Flattened IR Design for SPARK v0.6.1

Purpose

The SPARK pipeline cannot handle dynamically nested JSON arrays due to Ada's static typing constraints. This design document specifies a **flattened IR format** that represents control flow using block indices instead of nested arrays.

Problem Statement

Current Nested IR (Python/Rust)

```
{
  "op": "if",
  "condition": "x > 0",
  "then_block": [
    {"op": "assign", "target": "y", "value": "x + 1"},
    {"op": "assign", "target": "z", "value": "y * 2"}
  ],
  "else_block": [
    {"op": "assign", "target": "y", "value": "0"}
  ]
}
```

Issue: Ada cannot dynamically parse nested arrays without knowing their depth at compile time.

Flattened IR Solution

Key Concepts

1. **Single Flat Array**: All steps stored in one array, no nesting
2. **Block Indices**: Use indices to reference blocks instead of nested arrays
3. **Block Metadata**:
   - `block_start` : Index where block begins
   - `block_count` : Number of steps in block
   - `else_start` : Index where else block begins (if statements only)
   - `else_count` : Number of steps in else block

Flattened IR Example

```
{
  "steps": [
    {
      "op": "if",
      "condition": "x > 0",
      "block_start": 1,
      "block_count": 2,
      "else_start": 3,
      "else_count": 1
    },
    {
      "op": "assign",
      "target": "y",
      "value": "x + 1"
    },
    {
      "op": "assign",
      "target": "z",
      "value": "y * 2"
    },
    {
      "op": "assign",
      "target": "y",
      "value": "0"
    }
  ]
}
```

Interpretation:
- Step 0: `if (x > 0)`
- Then block: steps [1..2] (indices 1 and 2)
- Else block: step [3] (index 3)

Control Flow Structures

If Statement

```
{
  "op": "if",
  "condition": "condition_expr",
  "block_start": N,
  "block_count": M,
  "else_start": N+M,
  "else_count": K
}
```

Notes:
- `else_start` and `else_count` are 0 if no else block
- Then block: steps[block_start .. block_start + block_count - 1]
- Else block: steps[else_start .. else_start + else_count - 1]

While Loop

```
{
  "op": "while",
  "condition": "condition_expr",
  "block_start": N,
  "block_count": M
}
```

Notes:
- Body: steps[block_start .. block_start + block_count - 1]
- No else_start/else_count needed

For Loop

```
{
  "op": "for",
  "init": "i = 0",
  "condition": "i < 10",
  "increment": "i += 1",
  "block_start": N,
  "block_count": M
}
```

Notes:
- Body: steps[block_start .. block_start + block_count - 1]
- No else_start/else_count needed

Conversion Algorithm

Nested to Flat Conversion

```
def convert_nested_to_flat(nested_steps):
    flat_steps = []

    def flatten(steps, start_index):
        for step in steps:
            current_index = len(flat_steps)

            if step["op"] == "if":
                # Reserve space for if statement
                flat_steps.append(None)

                # Flatten then block
                then_start = len(flat_steps)
                flatten(step["then_block"], then_start)
                then_count = len(flat_steps) - then_start

                # Flatten else block if present
                else_start = 0
                else_count = 0
                if "else_block" in step:
                    else_start = len(flat_steps)
                    flatten(step["else_block"], else_start)
                    else_count = len(flat_steps) - else_start

                # Fill in if statement with indices
                flat_steps[current_index] = {
                    "op": "if",
                    "condition": step["condition"],
                    "block_start": then_start,
                    "block_count": then_count,
                    "else_start": else_start,
                    "else_count": else_count
                }

            elif step["op"] == "while":
                flat_steps.append(None)
                body_start = len(flat_steps)
                flatten(step["body"], body_start)
                body_count = len(flat_steps) - body_start

                flat_steps[current_index] = {
                    "op": "while",
                    "condition": step["condition"],
                    "block_start": body_start,
                    "block_count": body_count
                }

            elif step["op"] == "for":
                flat_steps.append(None)
                body_start = len(flat_steps)
                flatten(step["body"], body_start)
                body_count = len(flat_steps) - body_start

                flat_steps[current_index] = {
                    "op": "for",
                    "init": step["init"],
                    "condition": step["condition"],
                    "increment": step["increment"],
                    "block_start": body_start,
                    "block_count": body_count
                }
```

```
            else:
                # Regular step (assign, return, etc.)
                flat_steps.append(step)

        return flat_steps

    return flatten(nested_steps, 0)
```

Single-Level Nesting Limitation

v0.6.1 Scope: Single-level nesting only

This means:
- ✅ `if` with simple statements in then/else blocks
- ✅ `while` with simple statements in body
- ✅ `for` with simple statements in body
- ❌ Nested `if` inside `if`
- ❌ `while` inside `if`
- ❌ Any control flow inside control flow

Future v0.7.0: Bounded recursion with depth limits

SPARK Implementation Notes

IR_Step Record (stunir_ir_to_code.ads)

```
type IR_Step is record
   Op            : Name_String;  -- "if", "while", "for", "assign", "return"
   Target        : Name_String;  -- For assign
   Value         : Name_String;  -- For assign, return
   Condition     : Name_String;  -- For if, while, for
   Init          : Name_String;  -- For for
   Increment     : Name_String;  -- For for
   Block_Start   : Natural;      -- Start index of then/body block
   Block_Count   : Natural;      -- Number of steps in then/body block
   Else_Start    : Natural;      -- Start index of else block (0 if none)
   Else_Count    : Natural;      -- Number of steps in else block (0 if none)
end record;
```

Processing Algorithm (stunir_ir_to_code.adb)

```
procedure Process_Block (
   Steps      : in IR_Step_Array;
   Start_Idx  : in Natural;
   Count      : in Natural;
   Output     : in out Unbounded_String;
   Indent     : in Natural
) is
begin
   for I in Start_Idx .. Start_Idx + Count - 1 loop
      Process_Step(Steps(I), Output, Indent);
   end loop;
end Process_Block;

procedure Process_Step (
   Step   : in IR_Step;
   Output : in out Unbounded_String;
   Indent : in Natural
) is
begin
   case Step.Op is
      when "if" =>
         Append(Output, Indent_Str(Indent) & "if (" & Step.Condition & ") {");
         Process_Block(Steps, Step.Block_Start, Step.Block_Count, Output, Indent + 1);

         if Step.Else_Count > 0 then
            Append(Output, Indent_Str(Indent) & "} else {");
            Process_Block(Steps, Step.Else_Start, Step.Else_Count, Output, Indent + 1)
;
         end if;

         Append(Output, Indent_Str(Indent) & "}");

      when "while" =>
         -- Similar pattern

      when "for" =>
         -- Similar pattern

      when others =>
         -- Regular step processing
   end case;
end Process_Step;
```

Schema Version

- **Format**: `stunir_flat_ir_v1`
- **Compatible with**: Ada SPARK pipeline only
- **Generated by**: Python IR converter
- **Consumed by**: SPARK ir_to_code tool

Testing Strategy

1. **Unit Tests**: Test conversion of each control flow type
2. **Integration Tests**: Compare SPARK output with Python output
3. **Validation**: Ensure generated C code is functionally equivalent

Limitations

v0.6.1 Limitations

- Single-level nesting only (no nested control flow)
- Maximum 100 steps per function (SPARK array bounds)
- No recursive function calls

Future Enhancements (v0.7.0+)

- Bounded recursion with depth limits
- Ada 2022 Unbounded_Strings for dynamic sizing
- Full recursive control flow support

Benefits

1. **SPARK Compatible**: No dynamic nesting, static array bounds
2. **Formally Verifiable**: SPARK can prove safety properties
3. **Simple Processing**: Linear array traversal, no recursion needed
4. **Efficient**: O(n) conversion and processing

Compatibility

Pipeline	IR Format	Status
Python	Nested	✅ Full support
Rust	Nested	✅ Full support
SPARK v0.6.0	Flat (limited)	⚠️ No nesting
SPARK v0.6.1	Flat	✅ Single-level nesting
SPARK v0.7.0+	Flat	🔮 Bounded recursion

Conclusion

The flattened IR format enables SPARK to handle single-level nested control flow while maintaining formal verification guarantees. This is a pragmatic solution that balances Ada's static typing constraints with the need for control flow support.

Document Version: v0.6.1
Author: STUNIR Development Team
Date: 2026-01-31
Status: Design Complete