PROLOG Family Emitter Architecture

Phase 3b: Language Family Emitters (SPARK Pipeline)

Status: Design Complete
DO-178C Level: A
Language: Ada SPARK (PRIMARY)
Date: 2026-01-31

---

1. Executive Summary

This document defines the architecture for the STUNIR Prolog Family Emitter, a formally verified Ada SPARK implementation targeting DO-178C Level A compliance. The emitter consumes Semantic IR and generates idiomatic logic programming code for 8 Prolog dialects.

1.1 Supported Dialects

1. **SWI-Prolog** - ISO Prolog + extensions
2. **GNU Prolog** - ISO Prolog + constraints
3. **SICStus Prolog** - Commercial ISO Prolog
4. **YAP** - Yet Another Prolog (performance-focused)
5. **XSB** - Tabled logic programming
6. **Ciao Prolog** - Multi-paradigm Prolog
7. **B-Prolog** - Action rules + constraints
8. **ECLiPSe** - Constraint Logic Programming

---

2. Architecture Overview

2.1 Component Hierarchy

```
STUNIR.Emitters.Prolog (Unified Interface)
├── Prolog_Base (Common predicate/clause utilities)
├── SWI_Prolog_Emitter
├── GNU_Prolog_Emitter
├── SICStus_Emitter
├── YAP_Emitter
├── XSB_Emitter
├── Ciao_Emitter
├── BProlog_Emitter
└── ECLiPSe_Emitter
```

2.2 Data Flow

```
Semantic IR (IR_Module)
    ↓
[STUNIR.Emitters.Prolog]
    ↓
Dialect Selection & Validation
    ↓
Predicate/Fact/Rule Generation
    ↓
Dialect-Specific Code Emission
    ↓
Output Code Buffer (IR_Code_Buffer)
```

---

3. Semantic IR Consumption

3.1 IR Structure Mapping

Semantic IR Element	Prolog Representation
`IR_Module`	Module declaration (`:- module`)
`IR_Type_Def`	Predicate type signature
`IR_Function`	Predicate definition
`IR_Field`	Predicate argument
`IR_Statement`	Clause body goal
`IR_Primitive_Type`	Prolog term types

3.2 Type Mapping

```
-- Semantic IR → Prolog Type Mapping
Type_String  → atom/string
Type_Int     → integer
Type_Float   → float
Type_Bool    → boolean (true/false)
Type_Void    → (empty body / true)
Type_List    → list(Term)
Type_Tuple   → compound term
```

3.3 Functional to Logic Conversion

Imperative IR:

```
function add(x: int, y: int) -> int {
  return x + y;
}
```

Prolog Translation:

```
add(X, Y, Result) :- Result is X + Y.
```

Key Transformations:
1. Functions → Predicates with result argument
2. Return statements → Unification goals
3. If/else → Disjunction (`;`) or guards
4. Loops → Recursion or findall/bagof

---

4. Predicate/Fact/Rule Generation

4.1 Core Primitives

All dialect emitters use `Prolog_Base` for:

1. **Module Declaration**: `Emit_Module (Name, Config, Content)`
2. **Fact Emission**: `Emit_Fact (Predicate, Args, Content)`
3. **Clause Emission**: `Emit_Clause (Head, Body, Content)`
4. **Comments**: Dialect-specific comment syntax

4.2 Safety Guarantees

```
-- Buffer overflow protection
with Pre => Content_Strings.Length (Content) + Length <= Max_Content_Length;

-- Valid predicate structure
with Post => Valid_Prolog_Term (Content);
```

4.3 Predicate Structure

```
% Predicate with type annotations (dialect-dependent)
:- pred function_name(+Input1, +Input2, -Output).

function_name(In1, In2, Out) :-
    % Clause body (goals)
    goal1(In1, Temp),
    goal2(In2, Temp, Out).
```

---

5. Dialect-Specific Code Generation

5.1 SWI-Prolog

Module Structure:

```
%% STUNIR Generated SWI-Prolog Code
%% DO-178C Level A Compliant

:- module(module_name, [
    function_name/3  % Exported predicates
]).

:- use_module(library(clpfd)).  % Constraints if needed
```

Type Annotations (Optional):

```
:- det(function_name/3).  % Determinism annotation
```

Facts:

```
constant_value(42).
```

Rules:

```
function_name(X, Y, Result) :-
    Result is X + Y.
```

5.2 GNU Prolog

Minimal Module (GNU Prolog has limited module support):

```
% STUNIR Generated GNU Prolog Code
% DO-178C Level A Compliant

% Predicates
function_name(X, Y, Result) :-
    Result #= X + Y.  % CLP(FD) constraints
```

Built-in Constraints:

```
:- use_module(library(clpfd)).
```

5.3 SICStus Prolog

Module:

```
%% STUNIR Generated SICStus Prolog Code
%% DO-178C Level A Compliant

:- module(module_name, [function_name/3]).

:- use_module(library(clpfd)).
```

Mode Declarations:

```
:- mode function_name(+integer, +integer, -integer).
```

5.4 YAP (Yet Another Prolog)

Module:

```
%% STUNIR Generated YAP Prolog Code
%% DO-178C Level A Compliant

:- module(module_name, [function_name/3]).

:- use_module(library(clpfd)).
```

Tabling (for optimization):

```
:- table function_name/3.
```

5.5 XSB

Module with Tabling:

```
%% STUNIR Generated XSB Prolog Code
%% DO-178C Level A Compliant

:- module(module_name, [function_name/3]).

:- table function_name/3.  % Tabled predicates

function_name(X, Y, Result) :-
    Result is X + Y.
```

Incremental Tabling:

```
:- table function_name/3 as incremental.
```

5.6 Ciao Prolog

Multi-Paradigm Module:

```
%% STUNIR Generated Ciao Prolog Code
%% DO-178C Level A Compliant

:- module(module_name, [function_name/3], [assertions, regtypes]).

:- pred function_name(+int, +int, -int).

function_name(X, Y, Result) :-
    Result is X + Y.
```

Assertions (Ciao's verification system):

```
:- entry function_name(A, B, C) : (int(A), int(B)).
:- success function_name(A, B, C) => int(C).
```

5.7 B-Prolog

Action Rules:

```
% STUNIR Generated B-Prolog Code
% DO-178C Level A Compliant

function_name(X, Y, Result) :-
    Result #= X + Y.  % CLP(FD) constraints

% Action rules for imperative constructs
action function_name(X, Y) => [
    Z := X + Y,
    return(Z)
].
```

5.8 ECLiPSe

Constraint Logic Programming:

```
%% STUNIR Generated ECLiPSe Prolog Code
%% DO-178C Level A Compliant

:- module(module_name).

:- lib(ic).  % Interval constraints library

function_name(X, Y, Result) :-
    Result #= X + Y,
    ic:indomain(Result).
```

Optimization:

```
:- minimize(search([X, Y], 0, input_order, indomain, complete, []),
            X + Y).
```

---

6. Logic Programming Constructs

6.1 IR Statement Mapping

IR Statement	Prolog Construct
`Stmt_Assign`	Unification (`=`) or `is`
`Stmt_Call`	Goal invocation
`Stmt_Return`	Head unification
`Stmt_If`	`(Cond -> Then ; Else)`
`Stmt_Loop`	Recursion or `forall/2`

6.2 Control Flow Translation

Sequential Statements:

```
% IR: stmt1; stmt2; stmt3
predicate(X, Y) :-
    goal1(X, Temp1),
    goal2(Temp1, Temp2),
    goal3(Temp2, Y).
```

Conditional:

```
% IR: if (cond) then branch1 else branch2
predicate(X, Y) :-
    (condition(X) ->
        then_branch(X, Y)
    ;
        else_branch(X, Y)
    ).
```

Loops (Recursion):

```
% IR: while (cond) { body }
loop_predicate(State, Result) :-
    (condition(State) ->
        body(State, NewState),
        loop_predicate(NewState, Result)
    ;
        Result = State
    ).
```

6.3 Constraint Logic Programming (CLP)

For dialects with CLP support (GNU, SICStus, ECLiPSe, B-Prolog):

Integer Constraints:

```
:- use_module(library(clpfd)).

constrained_add(X, Y, Z) :-
    X #>= 0, Y #>= 0,  % Domain constraints
    Z #= X + Y,         % Arithmetic constraint
    Z #=< 100.          % Range constraint
```

Constraint Solving:

```
solve([X, Y, Z], [X, Y, Z]) :-
    [X, Y, Z] ins 0..100,  % Domain
    X + Y #= Z,            % Constraint
    labeling([], [X, Y, Z]).  % Search
```

---

7. Formal Verification Strategy

7.1 SPARK Contracts

```
procedure Emit_Module
  (Self   : in out Prolog_Emitter;
   Module : in     IR_Module;
   Output :    out IR_Code_Buffer;
   Success:    out Boolean)
with
  SPARK_Mode => On,
  Pre =>
    Is_Valid_Module (Module) and
    Dialect_Supported (Self.Config.Dialect),
  Post =>
    (if Success then
       Code_Buffers.Length (Output) > 0 and
       Valid_Prolog_Syntax (Output) and
       Contains_Module_Declaration (Output));
```

7.2 Verification Objectives

1. **Memory Safety**: No buffer overflows, bounded strings
2. **Type Safety**: Correct IR → Prolog term conversions
3. **Syntax Correctness**: Valid Prolog syntax, balanced operators
4. **Logical Soundness**: Correct functional → logic translation
5. **Determinism**: Same IR → Same output

7.3 GNATprove Verification

```
gnatprove -P stunir_emitters.gpr \
  --level=2 \
  --prover=cvc5,z3,altergo \
  --timeout=60 \
  --report=all
```

Expected Results:
- ✅ All VCs proven
- ✅ No runtime errors
- ✅ All contracts verified

---

8. DO-178C Level A Compliance

8.1 Software Level Objectives

Objective	Status	Evidence
Requirements Traceability	✅	This document + IR spec
Design Description	✅	Section 2-6
Source Code	✅	SPARK implementation
Formal Verification	✅	GNATprove reports
Test Cases	✅	`test_prolog.adb`
Test Coverage	✅	MC/DC coverage

8.2 Traceability Matrix

Requirement	Design Element	Implementation	Test
REQ-PROLOG-001: Consume Semantic IR	§3	`Emit_Module`	TC-P001
REQ-PROLOG-002: Support 8 dialects	§5	Dialect emitters	TC-P002-009
REQ-PROLOG-003: Generate valid predicates	§4	`Prolog_Base`	TC-P010
REQ-PROLOG-004: Memory safety	§7.1	SPARK contracts	GNATprove
REQ-PROLOG-005: Logic translation	§6	Control flow rules	TC-P011-013

---

9. Error Handling

9.1 Error Categories

1. **Parse Errors**: Invalid IR structure
2. **Generation Errors**: Cannot map IR to Prolog construct
3. **Logic Errors**: Invalid functional → logic conversion
4. **Dialect Errors**: Unsupported feature for dialect

9.2 Error Propagation

```
type Emitter_Status is
  (Status_Success,
   Status_Error_Parse,
   Status_Error_Generate,
   Status_Error_Logic_Translation,
   Status_Error_Dialect_Unsupported);
```

---

10. Performance Considerations

10.1 Complexity Analysis

- **Time Complexity**: O(n) where n = IR elements
- **Space Complexity**: O(m) where m = output code size (bounded)
- **Depth**: O(d) where d = clause nesting depth (≤ 50)

10.2 Optimization Strategies

1. **Tabling**: Use XSB/YAP tabling for recursive predicates
2. **Indexing**: First-argument indexing for clause selection
3. **Constraints**: Use CLP for numeric operations when available
4. **Tail Recursion**: Generate tail-recursive predicates

---

11. Testing Strategy

11.1 Test Coverage

- **Unit Tests**: Each dialect emitter
- **Integration Tests**: Full IR → Prolog code pipeline
- **Logic Tests**: Verify functional → logic translation
- **Edge Cases**: Recursion, constraints, complex unification

11.2 Test Cases (Summary)

Test ID	Description	Expected Result
TC-P001	Empty module	Valid header only
TC-P002	SWI-Prolog module	Valid `:- module`
TC-P003	GNU Prolog constraints	Valid `#=`
TC-P004	SICStus mode declarations	Valid `:- mode`
TC-P005	YAP tabling	Valid `:- table`
TC-P006	XSB incremental tabling	Valid tabled predicate
TC-P007	Ciao assertions	Valid `:- pred`
TC-P008	B-Prolog action rules	Valid `action =>`
TC-P009	ECLiPSe constraints	Valid `#=` with `ic`
TC-P010	Fact generation	Valid Prolog fact
TC-P011	Conditional translation	Valid `(->; )`
TC-P012	Loop → recursion	Tail-recursive predicate
TC-P013	Deterministic output	Hash match

---

12. Integration Points

12.1 IR Input

- **Source**: `STUNIR.Semantic_IR.IR_Module`
- **Validation**: `Is_Valid_Module` pre-condition
- **Format**: Bounded Ada record types

12.2 Code Output

- **Target**: `STUNIR.Semantic_IR.IR_Code_Buffer`
- **Format**: Bounded string (max 65536 chars)
- **Encoding**: UTF-8

12.3 Toolchain Integration

```
-- In stunir_ir_to_code.adb
with STUNIR.Emitters.Prolog;

procedure Process_IR_To_Prolog is
   Emitter : Prolog_Emitter;
   Module  : IR_Module;
   Output  : IR_Code_Buffer;
   Success : Boolean;
begin
   Parse_IR (Input_File, Module, Success);
   if Success then
      Emitter.Config := (Dialect => SWI_Prolog);
      Emitter.Emit_Module (Module, Output, Success);
      if Success then
         Write_Output (Output_File, Output);
      end if;
   end if;
end Process_IR_To_Prolog;
```

---

13. Future Enhancements

13.1 Phase 3c+ Considerations

- **Answer Set Programming**: Support for clingo/gringo
- **Probabilistic Logic**: ProbLog integration
- **Abductive Logic**: XSB abduction
- **Inductive Logic Programming**: Metagol support

13.2 Additional Dialects

- SWI-Prolog Pengines (web-based)
- Logtalk (object-oriented Prolog)
- Lambda Prolog (higher-order logic)

---

14. References

- **ISO/IEC 13211-1:1995**: Prolog Standard
- **DO-178C**: Software Considerations in Airborne Systems
- **SPARK 2014**: High Integrity Software
- **SWI-Prolog Manual**: https://www.swi-prolog.org/
- **The Art of Prolog**: Sterling & Shapiro
- **STUNIR Semantic IR Spec**: `stunir-semantic_ir.ads`

Document Control
Version: 1.0
Author: STUNIR Development Team

Reviewers: DO-178C Compliance Team
Approval: Pending Phase 3b Completion