Structured Use Cases OMG Proposal Draft v0.6

Doug Rosenberg – Parallel Agile, Inc.

www.parallelagile.com

Table of Contents

1. Overview

Structured Use Cases is a proposed alternative use case library for SysML v2. The proposal consists of two complementary parts: a Structured Use Case Specification Library and a Structured Use Case Diagram Library.

SysML v2 implements use case modeling through a standard library. Because the language is designed to evolve through its libraries, this proposal takes the form of an alternative use case library rather than a language revision. The intent is to improve usability, interoperability, and standardization while remaining compatible with the overall SysML v2 architecture. The increased emphasis on alternate and exception scenarios promotes more complete discovery of off-nominal requirements while improving behavioral test coverage.

The proposal standardizes structured use case specifications while restoring a simple and natural approach to creating use case diagrams. Together these libraries improve the value-to-pain ratio of use case modeling without requiring changes to the SysML v2 language itself.

2. Background

The modeling philosophy presented in this paper has evolved over several decades through industrial practice, consulting, commercial tool development, teaching, and published research, including several books:

- **Use Case Driven Object Modeling with UML: A Practical Approach** (1999)
- **Applying Use Case Driven Object Modeling with UML: An Annotated eCommerce Example** (2001)
- **Use Case Driven Object Modeling with UML: Theory and Practice** (2007)
- **Design Driven Testing: Test Smarter, Not Harder** (2010)

This proposal extends that body of work by standardizing both structured use case specifications and simplified use case diagrams.

Structured use case specifications have been successfully used in industry for many years. For example, the **Sparx Systems Structured Scenario Editor** (Figure 1), introduced in conjunction with the **Design Driven Testing** book, demonstrated the practical value of organizing behavior into basic, alternate, and exception scenarios composed of individual steps.

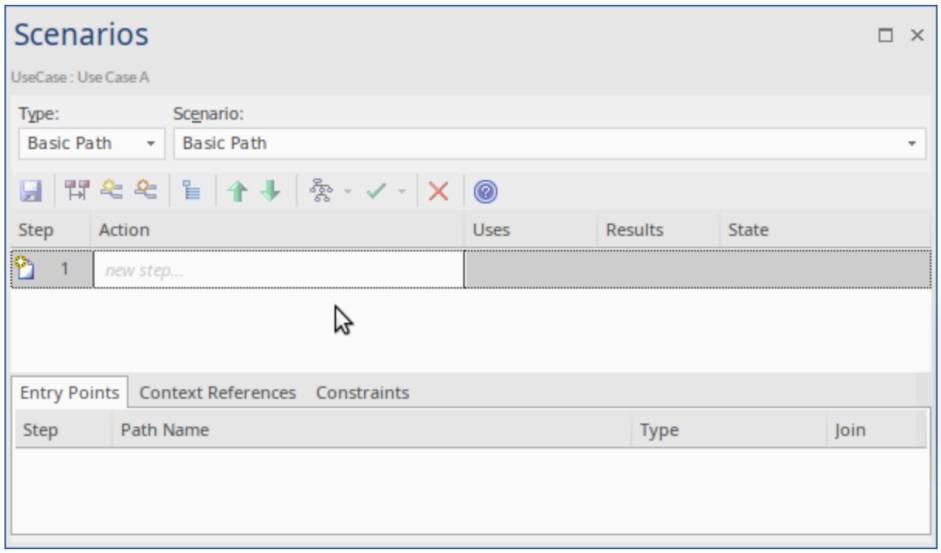


Figure 1 – Use Case Modelers have worked with scenarios for decades

More than fifteen years later, this methodology remains in active industrial use. Expressing use cases as a set of scenarios which consist of steps is much more aligned with current practice than is the existing SysML v2 use case library.

3. Use Case Modeling Philosophy

KEY PRINCIPLE

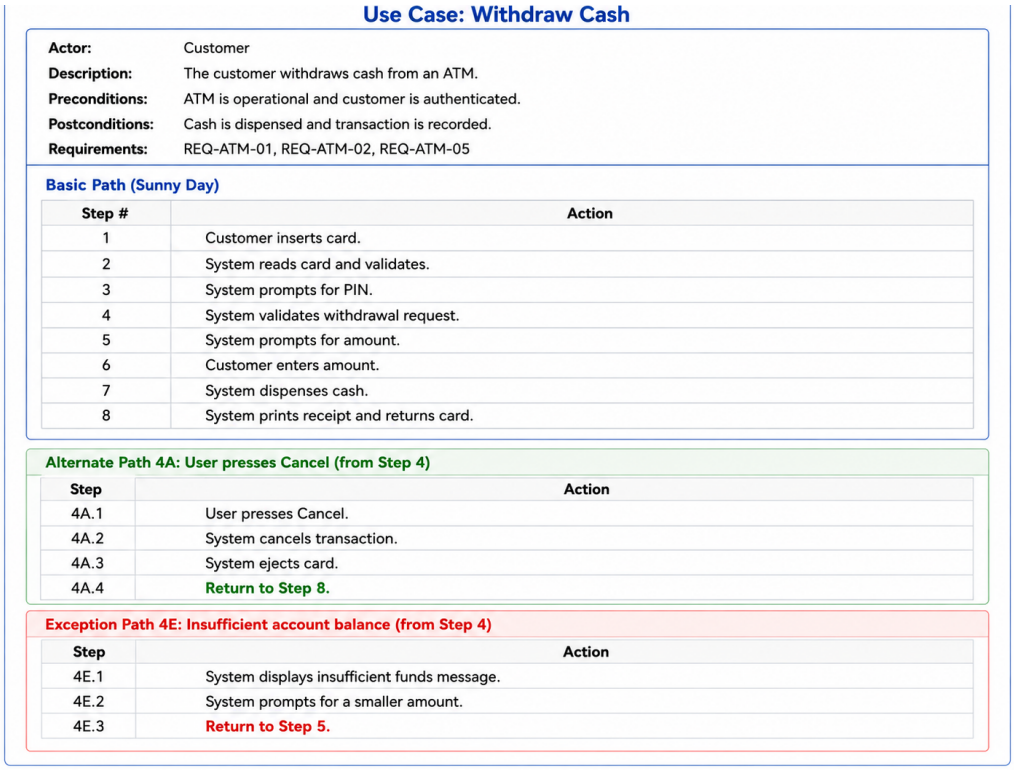
The primary engineering value of use case modeling resides in the use case specifications, not in the use case diagrams.

Use case diagrams provide a simple overview of system behavior by organizing system functionality into individual use cases and showing the relationships among those use cases and their actors. Their primary purpose is to help readers understand the functional organization of a system and navigate quickly to the corresponding use case specifications.

Because every system is different, use case diagrams should provide sufficient flexibility to organize and communicate functionality effectively. They should not be overly restrictive. The detailed semantics belong primarily in the accompanying use case specifications, while the diagram serves as an intuitive organizational and communication aid.

A use case specification captures the behavior associated with a single use case through scenarios and ordered steps.

Structured use case specifications are deliberately human-facing engineering artifacts. They are intended to support conversations with subject matter experts and other stakeholders, not merely to provide a machine-processable behavioral representation for systems engineers. For this reason, the proposed library retains human-readable scenario and step identifiers even though those identifiers do not define execution order. Identifiers such as 4A and 4A.2 make alternate and exception behavior easy to reference during reviews, discussions, documentation, testing, and change management. Authoring tools may automatically assign and renumber these identifiers as scenarios and steps are reorganized.



Use Case: Withdraw Cash

Actor:	Customer
Description:	The customer withdraws cash from an ATM.
Preconditions:	ATM is operational and customer is authenticated.
Postconditions:	Cash is dispensed and transaction is recorded.
Requirements:	REQ-ATM-01, REQ-ATM-02, REQ-ATM-05

Basic Path (Sunny Day)

Step #	Action
1	Customer inserts card.
2	System reads card and validates.
3	System prompts for PIN.
4	System validates withdrawal request.
5	System prompts for amount.
6	Customer enters amount.
7	System dispenses cash.
8	System prints receipt and returns card.

Alternate Path 4A: User presses Cancel (from Step 4)

Step	Action
4A.1	User presses Cancel.
4A.2	System cancels transaction.
4A.3	System ejects card.
4A.4	**Return to Step 8.**

Exception Path 4E: Insufficient account balance (from Step 4)

Step	Action
4E.1	System displays insufficient funds message.
4E.2	System prompts for a smaller amount.
4E.3	**Return to Step 5.**

Figure 2. Example Use Case Specification

To reiterate, the primary purpose of use case modeling is not merely to document known requirements, but to help discover missing requirements. The scenarios in a use case specification, particularly the alternate and exception scenarios, are where many off-nominal requirements are discovered. This is why the specification, rather than the diagram, contains the primary engineering value of the use case model: it is where requirements are discovered, system behavior is defined, and comprehensive behavioral tests can ultimately be derived.

3.1 Relationship Between Structured Scenarios and Activity Diagrams

Structured scenarios and activity diagrams are complementary modeling techniques that serve different purposes during systems engineering. Structured scenarios are particularly effective during requirements elicitation, while activity diagrams excel at refining and communicating the execution of those requirements.

A common misperception is that activity diagrams can serve as a complete replacement for structured use case scenarios. While activity diagrams are excellent for modeling execution behavior, they do not naturally guide analysts through the systematic exploration of alternate and exception behavior that is central to effective requirements elicitation. As a result, important behavioral requirements may never be discovered, even though the notation is fully capable of expressing them.

The difference lies less in the expressive power of the notations than in the way they guide the analyst's thinking.

A structured scenario approach provides more than a documentation format. It guides analysts through a systematic process of behavioral discovery. By explicitly documenting the Basic Scenario together with Alternate and Exception Scenarios, analysts are continually encouraged to ask:

- What must be true for this step to succeed?
- What could prevent this step from succeeding?
- What alternate choices could the actor make?
- How should the system respond when something unexpected occurs?
- Does execution rejoin the Basic Scenario, or does the use case end?

This structured thought process naturally uncovers prerequisite behaviors, validations, alternate outcomes, recovery paths, and exceptional conditions. Considering alternate and exception scenarios frequently improves the quality of the Basic Scenario itself. For example, asking what happens when an ATM customer has insufficient funds naturally reveals that the nominal behavior must first include a step in which the banking system verifies that sufficient funds are available. The exploration of off-nominal behavior often exposes missing requirements in the nominal behavior.

By contrast, when use case behavior is modeled directly as an activity diagram, the analyst's attention naturally gravitates toward the nominal execution path. A straightforward activity diagram progressing from start to finish often appears complete because there is a visible path through the system. Under schedule pressure, teams frequently do not go beyond the primary flow, leaving alternate and exception behavior unexplored or undocumented. The issue is not that activity diagrams cannot represent this behavior—they certainly can—but that the modeling process does not continually encourage analysts to discover it.

As additional alternate paths, exception paths, synchronization, timing constraints, events, and concurrent behavior are incorporated, activity diagrams naturally become larger and more interconnected. This additional detail is valuable once the required behavior has been discovered, but it can make activity diagrams less effective as the primary vehicle for eliciting behavioral requirements.

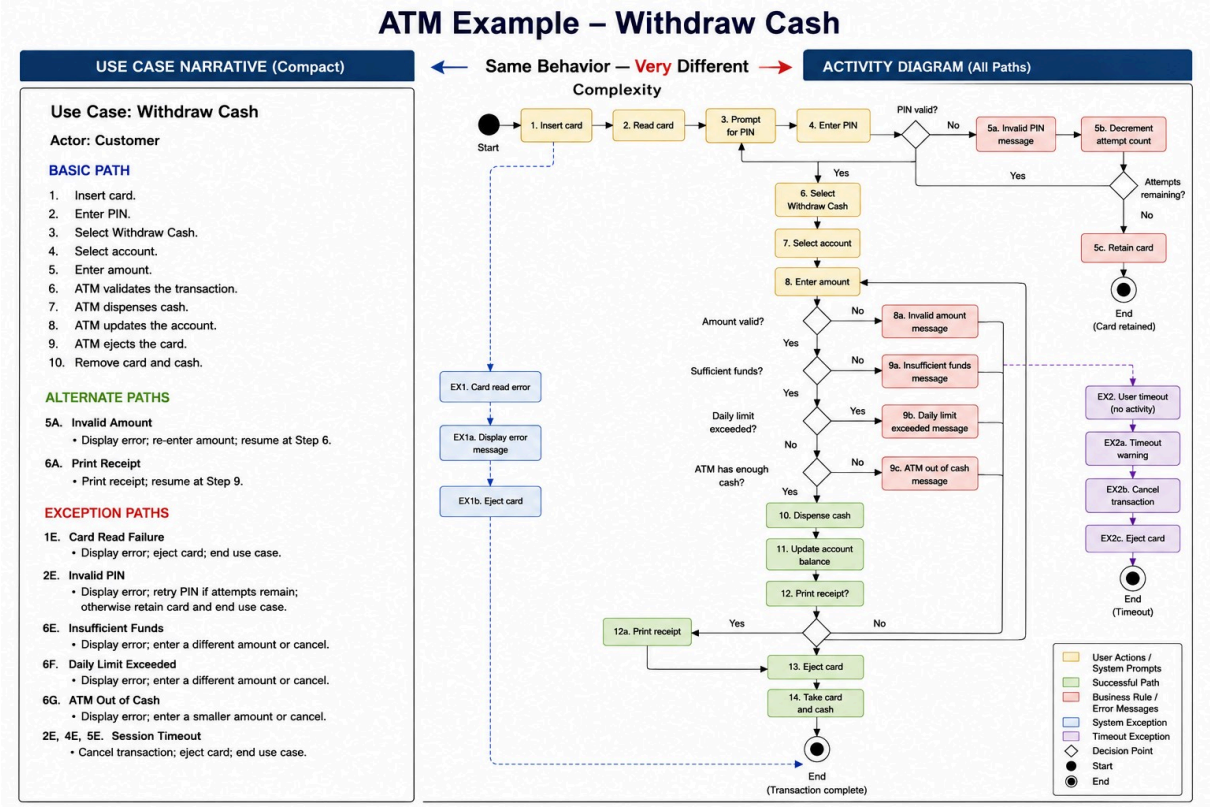


Figure 3 – Activity diagrams become cumbersome when off-nominal behavior is included.

Structured scenarios and activity diagrams therefore serve complementary roles within the modeling process.

Structured scenarios guide the discovery and organization of behavioral requirements by continually focusing attention on off-nominal behavior. Activity diagrams refine and communicate the execution semantics of those requirements, including timing, events, asynchronous behavior, concurrency, synchronization, and control flow.

The proposed library reinforces this distinction while remaining grounded in native SysML v2 behavioral semantics. A structured use case scenario is modeled as a specialization of a SysML Action, and its individual steps are likewise modeled as Actions. Standard SysML successions define the ordering of those steps. This use of native behavioral semantics does not turn the structured specification into an activity diagram. The structured scenario remains a deliberately constrained, narrative-oriented requirements artifact whose organization into basic, alternate, and exception scenarios guides requirements discovery. An activity diagram can subsequently

elaborate that discovered behavior when richer execution semantics such as concurrency, events, timing, or synchronization are needed.

For this reason, this specification treats structured scenarios as the primary mechanism for capturing use case behavior, while recognizing activity diagrams as an important complementary technique for elaborating and communicating the dynamic execution of that behavior.

4. The Structured Use Cases Library

4.1 Purpose and Motivation

Structured use case specifications have been used successfully in systems and software engineering for decades. Although specific templates vary, a common pattern has emerged in industrial practice: a use case describes a goal-oriented interaction through a basic scenario together with alternate and exception scenarios. Each scenario consists of an ordered sequence of steps describing interactions among actors and the system.

The contribution of the Structured Use Case Specification Library is not to invent this approach, but to provide a standard semantic representation for an established engineering practice. The SysML v2 use case library provides standard semantics for use cases, but does not provide a corresponding standard representation for the structured scenarios and steps that make up a detailed use case specification. As a result, structured specifications are commonly represented as text, document fields, or proprietary tool-specific data structures.

This is an important interoperability gap because the structured specification contains much of the engineering value of a use case. In particular, systematic exploration of alternate and exception scenarios helps engineers discover off-nominal requirements that may otherwise be missed. The resulting scenarios also provide a natural basis for behavioral modeling and behavioral testing.

An additional design objective is human readability. Use case specifications are intended to be read and reviewed by subject matter experts and other stakeholders who understand the operational behavior of the system but may not be SysML specialists. The metamodel must therefore support formal machine-readable semantics without sacrificing the simple numbered narrative that has made structured use cases effective as a communication technique.

4.2 Evolution of the Library

An earlier version of the Structured Use Case Specification Library represented concepts such as Use Case, Scenario, and Step as custom structural model elements. This approach demonstrated that structured use cases could be represented formally in SysML v2, but it effectively created a parallel use-case vocabulary on top of the language.

Experience with the reference implementation led to a simpler approach. The revised library uses SysML v2 semantic metadata to extend native SysML concepts rather than replacing them. A structured use case remains a native SysML UseCase. Scenarios and steps remain native SysML Actions. Preconditions and postconditions remain native SysML Constraints. Semantic metadata adds the additional Structured Use Cases meaning needed to identify these elements as structured use cases, scenarios, steps, preconditions, and postconditions.

This approach preserves two complementary views of the same model. A standard SysML v2 tool can understand the underlying elements according to their native SysML semantics, while tools that understand the Structured Use Cases Library can additionally recognize their structured-use-case roles. The result is an extension of SysML v2 rather than a parallel modeling language.

4.3 Core Semantic Concepts

The revised Specification Library is based on three primary concepts:

- Structured Use Case - a specialization of the native SysML UseCase that contains the complete behavioral specification of a use case.
- Scenario - a specialization of the native SysML Action representing one behavioral path through the use case.
- Step - a specialization of the native SysML Action representing one ordered unit within a scenario narrative.

A Step is intentionally not restricted to an action performed by the system. It may describe behavior performed by an actor, behavior performed by the system, an interaction between them, or scenario-control information meaningful to the structured narrative. The relationship between the concepts is straightforward: a Structured Use Case contains Scenarios, and each Scenario contains Steps as subactions.

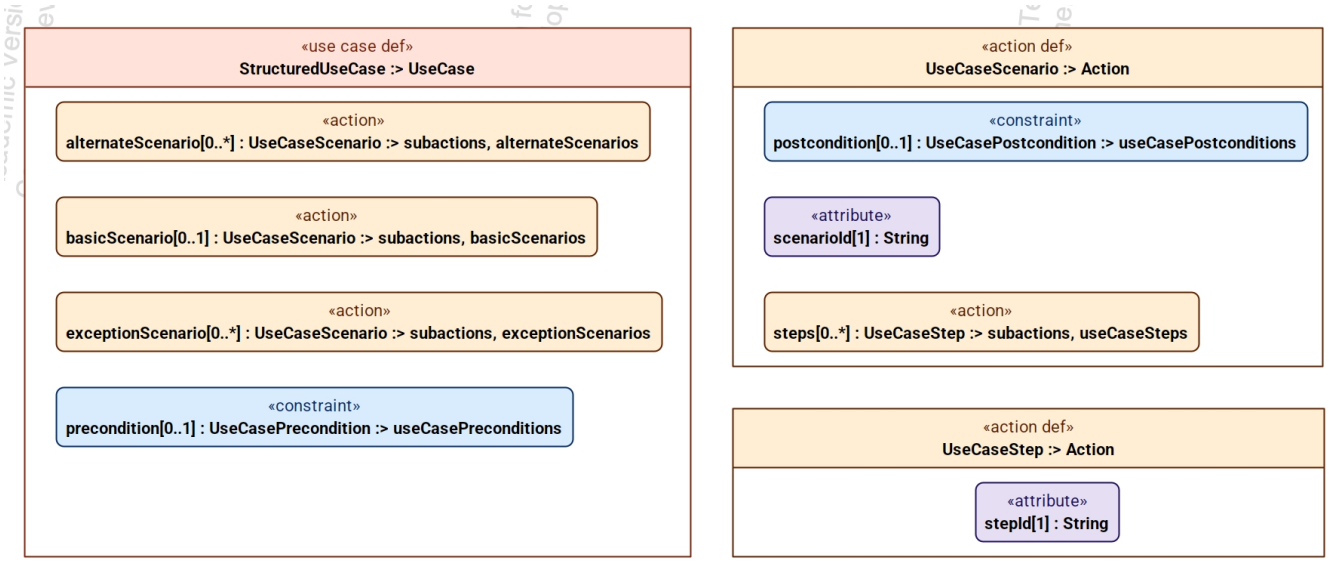


Figure 4. Structured Use Case Specification Library Metamodel

4.4 Basic, Alternate, and Exception Scenarios

The library recognizes three scenario classifications: basic, alternate, and exception. Basic is the nominal or expected behavioral path; alternate is a valid variation from the basic path; exception is an off-nominal path, typically resulting from a failure, error, or other exceptional condition.

These classifications do not require three different Scenario definitions. Basic, alternate, and exception scenarios have the same fundamental structure: each is a Scenario containing an ordered sequence of Steps and an associated Postcondition. The revised library therefore defines a single UseCaseScenario semantic type and uses semantic metadata to classify individual scenarios as #basic, #alternate, or #exception.

This avoids unnecessary duplication in the metamodel while preserving the distinctions that are important to engineers and subject matter experts. The different multiplicities of these classifications are properties of the Structured Use Case rather than properties of three different Scenario types.

4.5 Steps and Behavioral Ordering

A significant change from the earlier library is that a use case Step is modeled as a specialization of the native SysML Action. This creates a direct semantic connection between a structured use case specification and the SysML behavioral model: the Step is already an Action.

The ordering of Steps within a Scenario is represented using standard SysML successions. Step numbers are not used to define behavioral order. This distinction is deliberate: succession provides the machine-readable behavioral semantics, while step identifiers provide the human-readable numbering needed to review and discuss a use case specification.

This separation also allows an authoring tool to reorder steps and automatically renumber the human-readable identifiers without changing the fundamental model identity or behavioral ordering semantics.

4.6 Human-Readable Scenario and Step Identifiers

Structured use cases are collaborative engineering artifacts. During reviews, engineers and subject matter experts routinely refer to specific locations in the specification using identifiers such as “Step 6,” “Scenario 6A,” or “Exception 3E.” For this reason, the library explicitly retains human-readable scenarioId and stepId attributes even though behavioral ordering is represented independently through SysML successions.

For example, an alternate scenario can be presented as “6A Enter Smaller Amount,” with steps “6A.1 Notify customer of insufficient funds,” “6A.2 Prompt for smaller amount,” and “6A.3 Customer enters smaller amount.” The identifiers provide a concise vocabulary for human communication and can be maintained automatically by use case authoring tools.

4.7 Branch and Rejoin Semantics

Alternate and exception scenarios normally originate at a particular point in the basic scenario. The Structured Use Cases convention encodes this relationship in the human-readable Scenario identifier. For example, Scenario 6A represents an alternate scenario associated with Step 6 of the basic scenario, while 3E represents an exception scenario associated with Step 3.

In the SysML behavioral representation, the incoming succession to the first Step of the scenario may also be given a descriptive name such as “Branch after Step 6.” This makes the branch point visible in a behavioral view without introducing an artificial branch Step into the scenario.

When an alternate or exception scenario returns to the basic path, the rejoin semantics can be represented explicitly as the final Step of that scenario, such as “Rejoin before Step 6” or “Rejoin after Step 9.” The distinction is important: rejoin before means that the referenced basic step executes next, while rejoin after means that execution resumes following the referenced step. A terminating scenario simply has no rejoin Step.

This approach provides the information needed by a structured use case without requiring additional merge-node semantics or specialized branch/rejoin reference elements, and keeps the specification immediately understandable to subject matter experts.

4.8 Preconditions and Scenario-Specific Postconditions

A Structured Use Case may define a Precondition describing the condition that must hold before the use case begins. The Precondition applies to the use case as a whole and is represented as a native SysML Constraint with Structured Use Cases semantic metadata.

Each Scenario may define its own Postcondition describing the expected state when that particular behavioral path completes. A single use-case-level postcondition would be insufficient because basic, alternate, and exception scenarios may terminate in different valid system states.

Scenario-specific postconditions also provide a useful foundation for behavioral verification: each scenario represents a behavioral test thread, while its Postcondition defines an expected result that can be evaluated as an assertion.

4.9 Semantic Metadata

The revised library uses SysML v2 semantic metadata to identify the structured-use-case role played by native SysML elements. The principal metadata classifications are #structuredUseCase, #basic, #alternate, #exception, #step, #precondition, and #postcondition.

For example, a Step is fundamentally a native SysML ActionUsage. Applying #step adds the semantic meaning that the Action participates as a Step in a structured use case specification. Likewise, #basic, #alternate, and #exception classify native Action usages according to their roles as scenarios without requiring separate BasicScenario, AlternateScenario, and ExceptionScenario definitions.

This approach preserves native SysML semantics, minimizes redundant metamodel concepts, permits standard SysML tooling to continue recognizing the underlying elements, and provides machine-readable Structured Use Cases semantics for querying, validation, interchange, documentation, and automated processing.

4.10 Library Listing

Listing 1 shows the current Structured Use Cases library used for the implementation tests. In addition to the structured specification semantics, the library now defines a reusable UseCaseActor and #useCaseActor semantic metadata for simple use case diagrams. This Actor is external to individual use cases and may participate in connections to any number of use cases.

```
library package StructuredUseCases {

    doc /*
     * Structured Use Case Specification Library
     * Working Draft v0.2
     *
     * Design principles
     * -----------------
     * This library extends native SysML v2 semantics rather than creating
     * a parallel vocabulary from generic structural elements.
     *
     *   - A StructuredUseCase is a native SysML UseCase.
     *   - A UseCaseScenario is a native SysML Action.
     *   - A UseCaseStep is a native SysML Action.
     *   - A precondition or postcondition is a native SysML Constraint.
     *
     * Semantic metadata adds structured-use-case meaning while preserving
     * the underlying SysML element type.
     *
     * Behavioral order is defined by standard SysML successions.
     * Human-readable scenarioId and stepId values support review,
     * navigation, discussion, documentation, and automatic renumbering
     * by use-case authoring tools.
     *
     * A single precondition applies to the complete use case.
     * Each scenario has its own postcondition because different behavioral
```

```
 * paths may terminate in different valid system states.
 *
 * Branch and rejoin behavior is expressed directly in the scenario
 * narrative as ordinary UseCaseSteps, rather than through separate
 * reference properties.
 */

private import ScalarValues::String;
private import Actions::Action;
private import Actions::actions;
private import Constraints::ConstraintCheck;
private import Constraints::constraintChecks;
private import UseCases::UseCase;
private import UseCases::useCases;
private import Metaobjects::SemanticMetadata;
private import Parts::Part;
private import Parts::parts;


// ================================================================
// Semantic foundation
// ================================================================


// ----------------------------------------------------------------
// Use Case Step
// ----------------------------------------------------------------
//
// A UseCaseStep is the smallest ordered unit in a structured
// scenario narrative.
//
// A step may describe actor behavior, system behavior, or
// scenario-control behavior such as:
//
//   "Branch after Step 6"
//   "Rejoin before Step 6"
//   "Rejoin after Step 9"
//
// It specializes Action so that the step remains a native SysML
// behavioral element.
//
// stepId is human-readable presentation/navigation information.
// It does NOT define behavioral order; successions do that.
//
action def UseCaseStep :> Action {

    attribute stepId : String[1];
}

abstract action useCaseSteps : UseCaseStep[0..*] nonunique
    :> actions;


// ----------------------------------------------------------------
// Scenario Postcondition
// ----------------------------------------------------------------
//
// A scenario-specific expected outcome. Separate postconditions are
// important because nominal, alternate, and exception paths can end
// in different valid states.
//
constraint def UseCasePostcondition :> ConstraintCheck;
```

```
abstract constraint useCasePostconditions :
    UseCasePostcondition[0..*] nonunique
    :> constraintChecks;


// ---------------------------------------------------------------
// Use Case Scenario
// ---------------------------------------------------------------
//
// A UseCaseScenario is one behavioral path through a structured
// use case.
//
// Basic, alternate, and exception scenarios intentionally share this
// single semantic definition. Their distinction is classification,
// not structure.
//
action def UseCaseScenario :> Action {

    // Human-readable scenario identifier, e.g. "B", "6A", "3E".
    attribute scenarioId : String[1];


    // A scenario contains zero or more UseCaseSteps as native SysML
    // subactions.
    action steps : UseCaseStep[0..*]
        :> subactions, useCaseSteps;


    // Each scenario owns its own postcondition.
    constraint postcondition : UseCasePostcondition[0..1]
        :> useCasePostconditions;
}

abstract action useCaseScenarios : UseCaseScenario[0..*] nonunique
    :> actions;


// ---------------------------------------------------------------
// Scenario classifications
// ---------------------------------------------------------------
//
// These abstract usages provide distinct semantic categories for
// metadata binding while retaining one common UseCaseScenario type.
//
abstract action basicScenarios :
    UseCaseScenario[0..*] nonunique
    :> useCaseScenarios;

abstract action alternateScenarios :
    UseCaseScenario[0..*] nonunique
    :> useCaseScenarios;

abstract action exceptionScenarios :
    UseCaseScenario[0..*] nonunique
    :> useCaseScenarios;


// ---------------------------------------------------------------
// Use Case Precondition
// ---------------------------------------------------------------
//
// One precondition applies to the structured use case as a whole.
//
```

```
constraint def UseCasePrecondition :> ConstraintCheck;

abstract constraint useCasePreconditions :
    UseCasePrecondition[0..*] nonunique
    :> constraintChecks;


// ------------------------------------------------------------------
// Structured Use Case
// ------------------------------------------------------------------
//
// StructuredUseCase specializes the native SysML UseCase.
//
// Scenario multiplicities belong here:
//
//   basicScenario       0..1
//   alternateScenario   0..*
//   exceptionScenario   0..*
//
use case def StructuredUseCase :> UseCase {

    action basicScenario : UseCaseScenario[0..1]
        :> subactions, basicScenarios;

    action alternateScenario : UseCaseScenario[0..*]
        :> subactions, alternateScenarios;

    action exceptionScenario : UseCaseScenario[0..*]
        :> subactions, exceptionScenarios;


    // One use-case-level precondition.
    constraint precondition : UseCasePrecondition[0..1]
        :> useCasePreconditions;
}

abstract use case structuredUseCases :
    StructuredUseCase[0..*] nonunique
    :> useCases;




// ==================================================================
// Diagram Actor
// ==================================================================
//
// A Structured Use Cases Actor is a reusable, external participant
// that can be connected to any number of use cases on a use case
// diagram.
//
// This is deliberately NOT the native SysML v2 use-case "actor"
// parameter mechanism. The purpose is to restore the familiar
// first-class diagram actor used in traditional use case modeling.
//
part def UseCaseActor :> Part;

abstract part useCaseActors : UseCaseActor[0..*] nonunique
    :> parts;


// ==================================================================
// Semantic metadata and user-defined keywords
```

```
// ================================================================



// ----------------------------------------------------------------
// #useCaseActor
// ----------------------------------------------------------------
//
// Identifies a reusable external use-case Actor.
//
// The Actor is deliberately represented as a reusable structural
// participant rather than as a native SysML v2 actor parameter
// owned by an individual use case.
//
metadata def <useCaseActor>
    UseCaseActorMetadata :> SemanticMetadata {

    :> annotatedElement : SysML::PartDefinition;
    :> annotatedElement : SysML::PartUsage;
    :>> baseType =
        useCaseActors meta SysML::Usage;
}


metadata def <structuredUseCase>
    StructuredUseCaseMetadata :> SemanticMetadata {

    :> annotatedElement : SysML::UseCaseUsage;
    :>> baseType =
        structuredUseCases meta SysML::UseCaseUsage;
}


metadata def <scenario>
    UseCaseScenarioMetadata :> SemanticMetadata {

    :> annotatedElement : SysML::ActionUsage;
    :>> baseType =
        useCaseScenarios meta SysML::ActionUsage;
}


metadata def <basic>
    BasicScenarioMetadata :> SemanticMetadata {

    :> annotatedElement : SysML::ActionUsage;
    :>> baseType =
        basicScenarios meta SysML::ActionUsage;
}


metadata def <alternate>
    AlternateScenarioMetadata :> SemanticMetadata {

    :> annotatedElement : SysML::ActionUsage;
    :>> baseType =
        alternateScenarios meta SysML::ActionUsage;
}


metadata def <exception>
    ExceptionScenarioMetadata :> SemanticMetadata {
```

```
        :> annotatedElement : SysML::ActionUsage;
        :>> baseType =
            exceptionScenarios meta SysML::ActionUsage;
    }


    metadata def <step>
        UseCaseStepMetadata :> SemanticMetadata {

        :> annotatedElement : SysML::ActionUsage;
        :>> baseType =
            useCaseSteps meta SysML::ActionUsage;
    }


    metadata def <precondition>
        UseCasePreconditionMetadata :> SemanticMetadata {

        :> annotatedElement : SysML::ConstraintUsage;
        :>> baseType =
            useCasePreconditions meta SysML::ConstraintUsage;
    }


    metadata def <postcondition>
        UseCasePostconditionMetadata :> SemanticMetadata {

        :> annotatedElement : SysML::ConstraintUsage;
        :>> baseType =
            useCasePostconditions meta SysML::ConstraintUsage;
    }
}
```

4.11 Example: Withdraw Cash from ATM

The Withdraw Cash from ATM example exercises the revised Specification Library using a basic scenario, alternate scenarios, and an exception scenario. The basic scenario is represented as a sequence of native SysML Actions connected by successions. Alternate and exception scenarios use the same underlying Scenario semantics and are distinguished through semantic metadata.

Human-readable scenario and step identifiers are displayed directly in the specification. Branch information is associated with the incoming succession, while a returning scenario ends with an explicit Step such as “Rejoin before Step 6.” Each scenario has its own Postcondition.

The resulting model can therefore be viewed in two complementary ways. To a SysML tool, it consists largely of familiar Use Cases, Actions, Constraints, and Successions. To a subject matter expert, it remains a conventional structured use case consisting of numbered basic, alternate, and exception scenarios. This combination of native SysML semantics and SME-readable presentation is a primary design objective of the revised library.

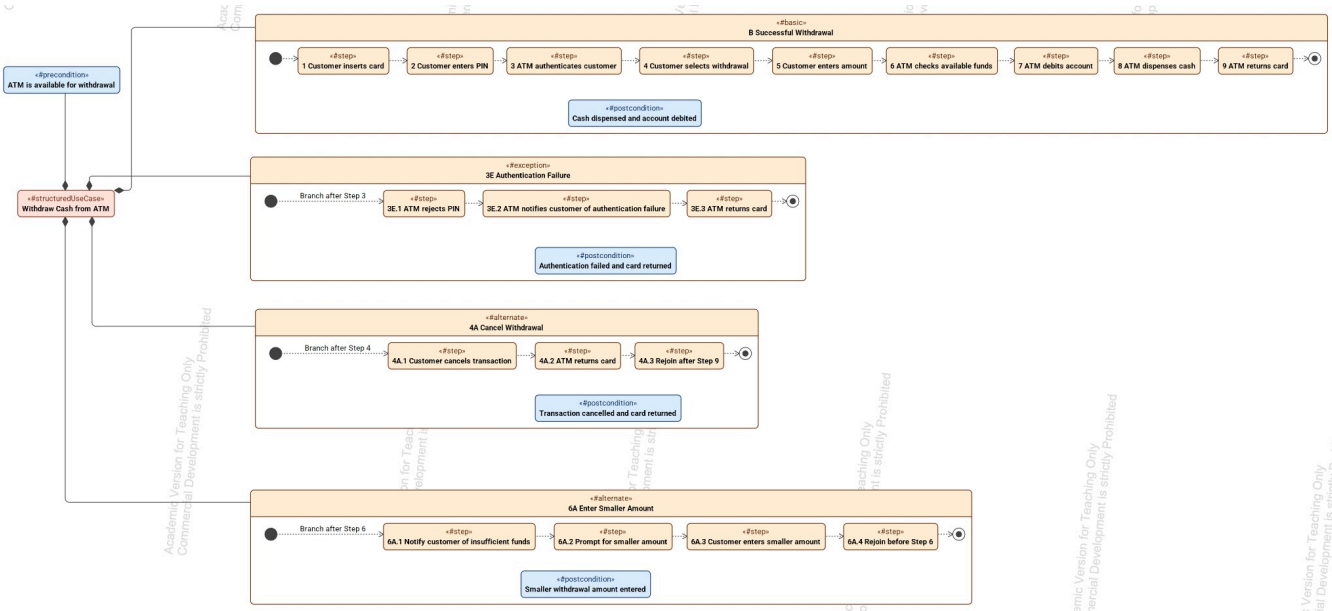


Figure 5. Withdraw Cash from ATM using the revised Structured Use Cases Library

Listing 2. Withdraw Cash from ATM Test Model

```
package WithdrawCashFromATMTest4 {

    private import StructuredUseCases::*;


    // ================================================================
    // Test Case 4: Withdraw Cash from ATM
    #structuredUseCase use case 'Withdraw Cash from ATM' {

        #precondition constraint 'ATM is available for withdrawal';


        // ============================================================
        // Basic Scenario
        // ============================================================

        #basic action 'B Successful Withdrawal' {

            attribute :>> scenarioId = "B";


            #step action '1 Customer inserts card' {
                attribute :>> stepId = "1";
            }

            #step action '2 Customer enters PIN' {
                attribute :>> stepId = "2";
            }

            #step action '3 ATM authenticates customer' {
                attribute :>> stepId = "3";
            }

            #step action '4 Customer selects withdrawal' {
                attribute :>> stepId = "4";
            }
```

```
    #step action '5 Customer enters amount' {
        attribute :>> stepId = "5";
    }

    #step action '6 ATM checks available funds' {
        attribute :>> stepId = "6";
    }

    #step action '7 ATM debits account' {
        attribute :>> stepId = "7";
    }

    #step action '8 ATM dispenses cash' {
        attribute :>> stepId = "8";
    }

    #step action '9 ATM returns card' {
        attribute :>> stepId = "9";
    }


    first start then '1 Customer inserts card';
    first '1 Customer inserts card'
        then '2 Customer enters PIN';
    first '2 Customer enters PIN'
        then '3 ATM authenticates customer';
    first '3 ATM authenticates customer'
        then '4 Customer selects withdrawal';
    first '4 Customer selects withdrawal'
        then '5 Customer enters amount';
    first '5 Customer enters amount'
        then '6 ATM checks available funds';
    first '6 ATM checks available funds'
        then '7 ATM debits account';
    first '7 ATM debits account'
        then '8 ATM dispenses cash';
    first '8 ATM dispenses cash'
        then '9 ATM returns card';
    first '9 ATM returns card' then done;


    #postcondition constraint
        'Cash dispensed and account debited';
}


// ============================================================
// Exception Scenario 3E
// ============================================================

#exception action '3E Authentication Failure' {

    attribute :>> scenarioId = "3E";


    #step action '3E.1 ATM rejects PIN' {
        attribute :>> stepId = "3E.1";
    }

    #step action
        '3E.2 ATM notifies customer of authentication failure' {
        attribute :>> stepId = "3E.2";
```

```
    }

    #step action '3E.3 ATM returns card' {
        attribute :>> stepId = "3E.3";
    }


    succession 'Branch after Step 3'
        first start
        then '3E.1 ATM rejects PIN';

    first '3E.1 ATM rejects PIN'
        then
            '3E.2 ATM notifies customer of authentication failure';

    first
        '3E.2 ATM notifies customer of authentication failure'
        then '3E.3 ATM returns card';

    first '3E.3 ATM returns card' then done;


    #postcondition constraint
        'Authentication failed and card returned';
}


// ============================================================
// Alternate Scenario 4A
// Demonstrates rejoin after
// ============================================================

#alternate action '4A Cancel Withdrawal' {

    attribute :>> scenarioId = "4A";


    #step action '4A.1 Customer cancels transaction' {
        attribute :>> stepId = "4A.1";
    }

    #step action '4A.2 ATM returns card' {
        attribute :>> stepId = "4A.2";
    }

    #step action '4A.3 Rejoin after Step 9' {
        attribute :>> stepId = "4A.3";
    }


    succession 'Branch after Step 4'
        first start
        then '4A.1 Customer cancels transaction';

    first '4A.1 Customer cancels transaction'
        then '4A.2 ATM returns card';

    first '4A.2 ATM returns card'
        then '4A.3 Rejoin after Step 9';

    first '4A.3 Rejoin after Step 9' then done;
```

```
            #postcondition constraint
                'Transaction cancelled and card returned';
        }


        // ================================================================
        // Alternate Scenario 6A
        // Demonstrates rejoin before
        // ================================================================

        #alternate action '6A Enter Smaller Amount' {

            attribute :>> scenarioId = "6A";


            #step action
                '6A.1 Notify customer of insufficient funds' {
                attribute :>> stepId = "6A.1";
            }

            #step action '6A.2 Prompt for smaller amount' {
                attribute :>> stepId = "6A.2";
            }

            #step action '6A.3 Customer enters smaller amount' {
                attribute :>> stepId = "6A.3";
            }

            #step action '6A.4 Rejoin before Step 6' {
                attribute :>> stepId = "6A.4";
            }


                first start
                then '6A.1 Notify customer of insufficient funds';

            first '6A.1 Notify customer of insufficient funds'
                then '6A.2 Prompt for smaller amount';

            first '6A.2 Prompt for smaller amount'
                then '6A.3 Customer enters smaller amount';

            first '6A.3 Customer enters smaller amount'
                then '6A.4 Rejoin before Step 6';

            first '6A.4 Rejoin before Step 6' then done;


            #postcondition constraint
                'Smaller withdrawal amount entered';
        }
    }
}
```

4.12 Less Restrictive Use Case Diagrams

In addition to supporting detailed structured use case specifications, the Structured Use Cases Library supports less restrictive use case diagrams. The intent is to preserve the simple, flexible diagramming practices familiar from traditional use case modeling while retaining the native SysML v2 semantics used by Structured Use Cases.

In particular, a UseCaseActor is a reusable external participant rather than a parameter owned by an individual use case. The #useCaseActor semantic metadata identifies this role while retaining a structural representation that can participate in ordinary connections. Figure 6 demonstrates the intended topology: one Customer actor participates in three Structured Use Cases. The same actor is reused across all three relationships; no actor parameter is declared inside any of the use cases.

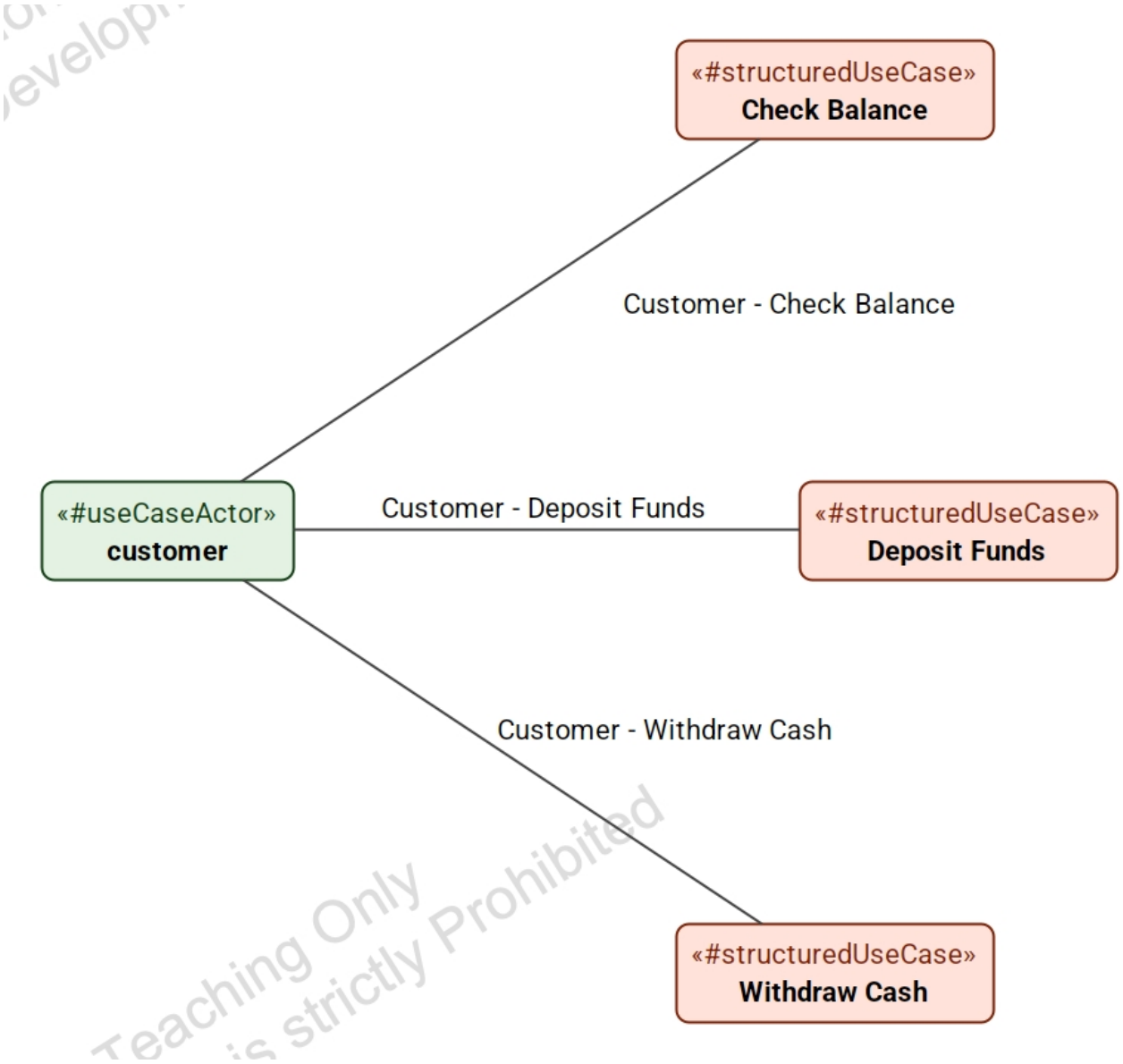


Figure 6. One reusable external Actor connected to multiple Structured Use Cases

Structured Use Cases may also participate in simple use-case-to-use-case connections. Figure 7 shows named connections representing include, extend, invoke, and precede relationships. This approach preserves a lightweight diagram vocabulary while allowing Structured Use Cases to retain their native SysML v2 UseCase specialization and their structured specification semantics.

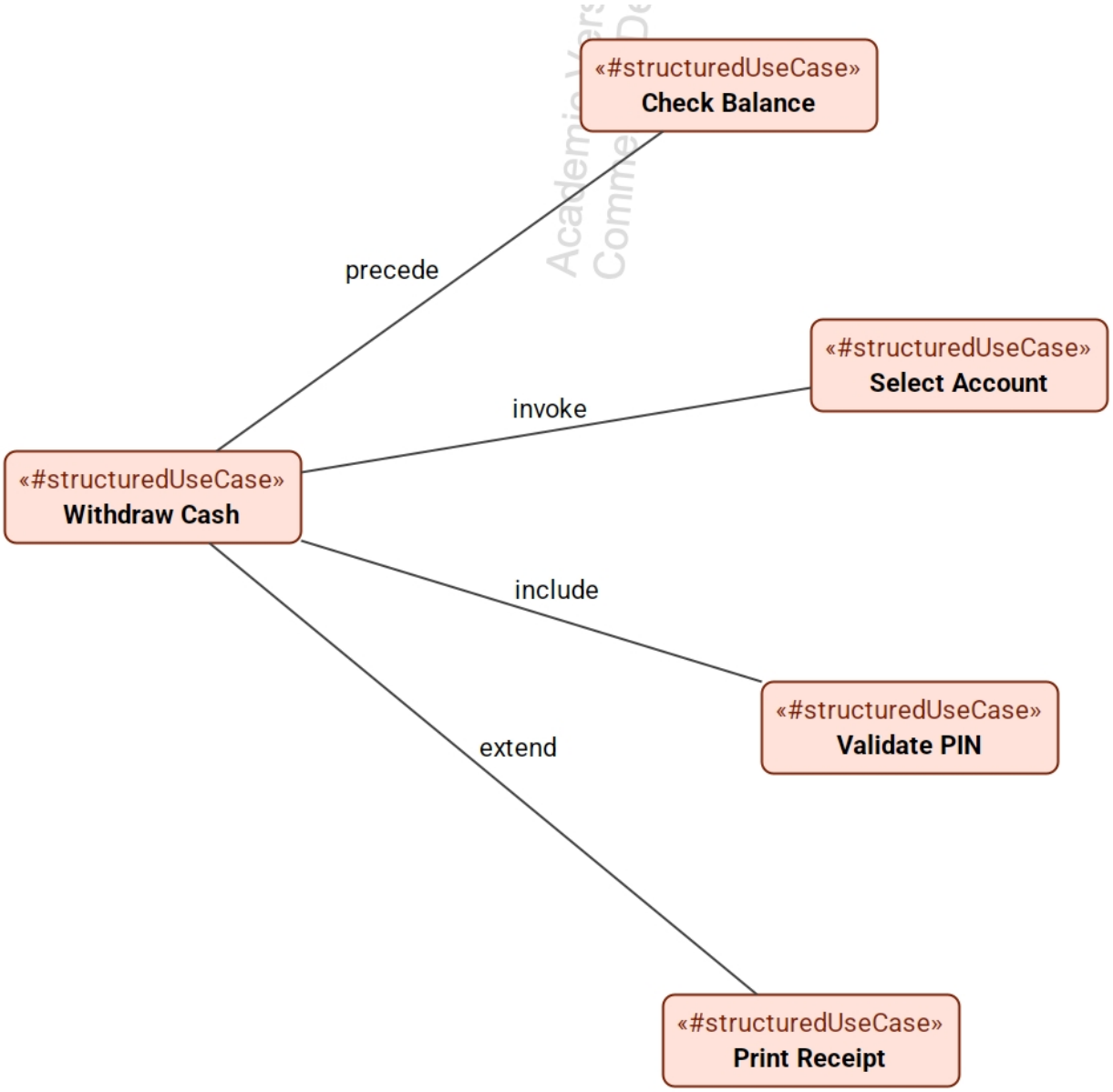


Figure 7. Example use-case-to-use-case connections

These implementation tests demonstrate that specializing the native SysML v2 UseCase does not, by itself, prevent the less restrictive diagram topology proposed here. The specification and diagram concerns can therefore coexist in one library: native UseCase semantics are retained for Structured Use Cases, while reusable external Actors and simple connections provide the diagram behavior needed for stakeholder communication.

4.13 Why Standardize Structured Use Case Specifications?

Structured use case specifications have been used successfully in industry for more than two decades, yet there is currently no standard semantic representation for them. Some modeling tools provide structured scenario editors, while others store specifications as text or proprietary

document fields. Even among tools that support structured scenarios, representations differ, limiting interoperability.

The Structured Use Case Specification Library provides a common semantic foundation without requiring changes to the SysML v2 language. Because the revised library builds on native Use Cases, Actions, Constraints, and Successions, it aligns the structured specification directly with the existing SysML behavioral model.

Standardizing this representation enables structured use cases to participate directly in model interchange, requirements traceability, behavioral modeling, documentation, test generation, and AI-assisted engineering workflows. Most importantly, standardization encourages the engineering practice that motivates this proposal: systematically exploring the alternate and exception scenarios in which many otherwise-missed requirements are discovered.

5. Use Case Modeling Language (UCML)

UCML consists of a textual specification language and a graphical diagram language. The specification captures the structured, human-readable form of a use case, while the diagram provides a concise visual representation suitable for communication among stakeholders. UCML is the reference editor's concise interchange and serialization notation; the normative SysML v2 representation is defined by the Structured Use Cases library described in Section 4.

The UCML representation preserves the same core concepts as the SysML library: a use case has a single use-case-level precondition and contains a basic scenario together with zero or more alternate and exception scenarios; each scenario has a human-readable scenario identifier, an ordered sequence of steps, and an optional scenario-specific postcondition; and each step has a human-readable step identifier. The editor may automatically maintain scenario and step identifiers when the model is edited. Behavioral order is represented by sequence and maps to SysML successions rather than being inferred from the textual form of an identifier.

5.1 Specification Grammar

Conceptual UCML specification grammar

```
UseCaseSpecification
 ::= "use case" Name
     LinkedRequirements*
     LinkedTestCases*
     Precondition?
     BasicScenario
     AlternateScenario*
     ExceptionScenario*

BasicScenario
 ::= "basic" ScenarioIdentifier?
     Step+
     Postcondition?

AlternateScenario
 ::= "alternate" ScenarioIdentifier
     BranchPoint?
     Step+
     Postcondition?

ExceptionScenario
 ::= "exception" ScenarioIdentifier
     BranchPoint?
     Step+
     Postcondition?

Step
 ::= StepIdentifier
     StepText

BranchPoint
 ::= BranchRelation StepReference

BranchRelation
 ::= "branch before"
  |  "branch after"
```

```
Precondition
 ::= Condition

Postcondition
 ::= Condition

StepReference
 ::= StepIdentifier
```

The grammar intentionally does not define a separate branch or rejoin model element. A branch point describes where an alternate or exception scenario leaves the basic scenario and may be rendered as a label on the incoming succession. When a scenario returns to the basic path, a human-readable rejoin instruction such as “Rejoin before Step 6” or “Rejoin after Step 9” may be represented as the final scenario step. This keeps the narrative understandable to subject matter experts without introducing artificial control-flow steps at the beginning of every off-nominal scenario.

5.2 Diagram Grammar

UseCaseDiagram

::= Node*

Association*

Node

::= Actor

| UseCase

Association

::= AssociationType

SourceNode

TargetNode

AssociationType

::= "participates"

| "include"

| "extend"

| "generalization"

| "invokes"

| "precedes"

The following figures illustrate the same Withdraw Cash use case in graphical and specification-editor form. The complete textual UCML listing that follows represents the canonical model produced by the reference editor.

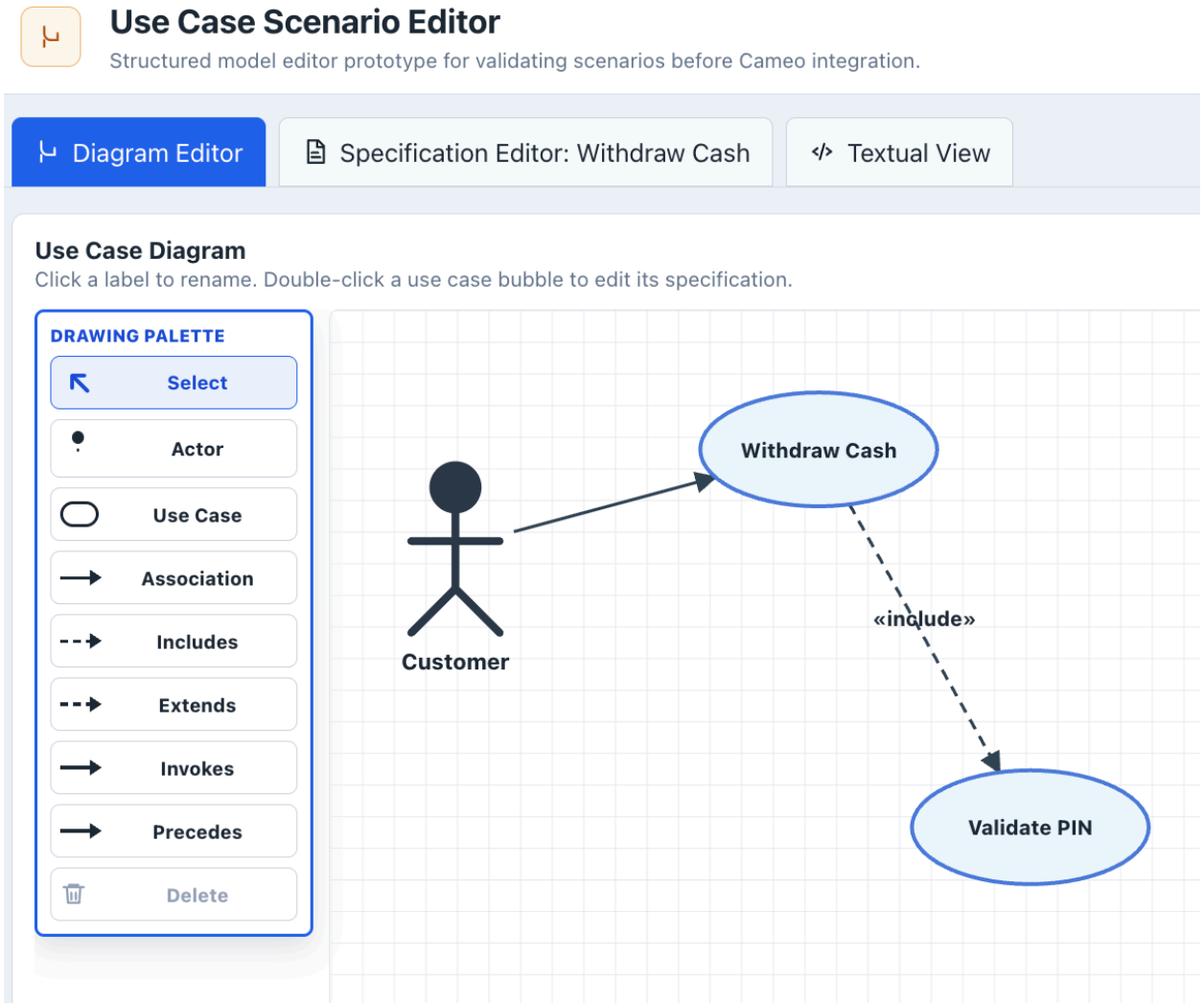


Figure 8. Withdraw Cash Use Case Diagram.

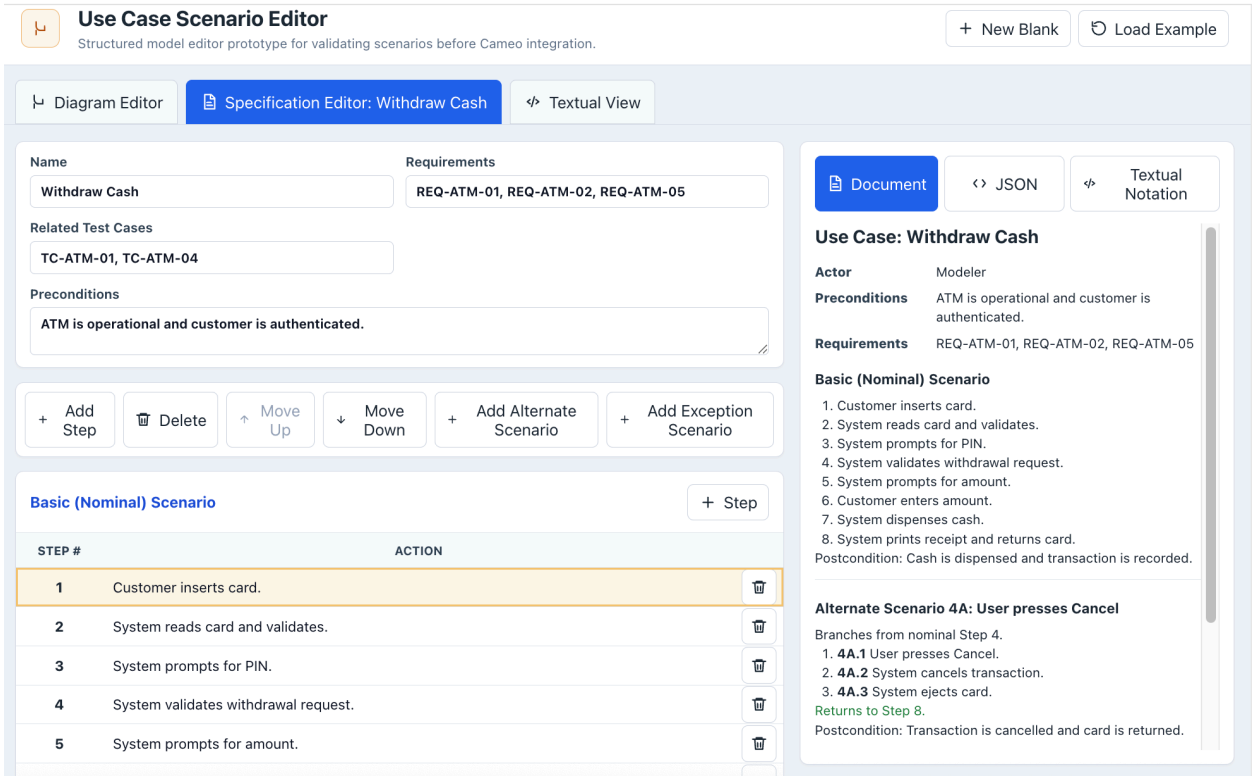


Figure 9. Withdraw Cash Specification Editor.

5.3 Example UCML Model

The following listing preserves the Withdraw Cash example while aligning the UCML representation with the revised Structured Use Cases library. It shows scenario IDs, step IDs, a use-case-level precondition, scenario-specific postconditions, and narrative branch and rejoin behavior.

Updated canonical UCML listing

```
model ATMUseCaseModel {
  actor Customer {
    id: "actor1";
    name: "Customer";
  }

  use case "Withdraw Cash" {
    id: "uc1";
    specification {
      primaryActorId: "actor1";
      preconditions {
        "ATM is available for withdrawal.";
      }

      scenario basic "B" {
        step "1" "Customer inserts card.";
        step "2" "Customer enters PIN.";
        step "3" "ATM authenticates customer.";
        step "4" "Customer selects withdrawal.";
        step "5" "Customer enters amount.";
        step "6" "ATM checks available funds.";
```

```
        step "7" "ATM debits account.";
        step "8" "ATM dispenses cash.";
        step "9" "ATM returns card.";
        postconditions {
          "Cash dispensed and account debited.";
        }
      }

      scenario exception "3E" {
        branch after "3";
        step "3E.1" "ATM rejects PIN.";
        step "3E.2" "ATM notifies customer of authentication failure.";
        step "3E.3" "ATM returns card.";
        postconditions {
          "Authentication failed and card returned.";
        }
      }

      scenario alternate "4A" {
        branch after "4";
        step "4A.1" "Customer cancels transaction.";
        step "4A.2" "ATM returns card.";
        step "4A.3" "Rejoin after Step 9.";
        postconditions {
          "Transaction cancelled and card returned.";
        }
      }

      scenario alternate "6A" {
        branch after "6";
        step "6A.1" "Notify customer of insufficient funds.";
        step "6A.2" "Prompt for smaller amount.";
        step "6A.3" "Customer enters smaller amount.";
        step "6A.4" "Rejoin before Step 6.";
        postconditions {
          "Smaller withdrawal amount entered.";
        }
      }
    }
  }

  association Customer -> "Withdraw Cash";
}
```

6. Why an Alternative Use Case Library?

This proposal has two objectives: to standardize structured use case specifications through the Structured Use Case Specification Library and to provide the Structured Use Case Diagram Library, which restores simplicity, flexibility, and natural communication while encouraging modeling practices that improve requirements discovery, behavioral completeness, systematic verification, and ultimately more robust systems.

The primary motivation for this proposal is the observation that most engineers construct use case models in essentially the same way. In practice, they typically begin by following a simple three-step process:

1. ***Identify an actor.***
2. ***Identify the goals (use cases) associated with that actor.***
3. ***Connect the actor to those use cases to establish the initial use case diagram.***

This straightforward workflow has proven effective for decades because it naturally organizes system functionality around its users.

The current SysML v2 Diagram Library makes this fundamental modeling workflow unreasonably difficult by modeling Actors as parameters of Use Cases. As a result, representing a single actor associated with multiple use cases becomes unnecessarily complex, even though this is how many engineers begin constructing a use case model. **In effect, it breaks the natural use case modeling workflow at its very first step.**

This illustrates a broader design philosophy that appears in several areas of SysML v2: optimizing for formal representation rather than for the people creating and reading the models. The result is a language that is too often *machine-friendly but human-hostile*. While such representations may simplify tool processing, they impose unnecessary complexity on engineers performing common modeling tasks. The current SysML v2 Use Case Library is a vivid illustration of this philosophy.

The Structured Use Case Diagram Library restores this natural modeling style while preserving the simplicity, flexibility, and natural communication that have traditionally made use case diagrams effective engineering tools.

The Structured Use Case Specification Library addresses an even more important objective. The current SysML v2 use case library does not provide a standardized semantic representation for structured use case specifications, even though they contain the primary engineering value of use case modeling. As a result, the language does not currently encourage the systematic exploration of alternate and exception scenarios that has become established industrial practice. Without a standard semantic representation for these scenarios, opportunities to discover off-nominal requirements,

improve behavioral completeness, and derive comprehensive behavioral tests are diminished.

Together, the Structured Use Case Specification Library and the Structured Use Case Diagram Library comprise the Structured Use Cases Library. The Diagram Library restores the intuitive modeling workflow that engineers have used successfully for decades, while the Specification Library standardizes the primary engineering artifact of use case modeling, encouraging more complete requirements discovery through systematic exploration of alternate and exception scenarios and providing a stronger foundation for comprehensive behavioral testing. Together, they provide a practical alternative that standardizes proven engineering practices while improving interoperability, model quality, and the overall effectiveness of use case modeling.

7. Open-Source Reference Editor

To encourage thorough evaluation and rapid adoption of the proposed Structured Use Cases Library, the authors have developed a free-to-use reference editor implementing the Structured Use Case Specification Library and the Structured Use Case Diagram Library described in this proposal. The reference editor will be released as open source software through the OpenMBEE community as part of the Pain-Free Use Cases Project.

The reference editor provides a human-oriented authoring environment for the semantics defined by the Structured Use Cases library. Modelers work with familiar use case concepts—basic, alternate, and exception scenarios; numbered steps; preconditions; and scenario-specific postconditions—while the underlying SysML representation preserves native SysML v2 semantics. Structured use cases map to native UseCase elements, scenarios and steps map to Actions, preconditions and postconditions map to Constraints, and scenario step ordering maps to standard SysML successions. Semantic metadata identifies the structured-use-case roles carried by those native elements.

The editor maintains human-readable scenario and step identifiers as an authoring and communication convenience. When steps are reordered or scenarios are changed, the editor can automatically update the displayed identifiers and dependent narrative references. This allows subject matter experts to work with familiar references such as 6A.2 while keeping behavioral ordering separate from the identifier itself.

The primary purpose of the reference editor is not to compete with commercial MBSE tools, but to provide a practical implementation of the proposed libraries that can be used for experimentation, education, interoperability evaluation, and everyday use. The editor demonstrates that the proposed libraries are practical, implementable, and suitable for real-world use case modeling.

A public landing page for the Structured Use Cases project and hosted reference editor is available at https://www.parallelagile.com/simple-use-cases.html.

Included with the reference editor is the ability to export a textual representation of the underlying metamodel for the current use case model. This notation, called UCML (Use Case Modeling Language), is modeled after the SysML v2 textual notation and provides a concise, human-readable representation of the Structured Use Case Specification Library and the Structured Use Case Diagram Library. Like SysML v2 textual notation, UCML provides a textual view of the underlying model, enabling use case models to be exchanged, version controlled, reviewed, and processed using text-based development workflows.

The reference editor, together with the Structured Use Case Specification Library, the Structured Use Case Diagram Library, and the UCML textual notation, comprise the

technical foundation of the Pain-Free Use Cases Project. The project will be released as open source software through the OpenMBEE community and is described in more detail in the following section.

8. Open Source Project (Pain-Free Use Cases)

The Pain-Free Use Cases Project is an open source initiative intended to encourage experimentation, community participation, and rapid adoption of the proposed Structured Use Cases Library. The project consists of the Structured Use Case Specification Library, the Structured Use Case Diagram Library, the UCML (Use Case Modeling Language) textual notation, and the open source reference editor introduced in the previous section. The project will be released through the OpenMBEE community, providing a collaborative environment in which researchers, practitioners, tool vendors, and OMG participants can evaluate, extend, and refine the proposed libraries.

The current public project landing page is https://www.parallelagile.com/simple-use-cases.html. The page provides a stable entry point for the hosted reference editor and related Parallel Agile training resources while the OpenMBEE project materials are being prepared.

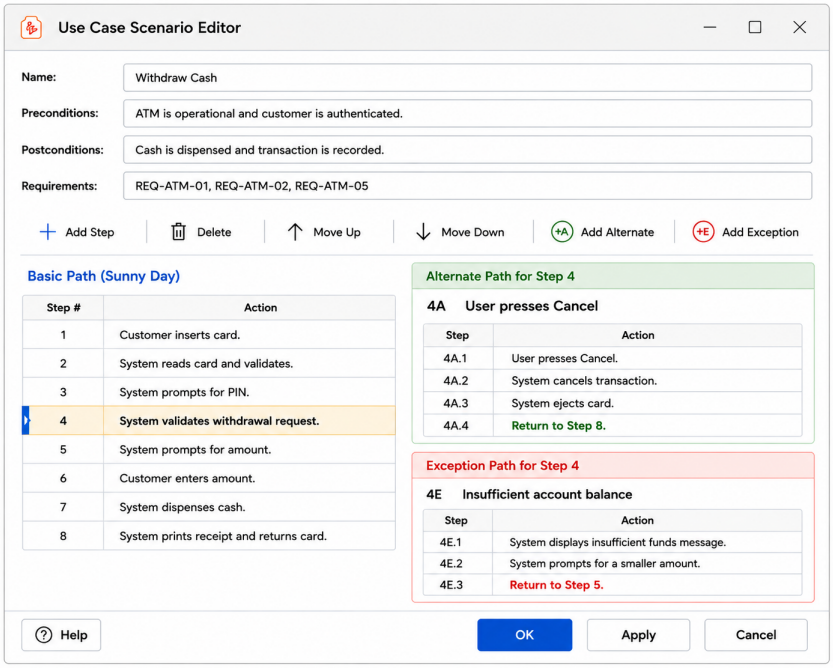


Figure 10: The Open Source editor will be published as part of Pain-Free Use Cases

Figure 10 illustrates the current implementation of the Pain-Free Use Cases reference editor. The editor provides a practical environment for developing structured use case specifications using the proposed metamodel. It supports creation and editing of the basic scenario together with alternate and exception scenarios, encouraging systematic exploration of off-nominal behavior while maintaining the semantic relationships defined by the Structured Use Case Specification Library. The editor also provides navigation among scenarios, making it practical to organize and maintain complete behavioral specifications for systems of realistic size.

The reference editor includes a graphical diagram editor for creating Structured Use Case diagrams and structured use case specifications. It also supports exporting the current use case model using UCML (Use Case Modeling Language), a human-readable textual notation modeled after the SysML v2 textual notation. UCML provides a concise textual representation of the underlying model suitable for review, version control, and interchange. In addition, the editor supports saving complete use case models in JSON format, providing a machine-readable representation for tool integration and automated processing.

The reference editor was developed using AI-Assisted MBSE (AIM) to produce the implementation specification. Initial implementation work was performed using Claude Code, with the production implementation completed using OpenAI Codex from the AIM specification. The resulting editor serves as the reference implementation of the proposed Structured Use Cases Library.

By making the **Pain-Free Use Cases Project** freely available through the **OpenMBEE** community, the authors hope to encourage experimentation and community participation while providing a practical reference implementation for the proposed OMG standard. Together, the proposed OMG standard and the **Pain-Free Use Cases Project** provide both a formal semantic foundation and a practical implementation, accelerating evaluation, adoption, and future evolution of structured use case modeling.

9. Proposed OMG Standardization

This proposal recommends that OMG adopt the Structured Use Cases Library as an alternative standard use case library for SysML v2. Because SysML v2 is designed to evolve through its libraries and extension mechanisms, the proposal requires no change to the SysML v2 grammar. Instead, it defines a compact structured-use-case vocabulary by specializing native SysML v2 concepts and applying semantic metadata to identify their roles.

This is an important architectural characteristic of the proposal. A StructuredUseCase remains a native SysML UseCase. A UseCaseScenario and a UseCaseStep remain native SysML Actions. Preconditions and postconditions remain native SysML Constraints, and behavioral ordering is expressed through standard SysML successions. The proposed metadata adds structured-use-case meaning without replacing the underlying SysML element types. Tools that understand the extension can therefore provide specialized structured-use-case behavior, while other SysML v2 tools can still recognize and process the underlying standard elements.

The Structured Use Cases Library addresses two complementary concerns. Its specification semantics standardize structured use case specifications organized into basic, alternate, and exception scenarios composed of ordered steps. Its diagram semantics provide the simple, flexible use case diagrams traditionally used to organize actors, goals, and relationships among use cases. Together, these capabilities provide a practical alternative use case modeling approach while remaining within the SysML v2 library architecture.

The accompanying Pain-Free Use Cases Project provides a practical reference implementation of the proposed libraries, including the open source reference editor, the UCML textual notation, and supporting software. Together, the proposed OMG standard and the open source project provide both a formal semantic foundation and a practical implementation, encouraging evaluation, adoption, and continued refinement by the broader MBSE community.

The public project landing page at https://www.parallelagile.com/simple-use-cases.html provides a stable URL that OMG reviewers, tool vendors, and community participants can use to access the hosted reference editor during proposal evaluation.

The authors welcome review, discussion, and collaboration from the OMG community and look forward to working with tool vendors, practitioners, and standards participants to refine the proposal and evolve the Structured Use Cases Library into a broadly adopted industry standard.

10. Increasing Behavioral Test Coverage

Although improved behavioral testing is not itself the primary objective of this proposal, it illustrates one of the important engineering benefits of standardizing structured use case specifications. By representing system behavior as organized scenarios composed of ordered steps, the proposed metamodel provides a natural semantic foundation for more systematic behavioral testing.

The Design-Driven Testing book introduced two complementary forms of testing derived directly from structured use case specifications. The first is requirement testing, in which one or more test cases are derived for each requirement to verify that the requirement has been correctly implemented. The second is the Use Case Thread Expander, which systematically expands the basic, alternate, and exception scenarios within a use case specification into a complete set of end-to-end behavioral test threads.

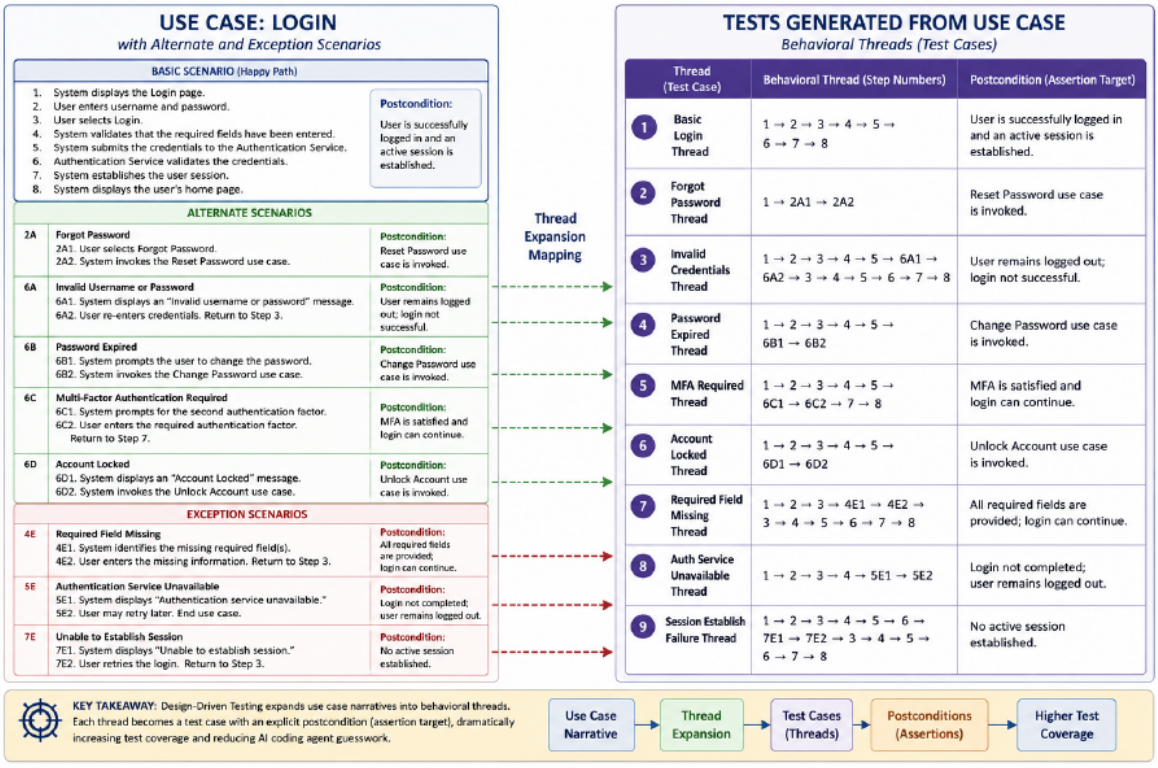


Figure 11 – Expanding use case threads dramatically improves test coverage

The Use Case Thread Expander can be implemented as a query over the structured use case model rather than as a separate manually maintained artifact. Because the Structured Use Cases Library gives explicit semantics to basic, alternate, and exception scenarios, ordered steps, branch and rejoin behavior, scenario identifiers, and scenario-specific postconditions, a tool can

traverse this information to derive the complete behavioral threads represented by a use case specification.

Conceptually, the query begins with the basic scenario and follows its ordered steps. At each point where an alternate or exception scenario branches from the current thread, the query creates the corresponding alternative thread, follows that scenario to its termination or rejoin point, and, when appropriate, resumes traversal of the basic scenario at the specified rejoin location. The process continues until every reachable basic, alternate, and exception path has been expanded into an end-to-end thread.

A conceptual Use Case Thread Expander query can therefore be stated as follows:

```
query expandUseCaseThreads(useCase : StructuredUseCase)
    returns BehavioralThread[*]

    start with the basic scenario

    for each reachable branch point:
        follow the basic continuation
        follow each alternate scenario
        follow each exception scenario

        if a scenario rejoins:
            resume at the specified basic-scenario step
        else:
            terminate that thread

    continue expansion until every reachable path terminates

    for each resulting thread return:
        useCase
        scenarios traversed
        ordered steps
        terminal postcondition
        scenarioId and stepId traceability
```

This pseudocode is conceptual rather than proposed normative SysML query syntax. Its purpose is to illustrate the engineering capability enabled by the library: thread expansion operates on explicit model semantics rather than requiring a test engineer or AI coding agent to infer behavioral paths from unstructured prose. The returned threads can include the scenarios traversed, the ordered steps executed, and the postcondition expected at completion. The human-readable scenarioId and stepId values provide traceability in generated test descriptions.

The effectiveness of both testing approaches depends directly on the completeness of the underlying requirements. By encouraging systematic exploration of alternate and exception scenarios, the proposed metamodel helps engineers discover additional off-nominal requirements that might otherwise be overlooked. Those newly discovered requirements naturally become candidates for additional requirement tests, while the corresponding scenarios become additional behavioral threads. More complete requirements therefore lead directly to more complete behavioral test coverage.

The proposed metamodel directly supports both testing approaches. Each Scenario defines a distinct behavioral path through the system, while the separate Postcondition associated with every Scenario provides the expected result for that behavioral thread. These scenario-specific postconditions naturally support assertion testing, allowing each test case to verify not only that the correct sequence of behavior occurred, but also that the expected outcome of the scenario was achieved.

When Design-Driven Testing was published in 2010, comprehensive requirement testing and use case thread expansion were considered too labor-intensive for routine industrial application. Recent advances in AI-assisted, specification-driven development have dramatically reduced this effort. In the authors' experience, AI coding agents such as Claude Code and OpenAI Codex are particularly effective at generating both assertion tests and use case thread tests from well-structured use case specifications, making these Design-Driven Testing techniques considerably more practical today.

Rather than requiring the expanded behavioral threads to be stored as part of the specification, AI coding agents can execute the thread expansion strategy directly from the structured use case narrative during test generation.

To summarize, the effectiveness of behavioral testing depends on the completeness of the requirements model, and the completeness of the requirements model depends on systematic exploration of alternate and exception scenarios.

11. Queries and Additional Model-Based Analysis

Section 3 of this proposal introduced a key principle:

KEY PRINCIPLE

The primary engineering value of use case modeling resides in the use case specifications, not in the use case diagrams.

The preceding sections have illustrated several reasons for this claim. Structured use case specifications capture system behavior as explicit basic, alternate, and exception scenarios composed of ordered steps. They encourage engineers to explore off-nominal behavior, help expose missing requirements, and provide a foundation for systematic behavioral testing.

An additional benefit emerges when this information is represented as model semantics rather than embedded in unstructured prose: the use case specification becomes queryable engineering information.

This is an important consequence of modeling the detailed behavior of a use case. The objective is not simply to replace a textual use case document with a more formal notation. By representing scenarios, steps, scenario types, ordering, branching and rejoining, preconditions,

postconditions, and relationships as explicit model semantics, engineering tools can ask useful questions about the behavioral model and use the answers to support requirements analysis, model quality, traceability, verification, and automation.

The examples below use simple pseudocode to illustrate the intent of each query. The pseudocode is conceptual and is not proposed as normative SysML v2 query syntax.

11.1 Immediately Useful Queries

The queries described below are not intended to define the limits of what can be done with structured use case models. We have barely begun to explore the analyses that become possible once detailed use case behavior is represented as explicit, machine-readable model semantics. As tools, APIs, AI agents, and engineering workflows begin to operate on these models, many additional applications will undoubtedly emerge.

Nevertheless, several queries appear immediately useful from an engineering perspective. Five examples are:

1. Find use cases with no alternate or exception scenarios. Identify use cases that may not yet have received adequate off-nominal analysis.
2. Extract requirements from off-nominal behavior. Identify alternate and exception scenarios and derive candidate requirements for handling the behavior they describe.
3. Find scenarios without postconditions. Identify behavioral paths whose expected outcome may not yet be sufficiently specified for verification.
4. Expand a use case into its complete set of behavioral threads. Traverse basic, alternate, and exception scenarios to derive the end-to-end behavioral paths that should be considered during testing.
5. Trace requirements through use case behavior to verification. Follow engineering intent from a requirement through the use cases, scenarios, and steps that realize it and onward to the behavioral threads and tests that verify it.

These examples illustrate only some of the immediate possibilities. Their significance is not the sophistication of the individual queries. Rather, they demonstrate the additional engineering value created by modeling the detailed behavior of the use case in the first place. Information that previously existed primarily for human readers becomes available for systematic analysis, automation, verification, and AI-assisted engineering.

11.2 Have We Explored Off-Nominal Behavior?

One of the most valuable practices in structured use case modeling is systematically asking what else can happen besides the basic scenario. Alternate and exception scenarios frequently expose requirements that are missed when analysis focuses only on the nominal path.

A simple model query can identify every use case that contains a basic scenario but no alternate or exception scenarios.

```
query useCasesWithoutOffNominalBehavior

for each structuredUseCase
    if structuredUseCase has a basic scenario
       and has no alternate scenarios
       and has no exception scenarios

        return structuredUseCase
```

The result does not necessarily indicate an error. Some use cases may genuinely have no meaningful off-nominal behavior. Instead, the query identifies use cases that deserve review:

Have we actually considered what could go wrong or what alternative choices could occur here?

This turns an important requirements-analysis discipline into a repeatable model-quality check. Rather than relying entirely on reviewers to notice that a use case contains only a happy path, the model itself can identify those use cases for further consideration.

11.3 What Requirements Can We Extract from Off-Nominal Behavior?

One of the primary engineering benefits of structured use case analysis is requirements discovery. Alternate and exception scenarios describe behavior that is frequently absent from the initial requirements model: alternative choices, validation behavior, error handling, recovery, degraded operation, and other responses to conditions outside the basic scenario.

Once alternate and exception scenarios are represented explicitly in the model, they can be queried to extract candidate requirements for this off-nominal behavior.

For example, an exception scenario named Authentication Failure represents required behavior for handling authentication failure. An alternate scenario named Enter Smaller Amount represents required behavior for allowing the interaction to continue with a smaller withdrawal amount. These behavioral concepts can be used to create candidate requirements for addition to the requirements model.

A conceptual query might therefore be:

```
query extractOffNominalRequirements

for each structuredUseCase
    for each alternate or exception scenario

        derive required behavior from scenario name

        create candidate requirement
            "Handle <required behavior>"

        relate candidate requirement to scenario
        add candidate requirement to requirements model

return candidate requirements
```

For the ATM example, this might produce candidate requirements such as:

Handle Authentication Failure

The system shall handle authentication failure as specified by the Authentication Failure exception scenario of the Withdraw Cash use case.

Handle Enter Smaller Amount

The system shall support entering a smaller amount as specified by the Enter Smaller Amount alternate scenario of the Withdraw Cash use case.

The generated requirement deliberately references the scenario rather than duplicating its detailed behavioral steps. The requirement records that the identified behavior is required, while the structured use case remains the authoritative behavioral specification of how that behavior occurs.

The generated requirement is also based on the semantic identity of the behavior rather than its scenario number. Scenario identifiers such as 3E or 6A are useful for human-readable navigation and may change as the use case evolves. The behavior represented by the scenario—such as Authentication Failure—is the more stable engineering concept and therefore provides the appropriate basis for the requirement.

Explore use case behavior → Discover alternate or exception scenario → Extract candidate requirement → Add requirement to requirements model → Maintain traceability to the scenario

This is more than a model-quality query. It makes one of the traditional benefits of use case modeling explicit and potentially automatable. Instead of relying solely on an engineer to recognize that newly discovered alternate and exception behavior may represent missing requirements, the model can systematically identify that behavior and propose corresponding requirements for engineering review.

11.4 Are Our Scenarios Complete Enough to Verify?

The Structured Use Cases Library associates a separate postcondition with each scenario because different behavioral paths may terminate in different valid system states.

A query can therefore identify scenarios that do not define a postcondition.

```
query scenariosWithoutPostconditions

for each structuredUseCase
    for each scenario in structuredUseCase

        if scenario has no postcondition
            return structuredUseCase,
                   scenario,
                   scenarioId
```

The purpose is not necessarily to impose a rigid modeling rule. Rather, the query identifies scenarios for which the expected outcome may not yet be sufficiently explicit.

This is particularly valuable for verification. If a scenario represents a behavioral path through the system, its postcondition describes the state that should exist when that path completes.

Missing postconditions therefore identify potential gaps in the information needed to construct effective assertion tests.

11.5 What Behavior Must We Test?

Section 10 described the Use Case Thread Expander, which derives complete end-to-end behavioral threads from the basic, alternate, and exception scenarios of a use case.

Thread expansion is itself an example of querying the structured behavioral model. Because scenario types, ordered steps, branch points, rejoin behavior, and postconditions are represented explicitly, a tool can systematically traverse the model and determine the behavioral paths that should be exercised by testing.

The conceptual query introduced in Section 10 can be summarized as:

```
query expandUseCaseThreads(useCase)

start with the basic scenario

for each reachable branch
    follow the basic continuation
    follow each alternate scenario
    follow each exception scenario

    if a scenario rejoins
        resume the originating scenario
        at the specified rejoin point
    else
        terminate that thread

continue until every reachable path terminates

return each complete behavioral thread
       with its ordered steps
       and terminal postcondition
```

The result is not merely a collection of test ideas. It provides a model-derived basis for answering an important verification question:

Have we tested every behavioral thread represented by the use case specification?

This illustrates a broader advantage of structured behavioral semantics: test scope can be derived systematically from the model rather than inferred manually from prose.

11.6 Can We Trace Engineering Intent Through Verification?

Structured use case models can also provide a bridge between requirements and verification.

Where appropriate trace relationships exist, a query can begin with a requirement and identify the use cases, scenarios, and steps that realize or exercise that requirement, followed by the behavioral threads and tests that verify the resulting behavior.

```
query traceRequirementToVerification(requirement)

find steps related to requirement
```

```
for each step
    find owning scenario
    find owning structuredUseCase

    find behavioral threads containing step
    find tests verifying those threads

return requirement,
       structuredUseCase,
       scenario,
       step,
       behavioralThread,
       test
```

Conversely, the same relationships could be traversed in the opposite direction. An engineer reviewing an alternate or exception scenario could ask which requirements justify that behavior and which tests demonstrate that it has been correctly implemented.

Requirement → Use Case → Scenario → Step → Behavioral Thread → Test

The specific trace relationships used by a project may vary, but the important point is that the structured behavioral model provides explicit semantic elements to which those relationships can be attached.

11.7 Engineering Value of Explicit Behavioral Semantics

These examples return to the central claim made in Section 3: the primary engineering value of use case modeling resides in the use case specifications, not in the use case diagrams.

When detailed use case behavior exists only as prose, much of its engineering content remains accessible primarily to human readers. When scenarios, steps, alternatives, exceptions, outcomes, and relationships are represented as explicit model semantics, that information becomes available to engineering tools as well.

The five queries described in this section illustrate some immediate applications. They allow engineers to ask practical questions: Have we explored what can go wrong? What requirements did that exploration uncover? Are our scenarios complete? What behavior needs to be tested? Can we trace that behavior from requirement through verification?

But these five examples barely scratch the surface. Once detailed use case behavior becomes machine-readable engineering information, new forms of analysis can be developed without changing the underlying use case specification. Queries, validation rules, automated transformations, AI agents, test generators, and other engineering tools can all operate on the same behavioral semantics.

This extensibility is itself part of the value proposition for standardizing structured use case semantics. The objective is not merely to provide a better notation for writing use cases. It is to make the engineering information contained in detailed use case specifications available for systematic analysis, automation, verification, and future applications that we have not yet anticipated.

12. Conclusions

This proposal has presented the Structured Use Cases Library as an alternative use case library for SysML v2 consisting of two complementary parts: the Structured Use Case Specification Library and the Structured Use Case Diagram Library. Together, these libraries provide a standardized semantic representation for structured use case specifications while restoring the simplicity, flexibility, and natural communication that have traditionally made use case diagrams effective engineering tools.

The proposal begins from a simple observation: the primary engineering value of use case modeling resides in the use case specifications, not in the diagrams themselves. Use case diagrams are most valuable as organizational and communication aids, while the specifications capture the behavioral semantics of the system. In particular, the systematic exploration of alternate and exception scenarios has long been recognized as one of the most effective techniques for discovering missing off-nominal requirements.

The Structured Use Cases Library standardizes this established industrial practice without creating a parallel behavioral language inside SysML. StructuredUseCase specializes the native SysML UseCase; scenarios and steps specialize native Actions; preconditions and postconditions specialize native Constraints; and standard successions represent behavioral order. Semantic metadata adds the structured-use-case roles needed by specialized tools while preserving the underlying SysML element types. This approach promotes interoperability while giving engineers and subject matter experts the familiar scenario-and-step structure needed for effective requirements discovery.

The Structured Use Case Diagram Library restores the straightforward modeling workflow that engineers have used successfully for decades. Rather than imposing unnecessary restrictions on how use case diagrams are constructed, it enables modelers to organize system functionality naturally around actors and their goals. By making diagrams simple and flexible while placing the engineering emphasis on the accompanying specifications, the proposed library provides a significantly higher value-to-pain ratio for everyday use case modeling.

The proposal is complemented by the Pain-Free Use Cases Project, an open source reference implementation consisting of the proposed libraries, the UCML textual notation, and a free reference editor. Together, the proposed OMG standard and the open source implementation provide both a formal semantic foundation and a practical environment for evaluation, experimentation, and community participation through the OpenMBEE community.

Finally, the proposal demonstrates that standardized structured use case specifications provide benefits extending well beyond documentation. By encouraging systematic

exploration of alternate and exception scenarios, the proposed metamodel promotes more complete requirements discovery while providing a natural semantic foundation for requirement testing, assertion testing, use case thread expansion, and comprehensive behavioral verification. Recent advances in AI-assisted, specification-driven development make these techniques considerably more practical than when they were first introduced, further increasing the value of standardized behavioral specifications.

The authors believe the Structured Use Cases Library represents a practical evolution of use case modeling that reflects established industrial practice while building directly on native SysML v2 semantics. By standardizing structured use case specifications, preserving human-readable scenario and step identifiers, using standard SysML behavioral ordering, restoring a simple and intuitive diagramming approach, and providing an open source reference implementation, the proposal offers the OMG community an opportunity to improve interoperability, increase the value-to-pain ratio of use case modeling, and encourage broader adoption of proven engineering practices.