# STUNIR Semantic IR Specification v1.0

**Status:** Design Document
**Target Implementation:** STUNIR 2.0
**Last Updated:** 2026-01-30

## Executive Summary

This document specifies the **Semantic Intermediate Reference (Semantic IR)** for STUNIR—a language-agnostic, AST-based intermediate representation that preserves program semantics across transformations. Unlike the current hash-based manifest system, Semantic IR enables true semantic equivalence checking, sophisticated code transformations, and reliable multi-language code generation.

## 1. Overview

### 1.1 Purpose

The Semantic IR serves as the canonical representation of program meaning in STUNIR's deterministic pipeline:

```
Input Spec → Parser → Semantic IR → Optimizer → Semantic IR → Code Generator → Target Code
```

### 1.2 Design Goals

1. **Semantic Preservation**: Different syntactic representations of semantically equivalent programs normalize to identical IR
2. **Language Agnostic**: Represents concepts common across all 24+ target categories (embedded, GPU, web, etc.)
3. **Formally Verifiable**: Structure amenable to SPARK formal verification
4. **Transformation Friendly**: Supports optimization passes, semantic analysis, and code generation
5. **Extensible**: Can represent new language features without breaking existing tools

### 1.3 Key Differences from Current System

| Aspect | Current (Hash Manifests) | Semantic IR |
|---|---|---|
| Representation | JSON file hashes | AST-based structure |
| Equivalence | Byte-level equality | Semantic equality |
| Transformations | None | Optimization passes |
| Type System | Implicit | Explicit, verifiable |
| Semantic Analysis | Limited | Full static analysis |

# 2. IR Node Type System

## 2.1 Core Node Categories

The Semantic IR consists of five primary node categories:

1. **Modules** - Top-level compilation units
2. **Declarations** - Named entities (functions, types, constants)
3. **Statements** - Imperative control flow and side effects
4. **Expressions** - Computations that produce values
5. **Types** - Type system representation

## 2.2 Node Structure

Every IR node has the following base structure:

```
-- Ada SPARK representation
type IR_Node_Base is record
   Node_ID     : Node_Identifier;          -- Unique node ID
   Kind        : IR_Node_Kind;             -- Node category
   Location    : Source_Location;          -- Original source location
   Type_Info   : Type_Reference;           -- Type information
   Attributes  : Attribute_Map;            -- Additional metadata
   Hash        : SHA256_Digest;            -- Node structural hash
end record;
```

```
// JSON representation
{
  "node_id": "n_12345",
  "kind": "binary_expr",
  "location": {
    "file": "main.spec",
    "line": 42,
    "column": 10
  },
  "type": "i32",
  "attributes": {
    "constant_folded": true
  },
  "hash": "sha256:abc123..."
}
```

# 3. Module System

## 3.1 Module Structure

```
type IR_Module is record
   Name         : Module_Name;
   Imports      : Import_List;
   Exports      : Export_List;
   Declarations : Declaration_List;
   Metadata     : Module_Metadata;
end record;
```

### 3.2 Module Example

```
{
  "kind": "module",
  "name": "mavlink_handler",
  "imports": [
    {
      "module": "std.io",
      "symbols": ["print", "error"]
    },
    {
      "module": "mavlink.protocol",
      "symbols": "*"
    }
  ],
  "exports": [
    "handle_heartbeat",
    "handle_command"
  ],
  "declarations": [
    // Functions, types, constants
  ],
  "metadata": {
    "target_categories": ["embedded", "realtime"],
    "safety_level": "DO-178C_Level_A"
  }
}
```

## 4. Type System

### 4.1 Primitive Types

```
type IR_Primitive_Type is (
   Type_Void,
   Type_Bool,
   Type_I8, Type_I16, Type_I32, Type_I64,
   Type_U8, Type_U16, Type_U32, Type_U64,
   Type_F32, Type_F64,
   Type_String,
   Type_Char
);
```

### 4.2 Composite Types

#### 4.2.1 Array Types

```
{
  "kind": "array_type",
  "element_type": "i32",
  "size": {
    "kind": "constant",
    "value": 256
  }
}
```

### 4.2.2 Struct Types

```json
{
  "kind": "struct_type",
  "name": "Point3D",
  "fields": [
    {"name": "x", "type": "f32", "offset": 0},
    {"name": "y", "type": "f32", "offset": 4},
    {"name": "z", "type": "f32", "offset": 8}
  ],
  "size": 12,
  "alignment": 4
}
```

### 4.2.3 Function Types

```json
{
  "kind": "function_type",
  "parameters": [
    {"name": "x", "type": "i32"},
    {"name": "y", "type": "i32"}
  ],
  "return_type": "bool"
}
```

### 4.2.4 Pointer Types

```json
{
  "kind": "pointer_type",
  "pointee": "u8",
  "mutability": "immutable"
}
```

# 5. Declarations

## 5.1 Function Declarations

```
{
  "kind": "function_decl",
  "node_id": "f_mavlink_handler",
  "name": "handle_heartbeat",
  "type": {
    "kind": "function_type",
    "parameters": [
      {
        "name": "msg",
        "type": {
          "kind": "pointer_type",
          "pointee": "mavlink_message_t",
          "mutability": "immutable"
        }
      }
    ],
    "return_type": "i32"
  },
  "attributes": {
    "inline": "never",
    "visibility": "public",
    "stack_usage": 128
  },
  "body": {
    // Statement block
  }
}
```

## 5.2 Type Declarations

```
{
  "kind": "type_decl",
  "name": "MessageHandler",
  "type_definition": {
    "kind": "function_pointer_type",
    "signature": {
      "parameters": [{"name": "msg", "type": "mavlink_message_t*"}],
      "return_type": "i32"
    }
  }
}
```

### 5.3 Constant Declarations

```json
{
  "kind": "const_decl",
  "name": "MAX_MESSAGE_SIZE",
  "type": "u32",
  "value": {
    "kind": "integer_literal",
    "value": 263,
    "type": "u32"
  },
  "attributes": {
    "compile_time": true
  }
}
```

## 6. Statements

### 6.1 Control Flow Statements

#### 6.1.1 Block Statement

```json
{
  "kind": "block_stmt",
  "statements": [
    // List of statements
  ],
  "scope_id": "scope_42"
}
```

#### 6.1.2 If Statement

```json
{
  "kind": "if_stmt",
  "condition": {
    "kind": "binary_expr",
    "op": "==",
    "left": {"kind": "var_ref", "name": "status"},
    "right": {"kind": "integer_literal", "value": 0}
  },
  "then_branch": {
    "kind": "block_stmt",
    "statements": [/* ... */]
  },
  "else_branch": {
    "kind": "block_stmt",
    "statements": [/* ... */]
  }
}
```

### 6.1.3 Loop Statements

```json
{
  "kind": "while_stmt",
  "condition": {
    "kind": "function_call",
    "function": "has_more_data",
    "arguments": []
  },
  "body": {
    "kind": "block_stmt",
    "statements": [/* ... */]
  },
  "attributes": {
    "loop_bound": 100,
    "unroll": false
  }
}
```

## 6.2 Variable Statements

```json
{
  "kind": "var_decl_stmt",
  "name": "buffer",
  "type": {
    "kind": "array_type",
    "element_type": "u8",
    "size": 256
  },
  "initializer": {
    "kind": "array_init",
    "elements": []
  },
  "attributes": {
    "storage": "stack",
    "alignment": 4
  }
}
```

## 6.3 Return Statement

```json
{
  "kind": "return_stmt",
  "value": {
    "kind": "integer_literal",
    "value": 0,
    "type": "i32"
  }
}
```

# 7. Expressions

## 7.1 Literal Expressions

```
{
  "kind": "integer_literal",
  "value": 42,
  "type": "i32"
}
```

```
{
  "kind": "string_literal",
  "value": "Hello, STUNIR!",
  "type": "string"
}
```

## 7.2 Variable References

```
{
  "kind": "var_ref",
  "name": "x",
  "binding": "var_decl_123",
  "type": "i32"
}
```

## 7.3 Binary Operations

```
{
  "kind": "binary_expr",
  "op": "+",
  "left": {
    "kind": "var_ref",
    "name": "a"
  },
  "right": {
    "kind": "var_ref",
    "name": "b"
  },
  "type": "i32"
}
```

### 7.4 Function Calls

```json
{
  "kind": "function_call",
  "function": {
    "kind": "function_ref",
    "name": "send_message",
    "binding": "func_decl_456"
  },
  "arguments": [
    {
      "kind": "var_ref",
      "name": "msg"
    }
  ],
  "type": "i32"
}
```

### 7.5 Member Access

```json
{
  "kind": "member_expr",
  "object": {
    "kind": "var_ref",
    "name": "point"
  },
  "member": "x",
  "type": "f32"
}
```

## 8. Semantic Normalization Rules

### 8.1 Expression Normalization

Different syntactic forms normalize to the same semantic IR:

**Input 1:** `x + 0`
**Input 2:** `0 + x`
**Normalized IR:** `{"kind": "var_ref", "name": "x"}`

**Input 1:** `x * 2`
**Input 2:** `2 * x`
**Normalized IR:** `{"kind": "binary_expr", "op": "*", "left": "x", "right": 2}` (canonical order)

### 8.2 Type Normalization

Platform-specific types normalize to canonical forms:

**Input:** `long` (on 64-bit system)
**Normalized:** `{"kind": "integer_type", "width": 64, "signed": true}`

### 8.3 Control Flow Normalization

Different control structures can normalize to equivalent forms:

**Input 1:**

```
for (i = 0; i < 10; i++) { ... }
```

**Input 2:**

```
i = 0;
while (i < 10) { ...; i++; }
```

**Normalized IR:** Canonical loop structure with explicit initialization, condition, increment

# 9. Target-Specific IR Extensions

## 9.1 Embedded Target Extensions

```json
{
  "kind": "function_decl",
  "name": "isr_handler",
  "attributes": {
    "interrupt_vector": 5,
    "save_context": true,
    "stack_size": 256,
    "priority": 10
  }
}
```

## 9.2 GPU Target Extensions

```json
{
  "kind": "function_decl",
  "name": "vector_add_kernel",
  "attributes": {
    "execution_model": "kernel",
    "workgroup_size": [256, 1, 1],
    "shared_memory": 1024
  },
  "parameters": [
    {
      "name": "a",
      "type": "f32*",
      "address_space": "global"
    }
  ]
}
```

## 9.3 WASM Target Extensions

```json
{
  "kind": "function_decl",
  "name": "exported_func",
  "attributes": {
    "wasm_export": "add_numbers",
    "wasm_type": "externref"
  }
}
```

# 10. Semantic Analysis Passes

## 10.1 Type Checking Pass

Validates that all operations are type-safe:
- Function call arguments match parameter types
- Binary operations have compatible operand types
- Array accesses are in bounds (where statically verifiable)

## 10.2 Name Resolution Pass

Resolves all variable and function references to their declarations:
- Creates binding edges from references to declarations
- Detects undefined variables
- Handles scoping rules

## 10.3 Constant Folding Pass

Evaluates constant expressions at compile time:

```
Input:  2 + 3 * 4
Output: 14
```

## 10.4 Dead Code Elimination Pass

Removes unreachable code:

```
if (false) {
  // This block is eliminated
}
```

# 11. Code Generation Interface

## 11.1 Visitor Pattern

Each code generator implements the IR visitor interface:

```
package STUNIR.Code_Generator is
   type Code_Generator is interface;

   procedure Visit_Module(Gen : in out Code_Generator; Node : IR_Module) is abstract;
   procedure Visit_Function(Gen : in out Code_Generator; Node : Function_Decl) is ab-
stract;
   procedure Visit_Expr(Gen : in out Code_Generator; Node : Expression) is abstract;
   -- ... more visit methods
end STUNIR.Code_Generator;
```

### 11.2 Language-Specific Emitters

```ada
package STUNIR.C_Generator is
   type C_Generator is new Code_Generator with private;

   overriding procedure Visit_Function(Gen : in out C_Generator; Node : Function_Decl)
;
   -- Emits: return_type function_name(parameters) { body }
end STUNIR.C_Generator;
```

## 12. Serialization Format

### 12.1 JSON Representation (Human-Readable)

The primary serialization format is JSON for human readability and tool interoperability.

### 12.2 CBOR Representation (Compact)

For efficiency, IR can be serialized to CBOR (Deterministic CBOR - dCBOR):
- Canonical key ordering
- Stable float encoding
- Efficient binary representation

### 12.3 Hash Computation

Each IR node has a structural hash computed from:
1. Node kind
2. All child node hashes (recursive)
3. Attribute values (in canonical order)

This enables:
- Efficient equivalence checking
- Content-addressed storage
- Incremental compilation

## 13. Verification Properties

### 13.1 SPARK Contracts

The Ada SPARK implementation includes formal contracts:

```ada
function Normalize_Expression(Expr : Expression) return Expression
   with Pre  => Is_Valid_Expression(Expr),
        Post => Is_Normalized(Normalize_Expression'Result) and then
                Semantically_Equivalent(Expr, Normalize_Expression'Result);
```

### 13.2 Invariants

- **Type Safety**: All nodes have valid type information
- **Name Resolution**: All references resolve to valid declarations
- **Structural Integrity**: No dangling references, cycles only in allowed contexts
- **Determinism**: Same input always produces same normalized IR

# 14. Examples

## 14.1 Example 1: Simple Function (Embedded Target)

**Input Spec (JSON):**

```json
{
  "functions": [
    {
      "name": "add",
      "parameters": [
        {"name": "a", "type": "i32"},
        {"name": "b", "type": "i32"}
      ],
      "return_type": "i32",
      "body": "return a + b;"
    }
  ]
}
```

**Semantic IR:**

```json
{
  "kind": "function_decl",
  "name": "add",
  "type": {
    "kind": "function_type",
    "parameters": [
      {"name": "a", "type": "i32"},
      {"name": "b", "type": "i32"}
    ],
    "return_type": "i32"
  },
  "body": {
    "kind": "block_stmt",
    "statements": [
      {
        "kind": "return_stmt",
        "value": {
          "kind": "binary_expr",
          "op": "+",
          "left": {"kind": "var_ref", "name": "a", "type": "i32"},
          "right": {"kind": "var_ref", "name": "b", "type": "i32"},
          "type": "i32"
        }
      }
    ]
  }
}
```

**Generated C Code:**

```c
int32_t add(int32_t a, int32_t b) {
    return a + b;
}
```

### 14.2 Example 2: GPU Kernel

**Input Spec:**

```json
{
  "functions": [
    {
      "name": "vector_add",
      "attributes": {"gpu_kernel": true},
      "parameters": [
        {"name": "a", "type": "f32[]"},
        {"name": "b", "type": "f32[]"},
        {"name": "result", "type": "f32[]"},
        {"name": "n", "type": "i32"}
      ],
      "body": "int i = get_global_id(0); if (i < n) result[i] = a[i] + b[i];"
    }
  ]
}
```

**Semantic IR:**

```json
{
  "kind": "function_decl",
  "name": "vector_add",
  "attributes": {
    "execution_model": "kernel"
  },
  "body": {
    "kind": "block_stmt",
    "statements": [
      {
        "kind": "var_decl_stmt",
        "name": "i",
        "type": "i32",
        "initializer": {
          "kind": "function_call",
          "function": "get_global_id",
          "arguments": [{"kind": "integer_literal", "value": 0}]
        }
      },
      {
        "kind": "if_stmt",
        "condition": {
          "kind": "binary_expr",
          "op": "<",
          "left": {"kind": "var_ref", "name": "i"},
          "right": {"kind": "var_ref", "name": "n"}
        },
        "then_branch": {
          "kind": "block_stmt",
          "statements": [
            {
              "kind": "assign_stmt",
              "target": {
                "kind": "array_access",
                "array": {"kind": "var_ref", "name": "result"},
                "index": {"kind": "var_ref", "name": "i"}
              },
              "value": {
                "kind": "binary_expr",
                "op": "+",
                "left": {
                  "kind": "array_access",
                  "array": {"kind": "var_ref", "name": "a"},
                  "index": {"kind": "var_ref", "name": "i"}
                },
                "right": {
                  "kind": "array_access",
                  "array": {"kind": "var_ref", "name": "b"},
                  "index": {"kind": "var_ref", "name": "i"}
                }
              }
            }
          ]
        }
      }
    ]
  }
}
```


**Generated OpenCL Code:**

```
__kernel void vector_add(__global float* a, __global float* b,
                         __global float* result, int n) {
    int i = get_global_id(0);
    if (i < n) {
        result[i] = a[i] + b[i];
    }
}
```

# 15. Future Extensions

## 15.1 Pattern Matching

Support for algebraic data types and pattern matching:

```
{
  "kind": "match_expr",
  "scrutinee": {"kind": "var_ref", "name": "option"},
  "cases": [
    {
      "pattern": {"kind": "constructor_pattern", "name": "Some", "fields": ["x"]},
      "body": {"kind": "var_ref", "name": "x"}
    },
    {
      "pattern": {"kind": "constructor_pattern", "name": "None"},
      "body": {"kind": "integer_literal", "value": 0}
    }
  ]
}
```

## 15.2 Effect System

Track side effects (IO, memory, exceptions) at the type level:

```
{
  "type": "i32",
  "effects": ["io", "memory_alloc"]
}
```

## 15.3 Dependent Types

Support for value-dependent types:

```
{
  "kind": "array_type",
  "element_type": "u8",
  "size": {
    "kind": "type_parameter",
    "name": "N"
  }
}
```

# 16. References

- **Current STUNIR Implementation**: `tools/spark/src/stunir_spec_to_ir.adb`

- **Target Requirements**: `contracts/target_requirements.json`
- **Type System**: Ada SPARK contracts in `tools/spark/src/stunir_ir_to_code.ads`
- **LLVM IR**: Inspiration for semantic representation
- **WebAssembly**: Module structure inspiration
- **Rust MIR**: Control flow representation

## Appendix A: Complete Node Type Hierarchy

```
IR_Node (base)
├── Module
├── Declaration
│   ├── FunctionDecl
│   ├── TypeDecl
│   ├── ConstDecl
│   └── VarDecl
├── Statement
│   ├── BlockStmt
│   ├── ExprStmt
│   ├── IfStmt
│   ├── WhileStmt
│   ├── ForStmt
│   ├── ReturnStmt
│   ├── BreakStmt
│   ├── ContinueStmt
│   └── VarDeclStmt
├── Expression
│   ├── LiteralExpr
│   │   ├── IntegerLiteral
│   │   ├── FloatLiteral
│   │   ├── StringLiteral
│   │   └── BoolLiteral
│   ├── VarRef
│   ├── BinaryExpr
│   ├── UnaryExpr
│   ├── FunctionCall
│   ├── MemberExpr
│   ├── ArrayAccess
│   ├── CastExpr
│   └── TernaryExpr
└── Type
    ├── PrimitiveType
    ├── ArrayType
    ├── PointerType
    ├── StructType
    ├── FunctionType
    └── TypeRef
```

## Appendix B: JSON Schema

See `schemas/semantic_ir_v1.schema.json` for the complete JSON Schema definition (to be created in implementation phase).

---

**Document Status:** ✅ Design Complete
**Next Steps:** See `docs/SEMANTIC_IR_IMPLEMENTATION_PLAN.md`