLEMENTATION-003
  statement: An implementation shall realize one or more semantic elements through explicit relationships.
```

T2S requirement declaration

```
requirement:
  id: T2S-IMPLEMENTATION-004
  statement: Every implementation shall declare its implementation language or realization technology.
```

T2S requirement declaration

```
requirement:
  id: T2S-IMPLEMENTATION-005
  statement: An implementation shall not alter the portable semantics of the elements it realizes.
```

T2S requirement declaration

```
requirement:
  id: T2S-IMPLEMENTATION-006
  statement: Implementation source locations and artifact references shall not participate in portable semantic identity.
```

T2S requirement declaration

```
requirement:
  id: T2S-IMPLEMENTATION-007
  statement: A generated implementation should identify the projector responsible for its realization.
```

T2S implementation declaration

```
implementation:
  id: org.text2system.example.s32.implementation
  name: ExampleImplementation
  language: rust
  generated: false
  version: 1.0.0
  artifacts:
  - kind: source
    location: src/example
```

T2S relationship declaration

```
relationship:
  id: org.text2system.example.s32.rel.implementation-realizes-interface
  subject: org.text2system.example.s32.implementation
  predicate: realizes
  object: org.text2system.example.s32.interface
```

33. Realization Profiles (Informative)

A realization profile defines deterministic realization policies used by a projector when transforming a Canonical System Model into one or more target artifacts.

A realization profile is not part of the Canonical System Model. It configures a projector and therefore does not alter the portable semantics defined by T2S.

A realization family identifies an implementation language, binary interface, documentation format, serialization model, or another realization domain.

T2S 1.0 defines the following reference realization profile families:

- rust
- zig
- python

Additional realization profile families, including C ABI, C++, Java, Lua, documentation, serialization, and diagram formats, may be defined independently of the core language.

A realization profile may define policies for:

- errors
- strings
- bytes
- sequences
- arrays
- options
- unions
- variants
- opaque types
- functions
- packages
- deployments

A realization profile may refine the representation of portable semantics but shall not alter their meaning.

Reference example

Informative architecture

Canonical System Model

Projector

Realization Profile

Projection Model

Artifacts

Reference projectors distributed with T2S 1.0 include:

- Rust Projector
- Zig Projector
- Python Projector
- C ABI Projector
- Documentation Projector
- Diagram Projector

Additional projectors and realization profiles may be developed independently of the T2S language specification.

The following chapters are informative. They describe reference projectors distributed with the T2S 1.0 reference implementation. They neither extend nor modify the normative semantics defined by this specification.

T2S realization_profile declaration

```
realization_profile:
  id: org.text2system.example.s33.rust_profile
  family: rust
  compatibility: '>=1.0'
  operating_system: gnu-linux
  architecture: x86_64
  conventions:
    error_model: explicit_result
    string_model: utf8
    sequence_model: vec
    option_model: option
    variant_model: enum
    opaque_model: opaque_struct
```

34. Rust Projector (Informative)

The Rust Projector transforms the Canonical System Model into Rust source code according to a selected realization profile.

The Rust Projector preserves the portable semantics of the Canonical System Model.

Typical mappings include:

- record → struct
- enum → enum
- sequence → Vec<T> or slice
- array → [T; N]
- option → Option<T>
- union → union or profile-defined equivalent
- variant → enum
- opaque → opaque Rust type or private struct
- function → fn
- error contract → Result<T, E> or profile-defined equivalent

Ownership, borrowing, visibility, lifetimes, asynchronous execution, trait implementation, module organization, and other Rust-specific language features are realization decisions defined by the selected realization profile.

The Rust Projector may generate additional implementation artifacts provided that the portable semantics of the Canonical System Model remain unchanged.

A conforming Rust Projector:

- preserves the portable semantics of every projected element;
- generates Rust types compatible with the corresponding portable T2S types;
- rejects unsupported portable constructs rather than silently changing their meaning;
- applies Rust-specific language features according to the selected realization profile.

Example realization profile

T2S realization_profile declaration

```
realization_profile:
  id: org.text2system.reference.rust.gnu_linux
  family: rust
  compatibility: '>=1.0'
  operating_system: gnu-linux
  conventions:
    error_model: explicit_result
    string_model: utf8
    sequence_model: vec
    option_model: option
    variant_model: enum
    opaque_model: opaque_struct
```

35. Zig Projector (Informative)

The Zig Projector transforms the Canonical System Model into Zig source code according to a selected realization profile.

The Zig Projector preserves the portable semantics of the Canonical System Model.

Typical mappings include:

- record → struct
- enum → enum
- sequence → []T or ArrayList(T)
- array → [N]T
- option → ?T
- union → union
- variant → tagged union
- opaque → opaque or struct
- function → fn
- error contract → error union or profile-defined equivalent

Memory management, allocators, error propagation, visibility, asynchronous execution, and other Zig-specific language features are realization decisions defined by the selected realization profile.

The Zig Projector may generate additional implementation artifacts provided that the portable semantics of the Canonical System Model remain unchanged.

A conforming Zig Projector:

- preserves the portable semantics of every projected element;
- generates Zig declarations compatible with the corresponding portable T2S semantic elements;
- rejects unsupported portable constructs rather than silently approximating or changing their meaning;
- applies Zig-specific language features according to the selected realization profile.

Reference realization profile

T2S realization_profile declaration

```
realization_profile:
  id: org.text2system.reference.zig.gnu_linux
  family: zig
  compatibility: '>=1.0'
  operating_system: gnu-linux
  conventions:
    error_model: error_union
    string_model: utf8_slice
    sequence_model: slice
    option_model: optional
    variant_model: tagged_union
    opaque_model: opaque_type
```

36. Python Projector (Informative)

The Python Projector transforms the Canonical System Model into Python source code according to a selected realization profile.

The Python Projector preserves the portable semantics of the Canonical System Model.

Typical mappings include:

- record → dataclass or class
- enum → Enum
- sequence → list
- array → list or profile-defined equivalent
- option → Optional[T] or None
- union → Union
- variant → profile-defined class hierarchy
- opaque → implementation-defined object
- function → def
- error contract → exception or profile-defined result object

Dynamic typing, object representation, runtime reflection, interpreter-specific behavior, and asynchronous execution are realization decisions defined by the selected realization profile.

The Python Projector may generate additional implementation artifacts provided that the portable semantics of the Canonical System Model remain unchanged.

A conforming Python Projector:

- preserves the portable semantics of every projected element;
- generates Python declarations compatible with the corresponding portable T2S semantic elements;
- rejects unsupported portable constructs rather than silently approximating or changing their meaning;
- applies Python-specific language features according to the selected realization profile.

Reference realization profile

T2S realization_profile declaration

```
realization_profile:
  id: org.text2system.reference.python.cpython
  family: python
  compatibility: '>=3.12'
  conventions:
    error_model: exception
    string_model: unicode
    sequence_model: list
    option_model: optional
    variant_model: class_hierarchy
    opaque_model: object
```

37. C ABI Projector (Informative)

The C ABI Projector transforms the Canonical System Model into a stable C Application Binary Interface according to a selected realization profile.

The C ABI Projector preserves the portable semantics of the Canonical System Model while producing a deterministic binary interface.

Typical mappings include:

- record → struct
- enum → integer enum representation
- sequence → pointer and length
- array → fixed-size array
- option → profile-defined presence representation
- union → union
- variant → tagged union
- opaque → incomplete struct pointer
- function → C function
- error contract → status code or profile-defined convention

Memory ownership, calling convention, symbol visibility, binary layout, alignment, structure packing, and other ABI-specific properties are realization decisions defined by the selected realization profile.

The C ABI Projector may generate header files, interface libraries, shared libraries, static libraries, and supporting artifacts provided that the portable semantics of the Canonical System Model remain unchanged.

A conforming C ABI Projector:

- preserves the portable semantics of every projected element;
- generates deterministic binary declarations according to the selected realization profile;
- rejects unsupported portable constructs rather than silently approximating or changing their meaning;
- applies ABI-specific realization rules according to the selected realization profile.

Reference realization profile

T2S realization_profile declaration

```
realization_profile:
  id: org.text2system.reference.c_abi.gnu_linux
  family: c_abi
  compatibility: '>=1.0'
  operating_system: gnu-linux
  conventions:
    error_model: status_code
    string_model: pointer_length
    sequence_model: pointer_length
    option_model: tagged_option
    variant_model: tagged_union
    opaque_model: incomplete_struct
```

38. Documentation Projector (Informative)

The Documentation Projector constructs a Documentation Model by combining the Narrative AST with the validated Canonical System Model.

The Documentation Model is an intermediate representation that combines narrative content, semantic information, traceability, cross-references, generated indexes, diagrams, and other documentation metadata.

The Documentation Projector transforms the Documentation Model into one or more human-readable documentation artifacts.

Typical outputs include:

- language specifications
- reference manuals
- API documentation
- architecture documentation
- requirements documentation
- traceability reports

Organization, layout, typography, styling, hyperlinks, indexing, navigation, and output formats are realization decisions defined by the selected realization profile.

The Documentation Projector does not modify the Canonical System Model.

A conforming Documentation Projector:

- constructs a Documentation Model from the Narrative AST and the validated Canonical System Model;
- preserves the semantic meaning of every projected semantic element;
- preserves traceability between documentation and the corresponding source declarations;
- generates documentation without modifying the Canonical System Model.

Reference realization profile

T2S realization_profile declaration

```
realization_profile:
  id: org.text2system.reference.documentation.markdown
  family: documentation
  compatibility: '>=1.0'
  conventions:
    output_format: markdown
    navigation: hierarchical
    cross_references: enabled
    diagrams: embedded
    indexes: generated
```

39. Diagram Projector (Informative)

The Diagram Projector transforms the Canonical System Model into graphical representations suitable for engineering analysis, visualization, and communication.

The Diagram Projector preserves the semantic relationships represented by the Canonical System Model.

Typical outputs include:

- system diagrams
- module diagrams
- interface diagrams
- dependency diagrams
- deployment diagrams
- package diagrams
- relationship graphs

Diagram layout, routing, styling, notation, rendering technology, and visual abstraction are realization decisions defined by the selected realization profile.

The Diagram Projector does not modify the Canonical System Model.

A conforming Diagram Projector:

- preserves the semantic relationships represented by the Canonical System Model;
- preserves traceability between graphical elements and their originating semantic elements;
- generates diagrams without modifying the Canonical System Model.

Reference realization profile

Informative architecture

Canonical System Model
|
▼
Diagram Projector
▲
|
Realization Profile
|
▼
Diagram Model
|
▼
Graphical Artifacts

T2S realization_profile declaration

```
realization_profile:
  id: org.text2system.reference.diagram.default
  family: diagram
  compatibility: '>=1.0'
  conventions:
    layout: hierarchical
    routing: orthogonal
    notation: t2s
    styling: default
    traceability: enabled
```

40. Validation

A conforming T2S processor shall validate the Canonical System Model before projection.

Validation may be performed as one or more implementation-defined passes, provided that the resulting validation outcome is equivalent.

Validation consists of the following categories:

- Source validation
- Structural validation
- Semantic validation
- Implementation consistency validation
- Test validation

Projectors may additionally perform realization-profile compatibility validation before projection begins.

Source validation verifies the lexical, syntactic, and structural correctness of the T2S source.

Structural validation verifies that every language construct conforms to the T2S grammar and structural constraints.

Semantic validation verifies the semantic consistency of the Canonical System Model, including:

- identifier resolution
- scope validation
- relationship validation
- type validation
- function validation
- configuration validation
- package validation
- deployment validation
- implementation validation

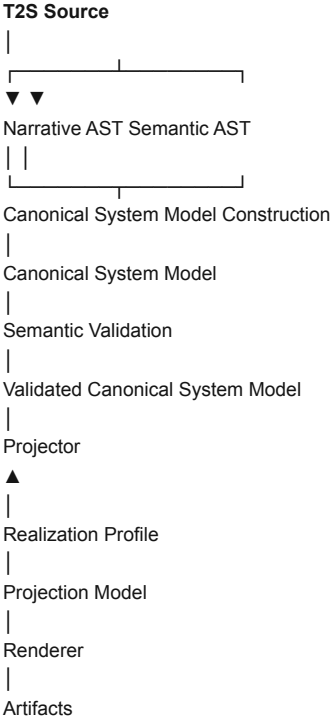
Implementation consistency validation verifies that implementation artifacts remain consistent with the portable semantic model.

Test validation verifies consistency between tests, requirements, and the semantic elements they verify.

The method by which validation is performed is implementation-defined.

Semantic requirements

Semantic processing pipeline



T2S requirement declaration

```
requirement:
  id: T2S-VALID-001
  statement: Every semantic reference shall resolve to exactly one compatible semantic element.
```

T2S requirement declaration

```
requirement:
  id: T2S-VALID-002
  statement: Duplicate stable identifiers shall be rejected.
```

T2S requirement declaration

```
requirement:
  id: T2S-VALID-003
  statement: Every relationship shall reference existing compatible semantic elements.
```

T2S requirement declaration

```
requirement:
  id: T2S-VALID-004
  statement: Cyclic type definitions shall be rejected unless explicitly permitted by the language semantics.
```

T2S requirement declaration

```
requirement:
  id: T2S-VALID-005
  statement: Every semantic element shall satisfy the constraints defined by this specification.
```

T2S requirement declaration

```
requirement:
  id: T2S-VALID-006
  statement: When a projector is invoked with a realization profile, every projected semantic element shall be compatible with that
```

T2S requirement declaration

```
requirement:
  id: T2S-VALID-007
  statement: Projection shall not begin until semantic validation has completed successfully.
```

T2S requirement declaration

```
requirement:
  id: T2S-VALID-008
  statement: Every reported validation error shall identify the affected semantic element.
```

T2S requirement declaration

```
requirement:
  id: T2S-VALID-009
  statement: Validation shall not modify the Canonical System Model. Validation produces diagnostics only.
```

41. Conformance

A conforming T2S processor shall:

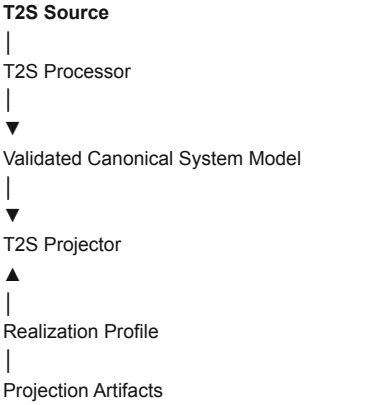
- accept every valid T2S source defined by this specification;
- reject every invalid T2S source with deterministic diagnostics;
- construct the Canonical System Model;
- perform semantic validation;
- preserve the portable semantics defined by this specification.

A conforming T2S projector shall:

- consume a validated Canonical System Model;
- preserve the portable semantics of the Canonical System Model;
- reject unsupported semantic constructs rather than silently changing their meaning;
- produce deterministic projection artifacts.

Semantic requirements

Conformance architecture



A conforming implementation may support additional language extensions, semantic elements, realization profiles, projectors, implementation technologies, or target artifacts, provided that they do not alter the normative semantics defined by this specification.

Language extensions should be identified explicitly and should not affect the interoperability of conforming T2S processors and projectors operating solely on the normative language defined by this specification.

T2S requirement declaration

```
requirement:
  id: T2S-CONFORMANCE-001
  statement: A conforming T2S processor shall satisfy every normative requirement defined by this specification.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONFORMANCE-002
  statement: A conforming T2S processor shall construct a validated Canonical System Model before projection.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONFORMANCE-003
  statement: A conforming T2S projector shall preserve the portable semantics of the Canonical System Model.
```

T2S requirement declaration

```
requirement:
  id: T2S-CONFORMANCE-004
  statement: Every conformance claim shall identify the implemented T2S language version.
```

T2S requirement declaration

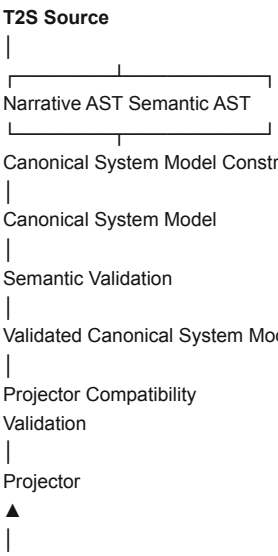
```
requirement:
  id: T2S-CONFORMANCE-005
  statement: Language extensions shall not alter the normative semantics defined by this specification.
```

42. Reference Processor Implementation (Informative)

The recommended implementation sequence for a reference T2S processor is:

1. Region scanning.
2. Source parsing.
3. Narrative AST construction.
4. Semantic AST construction.
5. Canonical System Model construction.
6. Semantic validation.
7. Projector compatibility validation (when projection is requested).
8. Projector execution.
9. Projection Model construction.
10. Artifact rendering.

The recommended processing pipeline is illustrated below.



Alternative implementation strategies are permitted provided that they produce equivalent observable behavior.

The internal organization of parser, validator, projector, renderer, and other implementation components is implementation-defined.

The reference processor implementation does not define the semantics of the T2S language.

The T2S Language Specification is the sole normative definition of the language.

43. Design Principles (Informative)

The design of T2S is guided by the following principles.

- Text first.
- Human-readable by default.
- Machine-readable by design.
- One canonical semantic model.
- Implementation-independent system definition.
- Language-neutral semantics.
- Separation of system definition and implementation.
- Separation of semantics and realization.
- Portable type system.
- Explicit system architecture.
- Explicit interfaces.
- Explicit function contracts.
- Explicit relationships.
- Explicit requirements and traceable verification.
- Deterministic parsing.
- Deterministic Canonical System Model construction.
- Deterministic semantic validation.
- Deterministic projection.
- Validation never modifies the Canonical System Model.
- Projectors consume validated Canonical System Models.
- Realization profiles configure projectors.
- Projectors preserve portable semantics.
- Unsupported semantic constructs are rejected rather than silently approximated.
- Algorithms remain in implementation languages.
- Documentation and semantics may coexist within a single source document.
- Embedded and standalone semantic declarations are semantically equivalent.
- The T2S Language Specification defines semantics; projectors define realization.
- The Canonical System Model is the authoritative representation of a system.

44. Summary (Informative)

T2S 1.0 defines a portable, implementation-independent, text-defined language for describing software and software-intensive systems.

The language separates narrative documentation, portable system semantics, realization policies, and implementation artifacts.

The Canonical System Model is the single authoritative semantic representation defined by the language.

The portable semantics of the Canonical System Model are independent of programming languages, operating systems, processor architectures, execution environments, and realization technologies.

A conforming T2S processor parses T2S source, constructs the Canonical System Model, and validates its semantic correctness.

A conforming T2S projector consumes a validated Canonical System Model together with a realization profile and produces deterministic realization artifacts while preserving the portable semantics of the model.

T2S does not describe implementation algorithms. Algorithms remain in ordinary implementation languages and are validated through testing.

The T2S Language Specification defines the normative semantics of the language. Realization profiles, projectors, validators, renderers, and other tooling implement those semantics but do not redefine them.

The result is a language that enables human-readable system definition, deterministic machine processing, implementation-independent architecture, and portable realization across multiple implementation technologies.

T2S establishes text as the canonical source of truth for system definition. All derived artifacts—including source code, documentation, diagrams, binary interfaces, packages, and deployment descriptions—are realizations of the same validated Canonical System Model.

45. Appendix A — Glossary (Informative)

This appendix defines the terminology used throughout this specification.

Unless explicitly stated otherwise, glossary entries are informative.

45.1 Abstract Syntax Tree (AST)

The structured representation produced by parsing a T2S source document.

Two abstract syntax trees are defined by this specification:

- Narrative AST
- Semantic AST

The internal representation of an AST is implementation-defined.

45.2 Canonical System Model

The implementation-independent semantic representation constructed from the Semantic AST.

The Canonical System Model is the authoritative semantic representation defined by the T2S language specification and consumed by projectors.

45.3 Compilation Unit

A single T2S source document processed as one logical unit.

A compilation unit shall define exactly one logical root semantic element.

45.4 Declaration

A semantic definition represented using the T2S semantic language.

Examples include:

- system
- module
- interface
- function
- type
- constant
- configuration
- relationship
- requirement
- constraint
- implementation
- test
- package
- deployment

45.5 Documentation Projector

A projector that transforms a Documentation Model into human-readable documentation artifacts.

Typical outputs include:

- HTML
- PDF
- Markdown
- DOCX

45.6 Embedded Semantic Source

Semantic declarations embedded within narrative text using the semantic envelope.

45.7 Function

A technology-independent callable contract belonging to exactly one interface.

45.8 Interface

A technology-independent interaction contract defining one or more callable functions.

45.9 Module

A logical architectural subdivision of a system.

45.10 Narrative

The human-readable documentation processed by the T2D parser.

45.11 Projector

A processor that transforms a validated Canonical System Model into one or more target artifacts according to a selected realization profile.

Projectors preserve portable semantics and do not modify the Canonical System Model.

45.12 Realization

A concrete implementation of portable semantics produced by a projector.

45.13 Realization Profile

A collection of realization policies that configures a projector.

A realization profile is not part of the Canonical System Model.

45.14 Relationship

A semantic connection between two semantic elements.

45.15 Semantic Builder

The processor that transforms the Semantic AST into the Canonical System Model.

45.16 Semantic AST

The parser output containing semantic declarations together with their source locations and source order.

45.17 Semantic Envelope

Exactly two leading SPACE characters introducing embedded semantic source.

The semantic envelope is removed before semantic parsing.

45.18 Standalone Semantic Source

Semantic declarations represented without accompanying narrative text.

45.19 Projection Model

The intermediate representation produced by a projector before rendering target artifacts.

45.20 T2D

The Text2Doc language for defining narrative documentation.

45.21 T2S

The Text2System language for defining portable system semantics.

45.22 Validated Canonical System Model

A Canonical System Model that has successfully completed semantic validation and is eligible for projection.

46. Appendix B — Reserved Words

This appendix defines the reserved words of the T2S language.

Reserved words have meanings assigned by this specification and shall not be assigned incompatible meanings by a conforming implementation.

46.1 Semantic declaration keywords

The following reserved words introduce first-class semantic declarations:

- system
- module
- interface
- function
- type
- constant
- configuration
- relationship
- requirement
- constraint
- implementation
- test
- package
- deployment

46.2 Semantic specification keywords

The following reserved words define the machine-readable schema of a semantic element within the T2S Language Specification:

- mandatory
- optional
- nested
- outgoing
- incoming

These keywords are interpreted according to the semantic scope established by the enclosing narrative structure.

For example, `mandatory` nested within the `Module` chapter defines mandatory properties of the `module` semantic element.

46.3 Language directives

The following reserved word introduces or identifies T2S language content:

- t2s

Future versions of the language may define additional reserved words or reserved word categories.

Semantic requirements

T2S requirement declaration

```
requirement:
  id: T2S-RESERVED-001
  statement: A conforming T2S processor shall recognize every reserved word defined by this specification.
```

T2S requirement declaration

```
requirement:
  id: T2S-RESERVED-002
  statement: Reserved words shall not be assigned incompatible language semantics by a conforming implementation.
```

T2S requirement declaration

```
requirement:
  id: T2S-RESERVED-003
  statement: Semantic specification keywords shall be interpreted within the semantic scope established by their enclosing narrative
```

T2S requirement declaration

```
requirement:
  id: T2S-RESERVED-004
  statement: Future language revisions may introduce additional reserved words while preserving the semantics of previously defined
```

47. Appendix C — Standard Diagnostics (Informative)

This appendix defines the reference diagnostic identifiers for conforming T2S processors.

Diagnostic identifiers provide implementation-independent identification of processing errors.

A processor may localize or otherwise customize diagnostic messages provided that the diagnostic identifier remains unchanged.

Reference diagnostic identifiers are grouped according to the recommended processing pipeline.

47.1 General Diagnostics

T2S constant declaration

```
constant:
  id: T2S-G001
  value: Compilation aborted.
```

T2S constant declaration

```
constant:
  id: T2S-G002
  value: Internal processor error.
```

T2S constant declaration

```
constant:
  id: T2S-G003
  value: Unsupported language version.
```

47.2 Narrative Diagnostics

T2S constant declaration

```
constant:
  id: T2S-N001
  value: Invalid document structure.
```

T2S constant declaration

```
constant:
  id: T2S-N002
  value: Invalid cover metadata.
```

T2S constant declaration

```
constant:
  id: T2S-N003
  value: Invalid section hierarchy.
```

47.3 Semantic Diagnostics

T2S constant declaration

```
constant:
  id: T2S-E001
  value: Duplicate semantic identifier.
```

T2S constant declaration

```
constant:
  id: T2S-E002
  value: Missing semantic identifier.
```

T2S constant declaration

```
constant:
  id: T2S-E003
  value: Unknown semantic declaration kind.
```

T2S constant declaration

```
constant:
  id: T2S-E004
  value: Invalid identifier.
```

T2S constant declaration

```
constant:
  id: T2S-E005
  value: Invalid semantic indentation.
```

T2S constant declaration

```
constant:
  id: T2S-E006
  value: Duplicate mapping key.
```

T2S constant declaration

```
constant:
  id: T2S-E007
  value: Unsupported semantic construct.
```

T2S constant declaration

```
constant:
  id: T2S-E008
  value: Invalid semantic region.
```

T2S constant declaration

```
constant:
  id: T2S-E009
  value: Unknown semantic reference.
```

T2S constant declaration

```
constant:
  id: T2S-E010
  value: Invalid semantic relationship.
```

47.4 Validation Diagnostics

T2S constant declaration

```
constant:
  id: T2S-V001
  value: Validation failed.
```

T2S constant declaration

```
constant:
  id: T2S-V002
  value: Type validation failed.
```

T2S constant declaration

```
constant:
  id: T2S-V003
  value: Relationship validation failed.
```

T2S constant declaration

```
constant:
  id: T2S-V004
  value: Constraint violation.
```

T2S constant declaration

```
constant:
  id: T2S-V005
  value: Semantic element does not satisfy specification.
```

47.5 Projection Diagnostics

T2S constant declaration

```
constant:
  id: T2S-P001
  value: Incompatible realization profile.
```

T2S constant declaration

```
constant:
  id: T2S-P002
  value: Unsupported semantic construct.
```

T2S constant declaration

```
constant:
  id: T2S-P003
  value: Projection aborted.
```

T2S constant declaration

```
constant:
  id: T2S-P004
  value: Renderer failure.
```

48. Appendix D — Implementation Notes (Informative)

This appendix provides implementation guidance for T2S processors and projectors.

It does not define or modify the normative semantics of the T2S language.

48.1 Recommended Processing Pipeline

see at 42.

The internal representation of every processing stage is implementation-defined.

48.2 Narrative Processing

Narrative text is processed according to the T2D Language Specification.

Narrative processing and semantic processing are independent.

A processor should preserve the relative source locations of narrative and semantic content.

48.3 Semantic Processing

The Semantic Builder transforms the Semantic AST into the Canonical System Model.

The Canonical System Model is independent of:

- programming language
- operating system
- processor architecture
- execution environment
- deployment technology
- rendering technology

48.4 Validation

Validation is performed using the Canonical System Model.

Typical validation includes:

- identifier validation
- scope validation
- relationship validation
- type validation
- function validation
- configuration validation
- package validation
- deployment validation
- implementation validation
- test validation

Projectors may additionally perform realization-profile compatibility validation before projection begins.

48.5 Projectors

Projectors transform a validated Canonical System Model into one or more target representations.

Typical projectors include:

- Documentation
- Diagram
- Rust
- Zig
- Python
- C ABI

Additional projectors may be developed independently without modifying the T2S Language Specification.

48.6 Renderers

A renderer transforms a Projection Model into one or more target artifacts.

Typical outputs include:

- HTML
- PDF
- Markdown
- DOCX
- Source code
- JSON
- SVG

Additional renderer implementations may be developed independently of both the T2S language and projector implementations.

48.7 Determinism

A conforming implementation should produce deterministic observable behavior for identical inputs and identical realization profiles.

Semantic validation shall produce deterministic diagnostics.

Projectors should produce deterministic projection models.

The order of semantic declarations should be preserved unless a projector explicitly specifies an alternative deterministic ordering.

48.8 Implementation Freedom

This specification intentionally does not define:

- parser implementation
- AST representation
- Canonical System Model representation
- memory management
- optimization strategy
- validation algorithms
- projector implementation
- renderer implementation

Conforming implementations may freely choose these implementation details provided that the externally observable semantics defined by this specification are preserved.

==== End of T2S 1.0 Language Specification ====