ATOMiK SDK - Multi-Language Code Generator

Version: 1.0.0 **Phase:** 4A - SDK Development **Status:** Complete

tests passing languages 5 license MIT

Overview

The ATOMiK SDK is a schema-driven code generation framework that produces delta-state computing primitives for multiple programming languages and hardware targets. Starting from a single JSON schema specification, it generates production-ready implementations in Python, Rust, C, JavaScript, and synthesizable Verilog RTL.

Key Features

- ☑ **Multi-Language Support**: Python, Rust, C, JavaScript, Verilog
- ☑ **Schema-Driven**: Single JSON schema → 5 language implementations
- ☑ **Hardware Integration**: Verilog RTL matching FPGA architecture
- ☑ **Namespace Consistency**: Automatic cross-language mapping
- ☑ **Comprehensive Testing**: 100% pass rate across all generators
- ☑ **Mathematical Guarantees**: Based on proven XOR algebra (92 Lean4 theorems)

Quick Start

Installation

```
# Clone repository
git clone https://github.com/your-org/ATOMiK.git
cd ATOMiK/software/atomik_sdk

# Install Python dependencies
pip install jsonschema
```

Basic Usage

```
from pathlib import Path
from generator.core import GeneratorEngine, GeneratorConfig
from generator.python_generator import PythonGenerator

# Create engine
engine = GeneratorEngine(GeneratorConfig(
    output_dir=Path("./output"),
    validate_schemas=True
))

# Register Python generator
```

```
engine.register_generator('python', PythonGenerator())

# Generate from schema
engine.load_schema(Path("schema.json"))
results = engine.generate(target_languages=['python'])
files = engine.write_output(results)

print(f"Generated {len(files)} files")
```

Generate All Languages

```
from generator.python_generator import PythonGenerator
from generator.rust_generator import RustGenerator
from generator.c_generator import CGenerator
from generator.verilog_generator import VerilogGenerator
from generator.javascript_generator import JavaScriptGenerator

# Register all generators
engine.register_generator('python', PythonGenerator())
engine.register_generator('rust', RustGenerator())
engine.register_generator('c', CGenerator())
engine.register_generator('verilog', VerilogGenerator())
engine.register_generator('javascript', JavaScriptGenerator())

# Generate all languages at once
results, files = engine.generate_and_write(
    schema_path=Path("schema.json"),
    target_languages=['python', 'rust', 'c', 'verilog', 'javascript']
)

print(f"Generated {len(files)} total files across 5 languages")
```

Supported Languages

Language	Target	Output Files	Validation
Python	Python 3.8+	3 files (module, **init**, tests)	py_compile
Rust	Rust 2021	5 files (lib, mod, module, Cargo.toml, tests)	cargo check
C	C99	4 files (.h, .c, test, Makefile)	gcc
Verilog	FPGA	3 files (module, testbench, constraints)	iverilog
JavaScript	Node.js 14+	4 files (module, index, package.json, tests)	node

Example Schema

```
{
  "catalogue": {
    "vertical": "System",
    "field": "Terminal",
    "object": "TerminalIO",
    "version": "1.0.0",
    "description": "Terminal I/O delta-state module"
  },
  "schema": {
    "delta_fields": {
      "command_delta": {
        "type": "parameter_delta",
        "width": 64,
        "description": "Command parameter deltas"
      },
      "response_delta": {
        "type": "parameter_delta",
        "width": 64,
        "description": "Response parameter deltas"
      }
    },
    "operations": {
      "accumulate": {
        "enabled": true
      },
      "reconstruct": {
        "enabled": true
      }
    }
  },
  "hardware": {
    "target": "FPGA",
    "device": "GW1NR-LV9QN88PC6/I5",
    "clock_mhz": 94.5,
    "interface": "UART",
    "data_width": 64
  }
}
```

Generated Code Examples

Python

```
from atomik.System.Terminal import TerminalIO

manager = TerminalIO()
manager.load(0x1234567890ABCDEF)
manager.accumulate(0x1111111111111111)
current_state = manager.reconstruct()
```

Rust

```rust
use atomik::system::terminal::TerminalIO;

let mut manager = TerminalIO::new();
manager.load(0x1234567890ABCDEF);
manager.accumulate(0x1111111111111111);
let current_state = manager.reconstruct();
```

C

```c
#include <atomik/system/terminal/terminal_io.h>

atomik_terminal_io_t manager;
atomik_terminal_io_init(&manager);
atomik_terminal_io_load(&manager, 0x1234567890ABCDEFULL);
atomik_terminal_io_accumulate(&manager, 0x1111111111111111ULL);
uint64_t current_state = atomik_terminal_io_reconstruct(&manager);
```

JavaScript

```javascript
import { TerminalIO } from '@atomik/system/terminal';

const manager = new TerminalIO();
manager.load(0x1234567890ABCDEFn);
manager.accumulate(0x1111111111111111n);
const currentState = manager.reconstruct();
```

Verilog

```verilog
atomik_system_terminal_terminal_io #(
    .DATA_WIDTH(64)
) dut (
    .clk(clk),
    .rst_n(rst_n),
    .load_en(load_en),
    .accumulate_en(accumulate_en),
    .read_en(read_en),
    .data_in(data_in),
    .data_out(data_out),
    .accumulator_zero(accumulator_zero)
);
```

Architecture

Component Structure

```
generator/
├── core.py               # GeneratorEngine orchestrator
├── schema_validator.py  # JSON Schema validation
├── namespace_mapper.py  # Cross-language namespace mapping
├── code_emitter.py      # Base classes for code generation
├── python_generator.py  # Python SDK generator
├── rust_generator.py    # Rust SDK generator
├── c_generator.py       # C SDK generator
├── verilog_generator.py # Verilog RTL generator
└── javascript_generator.py # JavaScript SDK generator
```

Generation Pipeline

```
JSON Schema
    ↓
SchemaValidator (validates schema)
    ↓
NamespaceMapper (extracts catalogue metadata)
    ↓
GeneratorEngine (orchestrates generation)
    ↓
CodeEmitter plugins (generate language-specific code)
    ↓
Generated Files (written to output directory)
```

Testing

Run All Tests

```
# Unit tests
python tests/test_generator_simple.py
python tests/test_python_generation.py
python tests/test_rust_generation.py
python tests/test_c_generation.py
python tests/test_verilog_generation.py
python tests/test_javascript_generation.py

# Integration tests
python tests/test_integration.py
```

Test Coverage

- ☑ Schema validation (JSON Schema Draft 7)
- ☑ Namespace mapping consistency

- ☑ Code generation for all 5 languages
- ☑ Syntax validation (language-specific compilers)
- ☑ Semantic equivalence across languages
- ☑ Cross-field dependency validation
- ☑ Hardware constraint checking

Test Results

All tests passing:

- **3 example schemas** tested
- **5 language generators** validated
- **57 total files** generated per test run
- **100% pass rate** on syntax validation

Documentation

- SDK User Manual - End-user guide
- SDK Developer Guide - Developer documentation
- SDK API Reference - API documentation for all languages
- Schema Guide - Schema specification guide
- Schema Validation Rules - Validation requirements

Project Status

Phase 4A - Complete ☑

Task	Status	Description
T4A.1	☑ Complete	JSON Schema Specification
T4A.2	☑ Complete	Generator Framework
T4A.3	☑ Complete	Python SDK Generator
T4A.4	☑ Complete	Rust SDK Generator
T4A.5	☑ Complete	C SDK Generator
T4A.6	☑ Complete	Verilog RTL Generator
T4A.7	☑ Complete	JavaScript SDK Generator
T4A.8	☑ Complete	Integration Tests
T4A.9	☑ Complete	SDK Documentation

Statistics

- **Generator files**: 10 Python modules
- **Test files**: 8 comprehensive test suites
- **Lines of code**: ~4,500+ across all generators
- **Documentation pages**: 5 comprehensive guides

- **Supported languages**: Python, Rust, C, Verilog, JavaScript

Mathematical Foundation

The ATOMiK SDK is based on formally verified delta algebra with:

- **92 Lean4 theorems** proven (Phase 1)
- **XOR-based operations** with mathematical guarantees:
  - Commutativity: `A ⊕ B = B ⊕ A`
  - Associativity: `(A ⊕ B) ⊕ C = A ⊕ (B ⊕ C)`
  - Self-inverse: `A ⊕ A = 0`
  - Identity: `A ⊕ 0 = A`

All generated SDKs preserve these properties.

Hardware Integration

FPGA Validation (Phase 3)

- ☑ Synthesized on Tang Nano 9K (GW1NR-9)
- ☑ Clock: 94.5 MHz (achieved 94.9 MHz)
- ☑ LUT utilization: 7%
- ☑ FF utilization: 9%
- ☑ All hardware tests passing (10/10)

Software-Hardware Equivalence

Generated Verilog RTL operations match hardware implementation:

- LOAD → `load_en` signal
- ACCUMULATE → `accumulate_en` signal
- READ → `read_en` signal
- STATUS → `accumulator_zero` output

Performance

Benchmark Results (Phase 2)

- **Memory traffic reduction**: 95-100% vs. traditional state management
- **Write-heavy workloads**: +22% to +55% speedup
- **Parallel efficiency**: 0.85
- **Statistical significance**: 75% of results

Contributing

See Developer Guide for contribution guidelines.

Adding a New Language

1. Create `generator/your_language_generator.py`
2. Implement `CodeEmitter` interface

3. Add test file `tests/test_your_language_generation.py`
4. Update integration tests
5. Document in API reference
6. Submit PR with 100% test coverage

License

Citation

```
@software{atomik_sdk_2026,
  title={ATOMiK SDK: Multi-Language Delta-State Code Generator},
  author={ATOMiK Development Team},
  year={2026},
  version={1.0.0},
  url={https://github.com/your-org/ATOMiK}
}
```

Acknowledgments

Built on:

- Phase 1: Mathematical foundations (92 Lean4 theorems)
- Phase 2: Performance validation (360 measurements)
- Phase 3: Hardware synthesis (Tang Nano 9K FPGA)

Version: 1.0.0 **Last Updated**: January 26, 2026 **Status**: Production Ready