STUNIR Semantic IR Implementation Plan

Version: 1.0
Status: Implementation Roadmap
Timeline: 20 weeks (5 months)
Target Release: STUNIR 2.0

Executive Summary

This document outlines the comprehensive implementation plan for migrating STUNIR from hash-based manifests to true Semantic IR (AST-based intermediate representation). The implementation is divided into 5 phases over 20 weeks, with clear deliverables, milestones, and success criteria.

1. Implementation Phases Overview

Phase	Duration	Description	Deliverables
Phase 1	2 weeks	IR Schema Design & Validation	JSON Schema, Ada types, validation suite
Phase 2	4 weeks	Parser Implementation	Spec → Semantic IR converter
Phase 3	8 weeks	Emitter Updates	Semantic IR → Code generators (all 28 targets)
Phase 4	4 weeks	Testing & Validation	Test suite, benchmarks, verification
Phase 5	2 weeks	Documentation & Deployment	User docs, migration guide, release

Total Duration: 20 weeks
Key Milestone: Week 14 (All emitters functional)
Release Target: Week 20

2. Phase 1: IR Schema Design & Validation (Weeks 1-2)

2.1 Objectives

- Define complete JSON Schema for Semantic IR
- Implement Ada SPARK data structures
- Create validation utilities
- Establish testing framework

2.2 Deliverables

2.2.1 JSON Schema (`schemas/semantic_ir_v1.schema.json`)

Complete JSON Schema covering:
- All node types (modules, declarations, statements, expressions, types)
- Validation rules (type constraints, reference integrity)
- Extensibility points for target-specific attributes

Example Schema Fragment:

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "STUNIR Semantic IR v1.0",
  "type": "object",
  "required": ["kind", "node_id"],
  "properties": {
    "kind": {
      "type": "string",
      "enum": ["module", "function_decl", "binary_expr", ...]
    },
    "node_id": {
      "type": "string",
      "pattern": "^n_[0-9]+$"
    }
  }
}
```

2.2.2 Ada SPARK Type Definitions (`tools/spark/src/stunir_semantic_ir.ads`)

```ada
-- tools/spark/src/stunir_semantic_ir.ads
pragma SPARK_Mode;

package STUNIR.Semantic_IR is
   -- Base node type
   type Node_ID is new Positive;
   type Node_Kind is (
      Module,
      Function_Decl, Type_Decl, Const_Decl, Var_Decl,
      Block_Stmt, If_Stmt, While_Stmt, Return_Stmt,
      Binary_Expr, Unary_Expr, Function_Call, Var_Ref,
      Primitive_Type, Array_Type, Pointer_Type, Struct_Type
   );

   type Source_Location is record
      File   : Bounded_String;
      Line   : Positive;
      Column : Positive;
   end record;

   type IR_Node_Base is tagged record
      ID       : Node_ID;
      Kind     : Node_Kind;
      Location : Source_Location;
      Type_Ref : Type_ID;
   end record;

   -- Node type hierarchy
   type Expression is new IR_Node_Base with record
      -- Expression-specific fields
   end record;

   type Binary_Expression is new Expression with record
      Operator : Binary_Op;
      Left     : Node_ID;
      Right    : Node_ID;
   end record;

   -- Validation functions
   function Is_Valid_Node(Node : IR_Node_Base) return Boolean
      with Global => null;

   function Is_Type_Safe(Expr : Expression; Expected : Type_ID) return Boolean
      with Pre => Is_Valid_Node(Expr);

end STUNIR.Semantic_IR;
```

2.2.3 Python Reference Implementation (`tools/semantic_ir.py`)

```python
from dataclasses import dataclass
from typing import List, Optional, Union
from enum import Enum

class NodeKind(Enum):
    MODULE = "module"
    FUNCTION_DECL = "function_decl"
    BINARY_EXPR = "binary_expr"
    # ... more kinds

@dataclass
class SourceLocation:
    file: str
    line: int
    column: int

@dataclass
class IRNode:
    node_id: str
    kind: NodeKind
    location: SourceLocation
    type: Optional[str] = None

@dataclass
class BinaryExpression(IRNode):
    op: str
    left: IRNode
    right: IRNode

    def validate(self) -> bool:
        """Validate type safety and structure"""
        # Type checking logic
        return True
```

2.2.4 Validation Suite

- JSON Schema validator
- SPARK verification proofs
- Property-based tests (Python)

2.3 Success Criteria

- ✅ Complete JSON Schema passes validation
- ✅ Ada SPARK code passes `gnatprove` at level 2
- ✅ All node types have validation tests
- ✅ Documentation complete

2.4 Tasks Breakdown

Task	Owner	Duration	Dependencies
Design JSON Schema	IR Team	3 days	Spec review
Implement Ada types	SPARK Dev	4 days	Schema complete
Python reference impl	Python Dev	2 days	Schema complete
Validation suite	QA Team	3 days	All impls done
Documentation	Tech Writer	2 days	All above

3. Phase 2: Parser Implementation (Weeks 3-6)

3.1 Objectives

- Implement Spec → Semantic IR conversion
- Handle all input formats (JSON specs, code snippets)
- Perform semantic analysis (type checking, name resolution)
- Generate validated IR with deterministic hashing

3.2 Architecture

```
Input Spec (JSON)
    ↓
Parser (SPARK)
    ↓
AST Construction
    ↓
Semantic Analysis
    ├── Name Resolution
    ├── Type Checking
    ├── Constant Folding
    └── Dead Code Detection
    ↓
Semantic IR (Normalized)
    ↓
Hash Computation
    ↓
IR Manifest + Receipt
```

3.3 Deliverables

3.3.1 Core Parser (`tools/spark/src/stunir_parser.adb`)

```
package body STUNIR.Parser is
   function Parse_Spec(Spec_File : String) return IR_Module
      with SPARK_Mode,
           Pre  => File_Exists(Spec_File),
           Post => Is_Valid_Module(Parse_Spec'Result);

   procedure Parse_Function(
      JSON     : in  JSON_Object;
      Function : out Function_Decl;
      Status   : out Parse_Status
   ) with SPARK_Mode;

   procedure Parse_Expression(
      JSON : in  JSON_Object;
      Expr : out Expression;
      Context : in Type_Context
   ) with SPARK_Mode;
end STUNIR.Parser;
```

3.3.2 Semantic Analyzer (`tools/spark/src/stunir_semantic_analyzer.adb`)

```
package STUNIR.Semantic_Analyzer is
   -- Name resolution
   procedure Resolve_Names(Module : in out IR_Module)
      with Pre  => Is_Valid_Module(Module),
           Post => All_Names_Resolved(Module);

   -- Type checking
   procedure Check_Types(Module : in out IR_Module)
      with Pre  => All_Names_Resolved(Module),
           Post => Is_Type_Safe(Module);

   -- Normalization
   procedure Normalize_IR(Module : in out IR_Module)
      with Pre  => Is_Type_Safe(Module),
           Post => Is_Normalized(Module);
end STUNIR.Semantic_Analyzer;
```

3.3.3 Hash Computer (`tools/spark/src/stunir_hash.adb`)

Deterministic hash computation for IR nodes:

```
function Compute_IR_Hash(Node : IR_Node_Base) return SHA256_Digest
   with SPARK_Mode,
        Post => Compute_IR_Hash'Result'Length = 32;
```

3.4 Implementation Steps

Week 3: Basic parser structure
- JSON parsing infrastructure
- AST node construction
- Error handling

Week 4: Semantic analysis (part 1)
- Name resolution pass
- Symbol table construction
- Scope management

Week 5: Semantic analysis (part 2)
- Type checking pass
- Type inference
- Error reporting

Week 6: Normalization & hashing
- Expression normalization
- Constant folding
- Deterministic hash computation
- Integration tests

3.5 Success Criteria

- ✅ All spec examples parse successfully
- ✅ Semantic errors detected correctly
- ✅ Normalized IR is deterministic (same input → same hash)
- ✅ SPARK proofs pass for all critical paths
- ✅ Performance: <100ms for typical specs

4. Phase 3: Emitter Updates (Weeks 7-14)

4.1 Objectives

Update all 28 target-specific emitters to consume Semantic IR instead of hash manifests:

Target Categories:
1. Embedded (ARM, AVR, MIPS, RISC-V, x86) - 5 emitters
2. Assembly (x86, ARM, MIPS) - 3 emitters
3. Polyglot (C89, C99, C11, Rust, Go, Zig) - 6 emitters
4. GPU (CUDA, OpenCL, SPIR-V, Metal, WGSL) - 5 emitters
5. Web (JavaScript, TypeScript, WASM) - 3 emitters
6. VM (Python, Ruby, JVM bytecode, CLR) - 4 emitters
7. Lisp (Common Lisp, Scheme, Clojure) - 3 emitters

4.2 Emitter Architecture

Each emitter implements the visitor pattern:

```ada
-- tools/spark/src/stunir_emitter_base.ads
package STUNIR.Emitter_Base is
   type Code_Emitter is interface;

   procedure Visit_Module(
      Emitter : in out Code_Emitter;
      Module  : in IR_Module
   ) is abstract;

   procedure Visit_Function(
      Emitter : in out Code_Emitter;
      Func    : in Function_Decl
   ) is abstract;

   procedure Visit_Expression(
      Emitter : in out Code_Emitter;
      Expr    : in Expression
   ) is abstract;

   function Get_Generated_Code(
      Emitter : Code_Emitter
   ) return String is abstract;
end STUNIR.Emitter_Base;
```

4.3 Prioritization Strategy

Priority 1 (Weeks 7-8): Critical Targets
- C99 emitter (most commonly used)
- Embedded ARM emitter (safety-critical)
- Python emitter (prototyping)

Priority 2 (Weeks 9-11): High-Value Targets
- WASM emitter
- Rust emitter
- CUDA/OpenCL emitters
- JavaScript/TypeScript emitters

Priority 3 (Weeks 12-14): Remaining Targets
- All other language targets
- Specialized emitters

4.4 Example: C99 Emitter Implementation

```ada
-- targets/spark/polyglot/c99/c99_emitter_semantic.adb
package body C99_Emitter_Semantic is
   overriding procedure Visit_Function(
      Emitter : in out C99_Emitter;
      Func    : in Function_Decl
   ) is
      Code : Bounded_String;
   begin
      -- Emit return type
      Append(Code, Map_Type_To_C99(Func.Return_Type));
      Append(Code, " ");

      -- Emit function name
      Append(Code, Func.Name);
      Append(Code, "(");

      -- Emit parameters
      for I in Func.Parameters'Range loop
         if I > Func.Parameters'First then
            Append(Code, ", ");
         end if;
         Append(Code, Map_Type_To_C99(Func.Parameters(I).Type));
         Append(Code, " ");
         Append(Code, Func.Parameters(I).Name);
      end loop;

      Append(Code, ") {");
      Append(Code, ASCII.LF);

      -- Visit function body
      Visit_Block(Emitter, Func.Body);

      Append(Code, "}");
      Append(Code, ASCII.LF);

      Emitter.Output := Code;
   end Visit_Function;

   overriding procedure Visit_Binary_Expr(
      Emitter : in out C99_Emitter;
      Expr    : in Binary_Expression
   ) is
   begin
      Append(Emitter.Output, "(");
      Visit_Expression(Emitter, Get_Node(Expr.Left));
      Append(Emitter.Output, " " & Map_Operator_To_C99(Expr.Operator) & " ");
      Visit_Expression(Emitter, Get_Node(Expr.Right));
      Append(Emitter.Output, ")");
   end Visit_Binary_Expr;
end C99_Emitter_Semantic;
```

4.5 Migration Strategy Per Emitter

For each emitter:

1. **Analyze Current Implementation** (0.5 days)
   - Document current input format

    - Identify transformation logic
    - List target-specific quirks

2. **Design IR → Code Mapping** (1 day)
    - Map IR nodes to target syntax
    - Handle target-specific features
    - Define code templates

3. **Implement Visitor Methods** (2 days)
    - Implement all required visit methods
    - Handle edge cases
    - Add SPARK contracts

4. **Test & Validate** (1 day)
    - Unit tests for each visit method
    - Integration tests with parser
    - Compare output with old emitter

5. **Document & Integrate** (0.5 days)
    - Update emitter documentation
    - Add examples
    - Integrate into build system

Total per emitter: ~5 days
Parallel development: 3-4 emitters simultaneously

4.6 Success Criteria

- ✅ All 28 emitters functional with Semantic IR
- ✅ Output quality matches or exceeds old emitters
- ✅ SPARK verification passes for SPARK emitters
- ✅ Performance acceptable (<500ms per emitter)
- ✅ All target examples compile and run

5. Phase 4: Testing & Validation (Weeks 15-18)

5.1 Testing Strategy

5.1.1 Unit Tests

Test individual components in isolation:
- Parser tests (500+ test cases)
- Semantic analyzer tests (300+ test cases)
- Emitter tests per target (50+ test cases each)
- Normalization tests (200+ test cases)

5.1.2 Integration Tests

End-to-end tests across the full pipeline:
- Spec → IR → C99 → Compilation
- Spec → IR → Rust → Compilation
- Spec → IR → WASM → Execution

Test matrix: 28 targets × 10 example specs = 280 integration tests

5.1.3 Property-Based Tests

Verify semantic properties:

```
@hypothesis.given(
    spec=generate_valid_spec(),
    permutation=st.permutations(range(10))
)
def test_semantic_equivalence(spec, permutation):
    """Different orderings of declarations should produce same IR hash"""
    spec1 = apply_permutation(spec, permutation)
    ir1 = parse_spec(spec)
    ir2 = parse_spec(spec1)
    assert compute_hash(ir1) == compute_hash(ir2)

@hypothesis.given(expr=generate_arithmetic_expr())
def test_constant_folding(expr):
    """Constant expressions should be evaluated at compile time"""
    ir = parse_expression(expr)
    if is_constant(expr):
        assert isinstance(ir, LiteralExpression)
```

5.1.4 Semantic Preservation Tests

Verify that transformations preserve meaning:

```
def test_for_to_while_equivalence():
    """For loops and equivalent while loops should have same semantics"""
    for_spec = parse_spec("for (i=0; i<10; i++) { f(i); }")
    while_spec = parse_spec("i=0; while(i<10) { f(i); i++; }")

    # Both should normalize to the same IR structure
    assert are_semantically_equivalent(for_spec, while_spec)
```

5.1.5 Regression Tests

Maintain test suite from current STUNIR:
- All existing test cases must pass
- Output quality must be preserved
- Performance must not regress significantly

5.2 Validation Criteria

Category	Metric	Target
Correctness	Unit test pass rate	100%
Coverage	Line coverage (SPARK)	>95%
Coverage	Branch coverage (Python)	>90%
Semantic	Property tests pass	100%
Performance	Parser speed	<100ms typical
Performance	Emitter speed	<500ms per target
Quality	SPARK proof level	Level 2 (all critical paths)

5.3 Benchmarking

Compare new vs. old implementation:

Benchmark	Current (Hash Manifests)	Target (Semantic IR)
Parse time	50ms	<100ms (+100% acceptable)
Emit time (C99)	200ms	<300ms (+50% acceptable)
Total pipeline	500ms	<800ms (+60% acceptable)
Memory usage	50MB	<100MB (+100% acceptable)

Performance targets allow for overhead from richer IR structure

5.4 Success Criteria

- ✅ All test suites pass
- ✅ No regressions in output quality
- ✅ Performance within acceptable ranges
- ✅ SPARK verification complete
- ✅ Documentation updated

6. Phase 5: Documentation & Deployment (Weeks 19-20)

6.1 Documentation Deliverables

6.1.1 User Documentation

- **Migration Guide** (`docs/MIGRATION_TO_SEMANTIC_IR.md`)
- Breaking changes
- Migration steps

- Compatibility notes
- **User Guide** (`docs/SEMANTIC_IR_USER_GUIDE.md`)
- How to use new IR
- Examples and tutorials
- Best practices
- **API Reference** (Auto-generated from SPARK)
- Complete API documentation
- Usage examples

6.1.2 Developer Documentation

- **Architecture Guide** (`docs/SEMANTIC_IR_ARCHITECTURE.md`)
- System design
- Component interactions
- Extension points
- **Emitter Development Guide** (`docs/DEVELOPING_EMITTERS.md`)
- How to create new emitters
- Testing emitters
- Best practices

6.1.3 Specification Updates

- Update `README.md` with Semantic IR information
- Update `ENTRYPOINT.md` with new IR format
- Add Semantic IR examples to `examples/`

6.2 Deployment Plan

6.2.1 Beta Release (Week 19)

- Release to internal testing team
- Gather feedback
- Fix critical bugs
- Performance tuning

6.2.2 Release Candidate (Week 20, Day 1-3)

- Public beta release
- Community feedback
- Final bug fixes
- Documentation polish

6.2.3 STUNIR 2.0 Release (Week 20, Day 4-5)

- Official release
- Announcement blog post
- Update package repositories
- Release notes

6.3 Rollout Strategy

Backward Compatibility:
- Keep hash-based system for 2 release cycles (deprecated)
- Provide migration tool: `stunir migrate-to-semantic-ir`
- Flag for legacy mode: `--use-legacy-ir`

Migration Timeline:
- **STUNIR 2.0:** Semantic IR default, legacy available
- **STUNIR 2.1:** Legacy deprecated, warnings issued
- **STUNIR 3.0:** Legacy removed, Semantic IR only

6.4 Success Criteria

- ✅ All documentation complete and reviewed
- ✅ Beta testing successful (no critical bugs)
- ✅ Community feedback positive
- ✅ Release published and announced
- ✅ Migration guide validated by early adopters

7. Resource Requirements

7.1 Team Structure

Role	Allocation	Responsibilities
Lead Architect	100% (20 weeks)	Overall design, technical decisions
SPARK Developer 1	100% (20 weeks)	Parser, semantic analyzer
SPARK Developer 2	100% (14 weeks)	Emitters (Priority 1 & 2)
SPARK Developer 3	50% (8 weeks)	Emitters (Priority 3)
Python Developer	50% (10 weeks)	Reference impl, testing
QA Engineer	100% (8 weeks)	Testing, validation
Technical Writer	50% (6 weeks)	Documentation

Total Effort: ~100 person-weeks

7.2 Infrastructure Requirements

- **Build Servers:** 4 CI runners for parallel builds
- **Test Infrastructure:** GPU test nodes for CUDA/OpenCL testing
- **Storage:** ~500GB for test artifacts and benchmarks

7.3 Dependencies

- **GNAT Compiler:** FSF GNAT 12+ with SPARK support
- **GNATprove:** For formal verification

- **JSON Schema Validator:** For schema validation
- **Test Frameworks:** pytest, hypothesis (Python), AUnit (Ada)

8. Risk Management

8.1 Identified Risks

Risk	Probability	Impact	Mitigation
SPARK verification fails	Medium	High	Start verification early, get expert support
Performance issues	Medium	Medium	Continuous benchmarking, optimization passes
Emitter complexity	Low	High	Prioritize critical emitters, parallel development
Breaking changes	High	Medium	Maintain backward compatibility, gradual migration
Schedule slippage	Medium	Medium	Buffer weeks, descope non-critical features

8.2 Contingency Plans

If SPARK verification proves too difficult:
- Fall back to Ada without SPARK mode for non-critical components
- Use Python reference implementation as temporary stopgap
- Extend timeline by 2-4 weeks for expert consultation

If emitters take longer than expected:
- Release STUNIR 2.0 with subset of emitters
- Phase remaining emitters into 2.1, 2.2 releases
- Use Python fallback emitters temporarily

If performance is unacceptable:
- Add caching layer for IR computations
- Implement incremental compilation
- Optimize hot paths identified by profiling

9. Success Metrics

9.1 Quantitative Metrics

Metric	Target	Measurement
Code Coverage	>95%	gcov, gnatcov
Test Pass Rate	100%	CI/CD dashboard
Performance	<2x slowdown	Benchmark suite
SPARK Proof	Level 2	gnatprove report
Bug Density	<0.1 bugs/KLOC	Issue tracker

9.2 Qualitative Metrics

- **Developer Satisfaction:** Survey of emitter developers
- **Code Maintainability:** Code review feedback
- **Documentation Quality:** User feedback surveys
- **Community Adoption:** Download stats, GitHub stars

10. Post-Release Plan

10.1 Immediate Post-Release (Weeks 21-24)

- **Bug Fixes:** Address critical bugs reported by users
- **Performance Tuning:** Optimize hot paths based on real-world usage
- **Documentation Updates:** Fix documentation issues
- **Community Support:** Active engagement on forums, GitHub issues

10.2 Future Enhancements (STUNIR 2.1+)

Phase 2.1 (3 months):
- Advanced optimizations (dead code elimination, inlining)
- Enhanced type inference
- Better error messages

Phase 2.2 (6 months):
- Pattern matching support
- Effect system
- Dependent types (research phase)

Phase 3.0 (12 months):
- Full CRIR integration (Church-Rosser confluence)
- Formal verification of semantic preservation
- Proof-carrying code generation

11. Conclusion

The migration to Semantic IR represents a fundamental improvement in STUNIR's architecture, enabling true semantic equivalence checking and sophisticated code transformations. The 20-week implementation plan is aggressive but achievable with proper resource allocation and risk management.

Key Success Factors:
1. Early and continuous SPARK verification
2. Parallel emitter development
3. Comprehensive testing strategy
4. Backward compatibility during transition
5. Clear documentation and migration path

Next Steps:
1. Review and approve this implementation plan
2. Allocate team resources
3. Begin Phase 1: IR Schema Design
4. Set up project tracking and CI/CD infrastructure

Document Status: ✅ Implementation Plan Complete
Next Steps: Begin Phase 1 implementation
Related Documents:
- `docs/SEMANTIC_IR_SPECIFICATION.md` (Design)
- `docs/MIGRATION_TO_SEMANTIC_IR.md` (User guide)
- `docs/SEMANTIC_IR_ARCHITECTURE.md` (Technical details)