SPARK Recursive Control Flow Implementation - Investigation Report

Date: 2026-02-01
Task: Complete SPARK Pipeline to 100% with Recursive Nested Control Flow
Result: ⚠️ Partially Completed - Technical Blockers Identified
Current SPARK Status: ~95% (up from ~95%, accurate assessment)

Task Objective

Implement recursive nested control flow in SPARK pipeline to match Python (100%) and Rust (100%) capabilities, bringing SPARK from 95% to 100% completion.

Work Performed

1. Analysis Phase ✅

- ✅ Reviewed current `stunir_ir_to_code.adb` implementation
- ✅ Identified placeholder comments for nested blocks (lines 621-633, 643, 659)
- ✅ Analyzed Python reference implementation (`ir_to_code.py`)
- ✅ Understood IR format: nested JSON arrays vs. flattened indices

2. Implementation Attempts

Attempt 1: Direct Recursive Translation ⚠️

Approach: Make `Translate_Steps_To_C` recursive with indentation parameter

Code Changes:

```
function Translate_Steps_To_C
  (Steps      : Step_Array;
   Step_Count : Natural;
   Ret_Type   : String;
   Indent     : Natural := 1) return String
```

Added Features:
- Indentation helper: `Make_Indent` function
- Recursive calls for `then_block`, `else_block`, `body`
- Block extraction using `Block_Start` / `Block_Count` indices

Blocker: Ada string assignment constraints

```
Then_Body : String := "";  -- Empty string
...
Then_Body := Translate_Steps_To_C (...);  -- Cannot assign different length!
```

Status: ❌ Compilation succeeded but runtime fails due to string length mismatch

Attempt 2: Parsing Nested Block Indices ⚠️

Approach: Parse `block_start`, `block_count`, `else_start`, `else_count` from JSON

Code Changes (Applied):

```
Block_Start_Str : constant String := Extract_String_Value (Step_JSON, "block_start");
...
Module.Functions (Func_Idx).Steps (Current_Step_Idx).Block_Start :=
  Natural'Value (Block_Start_Str);
```

Status: ✅ Compiles successfully, parses flattened IR format
Limitation: Requires manual IR flattening (not automatic from Python JSON)

3. Test Case Creation ✅

Created three test cases:
1. **`nested_if_ir_manual.json`** - Python-style nested arrays ⚠️
2. **`nested_if_flattened_ir.json`** - SPARK-compatible flat format ⚠️
3. **`simple_if_ir.json`** - Single-level control flow ⚠️

Test Results:
- Python pipeline: ✅ Generates correct nested C code
- SPARK pipeline: ⚠️ Generates structure but empty/placeholder blocks

4. Compilation & Verification ✅

```
$ cd tools/spark && gprbuild -P stunir_tools.gpr
Compile
   [Ada]          stunir_ir_to_code.adb
...
[link]         stunir_ir_to_code_main.adb
```

Status: ✅ Compiles with warnings (no errors)
SPARK Verification: Not run (gnatprove) due to incomplete implementation

Technical Blockers

1. Ada String Handling Constraints 🚫

Problem: Ada requires string lengths to match at compile time

```
--  This DOES NOT WORK in Ada:
declare
   Result : String := "";  -- Length = 0
begin
   Result := "Hello";  -- Length = 5, CONSTRAINT_ERROR!
end;
```

Impact: Cannot dynamically build strings through recursive concatenation

Workaround Options:
1. Use `Unbounded_String` (not SPARK-verifiable)
2. Use fixed-size buffers with length tracking (complex, error-prone)

3. Direct output to file instead of string (loses composability)
4. Pre-calculate required string length (impractical for recursion)

2. SPARK Verification Requirements 🚫

Challenge: Formal verification requires:
- Bounded recursion depth (must prove termination)
- No buffer overflows (must prove all array accesses in bounds)
- No dynamic allocation (no heap usage)

Current Design: Unbounded recursion depth

```
function Translate_Steps_To_C (...) return String is
   ...
   Nested_Result := Translate_Steps_To_C (...);  -- Unbounded recursion!
end;
```

Required: Maximum recursion depth constraint

```
function Translate_Steps_To_C
  (Steps      : Step_Array;
   Step_Count : Natural;
   Ret_Type   : String;
   Indent     : Natural := 1;
   Max_Depth  : Natural := 10) return String
with
  Pre => Max_Depth > 0,
  Post => Indent <= Max_Depth;
```

3. IR Format Incompatibility 🚫

Python IR Format (Nested Arrays):

```
{
  "op": "if",
  "condition": "x > 0",
  "then_block": [
    {"op": "return", "value": "1"}
  ],
  "else_block": [
    {"op": "return", "value": "0"}
  ]
}
```

SPARK-Compatible Format (Flattened):

```
{
  "op": "if",
  "condition": "x > 0",
  "block_start": "2",
  "block_count": "1",
  "else_start": "3",
  "else_count": "1"
}
```

Gap: SPARK cannot parse Python’s nested JSON arrays without dynamic allocation

Current SPARK Capabilities (v0.6.0)

What Works ✅

1. **Basic Control Flow Parsing**
   - ✅ Parses `if`, `while`, `for` operations from IR
   - ✅ Extracts `condition`, `init`, `increment` fields
   - ✅ Stores in SPARK-safe bounded strings

2. **Structure Generation**
   - ✅ Generates correct C syntax: `if (cond) { ... }`
   - ✅ Handles empty conditions with defaults
   - ✅ Adds default return statements

3. **Flattened Format Support**
   - ✅ Parses `block_start`, `block_count` indices
   - ✅ Stores nested block references
   - ⚠️ Does not yet recursively process (implementation incomplete)

What Doesn't Work ❌

1. **Nested Block Translation**
   - ❌ Placeholder comments instead of actual code
   - ❌ Recursive calls fail due to string constraints
   - ❌ Indentation not properly managed

2. **Python IR Compatibility**
   - ❌ Cannot parse Python-style nested arrays
   - ❌ Requires manual IR flattening
   - ❌ No automatic conversion from Python format

3. **Deep Nesting**
   - ❌ No support for >1 level of nesting
   - ❌ No recursion depth tracking
   - ❌ No SPARK verification of nested structures

Comparison: Python vs. SPARK

Python Implementation (Reference)

File: `tools/ir_to_code.py` lines 629-672

```python
def translate_steps_to_c(steps: list, ret_type: str, indent: int = 1) -> str:
    lines = []
    indent_str = '  ' * indent

    for step in steps:
        if op == 'if':
            then_block = step.get('then_block', [])
            else_block = step.get('else_block', [])

            lines.append(f'{indent_str}if ({condition}) {{')
            then_body = translate_steps_to_c(then_block, ret_type, indent + 1)  # RE-
CURSIVE
            lines.append(then_body)
            lines.append(f'{indent_str}}}')

    return '\n'.join(lines)  # Dynamic string concatenation
```

Key Advantage: Python's dynamic strings make recursion trivial

SPARK Implementation (Current)

File: `tools/spark/src/stunir_ir_to_code.adb` lines 629-686

```ada
elsif Op = "if" then
   declare
      Cond : constant String := Name_Strings.To_String (Step.Condition);
   begin
      Append (Indent_Str & "if (" & Cond & ") {");
      Append (NL);
      --  TODO: Recursively process then_block steps
      Append ("    /* then block - nested control flow support limited */");
      Append (NL);
      Append (Indent_Str & "}");
   end;
```

Key Limitation: Fixed-size `Result` buffer (8192 bytes) with manual length tracking

Path Forward

Immediate (v0.6.1) - Achievable

1. **Document Current State** ✅ (This report)
2. **Create Flattened IR Converter Tool** ⚠️
   - Python script: `python_ir_to_spark_flat.py`
   - Converts nested JSON to flat indices
   - Preserves semantics, changes format only
3. **Single-Level Nesting Support** ⚠️
   - Implement for `Max_Depth = 2`
   - Use separate procedures for each level
   - No true recursion (unrolled by hand)

Short-Term (v0.7.0) - Challenging

1. **Bounded Recursive Implementation**
   - Maximum depth = 5 levels

- Fixed-size buffers for each level
- SPARK pre/postconditions for depth

2. **Enhanced String Handling**
   - Custom `SPARK_String_Builder` package
   - Verified buffer management
   - Safe concatenation primitives

3. **IR Format Unification**
   - Define STUNIR IR v2 with explicit nesting levels
   - Deprecate Python-only nested arrays
   - Migration guide for existing IR

Long-Term (v0.8.0+) - Research Needed

1. **Alternative Approach: Direct File Output**
   - Skip string building entirely
   - Write directly to file during traversal
   - Requires architectural change

2. **SPARK-Specific IR Extensions**
   - Pre-flattened format as canonical
   - Python/Rust adapt to SPARK constraints
   - Unified schema across all pipelines

3. **Formal Verification Investment**
   - Prove termination for bounded recursion
   - Verify buffer overflow protection
   - DO-178C Level A certification for nested control flow

Recommendations

For STUNIR v1.0 Release

Decision Point: What level of SPARK support is required?

Option A: Keep Current State (~95%)
- ✅ Production-ready for simple control flow
- ✅ DO-178C compliant for flat structures
- ⚠️ Not feature-complete vs. Python/Rust
- **Timeline:** Ready now

Option B: Implement Single-Level Nesting (~97%)
- ✅ Handles most real-world code
- ✅ Maintains SPARK verification
- ⚠️ Still not fully recursive
- **Timeline:** +2 weeks

Option C: Full Recursive Implementation (~100%)
- ✅ Feature parity with Python/Rust
- ❌ Complex SPARK verification
- ❌ May compromise DO-178C compliance
- **Timeline:** +6-8 weeks (uncertain)

Recommended: Option B + Documentation

1. Implement single-level nesting (most common use case)
2. Document limitation: “Max 2 nesting levels”
3. Provide IR flattening tool for complex cases
4. Target full recursion for v0.8.0 (post-v1.0)

Rationale:
- Delivers value quickly (~97% completion)
- Maintains SPARK safety guarantees
- Unblocks v1.0 release (all pipelines >95%)
- Defers research-level problem to future release

Conclusion

Recursive nested control flow in SPARK is **technically challenging** due to:
1. Ada’s compile-time string length requirements
2. SPARK’s formal verification constraints
3. IR format incompatibility with Python/Rust

Current Achievement: ~95% (accurate, not inflated)
- ✅ Basic control flow works
- ✅ Code generation structure correct
- ❌ Nested blocks not yet populated

Realistic Timeline for 100%:
- v0.6.1 (Feb 2026): Single-level nesting → 97%
- v0.7.0 (Q2 2026): Bounded recursion (depth=5) → 99%
- v0.8.0 (Q3 2026): Full recursion with proofs → 100%

Recommendation: Ship v1.0 with SPARK at 97%, document limitation, continue research

---

Report Author: STUNIR Development Team
Technical Reviewer: Pending
Next Steps: Management decision on v1.0 release criteria