STUNIR Semantic IR Validation Guide

Version: 1.0.0
Status: Implementation Complete
Last Updated: 2026-01-30

Overview

This guide explains the validation framework for STUNIR Semantic IR, including schema validation, semantic checks, and multi-language validation support.

Validation Layers

The Semantic IR validation framework operates at four levels:

1. Schema Validation (Structural)

Validates IR JSON against JSON Schema definitions:

```
from semantic_ir.validator import SemanticIRValidator

validator = SemanticIRValidator()
result = validator.validate_file("my_ir.json")

if result.status == ValidationStatus.VALID:
    print("✓ Schema validation passed")
else:
    print(f"✗ Validation failed: {result.message}")
```

2. Type Validation (Semantic)

Checks type compatibility and operator semantics:

```
from semantic_ir.validation import check_binary_op_types
from semantic_ir.nodes import PrimitiveTypeRef
from semantic_ir.ir_types import BinaryOperator, IRPrimitiveType, TypeKind

left_type = PrimitiveTypeRef(kind=TypeKind.PRIMITIVE, primitive=IRPrimitiveType.I32)
right_type = PrimitiveTypeRef(kind=TypeKind.PRIMITIVE, primitive=IRPrimitiveType.I32)

result = check_binary_op_types(BinaryOperator.ADD, left_type, right_type)
assert result.status == ValidationStatus.VALID
```

3. Reference Validation (Consistency)

Ensures all node references resolve correctly:

```python
from semantic_ir.validation import validate_node_id

# Valid node ID
result = validate_node_id("n_func_add")
assert result.status == ValidationStatus.VALID

# Invalid node ID
result = validate_node_id("invalid_id")
assert result.status == ValidationStatus.INVALID
```

4. Module Validation (Complete)

Validates entire module structure:

```python
from semantic_ir.modules import IRModule
from semantic_ir.validation import validate_module

module = IRModule(
    node_id="n_mod_test",
    kind=IRNodeKind.MODULE,
    name="test_module"
)

result = validate_module(module)
if result.status == ValidationStatus.VALID:
    print("✓ Module is valid")
```

Multi-Language Validation

Python (Pydantic)

Automatic validation on instantiation:

```python
from semantic_ir.expressions import IntegerLiteral
from semantic_ir.ir_types import IRNodeKind, IRPrimitiveType
from semantic_ir.nodes import PrimitiveTypeRef, TypeKind

try:
    lit = IntegerLiteral(
        node_id="n_lit_42",
        kind=IRNodeKind.INTEGER_LITERAL,
        type=PrimitiveTypeRef(
            kind=TypeKind.PRIMITIVE,
            primitive=IRPrimitiveType.I32
        ),
        value=42
    )
    print("✓ Node created and validated")
except ValidationError as e:
    print(f"✗ Validation failed: {e}")
```

Rust (Compile-Time)

Type safety enforced at compile time:

```rust
use stunir_semantic_ir::*;

// Compile-time type checking
let lit = IntegerLiteral {
    base: ExpressionNode {
        base: IRNodeBase::new(
            "n_lit_42".to_string(),
            IRNodeKind::IntegerLiteral
        ),
        expr_type: TypeReference::PrimitiveType {
            primitive: IRPrimitiveType::I32,
        },
    },
    value: 42,
    radix: 10,
};

// Type errors caught at compile time
// let invalid = IntegerLiteral {
//     value: "not a number",  // Compile error!
// };
```

Ada SPARK (Formal Verification)

Formal contracts with proof:

```ada
with Semantic_IR.Validation;

procedure Validate_Node is
   Node_ID : constant Node_ID := To_Bounded_String ("n_test_123");
   Result  : Validation_Result;
begin
   Result := Validate_Node_ID (Node_ID);

   if Result.Status = Valid then
      Put_Line ("✓ Node ID valid");
   else
      Put_Line ("✗ " & To_String (Result.Message));
   end if;
end Validate_Node;
```

Haskell (Type-Level)

Type-safe validation:

```haskell
import STUNIR.SemanticIR.Validation

validateIR :: NodeID -> IO ()
validateIR nodeId = do
    let result = validateNodeID nodeId
    case valStatus result of
        Valid -> putStrLn "✓ Valid"
        Invalid -> putStrLn $ "✗ " ++ T.unpack (valMessage result)
```

Validation Rules

Node ID Rules

1. **Format**: Must match `^n_[a-zA-Z0-9_]+$`
2. **Uniqueness**: Each node ID must be unique within a module
3. **Reference**: All node ID references must resolve

```
# Valid
validate_node_id("n_func_add")        # ✓
validate_node_id("n_lit_42")          # ✓
validate_node_id("n_type_Point3D")    # ✓

# Invalid
validate_node_id("func_add")          # ✗ Missing 'n_' prefix
validate_node_id("n_")                # ✗ Too short
validate_node_id("n-func-add")        # ✗ Invalid character '-'
```

Hash Rules

1. **Format**: Must match `^sha256:[a-f0-9]{64}$`
2. **Computation**: Deterministic from node content
3. **Consistency**: Same content → same hash

```
# Valid
validate_hash("sha256:" + "a" * 64)   # ✓

# Invalid
validate_hash("sha256:abc")           # ✗ Too short
validate_hash("md5:" + "a" * 32)      # ✗ Wrong algorithm
validate_hash("invalid")              # ✗ Wrong format
```

Type Compatibility Rules

Arithmetic Operators (+, -, *, /, %)

- Require numeric types (i8..i64, u8..u64, f32, f64)
- Both operands must be numeric

```
check_binary_op_types(Op.ADD, i32, i32)  # ✓
check_binary_op_types(Op.MUL, f32, f32)  # ✓
check_binary_op_types(Op.ADD, i32, bool) # ✗ bool not numeric
```

Comparison Operators (==, !=, <, <=, >, >=)

- Require compatible types
- Types must match exactly

```
check_binary_op_types(Op.EQ, i32, i32)   # ✓
check_binary_op_types(Op.LT, f64, f64)   # ✓
check_binary_op_types(Op.EQ, i32, f32)   # ✗ Type mismatch
```

Logical Operators (&&, ||)

- Require boolean operands

- Both must be bool

```
check_binary_op_types(Op.AND, bool, bool) # ✓
check_binary_op_types(Op.OR, i32, i32)    # ✗ i32 not bool
```

Bitwise Operators (&, |, ^, <<, >>)

- Require integer types
- Both operands must be integers

```
check_binary_op_types(Op.BIT_AND, i32, i32) # ✓
check_binary_op_types(Op.SHL, u64, u64)     # ✓
check_binary_op_types(Op.BIT_OR, f32, f32)  # ✗ f32 not integer
```

Validation Workflow

Step 1: Load and Parse

```python
import json
from pathlib import Path

# Load IR file
ir_path = Path("my_module.json")
with open(ir_path, 'r') as f:
    ir_json = json.load(f)
```

Step 2: Schema Validation

```python
from semantic_ir.validator import SemanticIRValidator

validator = SemanticIRValidator()
result = validator.validate_json(ir_json)

if result.status != ValidationStatus.VALID:
    print(f"Schema errors: {result.message}")
    exit(1)
```

Step 3: Pydantic Validation

```python
from semantic_ir.modules import IRModule

try:
    module = IRModule(**ir_json["root"])
except ValidationError as e:
    print(f"Pydantic validation failed: {e}")
    exit(1)
```

Step 4: Semantic Validation

```python
from semantic_ir.validation import validate_module

result = validate_module(module)
if result.status != ValidationStatus.VALID:
    print(f"Semantic validation failed: {result.message}")
    exit(1)
```

Step 5: Generate Report

```python
report = validator.generate_report()
print(report)
```

CI/CD Integration

GitHub Actions

See `.github/workflows/semantic_ir_validation.yml`:

```yaml
- name: Validate IR files
  run: |
    for file in examples/semantic_ir/*.json; do
      python tools/semantic_ir/validator.py "$file" || exit 1
    done
```

Pre-commit Hook

```bash
#!/bin/bash
# .git/hooks/pre-commit

for file in $(git diff --cached --name-only | grep '\.json$'); do
    if [[ $file == examples/semantic_ir/* ]]; then
        python tools/semantic_ir/validator.py "$file"
        if [ $? -ne 0 ]; then
            echo "Validation failed for $file"
            exit 1
        fi
    fi
done
```

Testing Validation

Unit Tests

```
import pytest
from semantic_ir.validation import validate_node_id, ValidationStatus

def test_valid_node_id():
    result = validate_node_id("n_test_123")
    assert result.status == ValidationStatus.VALID

def test_invalid_node_id():
    result = validate_node_id("invalid")
    assert result.status == ValidationStatus.INVALID
    assert "must start with 'n_'" in result.message
```

Integration Tests

```
def test_round_trip_validation():
    # Create module
    module = IRModule(
        node_id="n_mod_test",
        kind=IRNodeKind.MODULE,
        name="test"
    )

    # Serialize
    json_str = module.model_dump_json()

    # Deserialize
    json_dict = json.loads(json_str)
    reconstructed = IRModule(**json_dict)

    # Validate
    result = validate_module(reconstructed)
    assert result.status == ValidationStatus.VALID
```

Common Validation Errors

Error: “Invalid node ID”

Cause: Node ID doesn’t match required pattern

Solution:

```
# Wrong
node_id = "func_add"

# Correct
node_id = "n_func_add"
```

Error: “Type mismatch in binary operation”

Cause: Operand types incompatible with operator

Solution:

```
# Wrong: bool + bool
BinaryExpr(op=Op.ADD, left=bool_var, right=bool_var)

# Correct: i32 + i32
BinaryExpr(op=Op.ADD, left=int_var, right=int_var)
```

Error: “Unresolved reference”

Cause: Node ID reference doesn’t exist

Solution:

```
# Ensure referenced nodes exist
statements = [
    VarDeclStmt(node_id="n_var_x", name="x", ...),
    AssignStmt(target="n_var_x", ...)  # Valid: n_var_x exists
]
```

Validation Best Practices

1. **Validate Early**: Check IR immediately after generation
2. **Validate Often**: Re-validate after each transformation
3. **Use Type System**: Leverage language type systems (Rust, SPARK)
4. **Write Tests**: Create comprehensive validation tests
5. **Check Examples**: Validate all example files in CI/CD
6. **Document Errors**: Provide clear error messages

See Also

- Schema Guide (SEMANTIC_IR_SCHEMA_GUIDE.md)
- Examples (../examples/semantic_ir/README.md)
- Specification (SEMANTIC_IR_SPECIFICATION.md)