VSAR User Manual

Writing VSARL Programs and Using the VSAR IDE

Version 0.3.3

January 2026

Vector Symbolic Architecture Reasoner

A Declarative Language for Approximate Logical Reasoning

Contents

1 Introduction

1.1 What is VSAR?

VSAR (Vector Symbolic Architecture Reasoner) is a reasoning system that uses high-dimensional vectors to perform logical inference. Unlike traditional logic programming systems (like Prolog), VSAR provides:

- **Approximate reasoning** with similarity scores
- **Graceful degradation** under uncertainty
- **Fast vectorized operations** (GPU-friendly)
- **Scalable inference** over large knowledge bases

1.2 What is VSARL?

VSARL (VSA Reasoning Language) is a declarative language for writing reasoning programs. It looks similar to Prolog/Datalog but uses vector symbolic architectures under the hood.

1.3 When to Use VSAR

Use VSAR when you need:

- Logical reasoning with approximate matching
- Tolerance to noisy or incomplete data
- Explainable results with confidence scores
- Fast inference over large fact sets
- Integration with neural/embedding-based systems

2 Getting Started

2.1 Installation

Install VSAR via pip:

```
# Install VSAR
pip install vsar

# Verify installation
vsar --version
```

2.2 Launching the IDE

Start the VSAR IDE from the command line:

```
# Start the VSAR IDE
vsar-ide

# Or specify a file to open
vsar-ide examples/02_family_tree.vsar
```

The IDE window will open with three main areas:

- **Editor** (center): Write your VSARL code
- **Console** (bottom): View program output and results
- **Menu bar** (top): File operations and controls

3 VSARL Language Guide

3.1 Program Structure

Every VSARL program has three main parts:

```
// 1. DIRECTIVES - Configure the reasoning engine
@model FHRR(dim=8192, seed=42);
@beam(width=50);
@novelty(threshold=0.95);

// 2. FACTS - Ground truths about the world
fact parent(alice, bob).
fact parent(bob, charlie).

// 3. RULES - Derive new facts from existing ones
rule grandparent(X, Z) :- parent(X, Y), parent(Y, Z).

// 4. QUERIES - Ask questions
query grandparent(alice, X)?
```

3.2 Directives

Directives configure how VSAR encodes and retrieves information.

3.2.1 Model Configuration

```
@model FHRR(dim=8192, seed=42);
```

- **FHRR**: The VSA backend (Fourier Holographic Reduced Representations)
- **dim**: Vector dimension (higher = more accurate but slower)
  - Recommended: 8192 for production, 512-2048 for testing
- **seed**: Random seed for reproducibility

3.2.2 Beam Search

```
@beam(width=50);
```

Controls how many candidates are explored during search.

- **Higher values** (50-100): Better coverage, slower
- **Lower values** (10-20): Faster, may miss results

3.2.3 Novelty Detection

```
@novelty(threshold=0.95);
```

Prevents inserting near-duplicate facts.

- **Higher values** (0.95-0.99): Stricter duplicate detection
- **Lower values** (0.7-0.9): More lenient

3.3 Facts

Facts are ground truths with no variables.

3.3.1 Syntax

```
fact predicate(arg1, arg2, ..., argN).
```

3.3.2 Valid Examples

```
fact parent(alice, bob).
fact likes(alice, pizza).
fact employee(bob, engineering, manager).
fact popular(inception).
```

3.3.3 Invalid Examples

```
fact parent(alice, bob);     // Wrong: semicolon instead of period
fact age(alice, 25).          // Wrong: numeric literals not supported
fact Parent(alice, bob).      // Wrong: predicate must be lowercase
```

3.3.4 Important Constraints

- **Predicates**: Must be lowercase (e.g., `parent`, `likes`, `works_in`)
- **Constants**: Must be lowercase (e.g., `alice`, `bob`, `engineering`)
- **No numbers**: Use symbolic constants instead (e.g., `twenty_five` not `25`)
- **End with period**: Facts always end with `.` never `;`

3.3.5 Classical Negation

You can assert negative facts using `~`:

```
fact ~enemy(alice, bob).      // Alice is NOT an enemy of Bob
fact ~likes(charlie, burgers). // Charlie does NOT like burgers
```

3.4 Rules

Rules derive new facts from existing ones.

3.4.1 Syntax

```
rule head(X, Y) :- body1(X, Z), body2(Z, Y).
```

- **Head**: What gets derived (single atom)
- **Body**: Conditions that must hold (one or more atoms, comma-separated)
- **Variables**: Uppercase (e.g., `X`, `Y`, `Person`, `Item`)

3.4.2 Examples

Simple derivation:

```
rule grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

Multiple rules for same predicate:

```
rule ancestor(X, Y) :- parent(X, Y).
rule ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).
```

Multi-body rule:

```
rule recommend(User, Item) :-
    likes(User, Item1),
    similar(Item1, Item2),
    genre(Item2, Genre),
    genre(Item, Genre).
```

3.4.3 Negation-as-Failure

Use `not` to test for absence:

```
rule safe(Person) :-
    employee(Person, Dept),
    not incident(Person, Location).
```

Important: Variables in `not` atoms should appear elsewhere in the rule body.

3.4.4 Valid Rules

```
rule ancestor(X, Y) :- parent(X, Y).
rule can_access(X, Resource) :-
    manages(X, Y), has_access(Y, Resource).
```

3.4.5 Invalid Rules

```
rule ancestor(X, Y) :- parent(X, Y);  // Wrong: semicolon
rule ancestor(X, Y) :- parent(X, _).  // Wrong: underscore wildcard
rule ancestor(x, y) :- parent(x, y).  // Wrong: lowercase variables
```

3.5 Queries

Queries ask questions about the knowledge base.

3.5.1 Syntax

```
query predicate(constant, X)?
```

- **Constants**: Lowercase values you know
- **Variables**: Uppercase unknowns to find
- **End with ?**: Queries always end with question mark

3.5.2 Examples

Single-variable query:

```
query parent(alice, X)?           // Who are Alice's children?
query likes(X, pizza)?             // Who likes pizza?
query employee(X, engineering)?    // Who works in engineering?
```

Ground query (yes/no):

```
query parent(alice, bob)?          // Is Alice Bob's parent?
```

Multiple queries:

```
query grandparent(alice, X)?
query grandparent(X, charlie)?
query ancestor(alice, X)?
```

3.5.3 Multi-Variable Queries

Multi-variable queries are not supported in the IDE. Use the Python API instead:

```
from vsar.language.ast import Query
result = engine.query(
    Query(predicate="parent", args=[None, None]),
    k=10
)
```

4 Using the VSAR IDE

4.1 IDE Layout

The IDE consists of three main areas:

- **Menu bar** (top): File operations and run controls
- **Editor** (center): Code editing area with syntax highlighting
- **Console** (bottom): Program output and query results

4.2 Keyboard Shortcuts

Shortcut	Action
F5	Run program
Ctrl+N	New file
Ctrl+O	Open file
Ctrl+S	Save file
Ctrl+Q	Interactive query dialog
Ctrl+/	Toggle comment

4.3 Running a Program

To run a VSARL program:

1. Load or create a program in the editor
2. Press **F5** or click **Run → Execute Program**
3. View results in the console below

4.3.1 Example Output

```
========================================
Parsing program...
Parsed successfully: 4 facts, 2 queries, 3 rules

Creating engine with directives: @model FHRR(dim=512, seed=42);
Inserting 4 facts into knowledge base...

Applying 3 rules (forward chaining)...
Iteration 1: derived 2 new facts
Iteration 2: derived 1 new facts
Iteration 3: no new facts (fixpoint reached)

Forward chaining complete: 3 iterations, 3 new facts

Executing 2 queries...
```

```
Query 1: grandparent(alice, X)?
  david (0.91)
  eve (0.88)

Query 2: ancestor(alice, X)?
  bob (0.95)
  charlie (0.94)
  david (0.87)
  eve (0.85)
========================================
```

4.4 Interactive Query Mode

To execute queries interactively:

1. Press **Ctrl+Q** or click **Run → Interactive Query**
2. Enter your query in the dialog (e.g., `parent(alice, X)?`)
3. View results in the console

4.5 File Operations

4.5.1 New File

- **Ctrl+N** or **File → New**
- Creates a blank program with default directives

4.5.2 Open File

- **Ctrl+O** or **File → Open**
- Browse to `.vsar` file
- Try the examples in `examples/` directory

4.5.3 Save File

- **Ctrl+S** or **File → Save**
- Saves current program
- Auto-adds `.vsar` extension if missing

4.6 Syntax Highlighting

The IDE provides syntax highlighting for:

- **Keywords**: `fact`, `rule`, `query`, `not`
- **Directives**: `@model`, `@beam`, `@novelty`

- **Comments**: Lines starting with `//`
- **Variables**: Uppercase identifiers
- **Constants**: Lowercase identifiers

5 Writing Your First Program

Let's build a simple family tree reasoning system step by step.

5.1 Step 1: Create a New File

1. Launch VSAR IDE: `vsar-ide`
2. Press **Ctrl+N** for new file

5.2 Step 2: Add Directives

```
@model FHRR(dim=512, seed=42);
@beam(width=50);
@novelty(threshold=0.95);
```

5.3 Step 3: Add Facts

```
// Parent relationships
fact parent(alice, bob).
fact parent(alice, charlie).
fact parent(bob, david).
fact parent(charlie, eve).

// Gender facts
fact male(bob).
fact male(david).
fact female(alice).
fact female(charlie).
fact female(eve).
```

5.4 Step 4: Add Rules

```
// Grandparent: X is grandparent of Z if X is parent of Y
// and Y is parent of Z
rule grandparent(X, Z) :- parent(X, Y), parent(Y, Z).

// Ancestor: Base case - direct parent
rule ancestor(X, Y) :- parent(X, Y).

// Ancestor: Recursive case - parent of ancestor
rule ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).

// Grandmother: Female grandparent
rule grandmother(X, Z) :- grandparent(X, Z), female(X).
```

5.5 Step 5: Add Queries

```
query grandparent(alice, X)?      // Who are Alice's grandchildren?
query ancestor(alice, X)?          // Who are Alice's descendants?
query grandmother(X, eve)?         // Who is Eve's grandmother?
```

5.6 Step 6: Run the Program

Press **F5** to execute.

5.6.1 Expected Output

```
Query 1: grandparent(alice, X)?
  david (0.92)
  eve (0.91)

Query 2: ancestor(alice, X)?
  bob (0.95)
  charlie (0.94)
  david (0.88)
  eve (0.87)

Query 3: grandmother(X, eve)?
  alice (0.89)
```

5.7 Step 7: Save Your Program

1. Press **Ctrl+S**
2. Name it `my_family_tree.vsar`
3. Save in your working directory

6 Common Patterns

6.1 Pattern 1: Transitive Closure

Computing transitive relationships (e.g., "can reach from X to Y"):

```
// Direct connection
fact connected(a, b).
fact connected(b, c).
fact connected(c, d).

// Reachability: base case
rule reachable(X, Y) :- connected(X, Y).

// Reachability: transitive case
rule reachable(X, Z) :- connected(X, Y), reachable(Y, Z).

query reachable(a, X)?  // Everything reachable from 'a'
```

6.2 Pattern 2: Hierarchies

Modeling organizational or taxonomic hierarchies:

```
// Organizational structure
fact reports_to(alice, bob).
fact reports_to(bob, charlie).
fact reports_to(david, charlie).

// Direct manager
rule manager(Manager, Employee) :- reports_to(Employee, Manager).

// Transitive supervision
rule supervises(X, Y) :- manager(X, Y).
rule supervises(X, Z) :- manager(X, Y), supervises(Y, Z).

query supervises(charlie, X)?  // Everyone Charlie supervises
```

6.3 Pattern 3: Classification

Deriving categories from properties:

```
// Movie facts
fact genre(inception, scifi).
fact genre(matrix, scifi).
fact genre(avatar, scifi).
fact rating(inception, high).
fact rating(matrix, high).

// Classify as recommended
rule recommended(Movie) :-
    genre(Movie, scifi),
```

```
    rating(Movie, high).

query recommended(X)?  // All recommended sci-fi movies
```

6.4 Pattern 4: Collaborative Filtering

Finding similar entities based on shared properties:

```
// User preferences
fact likes(alice, inception).
fact likes(alice, matrix).
fact likes(bob, inception).
fact likes(bob, interstellar).

// Similar taste
rule similar_taste(X, Y) :-
    likes(X, Item),
    likes(Y, Item).

// Recommendations
rule might_like(User, Item) :-
    similar_taste(User, OtherUser),
    likes(OtherUser, Item),
    not likes(User, Item).

query similar_taste(alice, X)?  // Similar taste to Alice?
query might_like(alice, X)?      // What might Alice like?
```

6.5 Pattern 5: Access Control

Security policies with negation:

```
// Access permissions
fact has_access(alice, server_room).
fact has_access(bob, lab).

// Incidents
fact incident(server_room, thursday).

// Trusted relationships
fact trusted(alice, bob).

// Suspect: has access but not trusted
rule suspect(Person, Location) :-
    has_access(Person, Location),
    incident(Location, Day),
    not trusted(Person, Anyone).

query suspect(X, server_room)?
```

7 Advanced Features

7.1 Multi-Variable Queries

While the IDE supports single-variable queries, you can use the Python API for multi-variable retrieval:

```
from vsar.language.parser import parse
from vsar.semantics.engine import VSAREngine
from vsar.language.ast import Query

# Load program
with open("my_program.vsar") as f:
    program = parse(f.read())

# Create engine
engine = VSAREngine(program.directives)

# Insert facts
for fact in program.facts:
    engine.insert_fact(fact)

# Multi-variable query: find ALL parent-child pairs
result = engine.query(
    Query(predicate="parent", args=[None, None]),
    k=10
)

for (parent, child), score in result.results:
    print(f"{parent} -> {child} (score: {score:.2f})")
```

7.2 Backward Chaining

Goal-directed proof search (alternative to forward chaining):

```
from vsar.reasoning.backward_chaining import BackwardChainer
from vsar.language.ast import Atom

# Create backward chainer
chainer = BackwardChainer(
    engine,
    rules=program.rules,
    max_depth=5,
    threshold=0.5
)

# Prove a specific goal
goal = Atom(predicate="ancestor", args=["alice", "eve"])
proofs = chainer.prove_goal(goal)

for proof in proofs:
    print(f"Proof: {proof.substitution}")
    print(f"Similarity: {proof.similarity:.2f}")
```

7.3 Understanding Similarity Scores

Every result comes with a similarity score (0-1):

- **1.0**: Perfect match (exact fact in KB)
- **0.9-0.99**: Very high confidence
- **0.7-0.89**: Good confidence
- **0.5-0.69**: Moderate confidence
- **¡ 0.5**: Low confidence (usually filtered out)

What affects scores:

- Direct facts: ∼0.95-1.0
- One-hop derivations: ∼0.85-0.95
- Multi-hop derivations: ∼0.7-0.9 (degrades with depth)
- High beam width improves accuracy

8 Troubleshooting

8.1 Common Errors

8.1.1 Error: "No terminal matches '0'"

Problem: Used numeric literals

```
fact age(alice, 25).  // WRONG
```

Solution: Use symbolic constants

```
fact age(alice, twenty_five).  // CORRECT
```

8.1.2 Error: "Expected: DOT"

Problem: Used semicolon instead of period

```
fact parent(alice, bob);  // WRONG
rule ancestor(X, Y) :- parent(X, Y);  // WRONG
```

Solution: Use period

```
fact parent(alice, bob).  // CORRECT
rule ancestor(X, Y) :- parent(X, Y).  // CORRECT
```

8.1.3 Error: "No terminal matches '?'"

Problem: Used `?` wildcard in query

```
query parent(?, ?)?  // WRONG
```

Solution: Use variables for single-variable queries

```
query parent(alice, X)?  // CORRECT (single variable)
```

For multi-variable queries, use the Python API.

8.1.4 Error: "Expected: UPPER_NAME"

Problem: Used lowercase for variable or underscore wildcard

```
rule ancestor(x, y) :- parent(x, y).  // WRONG
rule safe(X) :- person(X), not enemy(X, _).  // WRONG
```

Solution: Use uppercase variables

```
rule ancestor(X, Y) :- parent(X, Y).  // CORRECT
rule safe(X) :- person(X), not enemy(X, Person).  // CORRECT
```

8.2 IDE Not Starting

Problem: `vsar-ide` command not found

Solution:

```
# Reinstall VSAR
pip install --upgrade vsar

# Or install in development mode
pip install -e .
```

8.3 No Results for Query

Possible causes:

1. **Typo in predicate name**

```
fact Parent(alice, bob).  // Wrong: uppercase predicate
query parent(alice, X)?    // Won't match
```

2. **Facts not matching query**

```
fact parent(alice, bob).
query parent(bob, alice)?  // Returns nothing (wrong direction)
```

3. **Rules not firing (missing facts)**

```
rule grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
// If no parent facts exist, rule never fires
```

4. **Beam width too low**

```
@beam(width=5);  // Try increasing to 50
```

8.4 Low Similarity Scores

If you're getting unexpectedly low scores (¡ 0.7):

1. **Increase vector dimension**

```
@model FHRR(dim=8192, seed=42);  // Higher = more accurate
```

2. **Increase beam width**

```
@beam(width=100);  // Explore more candidates
```

3. **Check for deep derivations**

- Multi-hop rules naturally have lower scores
- This is expected behavior (approximate reasoning)

9 Quick Reference

9.1 Valid Identifiers

Type	Case	Examples
Predicate	lowercase	`parent`, `likes`, `works_in`
Constant	lowercase	`alice`, `bob`, `engineering`
Variable	uppercase	`X`, `Y`, `Person`, `Item`

9.2 Syntax Summary

```
// Directives (at top of file)
@model FHRR(dim=8192, seed=42);
@beam(width=50);
@novelty(threshold=0.95);

// Facts (end with period)
fact predicate(const1, const2).
fact ~negative(const1, const2).

// Rules (end with period)
rule head(X, Y) :- body1(X, Z), body2(Z, Y).
rule head(X, Y) :- body1(X, Z), not body2(Z, Y).

// Queries (end with question mark)
query predicate(const, X)?
query predicate(X, const)?
```

9.3 Common Directives

```
// Testing (small/fast)
@model FHRR(dim=512, seed=42);

// Development (medium)
@model FHRR(dim=2048, seed=42);

// Production (large/accurate)
@model FHRR(dim=8192, seed=42);

// Beam width
@beam(width=10);   // Fast, may miss results
@beam(width=50);   // Balanced (recommended)
@beam(width=100);  // Thorough, slower

// Novelty threshold
@novelty(threshold=0.7);   // Lenient duplicate detection
@novelty(threshold=0.95);  // Strict (recommended)
```

```
@novelty(threshold=0.99);  // Very strict
```

9.4 IDE Shortcuts

Shortcut	Action
F5	Run program
Ctrl+N	New file
Ctrl+O	Open file
Ctrl+S	Save file
Ctrl+Q	Interactive query
Ctrl+/	Toggle comment

9.5 Example Programs

VSAR includes 12 example programs in the `examples/` directory:

File	Description
`00_getting_started.vsar`	Beginner introduction
`01_basic_rules.vsar`	Simple derivation
`02_family_tree.vsar`	Classic Prolog example
`03_transitive_closure.vsar`	Recursive rules
`04_organizational_hierarchy.vsar`	Hierarchies
`05_knowledge_graph.vsar`	Multiple relations
`06_academic_network.vsar`	Complex interactions
`07_negation.vsar`	Negation-as-failure
`08_multi_variable_queries.vsar`	Multi-variable concepts
`09_backward_chaining.vsar`	Goal-directed search
`10_advanced_reasoning.vsar`	Enterprise security
`11_recommendation_system.vsar`	Collaborative filtering

10 Tips and Best Practices

10.1 Start Simple

Begin with ground facts and single-variable queries:

```
fact parent(alice, bob).
query parent(alice, X)?
```

Then add rules incrementally.

10.2 Use Descriptive Names

Good:

```
fact employee(alice, engineering).
rule can_access(User, Resource) :- ...
```

Bad:

```
fact e(a, eng).
rule ca(U, R) :- ...
```

10.3 Comment Your Code

```
// User preferences - movies they've watched
fact likes(alice, inception).
fact likes(alice, matrix).

// Collaborative filtering: users with similar taste
rule similar_taste(X, Y) :- likes(X, Item), likes(Y, Item).
```

10.4 Test Incrementally

Don't write everything at once. Test after adding:

- Facts only
- One rule at a time
- One query at a time

10.5 Check Similarity Scores

If scores are unexpectedly low:

- Increase dimension (`dim=8192`)
- Increase beam width (`width=100`)
- Check for multi-hop derivations (naturally lower scores)

10.6 Use the Examples

Learn from the 12 included examples:

```
vsar-ide examples/02_family_tree.vsar
```

Study the patterns and adapt them to your use case.

10.7 Understand Approximate Reasoning

VSAR is **not** exact like Prolog:

- Results have confidence scores
- Multi-hop inference degrades gracefully
- This is a feature, not a bug!

Use it when you need:

- Noise tolerance
- Similarity-based matching
- Scalable inference

11 Getting Help

11.1 Documentation

- **User Manual**: `docs/user_manual.pdf` (this document)
- **IDE Status**: `VSAR_IDE_AND_EXAMPLES_UPDATE.md`
- **Implementation Plan**: `IMPLEMENTATION_PLAN.md`
- **API Docs**: See `src/vsar/` docstrings

11.2 Examples

Load and study the 12 example programs in `examples/`:

```
ls examples/*.vsar
```

11.3 Command-Line Help

```
vsar --help
vsar-ide --help
```

11.4 Issues and Feedback

Report issues at: https://github.com/anthropics/vsar/issues

12 Appendix: Complete Example

Here's a complete, working program demonstrating multiple features:

```
// ====================================================
// Movie Recommendation System
// ====================================================

@model FHRR(dim=8192, seed=42);
@beam(width=50);
@novelty(threshold=0.95);

// User preferences
fact likes(alice, inception).
fact likes(alice, matrix).
fact likes(alice, interstellar).
fact likes(bob, inception).
fact likes(bob, matrix).
fact likes(charlie, interstellar).
fact likes(charlie, arrival).

// Movie metadata
fact genre(inception, scifi).
fact genre(matrix, scifi).
fact genre(interstellar, scifi).
fact genre(arrival, scifi).
fact director(inception, nolan).
fact director(interstellar, nolan).
fact director(arrival, villeneuve).

// Collaborative filtering
rule similar_taste(X, Y) :-
    likes(X, Item),
    likes(Y, Item).

// Content-based recommendation
rule might_like(User, Movie) :-
    likes(User, KnownMovie),
    genre(KnownMovie, Genre),
    genre(Movie, Genre),
    not likes(User, Movie).

// Director-based recommendation
rule recommend_by_director(User, Movie) :-
    likes(User, KnownMovie),
    director(KnownMovie, Dir),
    director(Movie, Dir),
    not likes(User, Movie).

// Queries
query similar_taste(alice, X)?
query might_like(alice, X)?
query recommend_by_director(alice, X)?
```

Save this as `recommendations.vsar`, press **F5**, and explore!

End of User Manual

For the latest updates and documentation, visit:
https://github.com/anthropics/vsar