STUNIR Semantic IR Schema Guide

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

Overview

This guide explains how to use the STUNIR Semantic IR JSON schemas for validation and development. The Semantic IR provides a language-agnostic, formally verifiable intermediate representation for STUNIR's deterministic pipeline.

Schema Structure

The Semantic IR schema is modular, consisting of 8 interconnected JSON Schema files:

Core Schemas

1. **ir_schema.json** - Main schema file
   - Root structure
   - Version information
   - Metadata definitions
   - Node base structure

2. **node_types.json** - Node type enumerations
   - IRNodeKind enumeration (all 35 node types)
   - BinaryOperator, UnaryOperator enumerations
   - StorageClass, Visibility, Mutability, InlineHint

3. **type_system.json** - Type system definitions
   - PrimitiveType, ArrayType, PointerType
   - StructType, FunctionType, TypeRef
   - Address spaces and mutability

Node Category Schemas

1. **expressions.json** - Expression nodes
   - Literals (integer, float, string, bool)
   - Variable references
   - Binary/unary operations
   - Function calls, member access, array access
   - Cast and ternary expressions

2. **statements.json** - Statement nodes
   - Block, expression, if, while, for statements
   - Return, break, continue
   - Variable declarations, assignments

3. **declarations.json** - Declaration nodes
   - Function declarations
   - Type declarations

   - Constant declarations
   - Variable declarations

4. **modules.json** - Module structure
   - Import/export statements
   - Module metadata
   - Target categories and safety levels

5. **target_extensions.json** - Target-specific extensions
   - Embedded attributes (interrupts, memory sections)
   - GPU attributes (kernels, workgroups)
   - WASM attributes (imports, exports)
   - Safety-critical attributes (DO-178C levels)

Using the Schemas

Schema Validation with Python

```python
import json
import jsonschema
from pathlib import Path

# Load schema
schema_path = Path("schemas/semantic_ir/ir_schema.json")
with open(schema_path, 'r') as f:
    schema = json.load(f)

# Load IR file
ir_path = Path("examples/semantic_ir/simple_function.json")
with open(ir_path, 'r') as f:
    ir_data = json.load(f)

# Validate
jsonschema.validate(instance=ir_data, schema=schema)
print("✓ IR is valid")
```

Using the Validator Tool

```
# Validate a single file
python tools/semantic_ir/validator.py examples/semantic_ir/simple_function.json

# Validate with verbose output
python tools/semantic_ir/validator.py --verbose examples/semantic_ir/embed-
ded_startup.json

# Specify custom schema directory
python tools/semantic_ir/validator.py --schema-dir /path/to/schemas file.json
```

Schema Design Principles

1. Node Base Structure

All IR nodes share a common base structure:

```
{
  "node_id": "n_<unique_identifier>",
  "kind": "<node_kind>",
  "location": {
    "file": "source_file.spec",
    "line": 10,
    "column": 5,
    "length": 20
  },
  "type": { /* type reference */ },
  "attributes": { /* custom attributes */ },
  "hash": "sha256:<64_hex_chars>"
}
```

2. Type References

Types are represented using discriminated unions:

```
// Primitive type
{
  "kind": "primitive_type",
  "primitive": "i32"
}

// Named type reference
{
  "kind": "type_ref",
  "name": "MyStruct",
  "binding": "n_type_decl_mystruct"
}

// Pointer type
{
  "kind": "pointer_type",
  "pointee": { /* type reference */ },
  "mutability": "immutable",
  "address_space": "global"
}
```

3. Node References

Nodes can reference other nodes by ID:

```
{
  "kind": "binary_expr",
  "op": "+",
  "left": "n_var_a",  // Reference to variable node
  "right": "n_var_b"  // Reference to variable node
}
```

4. Target-Specific Attributes

Target-specific features are stored in the `attributes` field:

```
{
  "kind": "function_decl",
  "name": "isr_handler",
  "attributes": {
    "interrupt_vector": 5,
    "embedded": {
      "save_context": true,
      "stack_size": 256
    }
  }
}
```

Validation Rules

Node ID Format

- Must start with `n_`
- Followed by alphanumeric characters and underscores
- Examples: `n_func_add`, `n_lit_42`, `n_module_main`

Hash Format

- Must be `sha256:` followed by exactly 64 hexadecimal characters
- Example: `sha256:0123456789abcdef...`

Type Safety

- All expressions must have valid type information
- Binary operations must have compatible operand types
- Function calls must match parameter types

Reference Resolution

- All node ID references must resolve to valid declarations
- No dangling references allowed
- Circular references only allowed in specific contexts (e.g., recursive functions)

Target Categories

The schema supports 24 target categories:

Core Categories

- **embedded**: Microcontrollers, IoT devices
- **realtime**: Real-time operating systems
- **safety_critical**: DO-178C certified systems
- **gpu**: CUDA, OpenCL, Vulkan
- **wasm**: WebAssembly
- **native**: General-purpose native code

Advanced Categories

- **jit**: Just-in-time compilation
- **interpreter**: Interpreted execution
- **functional**: Lisp, Scheme, Haskell

- **logic**: Prolog, Datalog
- **quantum**: Quantum computing
- **neuromorphic**: Neuromorphic hardware

See `target_extensions.json` for complete list and attributes.

Safety Levels

Supported safety assurance levels:

- **None**: No specific safety requirements
- **DO-178C_Level_D**: Software with no safety impact
- **DO-178C_Level_C**: Non-essential software
- **DO-178C_Level_B**: Essential software
- **DO-178C_Level_A**: Catastrophic failure prevention

Best Practices

1. Always Include Location Information

Include source location for debugging and error reporting:

```
"location": {
  "file": "module.spec",
  "line": 42,
  "column": 10
}
```

2. Compute Structural Hashes

Compute deterministic hashes for each node:

```
import hashlib
import json

def compute_node_hash(node_dict):
    # Remove hash field for computation
    node_copy = node_dict.copy()
    node_copy.pop('hash', None)

    # Canonical JSON encoding
    canonical = json.dumps(node_copy, sort_keys=True)

    # Compute SHA-256
    hash_bytes = hashlib.sha256(canonical.encode('utf-8')).digest()
    return f"sha256:{hash_bytes.hex()}"
```

3. Use Type References Consistently

Be consistent with type representations:

```
// Good: Use same representation throughout
{"kind": "primitive_type", "primitive": "i32"}

// Avoid: Mixing string and structured representations
"type": "i32"  // Don't do this
```

4. Validate Early and Often

Validate IR at every transformation step:

```
from semantic_ir.validation import validate_module

# After constructing IR
result = validate_module(my_module)
if result.status != ValidationStatus.VALID:
    raise ValueError(f"Invalid IR: {result.message}")
```

Schema Extensions

Adding Custom Attributes

You can add custom attributes to any node:

```
{
  "node_id": "n_func_1",
  "kind": "function_decl",
  "attributes": {
    "my_custom_attribute": "value",
    "optimization_hints": {
      "inline_threshold": 100
    }
  }
}
```

Custom attributes should:
- Not conflict with standard attribute names
- Be documented in your project
- Be validated separately if needed

Troubleshooting

Common Schema Errors

1. **“Invalid node ID”**
   - Ensure node IDs start with `n_`
   - Check for special characters

2. **“Type mismatch”**
   - Verify operand types match operator requirements
   - Check function parameter types

3. **“Missing required field”**
   - Ensure all required fields are present
   - Check for typos in field names

4. **“Invalid enum value”**
   - Use exact enum values from schemas
   - Check case sensitivity

See Also

- Semantic IR Specification (SEMANTIC_IR_SPECIFICATION.md)
- Validation Guide (SEMANTIC_IR_VALIDATION_GUIDE.md)
- Examples (../examples/semantic_ir/README.md)
- Implementation Plan (SEMANTIC_IR_IMPLEMENTATION_PLAN.md)