HLI: SPARK Migration Phase 1 - Critical Core Utilities

Document Information

- **Generated:** 2026-01-29
- **Phase:** 1 - Critical Core Migration
- **Target:** Ada SPARK 2014+
- **Source Repository:** /home/ubuntu/stunir_repo/
- **Framework Path:** /home/ubuntu/stunir_implementation_framework/

---

1. Executive Summary

Phase 1 SPARK Migration focuses on converting 5 critical Python utilities to formally verified Ada SPARK code. These utilities form the foundation of STUNIR's safety-critical operations:

#	Python Module	SPARK Component	Lines	Criticality	Proofs
1	`stunir_types/type_system.py`	`core/type_system/`	~800	HIGH	~40
2	`ir/control_flow.py`	`core/ir_transform/`	~600	HIGH	~35
3	`semantic/checker.py`	`core/semantic_checker/`	~400	HIGH	~25
4	`parsers/parse_ir.py`	`core/ir_validator/`	~200	MEDIUM	~15
5	`validators/validate_ir.py`	`core/ir_validator/`	~200	MEDIUM	~15

Total Estimated: ~2,200 Ada lines, ~130 SPARK proofs

---

2. Analysis of Python Components

2.1 Type System (`tools/stunir_types/type_system.py`)

Purpose: Comprehensive type system for cross-language code generation.

Key Abstractions:
- `TypeKind` enum (24 variants)
- `STUNIRType` abstract base class
- Type classes: `IntType` , `FloatType` , `PointerType` , `ArrayType` , `StructType` , etc.
- `TypeRegistry` for managing named types
- `parse_type()` for type string parsing

Critical Operations:
- Type creation with validation
- Type compatibility checking
- Type serialization to IR representation
- Type registry lookup (O(1) access required)

Safety Requirements:
- No invalid type states
- Bounded memory allocation
- Deterministic type comparison
- No runtime exceptions

2.2 Control Flow Analysis (`tools/ir/control_flow.py`)

Purpose: Control flow graph construction and analysis.

Key Abstractions:
- `BasicBlock` with predecessors/successors
- `ControlFlowGraph` container
- `LoopInfo` and `BranchInfo` structures
- `ControlFlowAnalyzer` for IR analysis
- `ControlFlowTranslator` for target code generation

Critical Operations:
- Dominator computation (iterative algorithm)
- Natural loop detection
- Branch merge point detection
- CFG serialization

Safety Requirements:
- No graph corruption
- Bounded traversal (no infinite loops)
- Correct dominator calculation
- Memory-safe block management

2.3 Semantic Checker (`tools/semantic/checker.py`)

Purpose: Semantic analysis and dead code detection.

Key Abstractions:
- `CheckKind` enum
- `CheckResult` with severity levels

- `DeadCodeDetector` for unused code
- `UnreachableCodeDetector` for control flow analysis
- `ConstantExpressionEvaluator` for compile-time evaluation

Critical Operations:
- Dead variable/function detection
- Unreachable code detection
- Constant folding
- Semantic issue reporting

Safety Requirements:
- No false negatives (missed issues)
- Bounded analysis time
- Deterministic results
- No side effects

2.4 IR Parser (`tools/parsers/parse_ir.py`)

Purpose: Parse and validate IR JSON files.

Key Operations:
- Schema validation
- Required/optional field checking
- Module name validation
- Function structure validation

2.5 IR Validator (`tools/validators/validate_ir.py`)

Purpose: Deep validation of IR against schema.

Key Operations:
- Schema lookup and validation
- Content hash computation
- Canonical JSON generation
- Error/warning reporting

---

3. SPARK Architecture Design

3.1 Package Structure

```
core/
├── type_system/
│   ├── stunir_types.ads          -- Type kind enumerations
│   ├── stunir_types.adb          -- Type kind implementations
│   ├── stunir_type_registry.ads  -- Type registry specification
│   ├── stunir_type_registry.adb  -- Type registry implementation
│   └── stunir_type_utils.ads     -- Type utility functions
├── ir_transform/
│   ├── ir_basic_blocks.ads       -- Basic block representation
│   ├── ir_basic_blocks.adb       -- Basic block implementation
│   ├── ir_control_flow.ads       -- CFG specification
│   ├── ir_control_flow.adb       -- CFG implementation
│   └── ir_dominators.ads         -- Dominator analysis
├── semantic_checker/
│   ├── semantic_analysis.ads     -- Analysis specification
│   ├── semantic_analysis.adb     -- Analysis implementation
│   ├── dead_code_detector.ads    -- Dead code detection
│   └── constant_evaluator.ads    -- Constant expression evaluation
├── ir_validator/
│   ├── ir_parser.ads             -- IR parsing specification
│   ├── ir_parser.adb             -- IR parsing implementation
│   ├── ir_validator.ads          -- Validation specification
│   └── ir_validator.adb          -- Validation implementation
└── common/
    ├── stunir_strings.ads        -- Bounded string types
    ├── stunir_hashes.ads         -- SHA-256 implementation
    └── stunir_json.ads           -- JSON handling
```

3.2 Key SPARK Design Decisions

1. **Bounded Containers:** Use `Ada.Containers.Formal_*` for provable containers
2. **No Exceptions:** Replace with discriminated result types
3. **Static Allocation:** Maximum sizes defined at compile time
4. **Explicit State:** All mutable state in explicit records
5. **Contract-Based Design:** Pre/postconditions on all operations

---

4. Data Structure Mappings

4.1 Type System Mappings

Python	Ada SPARK
`enum TypeKind`	`type Type_Kind is (...)`
`class STUNIRType(ABC)`	`type STUNIR_Type is tagged limited private`
`@dataclass IntType`	`type Int_Type is new STUNIR_Type with record...`
`Dict[str, STUNIRType]`	`Formal_Hashed_Maps.Map`
`List[StructField]`	`Formal_Vectors.Vector`
`Optional[Lifetime]`	`type Opt_Lifetime is (None, Some with ...)`

4.2 Control Flow Mappings

Python	Ada SPARK
`class BasicBlock`	`type Basic_Block is limited private`
`List[int] successors`	`Block_Set` (bounded vector)
`Set[int] dominators`	`Dominator_Set` (formal set)
`Dict[int, BasicBlock]`	`Block_Map` (formal map)
`LoopInfo`	`type Loop_Info is record...`

4.3 Semantic Checker Mappings

Python	Ada SPARK
`enum CheckKind`	`type Check_Kind is (...)`
`CheckResult`	`type Check_Result is record...`
`Dict[str, int]`	`Name_Count_Map`
`Set[str]`	`Name_Set`

---

5. Algorithm Translations

5.1 Dominator Computation

Python (iterative):

```python
def compute_dominators(self) -> None:
    all_blocks = set(self.blocks.keys())
    for block_id in self.blocks:
        if block_id == self.entry_id:
            self.blocks[block_id].dominators = {block_id}
        else:
            self.blocks[block_id].dominators = all_blocks.copy()

    changed = True
    while changed:
        changed = False
        for block_id in self.blocks:
            if block_id == self.entry_id:
                continue
            # Dom(n) = {n} ∪ (∩ Dom(p) for p in pred(n))
            new_doms = all_blocks.copy()
            for pred_id in self.blocks[block_id].predecessors:
                new_doms &= self.blocks[pred_id].dominators
            new_doms.add(block_id)
            if new_doms != self.blocks[block_id].dominators:
                self.blocks[block_id].dominators = new_doms
                changed = True
```

Ada SPARK:

```
procedure Compute_Dominators (CFG : in out Control_Flow_Graph)
  with
    Global => null,
    Pre => CFG.Entry_Block /= No_Block and CFG.Block_Count > 0,
    Post => All_Dominators_Valid (CFG)
is
   Changed : Boolean := True;
begin
   -- Initialize: entry dominates only itself
   Initialize_Dominators (CFG);

   -- Iterate until fixed point (bounded by block count squared)
   while Changed loop
      pragma Loop_Variant (Decreases => Remaining_Changes (CFG));
      Changed := False;

      for Block_Id in CFG.Blocks'Range loop
         if Block_Id /= CFG.Entry_Block then
            declare
               New_Doms : constant Dominator_Set :=
                  Compute_New_Dominators (CFG, Block_Id);
            begin
               if New_Doms /= Get_Dominators (CFG, Block_Id) then
                  Set_Dominators (CFG, Block_Id, New_Doms);
                  Changed := True;
               end if;
            end;
         end if;
      end loop;
   end loop;
end Compute_Dominators;
```

5.2 Dead Code Detection

Python:

```
def _is_terminating(self, stmt: Any) -> bool:
    if not isinstance(stmt, dict):
        return False
    stmt_type = stmt.get('type', '')
    return stmt_type in ('return', 'break', 'continue', 'goto', 'throw')
```

Ada SPARK:

```
function Is_Terminating (Stmt : Statement) return Boolean is
  (case Stmt.Kind is
     when Return_Stmt | Break_Stmt | Continue_Stmt | Goto_Stmt | Throw_Stmt => True,
     when others => False)
  with Inline;
```

---

6. Formal Specifications

6.1 Type System Contracts

```
package Stunir_Types is
   pragma SPARK_Mode (On);

   type Type_Kind is (
      Void_Kind, Bool_Kind, Int_Kind, Float_Kind, Char_Kind, String_Kind,
      Pointer_Kind, Reference_Kind, Array_Kind, Slice_Kind,
      Struct_Kind, Union_Kind, Enum_Kind, Tagged_Union_Kind,
      Function_Kind, Closure_Kind, Generic_Kind, Type_Var_Kind,
      Opaque_Kind, Recursive_Kind, Optional_Kind, Result_Kind,
      Tuple_Kind, Unit_Kind);

   Max_Type_Name_Length : constant := 256;
   subtype Type_Name is String (1 .. Max_Type_Name_Length);

   type Int_Bits is range 8 .. 128
     with Static_Predicate => Int_Bits in 8 | 16 | 32 | 64 | 128;

   type STUNIR_Type (Kind : Type_Kind := Void_Kind) is record
      case Kind is
         when Int_Kind =>
            Bits   : Int_Bits := 32;
            Signed : Boolean := True;
         when Float_Kind =>
            Float_Bits : Int_Bits := 64;
         when Pointer_Kind =>
            Pointee_Kind : Type_Kind;
            Nullable     : Boolean := True;
         when Array_Kind =>
            Element_Kind : Type_Kind;
            Array_Size   : Natural := 0;  -- 0 = dynamic
         when Struct_Kind =>
            Field_Count  : Natural := 0;
         when others =>
            null;
      end case;
   end record;

   function Is_Valid (T : STUNIR_Type) return Boolean
     with Post => Is_Valid'Result = (T.Kind in Type_Kind);

   function Is_Primitive (T : STUNIR_Type) return Boolean is
     (T.Kind in Void_Kind | Bool_Kind | Int_Kind | Float_Kind | Char_Kind |
Unit_Kind);

   function Is_Pointer_Like (T : STUNIR_Type) return Boolean is
     (T.Kind in Pointer_Kind | Reference_Kind | Slice_Kind);

end Stunir_Types;
```

6.2 Control Flow Contracts

```
package IR_Control_Flow is
   pragma SPARK_Mode (On);

   Max_Blocks : constant := 10_000;
   Max_Successors : constant := 64;

   type Block_Id is range 0 .. Max_Blocks - 1;
   No_Block : constant Block_Id := Block_Id'First;

   type Block_Type is (
      Entry_Block, Exit_Block, Normal_Block, Conditional_Block,
      Loop_Header_Block, Loop_Body_Block, Loop_Exit_Block,
      Switch_Block, Case_Block, Try_Block, Catch_Block, Finally_Block);

   type Successor_Array is array (Positive range <>) of Block_Id
     with Dynamic_Predicate => Successor_Array'Length <= Max_Successors;

   type Basic_Block is record
      Id            : Block_Id := No_Block;
      Block_Kind    : Block_Type := Normal_Block;
      Successors    : Successor_Count := 0;
      Predecessors  : Predecessor_Count := 0;
      Loop_Depth    : Natural := 0;
      Is_Loop_Header : Boolean := False;
   end record;

   type Control_Flow_Graph is limited private;

   procedure Add_Edge (
      CFG     : in out Control_Flow_Graph;
      From_Id : Block_Id;
      To_Id   : Block_Id)
     with
       Pre => From_Id /= No_Block and To_Id /= No_Block and
              Block_Exists (CFG, From_Id) and Block_Exists (CFG, To_Id),
       Post => Has_Edge (CFG, From_Id, To_Id);

   function Dominates (
      CFG : Control_Flow_Graph;
      A   : Block_Id;
      B   : Block_Id) return Boolean
     with
       Pre => Block_Exists (CFG, A) and Block_Exists (CFG, B);

private
   type Block_Array is array (Block_Id) of Basic_Block;

   type Control_Flow_Graph is record
      Blocks      : Block_Array;
      Block_Count : Natural := 0;
      Entry_Block : Block_Id := No_Block;
      Exit_Block  : Block_Id := No_Block;
   end record;

end IR_Control_Flow;
```

6.3 Semantic Checker Contracts

```
package Semantic_Analysis is
   pragma SPARK_Mode (On);

   type Check_Kind is (
      Type_Compatibility, Null_Safety, Bounds_Check, Overflow_Check,
      Dead_Code, Unreachable_Code, Resource_Leak, Constant_Expression,
      Purity, Side_Effects);

   type Warning_Severity is (Info, Warning, Error, Fatal);

   type Check_Result is record
      Kind     : Check_Kind;
      Passed   : Boolean;
      Severity : Warning_Severity := Info;
   end record;

   type Check_Results is array (Positive range <>) of Check_Result
     with Dynamic_Predicate => Check_Results'Length <= 1000;

   procedure Analyze_Dead_Code (
      IR      : IR_Data;
      Results : out Check_Results;
      Count   : out Natural)
     with
       Pre  => IR_Is_Valid (IR),
       Post => Count <= Results'Length and
               (for all I in 1 .. Count => Results (I).Kind = Dead_Code);

   function Is_Terminating (Stmt : Statement) return Boolean
     with Pure_Function;

end Semantic_Analysis;
```

---

7. Integration Strategy

7.1 Foreign Function Interface

```
-- C interface for Python interoperability
package Stunir_C_Interface is
   pragma SPARK_Mode (Off);  -- FFI not SPARK-provable

   function Create_Type_Registry return System.Address
     with Export, Convention => C, External_Name => "stunir_create_registry";

   function Register_Type (
      Registry : System.Address;
      Name     : Interfaces.C.Strings.chars_ptr;
      Kind     : Interfaces.C.int) return Interfaces.C.int
     with Export, Convention => C, External_Name => "stunir_register_type";

end Stunir_C_Interface;
```

7.2 Build Integration

GNAT Project File (stunir_core.gpr):

```
project Stunir_Core is
   for Source_Dirs use (
      "core/type_system",
      "core/ir_transform",
      "core/semantic_checker",
      "core/ir_validator",
      "core/common"
   );

   for Object_Dir use "obj";
   for Library_Dir use "lib";
   for Library_Name use "stunir_core";
   for Library_Kind use "static";

   package Compiler is
      for Default_Switches ("Ada") use (
         "-gnat2022",        -- Ada 2022
         "-gnata",           -- Enable assertions
         "-gnatwa",          -- All warnings
         "-gnatwe",          -- Warnings as errors
         "-gnatVa",          -- All validity checks
         "-gnato13"          -- Overflow checks
      );
   end Compiler;

   package Prove is
      for Proof_Switches ("Ada") use (
         "--level=2",        -- Medium proof effort
         "--timeout=60",     -- 60 seconds per VC
         "--prover=all"      -- Use all provers
      );
   end Prove;

end Stunir_Core;
```

---

8. Test Strategy

8.1 Unit Tests

```
-- Test package for type system
package Stunir_Types_Tests is

   procedure Test_Int_Type_Creation;
   procedure Test_Pointer_Type_Creation;
   procedure Test_Array_Type_Creation;
   procedure Test_Type_Registry_Lookup;
   procedure Test_Type_Equality;

end Stunir_Types_Tests;
```

8.2 Equivalence Tests

For each migrated component, create Python-SPARK equivalence tests:

```python
# tests/equivalence/test_type_system.py
import ctypes
import json
from tools.stunir_types.type_system import IntType, TypeRegistry

# Load SPARK library
spark_lib = ctypes.CDLL('./lib/libstunir_core.so')

def test_int_type_equivalence():
    """Verify Python and SPARK produce identical results."""
    # Python
    py_type = IntType(bits=32, signed=True)
    py_ir = py_type.to_ir()

    # SPARK (via FFI)
    spark_registry = spark_lib.stunir_create_registry()
    spark_type_id = spark_lib.stunir_create_int_type(32, 1)
    spark_ir = json.loads(spark_lib.stunir_type_to_json(spark_type_id))

    assert py_ir == spark_ir, f"Mismatch: {py_ir} vs {spark_ir}"
```

8.3 Property-Based Tests

```ada
-- Property tests using GNATtest
procedure Test_Dominator_Properties is
   CFG : Control_Flow_Graph;
begin
   -- Property: Every block dominates itself
   for B in CFG.Blocks'Range loop
      pragma Assert (Dominates (CFG, B, B));
   end loop;

   -- Property: Entry dominates all reachable blocks
   for B in CFG.Blocks'Range loop
      if Is_Reachable (CFG, B) then
         pragma Assert (Dominates (CFG, CFG.Entry_Block, B));
      end if;
   end loop;
end Test_Dominator_Properties;
```

---

9. Implementation Guidance

9.1 Priority Order

1. **Common utilities** (strings, hashes) - Foundation
2. **Type system** - Required by all other components
3. **IR validator/parser** - Input validation
4. **Control flow** - Depends on types
5. **Semantic checker** - Depends on all above

9.2 Key Patterns

Result Type Pattern (replacing exceptions):

```
type Result_Kind is (Ok, Err);

type Parse_Result (Kind : Result_Kind := Err) is record
   case Kind is
      when Ok =>
         Value : IR_Data;
      when Err =>
         Error_Code : Error_Kind;
         Error_Msg  : Error_Message;
   end case;
end record;
```

Bounded String Pattern:

```
Max_String_Length : constant := 1024;
subtype Bounded_String is String (1 .. Max_String_Length);
type Bounded_String_Holder is record
   Data   : Bounded_String := (others => ' ');
   Length : Natural := 0;
end record;
```

---

10. Expected Outcomes

10.1 Deliverables

- 10-15 Ada specification files (.ads)
- 10-15 Ada body files (.adb)
- 1 GNAT project file (.gpr)
- 1 Makefile for automation
- ~2,200 lines of Ada SPARK code
- ~130 proven verification conditions
- Documentation and examples

10.2 Proof Targets

Component	VCs Target	Effort
Type System	~40	Medium
Control Flow	~35	High
Semantic Checker	~25	Medium
IR Validator	~30	Low
Total	**~130**	

10.3 Success Criteria

- All SPARK proofs pass (`gnatprove` clean)
- Functional equivalence with Python versions

- No runtime exceptions possible
- All tests passing (2,138+ existing + new)
- Integration with existing STUNIR build

Appendix A: File Mapping Reference

Python Source	SPARK Target
`stunir_types/type_system.py`	`core/type_system/stunir_types.ads/adb`
`ir/control_flow.py`	`core/ir_transform/ir_control_flow.ads/adb`
`semantic/checker.py`	`core/semantic_checker/semantic_analysis.ads/adb`
`parsers/parse_ir.py`	`core/ir_validator/ir_parser.ads/adb`
`validators/validate_ir.py`	`core/ir_validator/ir_validator.ads/adb`

Document generated for STUNIR Phase 1 SPARK Migration