

Dynamic Model Harness — Describing Business Behavior

2026-10-12
SimpleModeling

SimpleModeling’s Dynamic Model Harness uses the checking and execution infrastructure of Model-Driven Development to constrain software behavior and AI-generated changes. Processing order, state transitions, and failure handling are error-prone even in conventional programming, so defining and executing them as models has value in its own right. AI development adds the need to control generated procedural code that embeds business rules and changes that diverge from requirements. This article explains the harness through checking and direct execution of workflows and state machines in CML and Textus, and through connections between requirements and realization models. This framework makes models serve as programs in the AI era.

1 The Need for a Dynamic Model Harness

Dynamic models offer two benefits: defining and executing behavior as models, and using those rules as a harness for AI generation and changes. Rules for sequence, transitions, and failure handling are collected in a common definition and applied during checking and execution. Connections between requirements and realization models also provide criteria for checking that implementations serve their intended purpose.

1.1 Why Behavior Needs Constraints

Processing order, state transitions, waiting, retries, and compensation are error-prone even in conventional programming. Individual operations can work correctly while an order is confirmed before inventory reservation and payment authorization, or while the normal path continues after failure. Model-Driven Development makes these rules explicit in workflows and state machines that the processing system checks and executes. It is useful whether or not AI is involved.

When AI implements behavior by accumulating locally convenient procedural code, rules become scattered across branches, flags, callbacks, and retry loops. A change that makes one operation work can break the overall sequence, state transitions, or failure rules. In addition to instructions to AI, a harness needs mechanisms that impose constraints on implementation and execution.

SimpleModeling provides workflows and state machines for this harness. Business rules are expressed in CML at an abstraction level close to domain knowledge, and generated or modified models and supplementary implementations are checked. Textus executes the defined workflows and state machines directly, applying model rules to process progression and state transitions.

Definition, checking, and execution together constrain the behavior AI can implement and the paths the runtime can execute.

The starting point is Model-Driven Development: describing software in models that form the basis of development and execution. Rules are defined in CML and checked and executed by the processing system, rather than merely being stored in reference documents or diagrams. This makes them operational constraints beyond instructions or reminders to AI. People judge business meaning in the models; AI produces model proposals and supplementary implementations that follow their rules.

1.2 Connections to the Previous Series and Episode 3

Episode 2 discussed how Model-Driven Development infrastructure functions as a harness for AI. Episode 3's Static Model Harness constrains identity, types, classification, ownership, and relationships. Episode 4 addresses the Dynamic Model Harness, which constrains operation order, state transitions, and progression after failure. It connects the dynamic modeling ideas of the previous series to SimpleModeling's checking, execution, and requirements realization.

Static and dynamic models describe structure and behavior. Domain and application models are distinguished by their purposes and concerns. These are different classification axes, but each model has a center of emphasis: the domain model centers on static models, while the application model centers on workflows originating in use cases.

1.3 Domain and Application Responsibilities

The domain model centers on static models of the concepts, structures, relationships, and rules of the subject domain. In the order example, it describes orders, order lines, products, and their relationships.

The application model starts from use cases and centers on workflows that achieve users' goals. Order confirmation combines domain operations and functions such as inventory reservation, payment authorization, and order confirmation.

1.4 Model Roles and Their Uses

SimpleModeling describes three types of dynamic models in CML: state machines, workflows, and use cases. Workflows and state machines are deductive models that determine process progression and state changes from general rules. Textus interprets and directly executes these CML rules, so the models function as programs.

A Use Case Model is an inductive model describing user goals and concrete scenarios. Although it cannot be executed directly, it is useful as a model of requirements to be realized. It supports example-based requirements checking and model-level checking of connections to corresponding workflows and state machines.

This distinction clarifies the roles: executable models constrain behavior, while their connec-

tions to requirements models help check whether that behavior achieves its purpose. We first explain the definition and execution of workflows and state machines, then their connections to use case models.

2 Defining and Executing Workflows and State Machines

Order confirmation illustrates where behavioral constraints are defined and how execution applies them. We define an order-confirmation workflow on the application side and an order state machine on the domain side. The example is draft order O-101 with one order line for product P-1, a quantity of one, and an amount of JPY 3,000. These identifiers and amounts are illustrative. CML describes progression, permitted state changes, operations, events, and failure handling; Textus applies the rules in execution.

2.1 Workflow Progression

A workflow defines the operations needed to achieve a goal in terms of sequence, branching, parallel processing, waiting, and completion conditions. The order-confirmation workflow is an application model realizing a use case goal. It reserves inventory, authorizes payment, and confirms the order, choosing subsequent steps according to earlier results. CML defines progression rules shared by individual orders, and Textus executes according to those rules.

The application workflow determines how multiple operations progress, while the domain's order state machine determines the object's state changes. The workflow uses operations and transitions permitted by the order state machine. A workflow can also have its own states, such as running or waiting for an external response. Order state and process state are distinguished and connected through operations and events.

2.2 Permitted State Transitions

A state machine describes the states an object can occupy and the transitions permitted in response to events. It connects current state, incoming event, transition condition, destination, and associated action. The model specifies what can be done in each state and under which conditions the next state can be reached.

The order state machine is a dynamic model within the domain model. Its states are draft, confirming, and confirmed. A draft may have zero order lines, but starting confirmation requires at least one. The order reaches confirmed only after inventory reservation and payment authorization succeed, preventing transitions that skip required processing. These are the business rules of this example, and they also apply when the application workflow uses the order.

The model also specifies what happens when no transition applies, when a condition is not met, and when evaluating a condition fails. If several transitions are candidates, the processing system needs selection rules such as exclusivity or priority. Listing state names alone leaves these behaviors unspecified.

2.3 Operations, Events, and Static Structure

Objects handled by workflows and state machines correspond to the entities, values, and relationships of Episode 3. Order O-101's lines, their references to product P-1, quantities, and amounts retain the meanings defined by the static model. Operations describe responsibilities, input and result types, and preconditions and postconditions, connecting model operations to actual objects and contracts.

We distinguish operations or commands requesting processing from events reporting its outcome. A request to confirm an order differs from the fact that the order has been confirmed. An event represents the result after the necessary conditions and processing have been satisfied; the object's state changes accordingly.

2.4 Waiting, Failure, and Compensation

Consider payment being declined after inventory has been reserved for order O-101. The workflow defines a failure branch that releases inventory, and the order state machine prevents transition to confirmed. Compensation performs the business follow-up required after earlier processing; it does not guarantee that every external effect can be undone. Compensation failure and human intervention are also addressed.

When an external service does not respond in time, the system must distinguish failure from success whose response was lost. Models and execution contracts include status queries, waiting deadlines, retry conditions and limits, duplicate-execution protection, and conditions for human handoff. Preventing progression under an assumed success or failure while the outcome remains unknown is another dynamic-model constraint.

2.5 Direct Model Execution in Textus

CML workflows and state machines contain processing and transition rules as deductive models, and Textus executes them directly. Order confirmation applies both the sequence of inventory reservation, payment authorization, and confirmation, and the order's transition conditions. The model becomes a description of functioning software as well as an explanation of its behavior.

Direct execution gives concrete form to the idea that models are programs in the AI era. AI supplements implementation details in a programming language where the model is insufficient, allowing developers to focus on models expressed through domain concepts and rules. The supplementary implementation remains subject to the model's types, operation contracts, and execution conditions.

3 Modeling Connections Between Use Cases and Executable Models

Use cases, the starting point of the application model, describe requirements to be realized. We model their connections to the workflows that realize their goals and scenarios and to the domain operations and state machines those workflows use, then check requirements realization.

3.1 Requirements in Use Cases

A use case model describes users' goals and scenarios as requirements. In the order-confirmation example, the goal is to confirm an order that satisfies inventory and payment conditions. The model describes the draft order, the operation requesting confirmation, and the expected outcome. The use case is mapped to workflows and state machines that realize its requirements rather than being executed directly.

The use case scenario describes requesting inventory reservation after an order-confirmation request, requesting payment authorization after successful reservation, and confirming the order after authorization. Making explicit who requests which operation on which object, and which result is received, identifies required operations and the objects responsible for them.

3.2 Modeling Requirements-to-Realization Connections

Connections between the use case model and executable models are themselves modeled. Starting from the order-confirmation use case, we make explicit how the application workflow realizes the requirement and uses domain operations and the order state machine. Scenario operations map to operations in the model; assumptions and expected outcomes map to object states and transition conditions. Failure and waiting scenarios also receive corresponding branches, states, and outcomes.

Connections from requirements to application workflows and domain state machines, structures, and operations also function as a harness. An executable model can be internally consistent without achieving a user's goal. Checking for paths and outcomes corresponding to requirements, and for consistency with domain rules, detects AI-generated changes that diverge from requirements. Modeling and checking this correspondence supports use case realization, that is, requirements realization. This connection is distinct from Model Up and Model Down.

3.3 Checking Examples Against General Rules

We validate the order-confirmation use case through examples. For the normal path, payment rejection, and waiting for an external response, we check that each scenario's goal, assumptions, outcome, and operation sequence correspond. If an example exposes a missing condition, we identify what must be reflected in the general rules.

We then compare these examples with workflow and state-machine rules. Operation contracts

and entity structures and invariants are also considered to determine whether each example’s inputs and state lead to the expected processing and outcome. A mismatch calls for reconsidering both the example’s assumptions and the general rules. Agreement with finitely many examples does not prove every path correct.

3.4 Comparing an Order Example with Each Model

First we compare the use case scenario for order O-101 with operations and entities. We check that the scenario’s objects exist and have responsibility for the operations, that order, order-line, and product identifiers and relationships are consistent, and that quantities, amounts, and operation inputs and results satisfy types and invariants.

Using the connection model, we compare the same example with the workflow and state machine. We check the mapping of normal operation order and transitions, and confirm that payment rejection has an inventory-release path and cannot confirm the order. Unknown-outcome examples map to waiting and status-query rules. Inconsistencies lead us to revisit requirement assumptions, realization rules, and their connections.

These checks operate on models expressed in CML. Alongside mechanical checks of types, references, inputs, outputs, and process connections, review with `textus-cbd-support` checks correspondence between use cases, static models, workflows, and state machines, feeding into build-time model checking. Use cases serve as a harness through checks on requirements-to-realization connections even though they are not executed directly. AI-assisted semantic comparison is combined with human judgments of validity.

4 A Harness for Checking, Review, and Change

Infrastructure for defining, checking, and executing dynamic models addresses behavioral errors in conventional development and also constrains AI generation and changes. Workflows and state machines constrain execution paths; connections to requirements models check fitness for purpose. Maintaining both supports changes while checking that the software runs and realizes its requirements.

4.1 A Harness Through Checking and Direct Execution

When the processing system interprets model rules and applies them during checking and execution, it constrains what AI can generate and change. Model checks are integrated into the build, and state changes and operations pass through execution boundaries governed by contracts. This Model-Driven Development infrastructure also serves as a harness for AI. Its defining feature is that the processing system applies the rules.

The order O-101 examples used in model-level comparison also carry over to implementation testing. Execution results show whether the normal path reaches confirmed after the required operations, and whether payment rejection releases inventory without confirming the order.

Consistency between models and implementation behavior are treated as separate verification subjects.

4.2 Model Up, Model Down, and Review

Model Up and Model Down, introduced in Episode 2, concern concretizing models toward realization and deriving abstractions suited to a purpose. Connections between use cases, workflows, and state machines express requirements-to-realization correspondence; Model Up and Model Down express abstraction and concretization. Different model types or a mapping between them alone do not constitute Model Down.

Model Up derives abstract models or diagrams from executable workflows or state machines to make business flows easier to review. Selecting detail for the review purpose while preserving correspondence with the source model lets people judge fitness for purpose and business rules, then return findings to the model under development. Here it is used to support review of executable models.

UML-derived interaction or activity diagrams may be produced as reference material when needed. When derived from CML models through review-oriented abstraction, they can express Model Up. An arbitrary reference diagram does not automatically do so. Such diagrams are outside the three dynamic-model types described in CML in SimpleModeling's main development flow and should be used with explicit correspondence to their source models.

4.3 AI Judgment and Established Procedures

If AI selects each next tool, branch, retry, and stopping point at runtime, maintaining compliance requires repeated reasoning and checking. Established procedures can instead be defined in workflows and state machines, while AI interprets user intent, proposes plans, selects authorized workflows, and analyzes exceptions. The runtime controls normal processing according to the accepted model.

Deterministic execution here means making execution decisions from received inputs and events, current state, and explicit rules. It does not mean that external service outcomes or event arrival order are always identical. Execution contracts define input handling, concurrency, and retries, returning failure events and compensation requests to boundaries where AI and people can judge them.

4.4 Workflows as Coherent Operations

A meaningful workflow such as order confirmation can be exposed to an agent as a coherent operation. A contract covering inputs, preconditions, outcomes, and failure meaning lets the agent use it without assembling each internal operation separately. For processes with an established normal path, this can reduce tool selection and repeated inference. The amount saved depends on the process and its configuration.

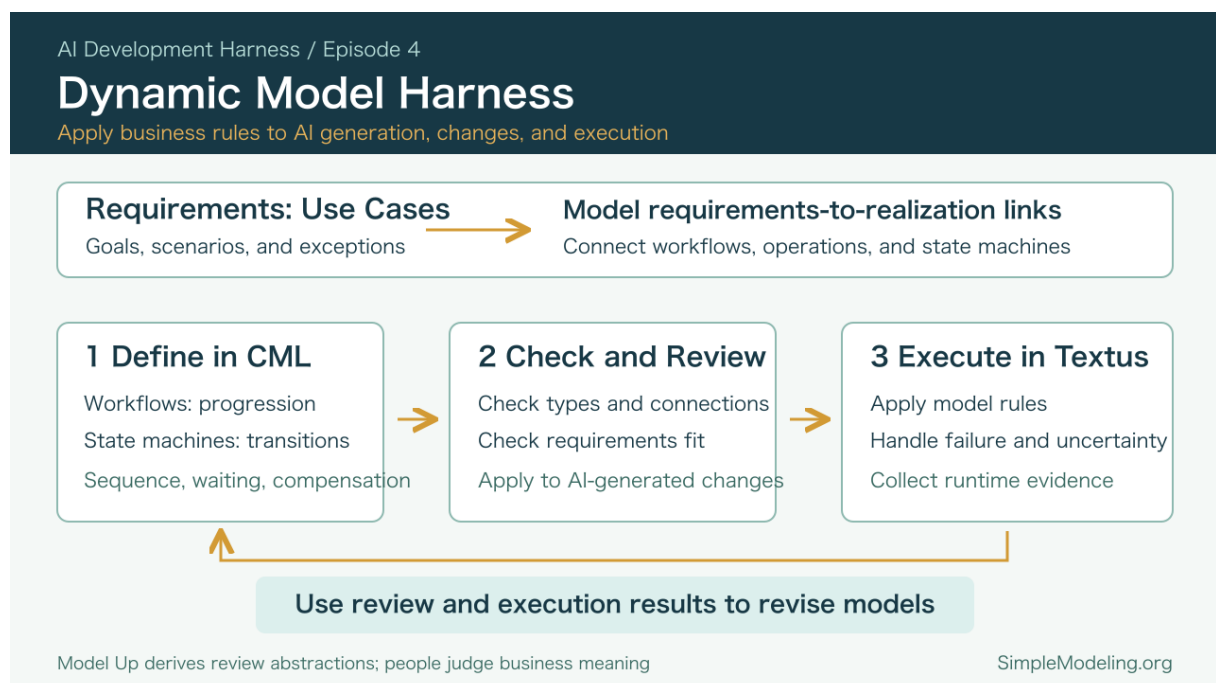
When workflows from dependent components are composed as subworkflows, invocation and result contracts govern their connections. A Capability Model retains correspondence between offered capabilities and their realizations, while a workflow describes how those capabilities are organized. Associated operations, workflows, and state machines perform execution, rather than the capability intermediate representation itself.

4.5 Managing Behavioral Changes Through Models

Behavioral changes begin by checking business meaning in the model and updating corresponding implementations and checks. Changing order-confirmation conditions, for example, requires rechecking the state machine, workflow, and use case examples. People judge permitted operations, failure handling, and human-handoff conditions. Mapping execution records to model operations, events, and transitions allows expected paths to be compared with actual progression. Diagrams are purpose-specific views derived from the same models.

When a model changes during a long-running process, the execution contract must specify which version governs work already in progress. A decision is needed on whether it completes under the existing version or migrates under defined conditions. Editing a model does not automatically assign new meaning to in-progress state. Version management and migration mechanisms belong to runtime responsibilities.

5 Summary and Later Episodes



Model-Driven Development with workflows and state machines makes sequence, transitions, and failure progression explicit, checkable, and executable, even where conventional programming is difficult. In AI development, it also acts as a harness against business-rule violations

caused by local procedural implementations and changes. SimpleModeling constrains behavior through CML model checks and direct execution in Textus, and checks connections from requirements to workflows, state machines, and domain models to support requirements realization. This framework applies Model-Driven Development infrastructure to AI generation and changes.

Episode 5 connects these operations, workflows, and state machines to component boundaries and contracts. Episode 6 covers the structure of supplementary programs, and Episode 7 covers the runtime that applies model rules. Episode 8's Knowledge Harness, Episode 9's Observed Model, and Episode 10's Verification Harness extend the series into shared meaning, execution results, and validation of changes.