STUNIR Confluence Specification v1.0

Document Status: DRAFT
Date: 2026-01-30
Author: STUNIR Team
Purpose: Define complete feature parity requirements across all four STUNIR pipelines

---

Executive Summary

Confluence is the state where all four STUNIR pipelines (SPARK, Python, Rust, Haskell) produce **bit-wise-identical outputs** from the same inputs. This specification defines what "complete parity" means, establishes testing criteria, and documents the implementation requirements for achieving confluence.

Critical Requirement

Different organizations have different language requirements for auditability and assurance:
- **Python**: Required by organizations prioritizing readability and human review
- **Haskell**: Required by organizations where maturity equals assurance (formal methods)
- **Rust**: Required by pure Rust environments (safety + performance guarantees)
- **SPARK**: Required by DO-178C Level A certification (avionics, medical devices)

All pipelines must be **production-ready** and capable of the **same functionality**. Humans must be able to review the ENTIRE pipeline in their preferred language.

---

1. What is Confluence?

1.1 Definition

Confluence is achieved when:
1. All four pipelines can process the same specification inputs
2. All four pipelines produce **bitwise-identical** Intermediate Reference (IR) outputs
3. All four pipelines can emit **bitwise-identical** code outputs for all 24 target categories
4. All four pipelines pass the same deterministic verification tests

1.2 Why Confluence Matters

1. **Organizational Acceptance**: Different organizations only accept specific languages for audit
2. **Formal Verification**: Each pipeline must be independently verifiable
3. **Deterministic Trust**: Identical outputs prove implementation correctness
4. **Language-Specific Assurance**: Teams can review code in their trusted language
5. **Cross-Validation**: Discrepancies between pipelines indicate bugs

1.3 Non-Goals

- **Performance parity**: Pipelines may have different speeds
- **Memory usage parity**: Pipelines may have different resource profiles

- **Build system parity**: Each language uses its native tooling
- **Line-by-line equivalence**: Implementations can differ algorithmically while producing identical outputs

---

2. The 24 Target Categories

All pipelines must support code generation for these 24 categories:

2.1 Core Categories (6)

#	Category	Description	Example Targets
1	**Assembly**	Low-level assembly code	ARM, x86, RISC-V, MIPS
2	**Embedded**	Embedded system code	ARM Cortex-M, AVR, bare-metal
3	**Polyglot**	Multi-language targets	C89, C99, C++, Rust, Go
4	**GPU**	GPU acceleration code	CUDA, ROCm, OpenCL, Metal, Vulkan
5	**WASM**	WebAssembly targets	WASM, WASI, WAT
6	**Native**	Native system code	Platform-specific optimizations

2.2 Language Families (8)

#	Category	Description	Example Targets
7	**Lisp**	Lisp family languages	Common Lisp, Scheme, Clojure, Racket, Emacs Lisp, Guile, Hy, Janet
8	**Prolog**	Logic programming	SWI-Prolog, GNU Prolog, YAP, XSB, Tau Prolog, Mercury, Logtalk, Datalog
9	**Functional**	Functional languages	Haskell, OCaml, Erlang, Elixir, F#
10	**OOP**	Object-oriented	Java, C#, Python, Ruby, Kotlin
11	**Systems**	Systems languages	C, C++, Rust, Zig, D
12	**Business**	Business logic	COBOL, RPG, PL/I
13	**Scientific**	Scientific computing	Fortran, MATLAB, Julia, R
14	**Mobile**	Mobile platforms	Swift, Kotlin, React Native

2.3 Specialized (10)

#	Category	Description	Example Targets
15	**ASM**	Assembly variants	NASM, MASM, GAS
16	**ASP**	Answer Set Programming	Clingo, DLV, Potassco
17	**BEAM**	BEAM VM targets	Erlang, Elixir, LFE, Gleam
18	**Bytecode**	Bytecode formats	JVM, CLR, Python bytecode
19	**Constraints**	Constraint solving	MiniZinc, Essence, OPL
20	**Expert Systems**	Rule engines	CLIPS, Drools, Jess, PDDL
21	**FPGA**	Hardware description	VHDL, Verilog, Chisel
22	**Grammar**	Parser generators	ANTLR, Bison, PEG, Yacc
23	**Lexer**	Lexical analyzers	Flex, Lex, Ragel
24	**Parser**	Parsing libraries	Parsec, Nom, Tree-sitter
25	**Planning**	AI planning	PDDL, STRIPS, HTN

---

3. Current Implementation Status

3.1 Pipeline Audit (as of 2026-01-30)

Pipeline	Files	Status	Completeness
SPARK	73	✅ Reference	24/24 categories (100%)
Python	121	⚠️ Partial	24/24 categories (varying depth)
Rust	0	❌ Missing	0/24 categories (0%)
Haskell	0	❌ Missing	0/24 categories (0%)

3.2 Category Coverage Matrix

See `tools/confluence/category_matrix.json` for detailed per-category status.

3.3 Critical Gaps

1. **Rust Pipeline**: Completely missing
   - No `spec_to_ir` implementation
   - No `ir_to_code` implementation
   - No target emitters (0/24 categories)

2. **Haskell Pipeline**: Completely missing
   - No `spec_to_ir` implementation
   - No `ir_to_code` implementation
   - No target emitters (0/24 categories)

3. **Python Pipeline**: Varying depth
   - Core tools exist but flagged as “reference implementation”
   - Some categories have minimal implementations
   - Needs enhancement to match SPARK completeness

---

4. Testing Criteria for Confluence

4.1 IR Generation Tests

For each pipeline, verify:

```
# Test spec_to_ir across all pipelines
spark_spec_to_ir spec.json > spark_ir.json
python_spec_to_ir spec.json > python_ir.json
rust_spec_to_ir spec.json > rust_ir.json
haskell_spec_to_ir spec.json > haskell_ir.json

# Verify bitwise identity
sha256sum spark_ir.json python_ir.json rust_ir.json haskell_ir.json
```

All four hashes must be **identical**.

4.2 Code Emission Tests

For each of the 24 categories and each pipeline:

```
# Test ir_to_code for category X
spark_ir_to_code ir.json --target=X > spark_output_X.txt
python_ir_to_code ir.json --target=X > python_output_X.txt
rust_ir_to_code ir.json --target=X > rust_output_X.txt
haskell_ir_to_code ir.json --target=X > haskell_output_X.txt

# Verify bitwise identity
sha256sum spark_output_X.txt python_output_X.txt rust_output_X.txt haskell_output_X.tx
t
```

All four hashes must be **identical** for **all 24 categories**.

4.3 Automated Confluence Test Suite

Location: `tools/confluence/test_confluence.sh`

This script:
1. Runs all four pipelines on a comprehensive test suite
2. Compares outputs using SHA-256 hashes
3. Reports any discrepancies
4. Generates a confluence report

Example output:

```
=== STUNIR Confluence Test Report ===
Date: 2026-01-30
Test Cases: 48

✅ spec_to_ir: 4/4 pipelines produce identical IR
✅ assembly/arm: 4/4 pipelines produce identical output
✅ assembly/x86: 4/4 pipelines produce identical output
❌ gpu/cuda: 3/4 pipelines produce identical output
   - Haskell output differs at line 42
✅ lisp/scheme: 4/4 pipelines produce identical output
...

Confluence Score: 95.8% (46/48 tests passing)
```

4.4 Success Criteria

Confluence is achieved when:
- **100%** of spec_to_ir tests pass (all 4 pipelines produce identical IR)
- **100%** of ir_to_code tests pass for all 24 categories (all 4 pipelines produce identical code)
- Automated test suite shows **100%** confluence score

---

5. Implementation Requirements

5.1 Core Toolchain Requirements

Each pipeline must implement:

1. **spec_to_ir**
   - Input: JSON specification file
   - Output: Canonical IR in JSON format
   - Hash: Deterministic SHA-256 computation
   - Verification: Must match reference (SPARK) hash

2. **ir_to_code**
   - Input: IR JSON + target category
   - Output: Generated code for specified target
   - Hash: Deterministic SHA-256 computation
   - Verification: Must match reference (SPARK) hash

5.2 Target Emitter Requirements

For each of the 24 categories, each pipeline must:
1. Accept IR input in canonical format
2. Generate code according to target specifications
3. Produce bitwise-identical output to reference implementation
4. Pass category-specific validation tests

5.3 Organizational Structure

```
targets/
├── spark/          # Ada SPARK implementations (DO-178C Level A)
│   ├── assembly/
│   ├── embedded/
│   ├── polyglot/
│   ├── gpu/
│   ├── wasm/
│   ├── lisp/
│   ├── prolog/
│   └── ... (24 categories)
├── python/         # Python implementations (readability focus)
│   ├── assembly/
│   ├── embedded/
│   └── ... (24 categories)
├── rust/           # Rust implementations (safety + performance)
│   ├── assembly/
│   ├── embedded/
│   └── ... (24 categories)
└── haskell/        # Haskell implementations (formal methods)
    ├── assembly/
    ├── embedded/
    └── ... (24 categories)
```

5.4 Build System Requirements

The build system must:
1. Support runtime selection: `--runtime=spark|python|rust|haskell`
2. Detect available implementations and fallback gracefully
3. Provide precompiled binaries for all pipelines
4. Support cross-platform builds (Linux, macOS, Windows)

Example:

```
# Build with specific runtime
./scripts/build.sh --runtime=rust --target=embedded

# Auto-detect best available runtime
./scripts/build.sh --runtime=auto

# Build all pipelines
./scripts/build.sh --runtime=all
```

---

6. Phased Implementation Plan

Phase 1: Infrastructure (PRIORITY)

- [ ] Create Rust core toolchain (`spec_to_ir` , `ir_to_code`)
- [ ] Create Haskell core toolchain (`spec_to_ir` , `ir_to_code`)
- [ ] Implement Python core toolchain (enhance existing)
- [ ] Create confluence test framework
- [ ] Update build system for multi-runtime support

Phase 2: Representative Emitters (PRIORITY)

For each pipeline, implement **one representative emitter per category** (24 total):
- [ ] Assembly (x86 or ARM)
- [ ] Embedded (ARM Cortex-M)
- [ ] Polyglot (C99)
- [ ] GPU (CUDA)
- [ ] WASM (basic WASM)
- [ ] Lisp (Common Lisp)
- [ ] Prolog (SWI-Prolog)
- [ ] ... (all 24)

Phase 3: Complete Emitters

Expand each category to support all targets within category (e.g., all 8 Lisp dialects).

Phase 4: Verification & Documentation

- [ ] Run full confluence test suite
- [ ] Document all implementations
- [ ] Create migration guides
- [ ] Generate precompiled binaries

---

7. Validation Methodology

7.1 Reference Implementation

SPARK is the reference implementation. All other pipelines must produce outputs that match SPARK's outputs exactly.

7.2 Test Vectors

Standard test vectors are located in `tests/confluence/` :
- `minimal.json` : Minimal spec (empty module)
- `simple.json` : Simple function with basic types
- `complex.json` : Full-featured specification
- `stress.json` : Large-scale specification

7.3 Verification Process

1. **Generate reference outputs** using SPARK pipeline
2. **Store reference hashes** in `tests/confluence/hashes/`
3. **Run other pipelines** and compare hashes

4. **Report discrepancies** with detailed diffs
5. **Iterate** until 100% confluence achieved

---

8. Documentation Requirements

Each pipeline must provide:

1. **Implementation Guide**: How the pipeline works internally
2. **Usage Guide**: How to use the pipeline
3. **Build Guide**: How to build from source
4. **Testing Guide**: How to run tests and verify correctness
5. **Assurance Case**: Why the implementation is trustworthy (language-specific)

Examples:
- `docs/PYTHON_PIPELINE.md`
- `docs/RUST_PIPELINE.md`
- `docs/HASKELL_PIPELINE.md`
- `docs/SPARK_PIPELINE.md`

---

9. Acceptance Criteria

Confluence is **officially achieved** when:

1. ✅ All four pipelines implement `spec_to_ir` and `ir_to_code`
2. ✅ All four pipelines implement representative emitters for all 24 categories
3. ✅ Automated confluence test suite passes at 100%
4. ✅ All pipelines produce bitwise-identical outputs for standard test vectors
5. ✅ Build system supports all four runtimes with `--runtime` flag
6. ✅ Precompiled binaries available for all pipelines (Linux, macOS, Windows)
7. ✅ Documentation complete for all pipelines
8. ✅ Each pipeline reviewed by language-specific experts

---

10. Maintenance Strategy

10.1 Adding New Targets

When adding a new target category:
1. Implement in SPARK first (reference)
2. Generate reference outputs and hashes
3. Implement in Python, Rust, Haskell
4. Verify confluence using test suite
5. Update documentation

10.2 Fixing Bugs

When fixing a bug:
1. Fix in all four pipelines simultaneously
2. Add regression test to confluence suite
3. Verify confluence maintained

10.3 CI/CD Requirements

CI pipeline must:
1. Build all four pipelines
2. Run confluence test suite
3. Report confluence score
4. Block merge if confluence < 100%

---

11. Current Progress Tracking

11.1 Metrics

- **Category Coverage**: 24/24 categories defined
- **SPARK Completeness**: 24/24 (100%)
- **Python Completeness**: 24/24 (varying depth, ~70% estimated)
- **Rust Completeness**: 0/24 (0%)
- **Haskell Completeness**: 0/24 (0%)
- **Confluence Score**: TBD (test suite not yet implemented)

11.2 Next Steps

1. Implement Rust core toolchain (highest priority)
2. Implement Haskell core toolchain (highest priority)
3. Create confluence test framework
4. Implement representative emitters for Rust (24 categories)
5. Implement representative emitters for Haskell (24 categories)
6. Run full test suite and measure confluence score
7. Iterate to 100% confluence

---

12. Appendices

Appendix A: Test Case Specifications

See `tests/confluence/TEST_CASES.md`

Appendix B: Category Definitions

See `contracts/target_requirements.json`

Appendix C: Hash Computation Standards

See `docs/hash_computation.md`

Appendix D: Organizational Requirements

See `docs/ORGANIZATIONAL_REQUIREMENTS.md`

Document Control:
- **Version**: 1.0
- **Status**: DRAFT
- **Next Review**: After Phase 1 completion
- **Owner**: STUNIR Team
- **Approvers**: TBD

CRITICAL NOTE: This is a multi-year effort. The goal is to create **infrastructure** and **representative examples** first, establishing patterns that can be replicated across all categories. Complete implementations of all variants within each category can be phased over time, but **core toolchain** and **one representative per category** are minimum requirements for confluence.