Implementation Guide: AIF-ASPIC+ Translation Framework

Executive Summary

This document provides a comprehensive implementation guide for bidirectional translation between the Argument Interchange Format (AIF) ontology and the ASPIC+ logical argumentation framework. It is based on the research paper "On Logical Specifications of the Argument Interchange Format" by Bex et al.

Core Purpose: Enable computational argumentation systems to interchange argument data and evaluate argument acceptability using formal logical semantics.

Key Components:

1. AIF Core Ontology (graph-based argument representation)
2. ASPIC+ Framework (structured argumentation with fixed preferences)
3. E-ASPIC+ Framework (extended with defeasible preferences)
4. Bidirectional translation functions
5. Identity-preserving properties and constraints

1. Conceptual Foundation

1.1 The Argument Interchange Format (AIF)

Purpose: A common ontology for expressing arguments across different computational argumentation systems, tools, and approaches.

Key Design Principles:

- **Interlingua approach:** Reduces translation complexity from $O(n^2)$ to $O(n)$ for n argumentation formats
- **Abstract representation:** Uses typed directed graphs, independent of specific logical formalisms
- **Minimal encoding bias:** Operates at knowledge level without symbol-level dependencies
- **Extensible:** Supports both formal logical systems and informal natural language arguments

Architecture: Two-layer structure

1. **Upper Ontology:** Defines basic building blocks (nodes and edges)
2. **Forms Ontology:** Defines argumentation-theoretic concepts (schemes, statement types)

1.2 The ASPIC+ Framework

Purpose: A logical framework for structured argumentation that:

- Constructs arguments as inference trees
- Defines attack relations between arguments
- Uses preferences to determine which attacks succeed as defeats
- Instantiates Dung's abstract argumentation frameworks
- Satisfies formal rationality postulates

Key Features:

- Intermediate abstraction level between fully abstract and fully concrete systems
- Accommodates both strict (deductive) and defeasible inference rules
- Supports three forms of attack: undercutting, rebutting, undermining
- Proven to subsume multiple other argumentation formalisms

1.3 E-ASPIC+ Framework

Extension of ASPIC+ that:

- Allows preferences to be argued for (not just given)
- Supports attacks on attacks (preference attacks)
- Instantiates Extended Argumentation Frameworks (EAFs)
- Enables full argumentation about argument preferences

2. AIF Core Ontology Specification

2.1 Node Types (Upper Ontology)

Information Nodes (I-nodes)

- **Purpose:** Represent propositions, statements, claims
- **Content:** Text, formulas, or other information
- **Attributes:** Identifier, content, optional metadata (creator, date, etc.)

Scheme Application Nodes (S-nodes)

Three subtypes:

1. **Rule Application Nodes (RA-nodes)**
    - Represent specific instances of inference
    - Fulfill inference schemes from Forms Ontology
    - Can be deductive or defeasible
2. **Conflict Application Nodes (CA-nodes)**
    - Represent specific instances of conflict
    - Fulfill conflict schemes from Forms Ontology
    - Not necessarily symmetric
3. **Preference Application Nodes (PA-nodes)**
    - Represent specific instances of preference
    - Fulfill preference schemes from Forms Ontology

- Not necessarily symmetric

2.2 Edge Types

Edges connecting to RA-nodes:

- `premise`: From I-node or RA-node to RA-node (input)
- `presumption`: From I-node to RA-node (defeasible input)
- `conclusion`: From RA-node to I-node or RA-node (output)

Edges connecting to CA-nodes:

- `conflicting element`: From I-node or RA-node to CA-node (attacker)
- `conflicted element`: From CA-node to I-node or RA-node (attacked)

Edges connecting to PA-nodes:

- `preferred element`: From I-node or RA-node to PA-node (stronger element)
- `dispreferred element`: From PA-node to I-node or RA-node (weaker element)

2.3 Formal Definition of AIF Argument Graph

Definition: AIF Argument Graph $G = (V, E)$

Where:

- $V = I \cup RA \cup CA \cup PA$ (set of all nodes)

- $E \subseteq V \times V \setminus I \times I$ (set of typed edges, no direct I-to-I connections)

Constraints:

1. I-nodes only connect via S-nodes (no direct I-I edges)

2. Each S-node has ≥1 predecessor and ≥1 successor

3. RA-nodes: ≥1 premise, exactly 1 conclusion

4. PA-nodes: exactly 1 preferred element, exactly 1 dispreferred element

5. CA-nodes: exactly 1 conflicting element, exactly 1 conflicted element

2.4 Forms Ontology

Inference Schemes:

- Define general principles of inference (like rules in logic)
- Can be deductive (guaranteed conclusions) or defeasible (presumptive conclusions)
- Examples: Modus Ponens, Argument from Expert Opinion

Conflict Schemes:

- Define general principles of conflict
- Two elements: conflicting (attacker), conflicted (attacked)
- Example: Conflict from Expert Unreliability

Preference Schemes:

- Define general principles of preference
- Two elements: preferred (stronger), dispreferred (weaker)
- Can apply to inferences or information
- Example: Expert Opinion preferred over General Knowledge

Statement Descriptions:

- Ordinary premises (can be attacked, compared by preference)
- Axioms (cannot be attacked)
- Assumptions (can be attacked, attacks always succeed)

3. ASPIC+ Framework Specification

3.1 Core Components

Argumentation System (AS)

AS = (L, ⁻, R, ≤)

Components:

- L: Logical language (set of well-formed formulas)

- ⁻: Contrariness function (L → 2^L)

- R = Rs ∪ Rd: Inference rules (strict ∪ defeasible)

- ≤: Partial preorder on Rd (rule preferences)

Knowledge Base (KB)

KB = (K, ≤')

Components:

- K = Kn ∪ Kp ∪ Ka

- Kn: Necessary axioms (cannot be attacked)

- Kp: Ordinary premises (can be attacked with preferences)

- Ka: Assumptions (can be attacked, always defeated)

- ≤': Partial preorder on K\Kn (premise preferences)

Argumentation Theory (AT)

AT = (AS, KB, <)

Components:

- AS: Argumentation system

- KB: Knowledge base

- <: Admissible argument ordering (derived from ≤ and ≤')

3.2 Inference Rules

Strict Rules: $\varphi_1, \ldots, \varphi_n \rightarrow \varphi$

- If antecedents hold, consequent necessarily holds
- Cannot be attacked on the inference itself

Defeasible Rules: $\varphi_1, \ldots, \varphi_n \Rightarrow \varphi$

- If antecedents hold, consequent presumably holds
- Can be undercut (attacked on the inference)

Rule Naming Convention:

- Subset LR ⊆ L contains formulas naming rules

- Each rule $r \in R$ has a corresponding name $r \in LR$
- Enables undercutting attacks in the object language

3.3 Argument Structure

Recursive Definition:

Argument A:

1. Base case: ϕ if $\phi \in K$

   - $Prem(A) = \{\phi\}$

   - $Conc(A) = \phi$

   - $Sub(A) = \{\phi\}$

   - $Rules(A) = \varnothing$

2. Inductive case: $A_1,...,A_n \rightarrow/\Rightarrow \psi$

   if $\exists$ rule $(Conc(A_1),...,Conc(A_n) \rightarrow/\Rightarrow \psi) \in R$

   - $Prem(A) = Prem(A_1) \cup ... \cup Prem(A_n)$

   - $Conc(A) = \psi$

   - $Sub(A) = Sub(A_1) \cup ... \cup Sub(A_n) \cup \{A\}$

   - $Rules(A) = Rules(A_1) \cup ... \cup Rules(A_n) \cup \{rule\}$

Argument Classifications:

- **Strict:** $DefRules(A) = \varnothing$ (no defeasible rules)
- **Defeasible:** $DefRules(A) \neq \varnothing$ (has defeasible rules)
- **Firm:** $Prem(A) \subseteq Kn$ (only axioms as premises)
- **Plausible:** $Prem(A) \not\subset Kn$ (has non-axiom premises)

3.4 Attack Relations

Three Forms of Attack:

1. **Undercutting:** A undercuts B on B'
   - $Conc(A) \in \bar{}r$ for some $B' \in Sub(B)$
   - Where B' has defeasible top rule r

- Attacks the inference itself

2. **Rebutting:** A rebuts B on B'
   - Conc(A) ∈ $\bar{}\phi$ for some B' ∈ Sub(B)
   - Where B' = $B''_1,...,B''_n \Rightarrow \phi$
   - Attacks the conclusion of an inference
   - **Contrary-rebut:** if Conc(A) is contrary (not contradictory) to ϕ
3. **Undermining:** A undermines B on ϕ
   - Conc(A) ∈ $\bar{}\phi$ for some ϕ ∈ Prem(B)\Kn
   - Attacks a non-axiom premise
   - **Contrary-undermine:** if Conc(A) is contrary to ϕ or ϕ ∈ Ka

3.5 Defeat Relations

Preference-Independent Attacks (always succeed):

- Undercutting
- Contrary-rebutting
- Contrary-undermining

Preference-Dependent Attacks (succeed only if not weaker):

- Regular rebutting (Conc(A) contradicts conclusion)
- Regular undermining (Conc(A) contradicts ordinary premise)

Defeat Definition:

A defeats B iff:

A attacks B on B', AND

(attack is preference-independent OR A $\nprec$ B')

Strict Defeat:

A strictly defeats B iff:

A defeats B AND B does not defeat A

3.6 Corresponding Dung Framework

DF_AT = (A, C)

Where:

- A: Set of all arguments constructible from AT

- C: Defeat relation on A (from Definition 3.9)

Apply Dung semantics (grounded, preferred, stable, etc.) to evaluate acceptability.

4. E-ASPIC+ Framework Specification

4.1 Key Differences from ASPIC+

Removed:

- Fixed preference orderings $\preceq$ and $\preceq'$
- Fixed argument ordering $<$

Added:

- Preference language Lm for expressing preferences
- Partial function P extracting orderings from preference arguments
- Second attack relation D for attacks on attacks (pref-attacks)

4.2 Extended Components

Extended Argumentation System (EAS)

EAS = (L, $^-$, R)

Components:

- L: Includes preference sublanguage Lm

- Lm = {l > l' | l, l' ∈ L} (preference expressions)

- $^-$: Contrariness function

- R = Rs ∪ Rd

- Rs includes axioms for partial preorder:

- o1: (z > y) ∧ (y > x) → (z > x) [transitivity]

- o2: (y > x) → ¬(x > y) [asymmetry]

Extended Knowledge Base (EKB)

EKB = K = Kn ∪ Kp ∪ Ka

Note: No orderings ≤ or ≤' (preferences now argued for)

Extended Argumentation Theory (EAT)

EAT = (EAS, EKB, P)

Where P is a partial function:

P: 2^A → Pow(A × A)

Interpretation:

If (X,Y) ∈ P(ϕ), then Y < X given arguments ϕ

(Y is less preferred than X based on arguments ϕ)

4.3 Preference Function P

Two Standard Definitions:

Weakest-Link Principle:

- B < A if ALL of B's defeasible rules are weaker than ALL of A's rules
- AND ALL of B's ordinary/assumption premises are weaker than ALL of A's premises

Last-Link Principle:

- B < A if B's LAST defeasible rule is weaker than ALL of A's last rules
- OR (if both strict) if B has a premise weaker than ALL of A's premises

4.4 Extended Argumentation Framework

EAF_C = (A, C, D)

Components:

- A: Set of arguments

- C: Attack relation (as in ASPIC+)

- D ⊆ (2^A \ ∅) × C: Pref-attacks (attacks on attacks)

Definition of D:

(ϕ, (A,B)) ∈ D iff:

1. **(A,B) ∈ C (A attacks B)**
2. **For all B' attacked by A:**

   ∃ϕ' ⊆ ϕ such that A < B' ∈ P(ϕ')
3. **ϕ is minimal satisfying condition 2**
4. **A's attack on B is NOT preference-independent**

S-Defeat (relative to set S):

A S-defeats B iff:

(A,B) ∈ C AND

¬∃ϕ ⊆ S such that (ϕ, (A,B)) ∈ D

Implementation Guide: AIF-ASPIC+ Translation Framework (TypeScript/Next.js/ Supabase Edition)

I'll revise the code examples to use TypeScript, Next.js, and Supabase with Prisma. Here are the key updated sections:

2.2 Prisma Schema for AIF Core Ontology

```
// prisma/schema.prisma

model Node {
  id          String   @id @default(cuid())
  type        NodeType
  content     String?
  fulfills    String?  // Reference to form in Forms
ontology
  attributes  Json?    // Flexible metadata
  graphId     String
  graph       Graph    @relation(fields: [graphId],
references: [id])

  // Relations
  edgesFrom   Edge[]   @relation("EdgeSource")
  edgesTo     Edge[]   @relation("EdgeTarget")

  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt

  @@index([graphId, type])
}

enum NodeType {
  I_NODE  // Information node
  RA_NODE // Rule application node
  CA_NODE // Conflict application node
  PA_NODE // Preference application node
}

model Edge {
  id          String   @id @default(cuid())
  type        EdgeType
  sourceId    String
  targetId    String
  graphId     String

  source      Node     @relation("EdgeSource", fields:
[sourceId], references: [id], onDelete: Cascade)
```

```
  target      Node     @relation("EdgeTarget", fields:
[targetId], references: [id], onDelete: Cascade)
  graph       Graph    @relation(fields: [graphId],
references: [id])

  createdAt   DateTime @default(now())

  @@index([graphId])
  @@index([sourceId])
  @@index([targetId])
}

enum EdgeType {
  PREMISE
  PRESUMPTION
  CONCLUSION
  CONFLICTING_ELEMENT
  CONFLICTED_ELEMENT
  PREFERRED_ELEMENT
  DISPREFERRED_ELEMENT
}

model Graph {
  id          String   @id @default(cuid())
  name        String?
  description String?
  nodes       Node[]
  edges       Edge[]
  metadata    Json?

  userId      String?
  user        User?    @relation(fields: [userId],
references: [id])

  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
}

model ArgumentationTheory {
  id              String   @id @default(cuid())
```

```
  graphId         String   @unique
  language        Json     // Set of formulas
  contrariness    Json     // Map of formula -> contraries
  rules           Json     // Strict and defeasible rules
  knowledgeBase   Json     // K_n, K_p, K_a
  preferences     Json     // Rule and premise preferences

  createdAt       DateTime @default(now())
  updatedAt       DateTime @updatedAt
}

model Argument {
  id              String   @id @default(cuid())
  theoryId        String
  premises        Json     // Set of premise IDs
  conclusion      String
  subArguments    Json     // Set of sub-argument IDs
  rules           Json     // Set of rule IDs
  topRule         String?
  isStrict        Boolean
  isFirm          Boolean

  createdAt       DateTime @default(now())

  @@index([theoryId])
}
```

11.2 Core TypeScript Data Structures

```
// lib/aif/types.ts

export type NodeType = 'I_NODE' | 'RA_NODE' | 'CA_NODE' |
'PA_NODE';
export type EdgeType =
  | 'PREMISE'
  | 'PRESUMPTION'
  | 'CONCLUSION'
  | 'CONFLICTING_ELEMENT'
  | 'CONFLICTED_ELEMENT'
  | 'PREFERRED_ELEMENT'
  | 'DISPREFERRED_ELEMENT';
```

```
export interface Node {
  id: string;
  type: NodeType;
  content?: string;
  fulfills?: string;
  attributes?: Record<string, any>;
}

export interface Edge {
  id: string;
  type: EdgeType;
  sourceId: string;
  targetId: string;
}

export interface AIFGraph {
  id: string;
  nodes: Map<string, Node>;
  edges: Edge[];
  metadata?: Record<string, any>;
}

export interface InferenceRule {
  name: string;
  antecedents: string[];
  consequent: string;
  type: 'strict' | 'defeasible';
}

export interface ArgumentStructure {
  id: string;
  premises: Set<string>;
  conclusion: string;
  subArguments: Set<string>;
  rules: Set<string>;
  topRule?: string;
}

export interface ArgumentationSystem {
```

```
  language: Set<string>;
  contrariness: Map<string, Set<string>>;
  strictRules: Set<InferenceRule>;
  defeasibleRules: Set<InferenceRule>;
  rulePreferences: Set<[string, string]>; // (weaker,
stronger)
}

export interface KnowledgeBase {
  axioms: Set<string>;        // K_n
  premises: Set<string>;      // K_p
  assumptions: Set<string>;   // K_a
  premisePreferences: Set<[string, string]>;
}

export interface ArgumentationTheory {
  argumentationSystem: ArgumentationSystem;
  knowledgeBase: KnowledgeBase;
  argumentOrdering: (a: ArgumentStructure, b:
ArgumentStructure) => boolean;
}
```

11.2 Graph Manipulation Class

```
// lib/aif/graph.ts

import { PrismaClient } from '@prisma/client';
import type { AIFGraph, Node, Edge, NodeType, EdgeType }
from './types';

export class AIFGraphManager {
  constructor(private prisma: PrismaClient) {}

  async createGraph(
    name?: string,
    userId?: string
  ): Promise<string> {
    const graph = await this.prisma.graph.create({
      data: {
        name,
        userId,
```

```
      nodes: { create: [] },
      edges: { create: [] }
    }
  });
  return graph.id;
}

async addNode(
  graphId: string,
  type: NodeType,
  content?: string,
  fulfills?: string
): Promise<string> {
  const node = await this.prisma.node.create({
    data: {
      type,
      content,
      fulfills,
      graphId
    }
  });
  return node.id;
}

async addEdge(
  graphId: string,
  sourceId: string,
  targetId: string,
  type: EdgeType
): Promise<string> {
  const edge = await this.prisma.edge.create({
    data: {
      type,
      sourceId,
      targetId,
      graphId
    }
  });
  return edge.id;
}
```

```
async getPredecessors(
  nodeId: string,
  edgeType?: EdgeType
): Promise<Node[]> {
  const edges = await this.prisma.edge.findMany({
    where: {
      targetId: nodeId,
      ...(edgeType && { type: edgeType })
    },
    include: {
      source: true
    }
  });
  return edges.map(e => e.source);
}

async getSuccessors(
  nodeId: string,
  edgeType?: EdgeType
): Promise<Node[]> {
  const edges = await this.prisma.edge.findMany({
    where: {
      sourceId: nodeId,
      ...(edgeType && { type: edgeType })
    },
    include: {
      target: true
    }
  });
  return edges.map(e => e.target);
}

async isInitialNode(nodeId: string): Promise<boolean> {
  const count = await this.prisma.edge.count({
    where: { targetId: nodeId }
  });
  return count === 0;
}
```

```
  async validateGraph(graphId: string): Promise<string[]> {
    const violations: string[] = [];

    const nodes = await this.prisma.node.findMany({
      where: { graphId },
      include: {
        edgesFrom: true,
        edgesTo: true
      }
    });

    for (const node of nodes) {
      // S-nodes must have predecessors and successors
      if (['RA_NODE', 'CA_NODE',
'PA_NODE'].includes(node.type)) {
        if (node.edgesTo.length === 0) {
          violations.push(`S-node ${node.id} has no
predecessors`);
        }
        if (node.edgesFrom.length === 0) {
          violations.push(`S-node ${node.id} has no
successors`);
        }
      }

      // RA-node specific constraints
      if (node.type === 'RA_NODE') {
        const premises = node.edgesTo.filter(e => e.type
=== 'PREMISE');
        const conclusions = node.edgesFrom.filter(e =>
e.type === 'CONCLUSION');

        if (premises.length === 0) {
          violations.push(`RA-node ${node.id} has no
premises`);
        }
        if (conclusions.length !== 1) {
          violations.push(`RA-node ${node.id} must have
exactly 1 conclusion`);
        }
```

```
      }

      // PA-node specific constraints
      if (node.type === 'PA_NODE') {
        const preferred = node.edgesTo.filter(e => e.type
=== 'PREFERRED_ELEMENT');
        const dispreferred = node.edgesFrom.filter(e =>
e.type === 'DISPREFERRED_ELEMENT');

        if (preferred.length !== 1 || dispreferred.length !
== 1) {
          violations.push(`PA-node ${node.id} must have 1
preferred and 1 dispreferred`);
        }
      }

      // CA-node specific constraints
      if (node.type === 'CA_NODE') {
        const conflicting = node.edgesTo.filter(e => e.type
=== 'CONFLICTING_ELEMENT');
        const conflicted = node.edgesFrom.filter(e =>
e.type === 'CONFLICTED_ELEMENT');

        if (conflicting.length !== 1 || conflicted.length !
== 1) {
          violations.push(`CA-node ${node.id} must have 1
conflicting and 1 conflicted`);
        }
      }
    }

    return violations;
  }

  async loadGraph(graphId: string): Promise<AIFGraph> {
    const graph = await this.prisma.graph.findUnique({
      where: { id: graphId },
      include: {
        nodes: true,
        edges: true
```

```
      }
    });

    if (!graph) {
      throw new Error(`Graph ${graphId} not found`);
    }

    const nodes = new Map(graph.nodes.map(n => [n.id, n]));

    return {
      id: graph.id,
      nodes,
      edges: graph.edges,
      metadata: graph.metadata as Record<string, any> |
undefined
    };
  }
}
```

5.1 Translation: AIF to ASPIC+ (TypeScript)

```
// lib/translation/aif-to-aspic.ts

import type {
  AIFGraph,
  ArgumentationTheory,
  ArgumentationSystem,
  KnowledgeBase,
  InferenceRule
} from '../aif/types';
import { AIFGraphManager } from '../aif/graph';
import { PrismaClient } from '@prisma/client';

export class AIFToASPICTranslator {
  constructor(
    private prisma: PrismaClient,
    private graphManager: AIFGraphManager
  ) {}

  async translate(
    graphId: string,
```

```
    formsOntology: Map<string, any>
  ): Promise<ArgumentationTheory> {
    const graph = await
this.graphManager.loadGraph(graphId);

    // Step 1: Construct language
    const language = this.constructLanguage(graph);

    // Step 2: Construct knowledge base
    const knowledgeBase = await
this.constructKnowledgeBase(
      graph,
      formsOntology
    );

    // Step 3: Construct rules
    const rules = await this.constructRules(graph,
formsOntology);

    // Step 4: Construct contrariness
    const contrariness = await
this.constructContrariness(graph);

    // Step 5: Construct preferences
    const preferences = await
this.constructPreferences(graph);

    // Create argumentation system
    const argumentationSystem: ArgumentationSystem = {
      language,
      contrariness,
      strictRules: rules.strict,
      defeasibleRules: rules.defeasible,
      rulePreferences: preferences.rules
    };

    // Create knowledge base with preferences
    const kb: KnowledgeBase = {
      ...knowledgeBase,
      premisePreferences: preferences.premises
```

```
    };

    // Derive argument ordering
    const argumentOrdering = this.deriveArgumentOrdering(
      preferences.premises,
      preferences.rules
    );

    return {
      argumentationSystem,
      knowledgeBase: kb,
      argumentOrdering
    };
  }

  private constructLanguage(graph: AIFGraph): Set<string> {
    const language = new Set<string>();

    // Add all I-nodes
    for (const [id, node] of graph.nodes) {
      if (node.type === 'I_NODE' && node.content) {
        language.add(node.content);
      }
    }

    // Add all RA-nodes (rule names)
    for (const [id, node] of graph.nodes) {
      if (node.type === 'RA_NODE') {
        language.add(id);
      }
    }

    return language;
  }

  private async constructKnowledgeBase(
    graph: AIFGraph,
    formsOntology: Map<string, any>
  ): Promise<Omit<KnowledgeBase, 'premisePreferences'>> {
    const axioms = new Set<string>();
```

```
    const premises = new Set<string>();
    const assumptions = new Set<string>();

    // Find initial nodes
    const initialNodeIds = new Set<string>();
    for (const [id, node] of graph.nodes) {
      const hasIncoming = graph.edges.some(e => e.targetId
=== id);
      if (!hasIncoming) {
        initialNodeIds.add(id);
      }
    }

    // Classify by fulfillment
    for (const nodeId of initialNodeIds) {
      const node = graph.nodes.get(nodeId);
      if (!node || node.type !== 'I_NODE' || !node.content)
continue;

      const form = node.fulfills ?
formsOntology.get(node.fulfills) : null;

      if (form?.type === 'axiom') {
        axioms.add(node.content);
      } else if (form?.type === 'assumption') {
        assumptions.add(node.content);
      } else {
        premises.add(node.content);
      }
    }

    return { axioms, premises, assumptions };
  }

  private async constructRules(
    graph: AIFGraph,
    formsOntology: Map<string, any>
  ): Promise<{
    strict: Set<InferenceRule>;
    defeasible: Set<InferenceRule>;
```

```
  }> {
    const strict = new Set<InferenceRule>();
    const defeasible = new Set<InferenceRule>();

    for (const [id, node] of graph.nodes) {
      if (node.type !== 'RA_NODE') continue;

      // Get predecessors (premises)
      const premiseEdges = graph.edges.filter(
        e => e.targetId === id && e.type === 'PREMISE'
      );
      const antecedents = premiseEdges
        .map(e => graph.nodes.get(e.sourceId)?.content)
        .filter((c): c is string => !!c);

      // Get successor (conclusion)
      const conclusionEdge = graph.edges.find(
        e => e.sourceId === id && e.type === 'CONCLUSION'
      );
      const consequent = conclusionEdge
        ? graph.nodes.get(conclusionEdge.targetId)?.content
        : null;

      if (!consequent) continue;

      // Determine rule type
      const form = node.fulfills ?
formsOntology.get(node.fulfills) : null;
      const isStrict = form?.type === 'deductive';

      const rule: InferenceRule = {
        name: id,
        antecedents,
        consequent,
        type: isStrict ? 'strict' : 'defeasible'
      };

      if (isStrict) {
        strict.add(rule);
      } else {
```

```
        defeasible.add(rule);
      }
    }

    return { strict, defeasible };
  }

  private async constructContrariness(
    graph: AIFGraph
  ): Promise<Map<string, Set<string>>> {
    const contrariness = new Map<string, Set<string>>();

    for (const [id, node] of graph.nodes) {
      if (node.type !== 'CA_NODE') continue;

      // Get conflicting element
      const conflictingEdge = graph.edges.find(
        e => e.targetId === id && e.type ===
'CONFLICTING_ELEMENT'
      );
      const conflicting = conflictingEdge
        ?
graph.nodes.get(conflictingEdge.sourceId)?.content
        : null;

      // Get conflicted element
      const conflictedEdge = graph.edges.find(
        e => e.sourceId === id && e.type ===
'CONFLICTED_ELEMENT'
      );
      const conflicted = conflictedEdge
        ? graph.nodes.get(conflictedEdge.targetId)?.content
        : null;

      if (conflicting && conflicted) {
        if (!contrariness.has(conflicting)) {
          contrariness.set(conflicting, new Set());
        }
        contrariness.get(conflicting)!.add(conflicted);
      }
```

```
    }

    return contrariness;
  }

  private async constructPreferences(
    graph: AIFGraph
  ): Promise<{
    premises: Set<[string, string]>;
    rules: Set<[string, string]>;
  }> {
    const premises = new Set<[string, string]>();
    const rules = new Set<[string, string]>();

    for (const [id, node] of graph.nodes) {
      if (node.type !== 'PA_NODE') continue;

      // Get preferred element
      const preferredEdge = graph.edges.find(
        e => e.targetId === id && e.type ===
'PREFERRED_ELEMENT'
      );
      const preferredNode = preferredEdge
        ? graph.nodes.get(preferredEdge.sourceId)
        : null;

      // Get dispreferred element
      const dispreferredEdge = graph.edges.find(
        e => e.sourceId === id && e.type ===
'DISPREFERRED_ELEMENT'
      );
      const dispreferredNode = dispreferredEdge
        ? graph.nodes.get(dispreferredEdge.targetId)
        : null;

      if (!preferredNode || !dispreferredNode) continue;

      // Check if both are I-nodes (premise preference)
      if (
        preferredNode.type === 'I_NODE' &&
```

```
        dispreferredNode.type === 'I_NODE' &&
        preferredNode.content &&
        dispreferredNode.content
      ) {
        premises.add([dispreferredNode.content,
preferredNode.content]);
      }

      // Check if both are RA-nodes (rule preference)
      if (
        preferredNode.type === 'RA_NODE' &&
        dispreferredNode.type === 'RA_NODE'
      ) {
        rules.add([dispreferredNode.id, preferredNode.id]);
      }
    }

    return { premises, rules };
  }

  private deriveArgumentOrdering(
    premisePrefs: Set<[string, string]>,
    rulePrefs: Set<[string, string]>
  ): (a: ArgumentStructure, b: ArgumentStructure) =>
boolean {
    // Returns true if a < b (a is less preferred than b)
    return (a, b) => {
      // Implement weakest-link or last-link principle
      // This is a simplified version
      return false; // Placeholder
    };
  }
}
```

11.3 Argument Construction (TypeScript)

```
// lib/reasoning/argument-constructor.ts

import type {
  ArgumentationTheory,
  ArgumentStructure,
```

```
  InferenceRule
} from '../aif/types';

export class ArgumentConstructor {
  private constructedArgs = new Map<string,
ArgumentStructure>();
  private argCounter = 0;

  constructor(private theory: ArgumentationTheory) {}

  constructAllArguments(): Set<ArgumentStructure> {
    const arguments = new Set<ArgumentStructure>();

    // Base arguments from knowledge base
    const { axioms, premises, assumptions } =
this.theory.knowledgeBase;

    for (const premise of
[...axioms, ...premises, ...assumptions]) {
      const arg: ArgumentStructure = {
        id: this.generateId(),
        premises: new Set([premise]),
        conclusion: premise,
        subArguments: new Set(),
        rules: new Set()
      };
      arg.subArguments.add(arg.id);
      arguments.add(arg);
    }

    // Iteratively build arguments using rules
    let changed = true;
    while (changed) {
      changed = false;
      const newArguments = new Set<ArgumentStructure>();

      const allRules = [
        ...this.theory.argumentationSystem.strictRules,
        ...this.theory.argumentationSystem.defeasibleRules
      ];
```

```
      for (const rule of allRules) {
        const matchingCombinations =
this.findMatchingArguments(
          arguments,
          rule.antecedents
        );

        for (const combination of matchingCombinations) {
          const newArg = this.applyRule(combination, rule);

          if (!this.argumentExists(arguments, newArg)) {
            newArguments.add(newArg);
            changed = true;
          }
        }
      }

      for (const arg of newArguments) {
        arguments.add(arg);
      }
    }

    return arguments;
  }

  private findMatchingArguments(
    arguments: Set<ArgumentStructure>,
    antecedents: string[]
  ): ArgumentStructure[][] {
    const matches: ArgumentStructure[][] = [];

    const backtrack = (
      index: number,
      current: ArgumentStructure[]
    ): void => {
      if (index === antecedents.length) {
        matches.push([...current]);
        return;
      }
```

```
    const antecedent = antecedents[index];
    for (const arg of arguments) {
      if (arg.conclusion === antecedent) {
        current.push(arg);
        backtrack(index + 1, current);
        current.pop();
      }
    }
  };

  backtrack(0, []);
  return matches;
}

private applyRule(
  subArguments: ArgumentStructure[],
  rule: InferenceRule
): ArgumentStructure {
  const newArg: ArgumentStructure = {
    id: this.generateId(),
    premises: new Set(),
    conclusion: rule.consequent,
    subArguments: new Set(),
    rules: new Set(),
    topRule: rule.name
  };

  // Combine premises from all sub-arguments
  for (const subArg of subArguments) {
    for (const premise of subArg.premises) {
      newArg.premises.add(premise);
    }
  }

  // Combine sub-arguments
  for (const subArg of subArguments) {
    for (const sub of subArg.subArguments) {
      newArg.subArguments.add(sub);
    }
```

```
    }
    newArg.subArguments.add(newArg.id);

    // Combine rules
    for (const subArg of subArguments) {
      for (const r of subArg.rules) {
        newArg.rules.add(r);
      }
    }
    newArg.rules.add(rule.name);

    return newArg;
  }

  private argumentExists(
    arguments: Set<ArgumentStructure>,
    arg: ArgumentStructure
  ): boolean {
    for (const existing of arguments) {
      if (
        existing.conclusion === arg.conclusion &&
        this.setsEqual(existing.premises, arg.premises) &&
        this.setsEqual(existing.rules, arg.rules)
      ) {
        return true;
      }
    }
    return false;
  }

  private setsEqual<T>(a: Set<T>, b: Set<T>): boolean {
    if (a.size !== b.size) return false;
    for (const item of a) {
      if (!b.has(item)) return false;
    }
    return true;
  }

  private generateId(): string {
    return `arg_${this.argCounter++}`;
```

```
  }
}
```

12.1 Next.js API Routes

```
// app/api/translate/aif-to-aspic/route.ts

import { NextRequest, NextResponse } from 'next/server';
import { PrismaClient } from '@prisma/client';
import { AIFGraphManager } from '@/lib/aif/graph';
import { AIFToASPICTranslator } from '@/lib/translation/
aif-to-aspic';
import { ArgumentConstructor } from '@/lib/reasoning/
argument-constructor';

const prisma = new PrismaClient();

export async function POST(request: NextRequest) {
  try {
    const body = await request.json();
    const { graphId, formsOntology } = body;

    if (!graphId) {
      return NextResponse.json(
        { error: 'graphId is required' },
        { status: 400 }
      );
    }

    // Initialize managers
    const graphManager = new AIFGraphManager(prisma);
    const translator = new AIFToASPICTranslator(prisma,
graphManager);

    // Translate
    const formsMap = new Map(Object.entries(formsOntology
|| {}));
    const theory = await translator.translate(graphId,
formsMap);

    // Construct arguments
```

```
    const constructor = new ArgumentConstructor(theory);
    const arguments = constructor.constructAllArguments();

    // Save to database
    const savedTheory = await
prisma.argumentationTheory.create({
      data: {
        graphId,
        language:
Array.from(theory.argumentationSystem.language),
        contrariness: Object.fromEntries(

Array.from(theory.argumentationSystem.contrariness).map(
            ([k, v]) => [k, Array.from(v)]
          )
        ),
        rules: {
          strict:
Array.from(theory.argumentationSystem.strictRules),
          defeasible:
Array.from(theory.argumentationSystem.defeasibleRules)
        },
        knowledgeBase: {
          axioms: Array.from(theory.knowledgeBase.axioms),
          premises:
Array.from(theory.knowledgeBase.premises),
          assumptions:
Array.from(theory.knowledgeBase.assumptions)
        },
        preferences: {
          premises:
Array.from(theory.knowledgeBase.premisePreferences),
          rules:
Array.from(theory.argumentationSystem.rulePreferences)
        }
      }
    });

    return NextResponse.json({
      success: true,
```

```
      theoryId: savedTheory.id,
      argumentCount: arguments.size,
      arguments: Array.from(arguments).map(arg => ({
        id: arg.id,
        conclusion: arg.conclusion,
        premises: Array.from(arg.premises),
        rules: Array.from(arg.rules)
      }))
    });
  } catch (error) {
    console.error('Translation error:', error);
    return NextResponse.json(
      { error: 'Translation failed', details: (error as
Error).message },
      { status: 500 }
    );
  }
}
// app/api/evaluate/route.ts

import { NextRequest, NextResponse } from 'next/server';
import { PrismaClient } from '@prisma/client';
import { DungFramework } from '@/lib/reasoning/dung-
framework';

const prisma = new PrismaClient();

export async function POST(request: NextRequest) {
  try {
    const body = await request.json();
    const { theoryId, semantics = 'preferred' } = body;

    if (!theoryId) {
      return NextResponse.json(
        { error: 'theoryId is required' },
        { status: 400 }
      );
    }

    // Load theory and arguments
```

```
    const theory = await
prisma.argumentationTheory.findUnique({
      where: { id: theoryId }
    });

    if (!theory) {
      return NextResponse.json(
        { error: 'Theory not found' },
        { status: 404 }
      );
    }

    const arguments = await prisma.argument.findMany({
      where: { theoryId }
    });

    // Create Dung framework
    const df = new DungFramework(arguments);
    await df.computeDefeats();

    // Compute extensions
    let extensions;
    switch (semantics) {
      case 'grounded':
        extensions = [await df.computeGroundedExtension()];
        break;
      case 'preferred':
        extensions = await df.computePreferredExtensions();
        break;
      case 'stable':
        extensions = await df.computeStableExtensions();
        break;
      default:
        return NextResponse.json(
          { error: `Unknown semantics: ${semantics}` },
          { status: 400 }
        );
    }

    return NextResponse.json({
```

```
      success: true,
      semantics,
      extensions: extensions.map(ext =>
        ext.map(arg => ({
          id: arg.id,
          conclusion: arg.conclusion
        }))
      )
    });
  } catch (error) {
    console.error('Evaluation error:', error);
    return NextResponse.json(
      { error: 'Evaluation failed', details: (error as
Error).message },
      { status: 500 }
    );
  }
}
```

12.2 React Components for Visualization

```
// components/argument-graph-visualizer.tsx

'use client';

import { useEffect, useRef } from 'react';
import { graphviz } from 'd3-graphviz';

interface Node {
  id: string;
  type: 'I_NODE' | 'RA_NODE' | 'CA_NODE' | 'PA_NODE';
  content?: string;
}

interface Edge {
  sourceId: string;
  targetId: string;
  type: string;
}

interface GraphVisualizerProps {
```

```
  nodes: Node[];
  edges: Edge[];
}

export function ArgumentGraphVisualizer({ nodes, edges }:
GraphVisualizerProps) {
  const containerRef = useRef<HTMLDivElement>(null);

  useEffect(() => {
    if (!containerRef.current) return;

    const dot = generateDotString(nodes, edges);

    graphviz(containerRef.current)
      .renderDot(dot)
      .transition(() => {
        return d3.transition().duration(500);
      });
  }, [nodes, edges]);

  return (
    <div
      ref={containerRef}
      className="w-full h-full min-h-[500px] border
rounded-lg"
    />
  );
}

function generateDotString(nodes: Node[], edges: Edge[]):
string {
  const nodeStyles: Record<string, string> = {
    I_NODE: 'shape=box, style=rounded',
    RA_NODE: 'shape=circle, style=filled,
fillcolor=lightblue',
    CA_NODE: 'shape=diamond, style=filled,
fillcolor=lightcoral',
    PA_NODE: 'shape=diamond, style=filled,
fillcolor=lightgreen'
  };
```

```
  const edgeStyles: Record<string, string> = {
    PREMISE: 'color=black',
    CONCLUSION: 'color=blue',
    CONFLICTING_ELEMENT: 'color=red',
    PREFERRED_ELEMENT: 'color=green'
  };

  let dot = 'digraph G {\n';
  dot += '  rankdir=TB;\n';
  dot += '  node [fontname="Arial"];\n\n';

  // Add nodes
  for (const node of nodes) {
    const label = node.content || node.id;
    const style = nodeStyles[node.type] || '';
    dot += `  "${node.id}" [label="${label}", ${style}];
\n`;
  }

  dot += '\n';

  // Add edges
  for (const edge of edges) {
    const style = edgeStyles[edge.type] || '';
    dot += `  "${edge.sourceId}" -> "${edge.targetId}" [$
{style}, label="${edge.type}"];\n`;
  }

  dot += '}\n';
  return dot;
}
```

12.3 Testing with Jest/Vitest

```
// __tests__/translation.test.ts

import { describe, it, expect, beforeAll, afterAll } from
'vitest';
import { PrismaClient } from '@prisma/client';
import { AIFGraphManager } from '@/lib/aif/graph';
```

```
import { AIFToASPICTranslator } from '@/lib/translation/
aif-to-aspic';

const prisma = new PrismaClient();

describe('AIF to ASPIC+ Translation', () => {
  let graphManager: AIFGraphManager;
  let translator: AIFToASPICTranslator;
  let testGraphId: string;

  beforeAll(async () => {
    graphManager = new AIFGraphManager(prisma);
    translator = new AIFToASPICTranslator(prisma,
graphManager);

    // Create test graph
    testGraphId = await graphManager.createGraph('test-
graph');

    // Add nodes: p -> r1 -> q
    const p = await graphManager.addNode(testGraphId,
'I_NODE', 'p');
    const r1 = await graphManager.addNode(testGraphId,
'RA_NODE');
    const q = await graphManager.addNode(testGraphId,
'I_NODE', 'q');

    await graphManager.addEdge(testGraphId, p, r1,
'PREMISE');
    await graphManager.addEdge(testGraphId, r1, q,
'CONCLUSION');
  });

  afterAll(async () => {
    // Cleanup
    await prisma.graph.delete({ where: { id:
testGraphId } });
    await prisma.$disconnect();
  });
```

```
  it('should construct language correctly', async () => {
    const graph = await
graphManager.loadGraph(testGraphId);
    const language = Array.from(graph.nodes.values())
      .filter(n => n.type === 'I_NODE' && n.content)
      .map(n => n.content!);

    expect(language).toContain('p');
    expect(language).toContain('q');
  });

  it('should translate graph to ASPIC+ theory', async () =>
{
    const formsOntology = new Map([
      ['form1', { type: 'ordinary_premise' }],
      ['form2', { type: 'defeasible_inference' }]
    ]);

    const theory = await translator.translate(testGraphId,
formsOntology);


expect(theory.argumentationSystem.language.size).toBeGreate
rThan(0);

expect(theory.knowledgeBase.premises.size).toBeGreaterThan(
0);
  });

  it('should validate graph constraints', async () => {
    const violations = await
graphManager.validateGraph(testGraphId);
    expect(violations).toHaveLength(0);
  });

  it('should reject invalid graphs', async () => {
    // Create invalid graph with RA-node without conclusion
    const invalidGraphId = await
graphManager.createGraph('invalid-graph');
```

```
    const p = await graphManager.addNode(invalidGraphId,
'I_NODE', 'p');
    const r1 = await graphManager.addNode(invalidGraphId,
'RA_NODE');
    await graphManager.addEdge(invalidGraphId, p, r1,
'PREMISE');
    // Missing conclusion edge

    const violations = await
graphManager.validateGraph(invalidGraphId);
    expect(violations.length).toBeGreaterThan(0);
    expect(violations[0]).toContain('must have exactly 1
conclusion');

    // Cleanup
    await prisma.graph.delete({ where: { id: invalidGraphId
} });
  });
});
```

13. CLI Tool (Node.js)

```
// scripts/cli.ts

import { Command } from 'commander';
import { PrismaClient } from '@prisma/client';
import { AIFGraphManager } from '../lib/aif/graph';
import { AIFToASPICTranslator } from '../lib/translation/
aif-to-aspic';
import { ArgumentConstructor } from '../lib/reasoning/
argument-constructor';
import fs from 'fs/promises';

const prisma = new PrismaClient();
const program = new Command();

program
  .name('aif-aspic')
  .description('AIF-ASPIC+ translation toolkit')
  .version('1.0.0');
```

```
program
  .command('translate')
  .description('Translate AIF graph to ASPIC+ theory')
  .argument('<graphId>', 'Graph ID')
  .option('-o, --output <file>', 'Output file path')
  .option('-f, --format <format>', 'Output format (json|
xml)', 'json')
  .action(async (graphId, options) => {
    try {
      const graphManager = new AIFGraphManager(prisma);
      const translator = new AIFToASPICTranslator(prisma,
graphManager);

      console.log(`Translating graph ${graphId}...`);

      const theory = await translator.translate(graphId,
new Map());
      const output = JSON.stringify(theory, null, 2);

      const outputPath = options.output || `theory_$
{graphId}.${options.format}`;
      await fs.writeFile(outputPath, output);

      console.log(`✓ Translated to ${outputPath}`);
    } catch (error) {
      console.error('Error:', (error as Error).message);
      process.exit(1);
    } finally {
      await prisma.$disconnect();
    }
  });

program
  .command('evaluate')
  .description('Evaluate argument acceptability')
  .argument('<theoryId>', 'Theory ID')
  .option('-s, --semantics <type>', 'Semantics (grounded|
preferred|stable)', 'preferred')
  .action(async (theoryId, options) => {
    try {
```

```
      console.log(`Evaluating theory ${theoryId} with ${options.semantics} semantics...`);

      // Load theory and compute extensions
      // Implementation similar to API route

      console.log('✓ Evaluation complete');
    } catch (error) {
      console.error('Error:', (error as Error).message);
      process.exit(1);
    } finally {
      await prisma.$disconnect();
    }
  });

program
  .command('visualize')
  .description('Generate graph visualization')
  .argument('<graphId>', 'Graph ID')
  .option('-o, --output <file>', 'Output file path')
  .option('-f, --format <format>', 'Output format (svg|png|pdf)', 'svg')
  .action(async (graphId, options) => {
    try {
      console.log(`Visualizing graph ${graphId}...`);

      // Implementation would use graphviz or similar

      console.log('✓ Visualization saved');
    } catch (error) {
      console.error('Error:', (error as Error).message);
      process.exit(1);
    } finally {
      await prisma.$disconnect();
    }
  });

program.parse();
```

Usage:

```
# Translate graph
npx tsx scripts/cli.ts translate graph123 -o theory.json

# Evaluate acceptability
npx tsx scripts/cli.ts evaluate theory456 --semantics
preferred

# Visualize graph
npx tsx scripts/cli.ts visualize graph123 -o graph.svg
```

This revised implementation guide now uses:

- **TypeScript** for type safety
- **Prisma** for database schema and ORM
- **Next.js** for API routes
- **React** for visualization components
- **Vitest/Jest** for testing
- **Commander** for CLI tool

All the core algorithms and logic remain the same, just adapted to the TypeScript/Next.js/Supabase stack!