Malmquist Safety Suite: Algorithm 1 PRD

Algorithm 1: UCCA Identification (Directly from thesis p. 100)

```
Algorithm 1: UCCA Identification

Input: A, C^int, S
Output: U^abs, U^ref sets of UCCAs, abstracted & refined (Tuple)
// A: what controller can provide what control action (Tuple)
// C^int: set of interchangeable controllers (Set)
// S: special interactions to consider in refinement (Tuple)

1. C^abs ← Enumerate-Combinations(A) // [Table 4-14]*
2. for x ∈ C^abs**
3.     if Context(C_x^abs) ≠ Ø**
4.         U^abs = U^abs ∪ (C_x^abs, context_x, U_x^rel)**
5. for x ∈ U^abs*
6.     C_x^ref ← Refine-Combinations(U_x^abs, A, S) // [Equation (17)]*
7.     C_x^ref' ← Prune-Equivalent(C_x^ref, C^int) // [Equation (21)]*
8.     U_x^ref ← Prioritize(C_x^ref', S) // [Heuristic, see discussion]*
9. return (U^abs, U^ref)*

// * Step automated; ** Step performed by human
```

Line 1: Automated Enumeration of Control Action Combinations

```
Input: A, C^int, S
Output: U^abs, U^ref sets of UCCAs, abstracted & refined (Tuple)
// A: what controller can provide what control action (Tuple)
// C^int: set of interchangeable controllers (Set)
// S: special interactions to consider in refinement (Tuple)

1. C^abs ← Enumerate-Combinations(A) // [Table 4-14]*
2. for x ∈ C^abs**
3.     if Context(C_x^abs) ≠ ∅**
4.         U^abs = U^abs ∪ (C_x^abs, context_x, U_x^rel)**
5. for x ∈ U^abs*
6.     C_x^ref ← Refine-Combinations(U_x^abs, A, S) // [Equation (17)]*
7.     C_x^ref' ← Prune-Equivalent(C_x^ref, C^int) // [Equation (21)]*
8.     U_x^ref ← Prioritize(C_x^ref', S) // [Heuristic, see discussion]*
9. return (U^abs, U^ref)*
// * Step automated; ** Step performed by human
```

Pseudocode:

```
FUNCTION Enumerate-Combinations(A)
    combinations = ∅
    N = A.Controllers  // vector of controllers
    M = A.ControlActions  // vector of control actions

    // Abstraction 2a Type 1-2
    FOR EACH u_a ∈ M
        FOR EACH u_b ∈ M WHERE b ≠ a
            combinations.add({
                a. ¬u_a ∧ ¬∃u_b
                b. ¬u_a ∧ ∃u_b
                c. u_a ∧ ¬∃u_b
                d. u_a ∧ ∃u_b
            } given U_a(c_N), U_b(c_N))
        END FOR
    END FOR

    // Abstraction 2b Type 1-2
    FOR EACH u_a ∈ M
        FOR EACH c_i ∈ N
            combinations.add({
                a. ¬U_a(c_i) ∧ ¬∃c_j U_a(c_j)
                b. U_a(c_i) ∧ ¬∃c_j U_a(c_j)
                c. U_a(c_i) ∧ ∃c_j U_a(c_j)
            } given i ≠ j ∈ N)
        END FOR
    END FOR

    // Abstraction 2a Type 3-4 (LTL temporal patterns)
    FOR EACH u_a ∈ M
        FOR EACH u_b ∈ M WHERE b ≠ a
            combinations.add({
                a. ∃u_b[(¬u_a ∧ ¬u_b) U (u_a ∧ ¬u_b) F u_b]
                b. ∃u_b[(¬u_a ∧ u_b) U (u_a ∧ u_b) F ¬u_b]
                c. ∃u_b[(u_a ∧ ¬u_b) U (¬u_a ∧ ¬u_b) F u_b]
                d. ∃u_b[(u_a ∧ u_b) U (¬u_a ∧ u_b) F ¬u_b]
                e. ∃u_b[(¬u_a ∧ ¬u_b) U (¬u_a ∧ u_b) F u_a]
                f. ∃u_b[(¬u_a ∧ u_b) U (¬u_a ∧ ¬u_b) F u_a]
                g. ∃u_b[(u_a ∧ ¬u_b) U (u_a ∧ u_b) F ¬u_a]
                h. ∃u_b[(u_a ∧ u_b) U (u_a ∧ ¬u_b) F ¬u_a]
            } given U_a(c_N), U_b(c_N))
        END FOR
    END FOR
```

```
    // Abstraction 2b Type 3-4
    FOR EACH u_a ∈ M
        FOR EACH c_i ∈ N
            combinations.add({
                a. (¬U_a(c_i) ∧ ¬U_a(c_j)) U (U_a(c_i) ∧ ¬U_a(c_j)) F U_a(c_j)
                b. (¬U_a(c_i) ∧ U_a(c_j)) U (U_a(c_i) ∧ U_a(c_j)) F ¬U_a(c_j)
                c. (U_a(c_i) ∧ ¬U_a(c_j)) U (¬U_a(c_i) ∧ ¬U_a(c_j)) F U_a(c_j)
                d. (U_a(c_i) ∧ U_a(c_j)) U (¬U_a(c_i) ∧ U_a(c_j)) F ¬U_a(c_j)
            } given i ≠ j ∈ N)
        END FOR
    END FOR

    RETURN combinations
END FUNCTION
```

Lines 2-4: Human-Identified Context of UCCAs

```
1. Algorithm 1: UCCA Identification
2.
3. Input: A, C^int, S
4. Output: U^abs, U^ref sets of UCCAs, abstracted & refined (Tuple)
5. // A: what controller can provide what control action (Tuple)
6. // C^int: set of interchangeable controllers (Set)
7. // S: special interactions to consider in refinement (Tuple)
8.
9. 1. C^abs ← Enumerate-Combinations(A) // [Table 4-14]*
10. 2. for x ∈ C^abs**
11. 3.     if Context(C_x^abs) ≠ ∅**
12. 4.         U^abs = U^abs ∪ (C_x^abs, context_x, U_x^rel)**
13. 5. for x ∈ U^abs*
14. 6.     C_x^ref ← Refine-Combinations(U_x^abs, A, S) // [Equation (17)]*
15. 7.     C_x^ref' ← Prune-Equivalent(C_x^ref, C^int) // [Equation (21)]*
16. 8.     U_x^ref ← Prioritize(C_x^ref', S) // [Heuristic, see discussion]*
17. 9. return (U^abs, U^ref)*
18.
19. // * Step automated; ** Step performed by human
```

Pseudocode:

Pseudocode(A): Data Structure

```
// Helper function to normalize text for consistent hashing.
FUNCTION NormalizeText(STRING text)
  RETURN text.Trim().ToLower().CollapseWhitespace()
END FUNCTION

// (Hazard, ControlAction, Controller, AbstractedCombination, StructuredContext classes remain the same)

CLASS UnsafeCombination (UCCA)
  STRING UCCA_ID                    // The unique key, potentially salted after a collision.
  AbstractedCombination SourceCombination
  StructuredContext Context
  LIST<Hazard> LinkedHazards
  LIST<ControlAction> RelevantActions
  STRING AnalystID
  DATETIME Timestamp

  // **ENHANCEMENT**: Now accepts an optional salt for collision recovery.
  FUNCTION GenerateUniqueKey(STRING salt = "")
    STRING contextWhen = NormalizeText(Context.When)
    STRING contextWhy = NormalizeText(Context.WhyUnsafe)
    STRING contextConstraint = NormalizeText(Context.ConstraintViolated)

    STRING contextKey = HASH(contextWhen + contextWhy + contextConstraint)
    STRING hazardsKey = GENERATE_KEY_FROM_SORTED_LIST(LinkedHazards.GetIDs())
    STRING actionsKey = GENERATE_KEY_FROM_SORTED_LIST(RelevantActions.GetIDs())

    RETURN HASH(SourceCombination.ID + contextKey + hazardsKey + actionsKey + salt)
  END FUNCTION

  // **ENHANCEMENT**: IsEqualTo now includes the ConstraintViolated field for parity with the key.
  FUNCTION IsEqualTo(UCCA otherUCCA)
    BOOL sourceMatch = (this.SourceCombination.ID == otherUCCA.SourceCombination.ID)
    BOOL contextMatch = (NormalizeText(this.Context.When) == NormalizeText(otherUCCA.Context.When) AND
                NormalizeText(this.Context.WhyUnsafe) == NormalizeText(otherUCCA.Context.WhyUnsafe) AND
                NormalizeText(this.Context.ConstraintViolated) == NormalizeText(otherUCCA.Context.ConstraintViolated))
    BOOL hazardsMatch = ARE_SETS_EQUAL(this.LinkedHazards.GetIDs(), otherUCCA.LinkedHazards.GetIDs())
    BOOL actionsMatch = ARE_SETS_EQUAL(this.RelevantActions.GetIDs(), otherUCCA.RelevantActions.GetIDs())

    RETURN sourceMatch AND contextMatch AND hazardsMatch AND actionsMatch
```

```
    END FUNCTION
END CLASS

// **NEW**: A dedicated result class to make the workflow explicit.
CLASS ReviewResult
    BOOL IsUnsafe, IsSafe, IsDeferred
    LIST<UCCA> UCCAs
    SafeReviewedCombination SafeCombination
END CLASS

```

Pseudocode(B): Identify Unsafe Combinations

```
//=====================================================================
// FUNCTION: IdentifyUnsafeCombinations (Final Version)
// PURPOSE: Orchestrates the complete human-in-the-loop process with all
//          enhancements, including the corrected bulk-triage logic.
//=====================================================================
FUNCTION IdentifyUnsafeCombinations (LIST<AbstractedCombination> potentialCombinations, LIST<Hazard> projectHazards, STRING currentAnalystID)
    // --- Initialize Data Stores ---
    LIST<UnsafeCombination> identifiedUCCAs = NEW LIST()
    LIST<SafeReviewedCombination> safeCombinations = NEW LIST()
    MAP<STRING, UCCA> uccaMap = NEW MAP()
    SET<STRING> reviewedCombinationIDs = NEW SET()

    // --- Main UI Initialization ---
    UI.DisplayUCCAAnalysisDashboard(potentialCombinations, identifiedUCCAs, safeCombinations)

    // --- 1. Bulk Triage Path (Corrected Logic) ---
    // The UI dashboard allows the analyst to multi-select rows and apply a "Mark Selected as Safe" action.
    LIST<AbstractedCombination> combinationsToBulkMarkSafe = UI.GetBulkSafeSelections()

    FOR EACH combo IN combinationsToBulkMarkSafe
        SafeReviewedCombination safeCombo = NEW SafeReviewedCombination()
        safeCombo.SourceCombination = combo
        safeCombo.AnalystID = currentAnalystID
        safeCombo.Timestamp = GetCurrentTimestamp()
        safeCombo.Justification = "Marked safe in bulk triage."

        safeCombinations.Add(safeCombo)

        // **CORRECTION**: Add the combo ID to the reviewed set to prevent it from
        // reappearing for individual review later in this session.
        reviewedCombinationIDs.Add(combo.ID)
    END FOR

    // Refresh the dashboard to reflect the updated status of bulk-reviewed items.
    IF combinationsToBulkMarkSafe IS NOT EMPTY
        UI.RefreshDashboard()
    END IF

    // --- 2. Individual Review Path ---
    // Loop through every potential combination for detailed analysis.
    FOR EACH combination IN potentialCombinations
        // Skip any combination that was already processed via the bulk action.
        IF reviewedCombinationIDs.Contains(combination.ID)
            CONTINUE LOOP
        END IF

        // Prompt the analyst for a detailed review of the specific combination.
        ReviewResult result = UI.PromptForAnalysis(combination, projectHazards, currentAnalystID, uccaMap)

        IF result.IsUnsafe
            // Add any newly defined UCCAs to the master list and the key map.
            FOR EACH ucca IN result.UCCAs
```

```
            identifiedUCCAs.Add(ucca)
            uccaMap.Put(ucca.UCCA_ID, ucca)
        END FOR
        reviewedCombinationIDs.Add(combination.ID)
        UI.UpdateCombinationStatus(combination.ID, "Reviewed - Unsafe")
      ELSE IF result.IsSafe
        // Add the explicitly marked-safe combination to the safe list.
        safeCombinations.Add(result.SafeCombination)
        reviewedCombinationIDs.Add(combination.ID)
        UI.UpdateCombinationStatus(combination.ID, "Reviewed - Safe")
      // If the result is IsDeferred, the combination remains pending, and no status is updated.
      END IF
    END FOR

    RETURN (identifiedUCCAs, safeCombinations)
END FUNCTION

```

Pseudocode(C): UI Screen

```
CLASS UI
  // ... (Dashboard display methods) ...

  STATIC FUNCTION PromptForAnalysis(AbstractedCombination combo, LIST<Hazard> hazards, STRING analystID, MAP<STRING, UCCA> existingUCCAMap)
    LIST<UCCA> identifiedUCCAsForThisCombo = NEW LIST()

    WHILE TRUE
      // ... (Display logic for the combination) ...

      // **ENHANCEMENT**: Clearer button labels for an explicit workflow.
      STRING userAction = ShowButtons(["Add Unsafe Context", "Mark as Safe", "Cancel / Return"])

      IF userAction == "Cancel / Return"
        // **ENHANCEMENT**: Return deferred status to keep item pending.
        RETURN NEW ReviewResult(IsDeferred=TRUE, UCCAs=identifiedUCCAsForThisCombo)

      ELSE IF userAction == "Mark as Safe"
        // ... (Logic for creating and returning a SafeReviewedCombination) ...

      ELSE IF userAction == "Add Unsafe Context"
        // ... (Logic to gather context, hazards, relevantActions, including validation hook) ...

        UCCA tempUCCA = NEW UnsafeCombination()
        // ... (Populate tempUCCA with user input) ...

        STRING newKey = tempUCCA.GenerateUniqueKey()

        // **ENHANCEMENT**: Full De-duplication and Hash Collision Handling.
        IF existingUCCAMap.ContainsKey(newKey)
          UCCA existingUCCA = existingUCCAMap.Get(newKey)

          IF tempUCCA.IsEqualTo(existingUCCA) // Deep comparison
            ShowAlertDialog("Duplicate Context: This exact context already exists.")
            CONTINUE LOOP // Prevent saving duplicate.
          ELSE
            // HASH COLLISION DETECTED
            LOG_WARNING("Hash collision detected on key: " + newKey + ". Attempting auto-recovery with salt.")

            // **ENHANCEMENT**: Auto-recovery attempt.
            STRING saltedKey = tempUCCA.GenerateUniqueKey(salt="_collision_retry_1")

            IF existingUCCAMap.ContainsKey(saltedKey)
              // Collision on the salted key is astronomically rare and requires manual intervention.
              ShowCriticalError("CRITICAL ERROR: Unrecoverable hash collision. Please contact system administrator.")
              LOG_ERROR("Unrecoverable hash collision on salted key: " + saltedKey)
```

```
                            CONTINUE LOOP // Halt this save.
                        ELSE
                            newKey = saltedKey // Use the salted key for this UCCA.
                        END IF
                    END IF
                END IF

                // If we passed the checks, finalize and add the UCCA.
                tempUCCA.UCCA_ID = newKey
                tempUCCA.AnalystID = analystID
                tempUCCA.Timestamp = GetCurrentTimestamp()
                identifiedUCCAsForThisCombo.Add(tempUCCA)

                // Add key to the map immediately to prevent duplicates within the same session.
                existingUCCAMap.Put(newKey, tempUCCA)

                ShowSuccessMessage("Unsafe context added.")
            END IF
        END WHILE
    END FUNCTION
END CLASS

```

Line 5-6: Automated Refinement of UCCAs:

```
Algorithm 1: UCCA Identification

Input: A, C^int, S
Output: U^abs, U^ref sets of UCCAs, abstracted & refined (Tuple)
// A: what controller can provide what control action (Tuple)
// C^int: set of interchangeable controllers (Set)
// S: special interactions to consider in refinement (Tuple)

1. C^abs ← Enumerate-Combinations(A) // [Table 4-14]*
2. for x ∈ C^abs**
3.     if Context(C_x^abs) ≠ Ø**
4.         U^abs = U^abs ∪ (C_x^abs, context_x, U_x^rel)**
5. for x ∈ U^abs*
6.     C_x^ref ← Refine-Combinations(U_x^abs, A, S) // [Equation (17)]*
7.     C_x^ref' ← Prune-Equivalent(C_x^ref, C^int) // [Equation (21)]*
8.     U_x^ref ← Prioritize(C_x^ref', S) // [Heuristic, see discussion]*
9. return (U^abs, U^ref)*

// * Step automated; ** Step performed by human

```

Pseudocode:

```
/*************************************************************************************************
*
* DEFINITIVE DATA STRUCTURE DEFINITIONS
*
*************************************************************************************************/

// Defines the authority mapping for the collaborative system. (Cross-ref: Thesis Alg. 1 Input A)
STRUCTURE AuthorityTuple
   STRING_LIST Controllers
    STRING_LIST ControlActions
    BOOLEAN_MATRIX AuthorityMatrix
END STRUCTURE

// Defines a special interaction rule. (Cross-ref: Thesis Alg. 1 Input S)
STRUCTURE SpecialInteractionRule
    STRING_LIST InputControlActions
    STRING InputVerb
    STRING RuleLogic
    STRING OutputVerb
END STRUCTURE

// Represents a high-level, abstracted UCCA.
STRUCTURE AbstractUCCA
    INTEGER Id
    STRING AbstractionType
    STRING UCCAType
    STRING_LIST AbstractControllers
    STRING_LIST AbstractActions
    STRING_LIST Verbs
    STRING Context
    STRING_LIST RelevantActions
END STRUCTURE

ENUM TemporalVerb { START, END }

// NOTE: AFTER and OVERLAP are included for future expansion and are not used by the current algorithm.
ENUM TemporalRelation { BEFORE, AFTER, OVERLAP }

STRUCTURE TemporalEvent
    STRING Controller
```

```
    STRING Action
    TemporalVerb Verb
    INTEGER SequencePosition
END STRUCTURE

STRUCTURE RefinedUCCA
    STRING Id
    // For T12, this is a MAP<STRING, STRING> (StateCombination).
    // For T34, this is a TEMPORAL_EVENT_LIST (TemporalSequence).
    VARIANT Combination
    TemporalRelation Relation // Populated for T34 UCCAs.
    STRING Context
END STRUCTURE

// ... (Main function RefineAllAbstractUCCAs and the dispatcher RefineSingleUCCA remain unchanged) ...

/****************************************************************************************************
*
* REFINEMENT HELPER FUNCTIONS
*
****************************************************************************************************/

// ... (RefineType12_Abs2a, RefineType12_Abs2b, RefineType34_Abs2a, RefineType34_Abs2b remain unchanged) ...


/****************************************************************************************************
*
* UTILITY FUNCTIONS
*
****************************************************************************************************/

FUNCTION ParseTemporalVerb(STRING verbString) -> TemporalVerb
    // Provides robust, fail-fast parsing of verb strings.
    IF verbString == "starts" THEN RETURN TemporalVerb.START
    ELSE IF verbString == "ends" THEN RETURN TemporalVerb.END
    ELSE
        THROW InvalidTemporalVerbException("Unknown verb: '" + verbString + "'. Must be 'starts' or 'ends'.")
    END IF
END FUNCTION

FUNCTION DetermineUccaCategory(STRING_LIST verbs) -> STRING
    // This helper must be robust to verb order.

    INTEGER providesCount = verbs.count("provides")
    INTEGER notProvidesCount = verbs.count("not_provides")

    // FIX: Added assertion to fail-fast on malformed input.
    ASSERT (providesCount + notProvidesCount) == verbs.size() AND verbs.size() == 2, "DetermineUccaCategory expects a list of exactly two 'provides'/'not_provides' verbs."

    IF providesCount == 0
        RETURN "gap"
    ELSE IF providesCount == 1
        RETURN "mismatch"
    ELSE // providesCount > 1
        RETURN "overlap"
    END IF
END FUNCTION

FUNCTION GenerateAllConcreteCombinations(AuthorityTuple authority) -> ITERABLE<CONCRETE_COMBINATION>
    // Implements the set P from Eq. (17). (Cross-ref: Thesis, pg. 104)
    // NOTE: This should be implemented as a generator/iterator to keep memory usage flat for large systems.
    // The implementation MUST only generate combinations where a "provides" verb is used if
    // authority.AuthorityMatrix allows it, preventing the creation of infeasible states.
    // ...
END FUNCTION
```

```
// ... (Other utility functions like GetCanonicalSignature, etc.) ...

/***************************************************************************************************
*
* SUGGESTED UNIT TESTS
* A comprehensive test harness would validate the algorithm's core properties:
*
* 1. Jam-Interference Rule Test: Confirms that a special interaction rule correctly maps a concrete
* overlap to a collective "gap" for both Abstraction 2a and 2b UCCAs.
*
* 2. RelevantActions Filter Test: A property-based test confirming that adding an irrelevant
* control action to an abstract UCCA's definition has zero impact on the final set of
* refined UCCAs.
*
* 3. Interchangeable Pruning Test: Confirms that for any pair of interchangeable controllers
* (c_i, c_j), a refined combination and its commuted version appear exactly once.
*
* 4. Pruning Fuzz Test: A property-based test that generates random controller groups and
* interchangeable sets, then asserts that no two items in the pruned output map to the
* same canonical signature, ensuring the pruning logic is robust.
*
****************************************************************************************************/
```

Lines 7: Automated Pruning of Additional Equivalent Combinations

```
Algorithm 1: UCCA Identification

Input: A, C^int, S
Output: U^abs, U^ref sets of UCCAs, abstracted & refined (Tuple)
// A: what controller can provide what control action (Tuple)
// C^int: set of interchangeable controllers (Set)
// S: special interactions to consider in refinement (Tuple)

1. C^abs ← Enumerate-Combinations(A) // [Table 4-14]*
2. for x ∈ C^abs**
3.     if Context(C_x^abs) ≠ Ø**
4.         U^abs = U^abs ∪ (C_x^abs, context_x, U_x^rel)**
5. for x ∈ U^abs*
6.     C_x^ref ← Refine-Combinations(U_x^abs, A, S) // [Equation (17)]*
7.     C_x^ref' ← Prune-Equivalent(C_x^ref, C^int) // [Equation (21)]*
8.     U_x^ref ← Prioritize(C_x^ref', S) // [Heuristic, see discussion]*
9. return (U^abs, U^ref)*

// * Step automated; ** Step performed by human

```

Pseudocode:

```
/*****************************************************************************************************
*  MODULE: UCCA_RefineAndPrune
*
*  Implements Algorithm 1 (Chapter 4 of Kopeikin PhD thesis) — Lines 5-7
*  ------------------------------------------------------------------
*    • Line 5   — iterate over each abstract UCCA (done in caller)
*    • Line 6   — RefineCombinations   (Eq. 17)  → Cxref
*    • Line 7   — PruneEquivalentCombinations (Eq. 21) → Cxref'
*
*  Scope
*  -----
*  • Supports Type 1-2 UCCAs (simultaneous "provide / not-provide" cases).
*  • Type 3-4 (temporal permutations) is SAFELY GUARDED with NotImplementedException.
*
*  Complexity
*  ----------
*  • Refinement   : O( 2P ) where P = authorised (controller,action) pairs.
*  • Pruning      : O( M ) where M = |Cxref|  (canonical-signature hash).
*
*  Key symbols (directly from thesis):
*    A   = AuthorityTuple (N, M, A_matrix)
*    S   = set of special-interaction rules (tuple ⟨U, Tin, Σ, Tout⟩)
*    Cint= set of interchangeable-controller groups
*****************************************************************************************************/



/*——————————————————————————— LINE 6 — REFINEMENT ———————————————————————————*/

FUNCTION RefineCombinations(abstractedUcca, authorityTuple, specialInteractions)
    /*
     * Returns Set<RefinedUCCA>  (Cxref_x in thesis)
     * Implements Eq. (17):  ∪ {Pe | Pe′ ≡ Uabs′ }
     */
    primitiveSpace ← GeneratePrimitiveCombinations(authorityTuple,
                                  abstractedUcca.getType())

    refined ← new Set()

    FOR EACH primitive IN primitiveSpace
```

```
        IF abstractedUcca.isType34() THEN
            collective ← CalculateUnionOfEfforts(primitive)              // 𝓢 ignored
        ELSE
            collective ← ApplySpecialInteractions(primitive, specialInteractions)
        END IF

        IF IsEquivalentToAbstractedUCCA(collective, abstractedUcca) THEN
            refined.add(primitive)
        END IF
    END FOR

    RETURN refined
END FUNCTION



/*——————————————————————— Primitive-space generator ———————————————————————*/

FUNCTION GeneratePrimitiveCombinations(authorityTuple, uccaType)
    IF uccaType == TYPE_1_2 THEN
        authorisedPairs ← authorityTuple.getAuthorisedPairs()            // (ci,aj)
        primitiveSet   ← new Set()
        limit          ← 2^(authorisedPairs.length)                      // bitmask loop

        FOR mask = 0 TO limit-1
            pc ← new RefinedUCCA()
            FOR idx = 0 TO authorisedPairs.length-1
                pair       ← authorisedPairs[idx]
                isProvided ← ((mask >> idx) & 1) == 1
                pc.setEffort(pair.controller, pair.action, isProvided)
            END FOR
            primitiveSet.add(pc)
        END FOR

        RETURN primitiveSet                                              // 2P combos

    ELSE IF uccaType == TYPE_3_4 THEN
        THROW new NotImplementedException(
              "Temporal permutation generation for Type 3-4 UCCAs not yet implemented. " +
              "See thesis §4.2.3, Eq.(19-20) for algorithmic requirements.")
    END IF
END FUNCTION



/*——————————————————————— 𝓢-rule evaluator ———————————————————————*/

FUNCTION ApplySpecialInteractions(primitiveCombination, specialInteractionRules)
    /*
     * Each rule S = ⟨U, Tin, Σ, Tout⟩
     * Tin  — input effort pattern
     * Σ    — predicate
     * Tout — rewritten collective output
     */
    collective ← CalculateUnionOfEfforts(primitiveCombination)

    FOR EACH rule IN specialInteractionRules
        IF rule.appliesTo(primitiveCombination) THEN
            collective ← rule.transform(collective)
        END IF
    END FOR
    RETURN collective
END FUNCTION
```

```
/*———————————————————— Abstract-vs-collective test ————————————————*/
FUNCTION IsEquivalentToAbstractedUCCA(collectiveOutput, abstractedUcca)
    relevant ← abstractedUcca.getRelevantActions()
    // Early rejection
    IF NOT collectiveOutput.hasAllActions(relevant) THEN RETURN FALSE
    FOR EACH action IN relevant
        IF abstractedUcca.isProvided(action) ≠ collectiveOutput.isProvided(action) THEN
            RETURN FALSE
        END IF
    END FOR
    RETURN TRUE
END FUNCTION
/*——————————————————— LINE 7 — PRUNING (O(M)) ————————————————————*/
FUNCTION PruneEquivalentCombinations(refinedUccaSet, interchangeableControllerSets)
    controllerToSet ← buildControllerToSetMap(interchangeableControllerSets)
    unique   ← new Set()
    seenHash ← new Set()
    ordered  ← convertSetToSortedList(refinedUccaSet)       // deterministic output
    FOR EACH ucca IN ordered
        sig ← GenerateCanonicalSignature(ucca, controllerToSet)
        IF NOT seenHash.contains(sig) THEN
            seenHash.add(sig)
            unique.add(ucca)
        END IF
    END FOR
    RETURN unique                                    // Cxref'
END FUNCTION
/*————————— Canonical signature  (implements thesis Eq. 21) ——————————*/
FUNCTION GenerateCanonicalSignature(ucca, controllerToSetMap)
    /*
     * Eq. 21 equivalence:
     *   1) swap efforts of any c_i,c_j ∈ same Cint_z
     *   2) others identical
     * → We sort efforts *within* each interchangeable set and
     *   list non-interchangeables verbatim → identical hashes for equivalents.
     */
    nonInt  ← new List()
    intSets ← new Map()      // groupID → List<string>
    FOR EACH ctrl IN ucca.getAllControllers()
        vector ← GetControlEfforts(ucca, ctrl).toString()
        IF controllerToSetMap.hasKey(ctrl) THEN
            gid ← controllerToSetMap.get(ctrl).getID()
            intSets.setdefault(gid, new List()).add(vector)
        ELSE
            nonInt.add(ctrl + ":" + vector)
        END IF
    END FOR
    sort(nonInt)                             // α-order for determinism
    groupStrings ← new List()
    FOR EACH gid IN intSets.getKeys().sort()
        list ← intSets[gid]; sort(list)          // sort efforts in group
```

```
        groupStrings.add("G" + gid + ":" + list.join("|"))
    END FOR

    RETURN nonInt.join(";") + ";" + groupStrings.join(";")
END FUNCTION



/*────────────────────────────── Helper utilities ────────────────────────────────*/

FUNCTION buildControllerToSetMap(interchangeableSets)
    map ← new Map()
    id ← 0
    FOR EACH s IN interchangeableSets
        FOR EACH c IN s
            map[c] = { set: s, id: id }
        END FOR
        id += 1
    END FOR
    RETURN map
END FUNCTION

FUNCTION GetControlEfforts(ucca, controllerID)
    RETURN ucca.efforts[controllerID]        // thin accessor
END FUNCTION
```

Developer Notes & How-To

File Layout

```
src/
└─ ucca/
    ├─ AuthorityTuple.ts     // typedefs & matrix utilities
    ├─ UCCA_Core.ts          // AbstractUCCA, RefinedUCCA classes
    ├─ refine_prune.ts     // **DROP CODE ABOVE HERE**
    └─ tests/
        └─ ucca_table4_15.spec.ts

```

Special Interaction Rule API

```
interface SpecialInteractionRule {
  appliesTo(pc: RefinedUCCA): boolean   // match on Tin + Σ
  transform(out: CollectiveOutput): CollectiveOutput // apply Tout
}

```

See thesis §4.2.2 for the full tuple ⟨U, Tin, Σ, Tout⟩.

Unit Test (Table 4-15)

- Reproduce the seven primitives (4.1–4.7) for “Team ¬TRACK ∧ STRIKE”.
- PruneEquivalentCombinations **must** drop 4.3 & 4.5 only.
- Add to CI; runtime O(2p)=128 combinations in thesis example.

Layer	Worst-case	Notes
GeneratePrimitiveCombinations	2^P	P=#authorised (c,a) pairs; guards quickly for p≤15 in UAV case.

Layer	Worst-case	Notes
ApplySpecialInteractions	O(#rules) per primitive	Usually ≤10.
PruneEquivalentCombinations	O(M)	M=

Extending to Type 3-4

- Replace the NotImplementedException with a permutation generator:
  – Enumerate **temporal events** ⟨controller,action,START/END⟩.
  – Filter permutations against LTL patterns (§4.2.3, Eq. 19-20).
  – Apply same ***S***-bypass rule.

Traceability

All major functions cite the exact thesis artefact they implement:

Function	Thesis reference
RefineCombinations	Eq. 17
GeneratePrimitiveCombinations	§4.2.3 lines 34-46
ApplySpecialInteractions	§4.2.2 tuple ⟨U,Tin,Σ,Tout⟩
PruneEquivalentCombinations	Eq. 21
GenerateCanonicalSignature	Eq. 21, cond. 1-3

Lines 8-9: Automated Prioritization and Output of UCCAs

```
Algorithm 1: UCCA Identification

Input: A, C^int, S
Output: U^abs, U^ref sets of UCCAs, abstracted & refined (Tuple)
// A: what controller can provide what control action (Tuple)
// C^int: set of interchangeable controllers (Set)
// S: special interactions to consider in refinement (Tuple)

1. C^abs ← Enumerate-Combinations(A) // [Table 4-14]*
2. for x ∈ C^abs**
3.     if Context(C_x^abs) ≠ ∅**
4.         U^abs = U^abs ∪ (C_x^abs, context_x, U_x^rel)**
5. for x ∈ U^abs*
6.     C_x^ref ← Refine-Combinations(U_x^abs, A, S) // [Equation (17)]*
7.     C_x^ref' ← Prune-Equivalent(C_x^ref, C^int) // [Equation (21)]*
8.     U_x^ref ← Prioritize(C_x^ref', S) // [Heuristic, see discussion]*
9. return (U^abs, U^ref)*

// * Step automated; ** Step performed by human

```

Pseudocode:

```
// Main algorithm entry point.
ALGORITHM UCCA_Identification(
    IN authority_tuple,
    IN interchangeable_controllers,
    IN special_interactions,
    IN enable_prioritization = TRUE // Optional flag to control prioritization flow.
)
    // ... Lines 1-7 of Algorithm 1 are executed first to produce ...
    // 1. abstracted_uccas: A set of UCCA tuples.
    // 2. refined_and_pruned_uccas_map: A map where each key is an abstracted UCCA and its
    //    value is the corresponding set of refined and pruned UCCAs (C_x_ref').

    // This map holds the final output. The name reflects that its contents
    // may be unordered if prioritization is disabled.
    LET final_refined_uccas_map = NEW MAP()

    // Corresponds to the loop starting at Line 5 of the source algorithm.
    FOR EACH abstracted_ucca IN abstracted_uccas:
        LET C_x_ref_prime = refined_and_pruned_uccas_map.get(abstracted_ucca)
        LET U_x_ref // This will hold the final output structure for this iteration.

        // =================================================================
        // FINAL PSEUDOCODE FOR LINE 8: Automated Prioritization
        // =================================================================
        IF enable_prioritization THEN
            // Prioritization is enabled; call the function.
            U_x_ref = Prioritize(C_x_ref_prime, special_interactions, abstracted_ucca)
        ELSE
            // When disabled, the ordered_view is NULL, indicating original set order.
            U_x_ref = { set: C_x_ref_prime, ordered_view: NULL }
        END IF

        final_refined_uccas_map.put(abstracted_ucca, U_x_ref)

    // =================================================================
    // FINAL PSEUDOCODE FOR LINE 9: Automated Output
    // =================================================================
    RETURN (abstracted_uccas, final_refined_uccas_map)

END ALGORITHM
```

```

//-------------------------------------------------------------------------------
// FINAL PRIORITIZE FUNCTION (ALGORITHM 1, LINE 8)
//-------------------------------------------------------------------------------
FUNCTION Prioritize(
    IN C_x_ref_prime,              // The refined, pruned UCCA set.
    IN special_interactions,       // The set S.
    IN abstracted_ucca             // The parent abstracted UCCA.
)
    // Implements the heuristic from the thesis.
    //
    // RETURNS: A structure containing:
    //  - {set}: The original, unordered set of UCCAs (C_x_ref_prime).
    //  - {ordered_view}: A list of the UCCAs with human-readable priority labels,
    //    sorted for presentation purposes. The ordering does not change the semantic
    //    nature of the output, which is still a set (𝓤ref).

    LET temp_list = NEW LIST()
    LET abstraction_type = abstracted_ucca.type

    FOR EACH ucca IN C_x_ref_prime:
        LET priority_level = 1 // Default: High

        // --- Precedence Rule 1: Special Interactions (S) ---
        // S rules have the highest precedence.
        // If multiple S rules match, choose the rule with the lowest numeric_priority (highest importance).
        LET special_priority = NULL
        FOR EACH interaction IN special_interactions:
            IF DoesInteractionApply(ucca, interaction) THEN
                IF special_priority == NULL OR interaction.priority_level < special_priority THEN
                    special_priority = interaction.priority_level
                END IF
            END IF
        END FOR

        IF special_priority != NULL THEN
            priority_level = special_priority
        ELSE
            // --- Precedence Rule 2: Default Heuristics ---
            // Heuristic 1: Single controller cases are lowest priority.
            IF GetActiveControllerCount(ucca) == 1 THEN
                priority_level = 4 // Lowest
            ELSE
                action_counts = GetActionCounts(ucca)
                // Heuristic 2: Abstraction 2a defaults duplicates (>=2) to low priority.
                // Exceptions must be handled by an S rule to be promoted.
                IF abstraction_type == "Abstraction 2a" AND any count in action_counts >= 2 THEN
                    priority_level = 3 // Low
                // Heuristic 3: Abstraction 2b demotes triplicates (>=3) or more.
                ELSE IF abstraction_type == "Abstraction 2b" AND any count in action_counts >= 3 THEN
                    priority_level = 3 // Low
                END IF
            END IF
        END IF
        ADD {ucca: ucca, numeric_priority: priority_level} TO temp_list

    sorted_list = SortBy(temp_list, "numeric_priority", ASCENDING)

    LET ordered_view_with_labels = NEW LIST()
    FOR EACH item IN sorted_list:
```

```
        ADD {ucca: item.ucca, priority: MapPriorityToString(item.numeric_priority)} TO ordered_view_with_labels
    END FOR

    RETURN { set: C_x_ref_prime, ordered_view: ordered_view_with_labels }

END FUNCTION

//--------------------------------------------------------------------------------
// HELPER FUNCTION for mapping numeric priorities to strings.
//--------------------------------------------------------------------------------
// This mapping makes the output more intuitive for analysts.
// Mapping: 1 -> "High", 2 -> "Medium", 3 -> "Low", 4 -> "Lowest"
FUNCTION MapPriorityToString(IN numeric_priority)
    SWITCH numeric_priority:
        CASE 1: RETURN "High"
        CASE 2: RETURN "Medium"
        CASE 3: RETURN "Low"
        CASE 4: RETURN "Lowest"
        DEFAULT: RETURN "Undefined"
    END SWITCH
END FUNCTION

//--------------------------------------------------------------------------------
// IMPLEMENTATION NOTES
//--------------------------------------------------------------------------------
// **Unit Testing:** A robust test suite is critical for this algorithm.
// 1.  **Optionality Test:** Run the main algorithm twice on the same input, once with
//     `enable_prioritization = TRUE` and once with `FALSE`. Assert that the output `set` is
//     identical in both runs and that `ordered_view` is `NULL` only when the flag is `FALSE`.
// 2.  **Heuristic Validation:**
//     - Use the refined UCCAs from the MUM-T case study's **Table 5-5** as a test fixture.
//     - Create a special interaction rule for the "two fix" case: `IF action_counts['fix'] == 2 THEN priority = 2`.
//     - Assert that UCCA 10.5.4 is correctly assigned "Medium" (Priority 2).
//     - Assert that UCCA 10.5.1 and 10.5.2 are assigned "High" (Priority 1).
//     - Assert that UCCA 10.5.3, 10.5.5, and 10.5.6 are assigned "Low" (Priority 3).
// 3.  **Special Interaction Precedence:** Create a test for a single-controller UCCA. Without an `S`
//     rule, assert its priority is "Lowest". With an `S` rule that promotes it to priority 1, assert
//     its priority is "High".

```