VSAR-DSL: A Specification Language for Multi-Mode VSA Reasoning Programs

This document defines **VSAR-DSL**, a small, compositional specification language for writing VSAR reasoning programs that leverage:

- the **FHRR encoding substrate** (roles, tags, typed symbols)
- the **unification kernel** (unbind → typed cleanup)
- the **multi-mode inference engine** (deduction, DL, defaults, paraconsistency, probability, argumentation, epistemic, abduction, induction, analogy, CBR)

VSAR-DSL is designed to be:

- **declarative**: state knowledge and policies; let the engine run
- **multi-semantics**: select which controllers apply and how they interact
- **explainable**: every conclusion carries a proof/argument trace
- **VSA-native**: explicit about types, cleanup domains, and similarity thresholds

---

1) Core ideas

1.1 A VSAR program is four blocks

1) **SIGNATURE**: symbols and types (typed codebooks) 2) **KB**: facts, rules, DL axioms, cases, arguments 3) **SEMANTICS**: which reasoning modes are active and how conflicts are handled 4) **QUERIES**: what to ask (deductive entailment, DL entailment, abduction, analogy, CBR, epistemic, etc.)

1.2 Everything is an item with metadata

Facts/rules/axioms/edges/cases are inserted as items with:

- weight (probability / confidence)
- priority (for defaults/defeasible)
- agent (for epistemic)
- provenance (source/time)

VSAR-DSL makes this explicit.

---

2) Syntax overview (EBNF-ish)

```
program      := signature semantics? kb queries?
```

```
signature    := "SIGNATURE" "{" decl* "}"

decl         := type_decl | pred_decl | func_decl | role_decl | agent_decl

type_decl    := "type" IDENT ("<:" IDENT)? ";"

pred_decl    := "pred" IDENT ":" "(" type_list ")" pred_attrs? ";"
func_decl    := "func" IDENT ":" "(" type_list ")" "->" IDENT ";"
role_decl    := "role" IDENT ";"          // argument/structural roles if user
wants custom
agent_decl   := "agent" IDENT ";"

type_list    := IDENT ("," IDENT)*

pred_attrs   := "[" attr ("," attr)* "]"
attr         := "dl" | "closed" | "open" | "symmetric" | "transitive" |
"functional"

semantics    := "SEMANTICS" "{" setting* "}"
setting      := IDENT ":" value ";"
value        := NUMBER | STRING | IDENT | list
list         := "[" value ("," value)* "]"

kb           := "KB" "{" stmt* "}"

stmt         := fact | rule | default_rule | dl_axiom | arg_edge | case_stmt |
map_stmt

fact         := lit meta? "."

lit          := atom | "~" atom | "not" atom
atom         := IDENT "(" term_list? ")"
term_list    := term ("," term)*
term         := IDENT | STRING | NUMBER | IDENT "(" term_list? ")"   // constant
or function term

rule         := "rule" IDENT? ":" atom "<-" body meta? "."
body         := lit ("," lit)*

default_rule := "default" IDENT? ":" atom "<-" body ("unless" body)? meta? "."

dl_axiom     := "tbox" concept "<:" concept meta? "." |
                "tbox" concept "==" concept meta? "." |
                "abox" concept "(" IDENT ")" meta? "." |
                "abox" role_atom meta? "."

concept      := IDENT |
                "(" concept "and" concept ")" |
                "(" concept "or" concept ")" |
                "(" "not" concept ")" |
                "(" "exists" IDENT "." concept ")" |
                "(" "forall" IDENT "." concept ")"

role_atom    := IDENT "(" IDENT "," IDENT ")"

arg_edge     := "support" "(" lit "," lit ")" meta? "." |
                "attack"  "(" lit "," lit ")" meta? "."

case_stmt    := "case" IDENT ":" "{" case_field+ "}" meta? "."
case_field   := ("problem" ":" term) | ("solution" ":" term) | ("context" ":"
term) | ("outcome" ":" term)

map_stmt     := "map" IDENT? ":" term "->" term meta? "."

meta         := "{" meta_kv ("," meta_kv)* "}"
meta_kv      := IDENT ":" value

queries      := "QUERIES" "{" query* "}"
query        := "ask" ask_expr meta? "."
ask_expr     := lit |
                "entails" "(" lit ")" |
                "dl_entails" "(" concept "(" IDENT ")" ")" |
                "explain" "(" lit ")" |
                "analogize" "(" term "," term ")" |
                "retrieve_case" "(" term ")" |
                "epistemic" "(" term ")"
```

---

3) Semantics block (selecting reasoning modes)

The **SEMANTICS** block chooses controllers and conflict policies.

Example:

```
SEMANTICS {
  logic: horn;                  // horn | datalog | backward | mixed
  dl: alc;                      // off | alc
  paraconsistent: belnap;       // off | belnap
  defaults: enabled;            // enabled | off
  defeat: [priority, weight, specificity];
  uncertainty: probabilistic;   // off | probabilistic | intervals
  t_norm: product;              // product | min | lukasiewicz
  t_conorm: noisy_or;           // noisy_or | max | logsumexp
  argumentation: gradual;       // off | dung | gradual
  epistemic: enabled;           // enabled | off
  cwa: [closed(pred1), open(pred2)];
  cleanup_threshold: 0.25;      // abstain if below
  retrieval_k: 25;              // for exploratory similarity
  proof_trace: full;            // none | brief | full
}
```

Notes:

- `cleanup_threshold` controls symbol commitment; below threshold the engine returns `UNKNOWN`.
- `retrieval_k` controls exploratory similarity search for analogy/CBR.

---

4) Knowledge block: facts, rules, defaults, DL axioms, arguments

4.1 Facts and literals

```
KB {
  parent(alice, bob) {w: 0.9}.
  ~parent(alice, bob) {w: 0.4}.       // classical negation
  not sick(bob) {w: 0.6}.             // default/NAF literal
}
```

Interpretation:

- `~A` encodes **classical negation** using the `NEG ⊗ TAG_LIT` wrapper.
- `not A` is a **meta-literal**; its meaning is controlled by the defaults/CWA semantics.

4.2 Horn rules

```
KB {
  rule r1: grandparent(x,z) <- parent(x,y), parent(y,z) {w: 0.95}.
}
```

4.3 Default (defeasible) rules

```
KB {
  default d1: flies(x) <- bird(x) unless penguin(x) {prio: 10, w: 0.8}.
  rule r2: ~flies(x) <- penguin(x) {w: 0.9}.
}
```

This exercises:

- defaults
- exceptions
- paraconsistency (if both `flies` and `~flies` derived)
- argumentation/defeat resolution

4.4 Description Logic axioms

```
SIGNATURE {
  type Individual;
  type Concept;
  pred hasChild: (Individual, Individual) [dl];
  pred Person: (Individual) [dl];
  pred Doctor: (Individual) [dl];
}

KB {
  tbox Doctor <: Person.
  abox Doctor(alice).
  abox hasChild(alice, bob).
  abox Person(bob).
}
```

DL concept expressions:

```
KB {
  tbox (exists hasChild . Doctor) <: Person.
}
```

4.5 Argumentation edges

```
KB {
  support( bird(tweety), flies(tweety) ) {w: 0.7}.
  attack(  penguin(tweety), flies(tweety) ) {w: 0.9}.
}
```

The argumentation controller will compute acceptability and warranted conclusions.

4.6 Epistemic statements

Epistemic operators are written with `K(agent, φ)` and `B(agent, φ)`.

```
SIGNATURE { agent alice; agent bob; }
KB {
  B(alice, parent(alice,bob)) {w: 0.9}.
  K(bob, ~parent(alice,bob)) {w: 0.8}.
}
```

Agents have separate KB partitions; trust/communication are set via SEMANTICS.

4.7 Cases and mappings (CBR + analogy)

```
KB {
  case c1: {
    problem: diagnose(symptoms(fever,cough));
    context: patient(age(8));
    solution: treat(viral_support);
    outcome: success;
  } {w: 0.8}.

  map m1: doctor -> mechanic {w: 0.6}.
}
```

---

5) Query block

5.1 Deductive queries

```
QUERIES {
  ask entails(grandparent(alice, z)).
}
```

5.2 Paraconsistent query

```
QUERIES {
  ask flies(tweety) {return: [support_pos, support_neg]}.
}
```

5.3 Abductive query

```
QUERIES {
  ask explain(sick(bob)) {k: 5}.
}
```

5.4 DL entailment query

```
QUERIES {
  ask dl_entails(Person(alice)).
}
```

5.5 Analogy and cases

```
QUERIES {
  ask analogize(domainA, domainB) {k: 10}.
  ask retrieve_case(diagnose(symptoms(fever,cough))) {k: 3}.
}
```

5.6 Epistemic query

```
QUERIES {
  ask entails(B(alice, parent(alice,bob))).
  ask entails(K(bob, ~parent(alice,bob))).
}
```

---

6) Compilation to VSAR operations (operational semantics)

VSAR-DSL compiles each construct into:

1) **Encodings** (FHRR role–filler binding + tags) 2) **Indexes** (predicate/concept/role indexes for crisp narrowing) 3) **Controller selection** (semantics portfolio)

6.1 How an atom compiles

`p(t1,t2)` compiles to the FHRR encoding:

$$enc(p(t_1,t_2)) = (P_p \otimes TAG_{ATOM}) \otimes ((ARG_1 \otimes enc(t_1)) \oplus (ARG_2 \otimes enc(t_2)))$$

6.2 How crisp matching works

To answer `p(a, ?)`:

- use predicate index for `p`
- decode `ARG1` and verify equals `a` via typed cleanup
- decode `ARG2` and return cleanup result(s)

This avoids whole-vector fuzziness.

6.3 Cleanup and similarity roles

- **cleanup**: typed nearest-neighbor symbol commitment
- **retrieval**: exploratory similarity over stored items (facts/rules/cases)

The DSL exposes both via `cleanup_threshold`, `retrieval_k`, and query-level overrides.

---

7) Safety and clarity features

7.1 Explicit abstention

If cleanup confidence is below threshold, the engine returns `UNKNOWN` and can optionally trigger:

- abductive expansion
- argumentation resolution
- case retrieval

7.2 Proof and argument traces

Every derived conclusion can return:

- proof tree (Horn/DL)
- argument graph slice (argumentation)
- explanation set (abduction)
- mapping witness (analogy)

---

8) Minimal viable DSL subset (MVP)

Start with:

- SIGNATURE (types, predicates)
- KB facts
- Horn rules
- QUERIES entails()
- SEMANTICS cleanup_threshold + retrieval_k

Then add modules:

- defaults + defeat
- paraconsistent belief state
- argumentation
- DL axioms
- epistemic operators
- abduction
- analogy + cases

9) Example: a single program exercising many modes

```
SIGNATURE {
  type Individual;
  pred parent: (Individual, Individual);
  pred bird: (Individual);
  pred penguin: (Individual);
  pred flies: (Individual);
  agent alice;
}

SEMANTICS {
  logic: horn;
  paraconsistent: belnap;
  defaults: enabled;
  argumentation: gradual;
  uncertainty: probabilistic;
  t_norm: product;
  t_conorm: noisy_or;
  cleanup_threshold: 0.25;
  retrieval_k: 25;
  proof_trace: full;
}

KB {
  parent(alice, bob) {w: 0.9}.

  rule r1: grandparent(x,z) <- parent(x,y), parent(y,z) {w: 0.95}.

  default d1: flies(x) <- bird(x) unless penguin(x) {prio: 10, w: 0.8}.
  rule r2: ~flies(x) <- penguin(x) {w: 0.9}.

  bird(tweety) {w: 0.7}.
  penguin(tweety) {w: 0.9}.

  B(alice, parent(alice,bob)) {w: 0.9}.
}

QUERIES {
  ask entails(grandparent(alice, z)).
  ask flies(tweety) {return: [support_pos, support_neg]}.
  ask explain(flies(tweety)) {k: 3}.
}
```

10) Implementation notes

- Prefer a **parser + AST** that compiles into engine API calls.
- Keep DSL surface syntax stable while internal engine evolves.
- Provide an interactive REPL where users can:
- add facts incrementally
- run queries
- inspect traces and decoded slots

---

11) Final view

VSAR-DSL makes VSAR usable as a **programmable reasoning system**:

- users write structured knowledge
- users select semantics portfolios
- VSAR compiles into FHRR encodings and invokes the inference engine

This gives a practical path from the math of VSAX/FHRR to usable reasoning programs.