

EF BUILDER

User's Guide

Designing DEVS Experimental Frames with the EFSUT Methodology

Concepts, Workflow, and Usage for the ef-builder Skill

Created and Edited by Luiza Cintra and Valdemar Neto

Table of Contents

I. Why Experimental Frames Matter	4
A. What This Guide Covers	4
B. A Note on How to Read the Verbatim Boxes	4
II. Core Concepts You Need First	6
A. The Experimental Frame	6
B. The EF Architecture: Generator, SUT, Transducer	6
C. Five Core EFSUT Principles	7
D. The Derivability Relation and EF Partial Order	7
E. Best-Model Table and Applicability	8
III. Getting Started: The Conceptual Phase	10
A. Step 0: System Intake	10
B. C1: Categorizing Stakeholder Questions	11
C. C2: Generating Candidate Frames	11
Scenario design	12
D. C3: Partial Order, Best-Model Table, and the EFSUT Table	12
Establishing derivability and drawing the partial order	12
The formal derivability relation	13
The Best-Model Table	13
The Stakeholder Requirements EFSUT Table	13
E. C4 & C5: Presenting Frames and Getting Your Approval	14
F. Conceptual Verification: The Maximal EF Check	14
IV. From Approved Frame to Java Code: The Developmental Phase	16
A. D1: Designing the Java Class Structure	16
B. D2 – D4: Generating the Generator, Transducer, System Model, and Coupled EF	16
C. D5: Verify and Write the Java Files	17
Structural verification (Step D5.1)	17
EFSUT consistency verification (Step D5.2)	17
D. Developmental Verification: The Maximal EF Cross-Check	18
V. Two Patterns Worth Understanding Before You Read the Code	19
A. The Pending-Emit Pattern	19
B. Stop Condition Enforcement	20
VI. Output Files Reference	22
A. File-by-File Guide	22
VII. Troubleshooting	24
Candidate frame specifies ports not present on the system	24
Partial order has no derivability edges	24
Best-model table maps all frames to the same model	24
Generator output ports do not match System Model input ports	24

System Model missing accessor method called by an EF	24
Pending emit values lost in internalTransition	24
Observation window too short for scenario count	25
Missing DEVS methods in Generator or Transducer	25
SCENARIO_NAMES length does not match SCENARIOS row count	25
Coupling count mismatch in Coupled EF constructor	25
runTestCaseSimulation() has fewer than 5 verification checks	25
runTestCaseSimulation() runs infinitely / never terminates	25
VIII. Reference Section	27
A. Glossary	27
B. Related Skills	27
C. References	28

I. Why Experimental Frames Matter

Every simulation study begins with a question, not a model. A stakeholder does not wake up wanting a DEVS atomic model of a transport node — they want to know whether the network survives a solar flare, or how long custody is maintained on a track, or which of two architectures degrades more gracefully under load. The model is only useful in service of that question.

An Experimental Frame (EF) is the formal bridge between a question and a model. It specifies exactly how a system under test (SUT) is stimulated, what is observed about it, and when the experiment stops. Two analysts asking different questions about the very same system need two different experimental frames — and, often, two different models of the system, at two different levels of fidelity. Building the wrong-fidelity model for a question either wastes enormous effort (a full physics-based model to answer a yes/no reachability question) or gives an answer that cannot be trusted (a coarse lumped model asked to predict millisecond-level latency).

EF Builder is a skill that automates this bridge. Given a description of a system and the questions stakeholders want answered, it walks through a disciplined, two-phase methodology — grounded in the EFSUT (Experimental Frame–System Under Test) framework of Zeigler (2026) — to (1) design and organize the right set of experimental frames, and (2) generate the compilable DEVS Java source code that implements them for the MS4 Systems framework.

A. What This Guide Covers

This guide follows the same explanatory, step-by-step format as the MS4 Me User's Guide, adapted to EF Builder. It is organized so you can either read it start to finish to learn the methodology, or jump directly to the section that matches where you are in a project:

- Section II introduces the core EFSUT vocabulary and concepts you need before running EF Builder — what an EF is, its architecture, and the theory behind ordering and comparing frames.
- Section III walks through the Conceptual Phase step by step: turning stakeholder questions into a reviewed, approved set of candidate experimental frames.
- Section IV walks through the Developmental Phase: turning an approved frame into compilable Java source files.
- Section V documents two implementation patterns that are easy to get wrong and are checked mechanically before any file is written.
- Section VI is a reference to every file EF Builder produces and where it is written.
- Section VII is a troubleshooting guide for the failure modes you are most likely to hit.
- Section VIII is a glossary and reference list.

B. A Note on How to Read the Verbatim Boxes

Passages shown in a shaded box with an amber left border, like this one, are quoted exactly — word for word — from the ef-builder skill's own reference documentation. Everywhere else, the text explains, motivates, and illustrates the methodology in plain language.

Keeping the verbatim text visually distinct means you can treat the shaded boxes as ground truth for exactly what the skill checks or requires, while the surrounding prose gives you the context and reasoning for why it matters.

II. Core Concepts You Need First

Before running EF Builder on a real system, it helps to have the vocabulary firmly in hand. This section defines the five ideas that everything else in this guide builds on: the Experimental Frame itself, its three-part architecture, the five EFSUT principles, the derivability relation that lets frames be compared and ordered, and the best-model table that connects a frame to the simplest model that can serve it.

A. The Experimental Frame

Formal definition (Zeigler, 2026): “Experimental Frame (EF): A specification of the conditions under which a system is observed or experimented with. Formally, an EF specifies: Admissible input segments (Generator): the set of input trajectories that may be applied; Observation conditions (Transducer): what outputs are measured and how they are processed; Control conditions: when the experiment starts and stops. System Under Test (SUT): The DEVS model (atomic or coupled) being experimented upon.”

Think of an EF as a laboratory protocol, not the laboratory itself. The SUT is the thing being studied; the EF is the written procedure that says exactly what you will do to it, what you will measure, and when you will stop — detailed enough that a different experimenter, or a different tool, could run the same experiment and get a comparable answer.

B. The EF Architecture: Generator, SUT, Transducer

Every experimental frame that EF Builder produces is a small, self-contained DEVS coupled model built from three pieces:

- A Generator (atomic) that injects scenario-based stimuli into the SUT's input ports.
- The System Under Test (SUT) itself — atomic or coupled — unchanged by the experiment.
- A Transducer (atomic) that observes the SUT's output ports, collects per-port statistics, and computes summary measures such as throughput.

Experimental Frame (Coupled Model)

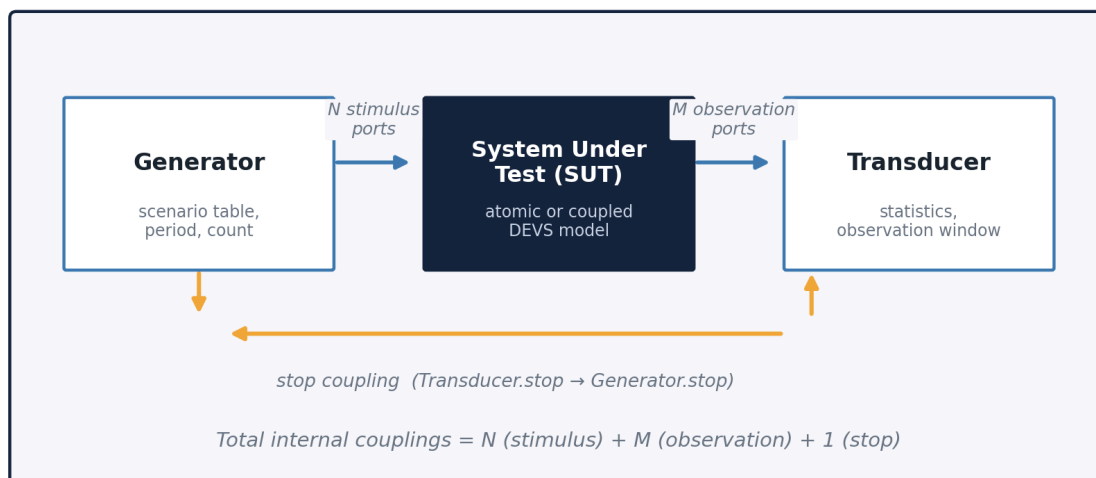


Figure 1. The three-part EF architecture: Generator $\rightarrow$ SUT $\rightarrow$ Transducer, closed by a stop coupling back to the Generator.

The Generator and Transducer are wired to the SUT and to each other by Internal Couplings only — the EF is self-contained and needs nothing from outside itself to run:

```
Couplings (IC only -- EF is self-contained):
Generator.out_ports -> SUT.in_ports      (N stimulus couplings)
SUT.out_ports -> Transducer.in_ports      (M observation couplings)
Transducer.stop -> Generator.stop        (1 control coupling)
Total: N + M + 1
```

That last coupling is what closes the loop: the Transducer decides when the experiment is over and tells the Generator to stop producing stimuli. Every EF Builder file you will see later has exactly $N + M + 1$ `addCoupling()` calls in its constructor — this is one of the first things the Developmental Verification step checks.

C. Five Core EFSUT Principles

Everything EF Builder does traces back to five principles from the EFSUT methodology:

#	Principle	What It Means
1	Multiple EFs per system	Any given system can have many experimental frames, each addressing different stakeholder questions and observing different aspects of the system.
2	EF Partial Order	Experimental frames are organized by a derivability relation using three operations — selection, aggregation, and coarsening — forming a partial order from finer to coarser observation.
3	Applicability Relation	An EF is applicable to a model if and only if the model's ports are compatible with the frame's observation and stimulation requirements.
4	Best-Model Table	For each EF in the partial order, there exists a minimally sufficient accommodating model — the simplest model that can answer the questions posed by that frame.
5	Mission Thread	A sequence of EF-Model pairs that together support end-to-end decision making.

Principle 4 is worth dwelling on, because it is the one that saves the most wasted effort in practice: EF Builder actively steers you away from over-engineering. If a question can be answered by a lumped, three-state model, the methodology does not want you building (and validating, and maintaining) a full high-fidelity model just because one exists. The Best-Model Table in Section II.E formalizes exactly this choice.

D. The Derivability Relation and EF Partial Order

Once you have more than one candidate frame, a natural question follows: are any of them related? Can one frame's answer be obtained from another's, without re-running an experiment? The derivability relation makes this precise.

Formal definition (Zeigler, 2026): “Derivability Relation: $EF1 \leq EF2$ ($EF1$ is derivable from $EF2$) if $EF1$'s data can be obtained from $EF2$'s data by applying one or more of: *Selection*: Restricting the time interval or choosing a subset of input/output trajectories; *Aggregation*: Combining multiple observations into summary statistics (e.g., averaging over time); *Coarsening*: Reducing resolution of observation (e.g., discretizing continuous outputs).”

In other words: $EF1 \leq EF2$ means $EF1$ is the “cheaper”, more specific experiment — its results can be squeezed out of $EF2$'s richer data without running anything new. Applying selection, aggregation, and coarsening repeatedly to the richest, most comprehensive frame in a family produces every coarser frame beneath it, which is exactly what forms a partial order: a hierarchy running from the finest observation at the top down to the coarsest summaries at the bottom.

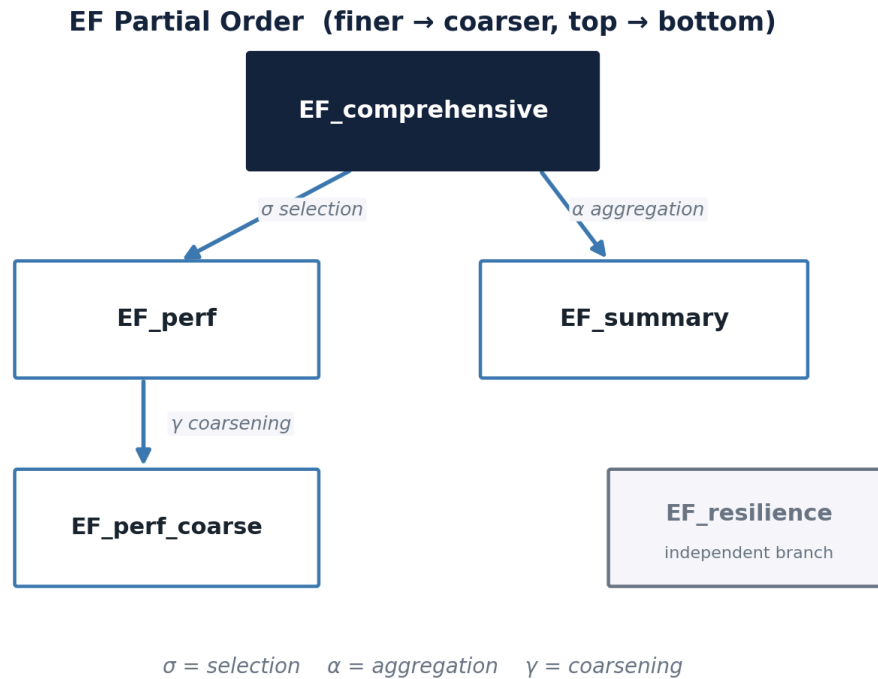


Figure 2. An example EF partial order. Each downward edge is labeled with the operation — σ (selection), α (aggregation), or γ (coarsening) — that derives the lower frame from the one above it. $EF_resilience$ sits on its own branch because it answers an orthogonal question that none of the other frames subsume.

EF Builder formalizes this diagram mathematically in Step C3.4 of the Conceptual Phase (Section III), computing the transitive closure of the derivability edges and reporting four properties of the resulting partial order: its greatest element (the single finest frame, if one exists), any independent branches, its width (the size of the largest antichain — the most frames that are mutually incomparable), and its height (the length of the longest derivability chain).

E. Best-Model Table and Applicability

Formal definition (Zeigler, 2026): “*Applicability: An EF is applicable to a model M if M has the required input/output ports and can generate the behaviors the EF needs to observe. The model must be able to ‘answer’ all questions posed by the frame. Best-Model Table: Maps each EF to the simplest (most abstract) model that can accommodate it. This prevents over-engineering by ensuring analysts use the minimally sufficient model for each question.*”

Applicability and the best-model table work together as a two-step filter. First, applicability is a yes/no test: can this model even answer this frame's questions, given its ports and behaviors? Second, among every model that passes that test, the best-model table picks out the cheapest one — so a frame that only needs aggregate

throughput numbers is matched to a lumped model, not to the most detailed model available, even though the detailed model would technically also work.

EF Builder records this as an applicability matrix, with frames as rows and candidate models as columns, marking each cell Best, Yes (applicable but over-specified), or — (not applicable):

```
| | Model1 | Model2 | ... | Modeln |
|---|---|---|---|
| EF_frame1 | Best | Yes | - | - |
| EF_frame2 | Yes | Best | - | - |
```

Legend: Best = minimally sufficient model; Yes = applicable but over-specified; - = not applicable (missing required ports or behaviors)

III. Getting Started: The Conceptual Phase

The Conceptual Phase is where EF Builder earns its keep before a single line of Java is written. It takes you from a loose description of a system and a handful of stakeholder questions, through a disciplined analysis, to a reviewed and approved set of experimental frames — backed by a formal partial order and a best-model table. Everything in this phase is captured in a shared working object the skill calls `EF_CONTEXT`, which persists across every step so nothing has to be re-entered.

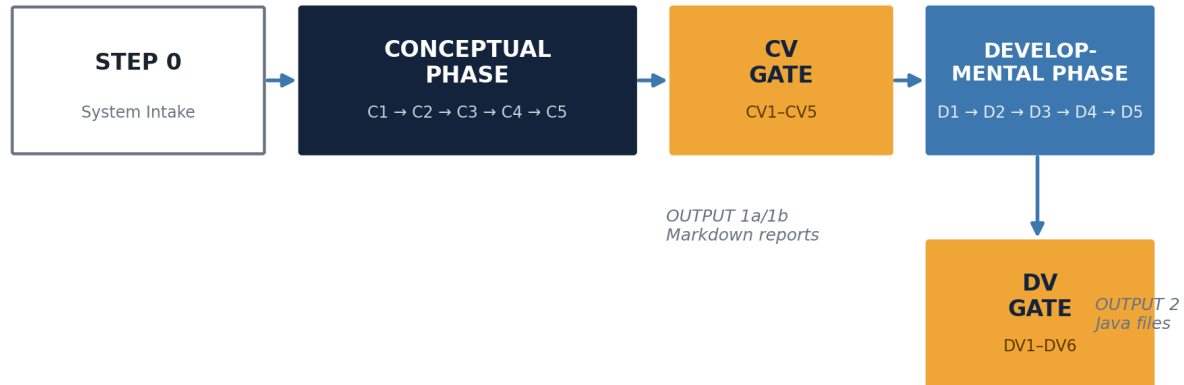


Figure 3. The full pipeline: System Intake, the Conceptual Phase (C1–C5), a Conceptual Verification gate (CV), the Developmental Phase (D1–D5), and a Developmental Verification gate (DV).

Let's walk through it step by step, the way you will actually experience it.

A. Step 0: System Intake

Before any question can be analyzed, EF Builder needs to know what the system actually is: its name, what it does, and — critically — its interface. If you have not already described your system, EF Builder will ask:

“Describe the system you want to design experimental frames for. Include what the system does, what inputs it receives, and what outputs it produces.”

From your description, EF Builder extracts five elements of the system interface:

Element	Description	Example
System name	Identifier for the SUT	TransportLayerNode
Input ports (X)	Signals/data the system receives	message_in, noise_level
Output ports (Y)	Signals/data the system produces	message_out, latency_report
States (S)	Distinct operational phases	IDLE, PROCESSING, DEGRADED, FAILED
State variables	Observable quantities	queue_depth, error_rate, throughput

- 1) Provide (or let EF Builder infer from your description) the system's name and a short description of what it does.

- 2) Identify every input port and output port — name, type, and a short description of what it carries.
- 3) List the distinct states the system can be in, and the observable state variables an experiment might want to read.
- 4) Review the extracted interface EF Builder presents back to you before moving on — this is the single source of truth every later step checks against.

It is worth getting Step 0 right: nearly every verification failure documented in Section VII traces back to a mismatch between a port used in a frame and a port declared here.

B. C1: Categorizing Stakeholder Questions

With the system interface settled, EF Builder turns to the questions themselves. If you have not supplied them, it asks directly:

“What questions do you want this experiment to answer about the system? You can provide multiple questions.”

EF Builder does not only take questions at face value — it also infers implicit ones from the system description. A description that mentions “failure” implies resilience questions; ports named `error_rate` imply performance questions. Every question, stated or inferred, is then classified into one of six EFSUT categories:

Category	Description	Example
Performance	Throughput, latency, capacity	What is the maximum throughput under load?
Resilience	Behavior under degradation or failure	How does the system respond to node loss?
Behavioral	State reachability, transition correctness	Does the system reach all declared states?
Parametric	Sensitivity to parameter changes	How does response time change with load?
Comparative	Contrasting configurations or designs	Which configuration yields lower latency?
Boundary	Edge-case and limit behavior	What happens at maximum capacity?

Each question is then assigned a priority — High, Medium, or Low — based on how strongly the stakeholder emphasized it, how much of the system's behavior it would exercise, and whether it is actually answerable with the ports on hand. The result is written into `EF_CONTEXT` as a numbered, categorized, prioritized question list (Q1, Q2, Q3, ...) that every later step refers back to.

C. C2: Generating Candidate Frames

This is where questions become frames. For every question, or cluster of related questions, EF Builder proposes one or more candidate experimental frames — each one a full specification of a Generator, a Transducer, and control conditions.

To keep the candidate set useful rather than exhaustive, EF Builder deliberately varies the scope of the frames it proposes:

Frame Type	Ports Used	Complexity	Typical Purpose
Full-scope	All N inputs, all M outputs	High	Comprehensive system characterization
Single-concern	Subset of ports related to one question	Low	Targeted investigation of one aspect
Stress	All inputs at extreme values, all outputs	High	Boundary and failure behavior
Comparative	Same ports, different scenario sets	Medium	A/B comparison of configurations
Focused-output	Minimal inputs, specific outputs	Low	Isolate one observable behavior

Two rules keep this generation step honest: every stakeholder question must be addressed by at least one candidate frame, and the frames as a whole should vary in scope rather than all converging on the same shape of experiment.

Scenario design

Each candidate frame's Generator is driven by a small table of scenarios — typically 3 to 8 — that follow a deliberate narrative arc, not a random walk:

```

1. Baseline scenario      -- nominal/default values for all ports
2. Stress scenario        -- push one or two ports toward adverse values
3. Edge case scenario     -- boundary values (0.0, 1.0, max, min)
4. Combined pressure      -- multiple ports at adverse values simultaneously
5. Recovery scenario      -- favorable values to enable recovery transitions
6. Stabilization          -- return to near-nominal conditions
7. Optimal scenario       -- best-case values for all ports
8. Adversarial scenario   -- worst-case combination

```

Each scenario should target a different SUT state or transition.
The scenario sequence should follow a narrative arc:
stable -> degrading -> crisis -> recovery -> stable

Timing is not left to guesswork either. The Generator's inter-stimulus period, its scenario count, and the Transducer's observation window all follow fixed derivation rules, with a hard constraint that the observation window must be long enough to see every scenario play out:

```

generator.period          = max(SUT time advances) * 2      (default: 10.0)
generator.scenario_count  = number of scenarios              (default: 8)
transducer.observation_time = period * scenario_count * 1.25 (default: 100.0)

CONSTRAINT: observation_time >= period * scenario_count

```

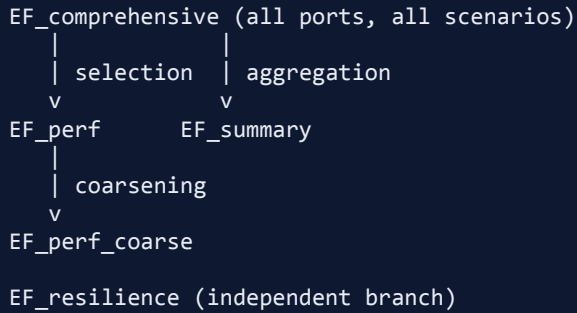
D. C3: Partial Order, Best-Model Table, and the EFSUT Table

With a set of candidate frames in hand, C3 is where they are organized and connected back to models. This step has several parts, run in sequence.

Establishing derivability and drawing the partial order

EF Builder compares every pair of candidate frames and tests whether one is derivable from the other via selection, aggregation, or coarsening (see Section II.D). The result is presented as a directed graph, finer frames at the top:

EF Partial Order for [SystemName]:



The formal derivability relation

The diagram above is then made mathematically precise. Frames are indexed E_1 through E_n from finest to coarsest; every derivability edge is labeled with its operation — σ for selection, α for aggregation, γ for coarsening; the transitive closure is computed (so if $E_4 \leq E_2 \leq E_1$, EF Builder records that $E_4 \leq E_1$ as well, via the composed operation); a Hasse diagram is drawn showing only the direct, non-transitive edges; and four structural properties of the order are reported:

Property	Meaning
Greatest element	The single finest frame, if one exists
Independent branches	Frames incomparable with the main chain
Width	Size of the largest antichain
Height	Length of the longest derivability chain

The Best-Model Table

For every frame in the order, EF Builder names the minimally sufficient model that can accommodate it, with a one-line justification:

Frame	Accommodating Model	Justification
EF_comprehensive	Full DEVS Model (all states, all ports)	Needs complete port interface
EF_perf	Reduced Model (performance-relevant states only)	Only needs throughput-related transitions
EF_resilience	Stochastic Model (failure states)	Needs failure modes and recovery transitions
EF_summary	Lumped Model (aggregated outputs)	Summary statistics derivable from lumped model

The Stakeholder Requirements EFSUT Table

Finally, C3 synthesizes everything — the original questions from C1, the frames from C2, and the best-model mapping just built — into a single decision-oriented table: one row per stakeholder question, showing which frame answers it, which model that frame maps to, and what the model's objectives are. This is the table stakeholders themselves are most likely to actually read.

Stakeholder Question	Experimental Frame (EF)	Primary Model	Modeling Objectives
[Question as plain sentence?]	EF_[Name]	[Model Name]	[Obj 1]; [Obj 2]; [Obj 3].

Every stakeholder question must appear in at least one row; every candidate frame and every model from the best-model table must be referenced somewhere in the table — EF Builder checks this completeness before moving on.

E. C4 & C5: Presenting Frames and Getting Your Approval

C4 is purely presentational: EF Builder lays out the full picture — the stakeholder questions, every candidate frame with its Generator/Transducer specification, the partial order, and the best-model table — in one consolidated view so you can evaluate the whole set at once rather than frame by frame.

C5 then hands the decision to you. EF Builder asks explicitly which frame or frames should proceed into the Developmental Phase:

“Which experimental frame(s) would you like to develop into DEVS Java source files?”

You can select one frame, several, or all of them. If you select a frame that is derivable from a richer one already in the candidate set, EF Builder will point this out and suggest developing the richer frame instead — since the frame you picked could then be obtained from it for free, without writing and maintaining two separate sets of Java files. You can also ask for a candidate frame to be modified before it is approved; EF Builder updates its specification in place.

F. Conceptual Verification: The Maximal EF Check

Before any code is generated, EF Builder runs a self-check called Conceptual Verification (CV). It is easy to underestimate how useful this gate is: it catches most port and coverage mistakes while they are still cheap to fix — in a markdown report — rather than after they have propagated into four Java files per frame.

The idea is to construct a synthetic Maximal EF: the union of every stimulus port, every observation port, and every scenario across all the candidate frames. If the Maximal EF is well-formed, every individual candidate frame is guaranteed to be derivable from it — which confirms the whole lattice is structurally sound.

Check	What It Verifies	Failure Means
CV1 – Construct Maximal EF	Union of all stimulus/observation ports and scenarios across all candidate frames	— (always constructible)
CV2 – Port Completeness Audit	Every port a frame uses is declared on the system; flags orphan system ports no frame exercises	A frame references a port the system doesn't have
CV3 – State Reachability Audit	The combined scenarios can drive the system through every declared state	A declared state is never targeted by any scenario
CV4 – Derivability Soundness	Each candidate frame's ports, scenarios, and coupling count are a valid subset of the Maximal EF's	A frame was not correctly derived during union computation
CV5 – Verification Report	Summarizes CV1–CV4 with an overall PASS/FAIL	Any prior step failed

If CV fails, EF Builder stops and reports exactly what to fix — typically an update to the system description (Step 0), an additional scenario or frame to reach a missing state, or a review of a frame's port selection. Only once CV passes does the Developmental Phase begin.

IV. From Approved Frame to Java Code: The Developmental Phase

“The Developmental Phase takes the user-approved experimental frame(s) from the Conceptual Phase and generates compilable DEVS Java source files. First, one System Model Java file is produced: a self-contained atomic DEVS model (extends PhaseAtomic) declaring the union of ALL ports referenced across all selected frames, with a complete state machine and accessor methods. Then, for each selected frame, three EF Java files are produced: a Generator (extends PhaseAtomic), a Transducer (extends PhaseAtomic), and a Coupled EF model (extends CoupledModelImpl) that wires the Generator, System Model, and Transducer together.”

If you selected more than one frame in C5, the System Model file is built once and shared by every frame; only the Generator, Transducer, and Coupled EF files are produced per frame. This keeps the SUT's state machine defined in exactly one place, no matter how many experiments observe it.

A. D1: Designing the Java Class Structure

Before generating any code, EF Builder plans the file set. For a system with F selected frames, that is always $1 + 3F$ files:

File	Class	Extends	Role
[SystemName]Model.java	[SystemName]Model	PhaseAtomic	Self-contained SUT with union of ALL ports across all frames, complete state machine, and accessor methods
[SystemName]_[FrameName]_EFGenerator.java	..._EFGenerator	PhaseAtomic	Injects scenario-based stimuli into the SUT
[SystemName]_[FrameName]_EFTtransducer.java	..._EFTransducer	PhaseAtomic	Observes SUT outputs and collects statistics
[SystemName]_[FrameName]_EF.java	..._EF	CoupledModelImpl	Wires Generator, SUT, and Transducer with $N+M+1$ couplings

All four classes live in package Models.java and import the MS4 Systems DEVS framework. D1 also fixes the port declarations (Generator output port names must exactly match System Model input port names, and System Model output port names must exactly match Transducer input port names), the scenario table shape, and which domain-specific events the Transducer should count.

B. D2 – D4: Generating the Generator, Transducer, System Model, and Coupled EF

D2 through D4 generate the four Java files themselves, following the class structure plan from D1. Each file implements the same six DEVS methods you already know from an atomic model — initialize(), getTimeAdvance(), getOutput(), internalTransition(), externalTransition(), and confluentTransition() — but each class fills them in differently, according to its role in the experiment:

Method	Generator	Transducer	System Model
getOutput()	Emit next scenario row	Emit stop w/ stats summary	Emit pending values

Method	Generator	Transducer	System Model
internalTransition()	count++, hold or passivate	Compute avg/throughput, DONE	Save+clear pending emit
externalTransition()	Read stop, passivate	Log arrival, increment counters	Parse input, scheduleEmit()
getTimeAdvance()	return sigma	return sigma	return sigma

The System Model's column is the most involved of the three, because it is the one carrying real domain behavior — the Generator and Transducer are comparatively mechanical (a scenario player and a statistics collector, respectively). The System Model's state machine is designed in D3B from the states and state variables you supplied back in Step 0: each state gets a name, a time advance, an internal-transition target, and a set of external-transition handlers describing how each input port affects it. Section V.A covers the one pattern in the System Model that is easy to get subtly wrong: the pending-emit mechanism used to produce output in response to an external event.

D3B also generates every accessor method (getXxx()) that any Coupled EF's test harness will call on the System Model or Transducer — EF Builder scans ahead through the couplings and test code to make sure nothing is missing, rather than discovering a missing accessor at compile time.

C. D5: Verify and Write the Java Files

D5 is the code-side counterpart of Conceptual Verification: nothing is written to disk until it passes two rounds of checks.

Structural verification (Step D5.1)

A checklist confirming the generated files are, mechanically, valid MS4 Systems DEVS Java: the package declaration, required imports, correct base classes (PhaseAtomic for Generator/Transducer/System Model, CoupledModelImpl for the Coupled EF), a serialVersionUID, correctly typed and directed port declarations, the right constructor signatures per class, all six DEVS methods present, the getMessages()/getFirstMessageValue() helpers, a well-formed scenario table, standalone main() methods with a runTestCaseSimulation(), the System Model's pending-emit mechanism, correct Display usage, and the use of sim.getCurrentSimulationTime() (never getSimulationTime() or getTime()) in the Coupled EF's test harness.

EFSUT consistency verification (Step D5.2)

A second, semantic checklist confirming the code actually satisfies the methodology, not just the Java compiler: every stakeholder question is addressed, the System Model declares the full port union across all selected frames, Generator/System Model/Transducer port names line up exactly, the coupling count is exactly $N+M+1$, the scenario table has at least 3 rows with consistent dimensions, timing constraints hold, every accessor called anywhere has a matching method, at least 5 PASS/FAIL verification checks exist in the test harness, and — checked in full detail in Section V.B — all three mandatory stop conditions are present and satisfiable.

Only after both checklists pass are the files written to the output directory (Section VI).

D. Developmental Verification: The Maximal EF Cross-Check

After D5 writes the files, EF Builder runs one more gate: Developmental Verification (DV). Where Conceptual Verification checked the plan, Developmental Verification checks the actual generated code against that same Maximal EF specification — catching anything that drifted during code generation itself.

Check	What It Verifies
DV1 – System Model Port Audit	Every port in the Maximal EF's port union actually appears as an <code>addInputPort()/addOutputPort()</code> in the generated System Model
DV2 – Accessor Coverage Matrix	Every <code>sys.getXxx()</code> / <code>trans.getXxx()</code> call in any Coupled EF's test harness resolves to a real method
DV3 – Port Activity Matrix	Every declared output port is actually emitted in at least one state — flags “dead” ports that will never produce output
DV4 – Coupling Subsumption Check	Each frame's <code>addCoupling()</code> calls total exactly $N+M+1$ and are a valid subset of the Maximal EF's port space
DV5 – State Reachability Trace	Each Generator's scenario table, traced through the System Model's actual transition thresholds, reaches the target states identified in CV3
DV6 – Verification Report	Summarizes DV1–DV5 with an overall PASS/FAIL

DV3 (the port activity matrix) is a particularly good example of why the Maximal EF cross-check exists: a syntactically valid Java file can declare an output port that no code path ever actually writes to. The file compiles fine, and nothing looks wrong until a Transducer wired to that port silently never receives a single message. DV3 catches this mechanically, before you spend time debugging a simulation that runs “successfully” but produces empty statistics.

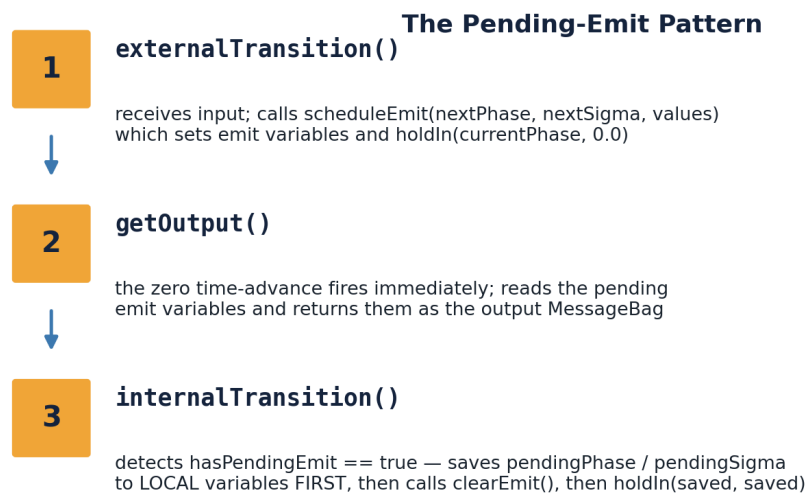
V. Two Patterns Worth Understanding Before You Read the Code

Two rules recur throughout the Developmental Phase and its verification steps, and both are flagged in the reference documentation with the strongest severity words it uses — CRITICAL and MANDATORY. Understanding them ahead of time makes the generated Java files far easier to read, and makes the two most common failure modes in Section VII immediately recognizable.

A. The Pending-Emit Pattern

DEVS has a subtle timing rule: `getOutput()` only ever fires immediately before `internalTransition()`, never in direct response to an external event. That creates a real problem for the System Model — how do you produce output right after receiving an input, if `getOutput()` cannot be called from inside `externalTransition()`?

The answer is a small piece of bookkeeping called the pending-emit mechanism, and it runs the same three-step sequence every time the System Model needs to respond to an input with output:



CRITICAL: always save pendingPhase / pendingSigma to locals BEFORE calling clearEmit()
 — clearEmit() resets both to "" / INFINITY.

Figure 4. The pending-emit pattern: `externalTransition()` schedules the emit, a zero time-advance immediately triggers `getOutput()`, and `internalTransition()` then clears the pending state and resumes normal timing.

The state that makes this work is a small set of fields declared once on the System Model:

```

private boolean hasPendingEmit = false;
private String pendingPhase = "";
private double pendingSigma = Double.POSITIVE_INFINITY;
private String emit[Output1] = null;
private String emit[Output2] = null;
// ... one emit variable per output port
  
```

The one rule that must never be violated is the order of operations inside `internalTransition()`: `pendingPhase` and `pendingSigma` must be copied to local variables before `clearEmit()` runs, because `clearEmit()` resets both

fields to "" and INFINITY respectively. Reverse the order, and the values you meant to resume with are gone before you read them.

```
// CORRECT pattern:
if (hasPendingEmit) {
    String nextPh = pendingPhase;
    double nextSg = pendingSigma;
    clearEmit();
    holdIn(nextPh, nextSg);
    return;
}

// INCORRECT -- pendingPhase/pendingSigma are destroyed by clearEmit():
// if (hasPendingEmit) {
//     clearEmit();
//     holdIn(pendingPhase, pendingSigma); // BUG: already cleared!
// }
```

This exact save-before-clear ordering is checked explicitly in Step D5.1 (“Pending emit mechanism”) and is the single most common root cause listed in the Troubleshooting section (VII) under “Pending emit values lost in internalTransition.”

B. Stop Condition Enforcement

The second pattern exists to answer an uncomfortable question: what stops the simulation loop from running forever? Every `runTestCaseSimulation()` method that EF Builder generates is required to answer that question three independent ways, so that no single failure mode — a bug, a livelock, a scenario that never quite reaches its target state — can leave the simulation spinning.

“Every `runTestCaseSimulation()` method MUST contain explicit stop conditions that are guaranteed to be satisfiable within the simulation's configured bounds. A simulation that cannot satisfy its stop condition will run infinitely through the iteration loop, consuming resources and producing no useful results.”

The three mandatory, independent conditions, any one of which ends the loop:

- (a) Logical completion — `top.getTransducer().phaseIs("DONE") && top.getGenerator().phaseIs("PASSIVE")`: the Transducer's observation window has expired and the Generator has exhausted its stimuli or received the stop signal.
- (b) Max simulation time — `simTime >= maxTime`, where `maxTime` must be at least $1.2 \times \text{OBSERVATION_TIME}$, giving the Transducer enough headroom to actually reach DONE.
- (c) Max iteration count — the while loop's `iteration < maxIterations` guard, where `maxIterations` must be at least $2 \times \text{GEN_MAX_COUNT} \times 10$, protecting against deadlock or livelock even if the model itself misbehaves.

Domain-specific stop conditions (a total system collapse, a queue overflow, a mission-abort signal) are welcome as additional termination criteria — but they may never be the only stop condition present. Before any file is written, EF Builder runs a ten-row checklist against the generated code and will not proceed if any row fails:

#	Verification	How It Is Checked
1	Loop has a finite upper bound	<code>while (iteration < maxIterations && !stopConditionMet)</code> — both guards present

#	Verification	How It Is Checked
2	maxIterations is a finite positive integer	Declared as final int MAX_ITERATIONS = N; where N > 0
3	maxTime is checked inside the loop	if (simTime >= maxTime) with a break statement
4	Logical completion condition is present	if (trans.phasels("DONE") && gen.phasels("PASSIVE")) sets stopConditionMet = true
5	stopConditionMet initialized to false	Declared before the loop
6	Every stopConditionMet = true has a reason	No path sets the flag without a stopReason assignment
7	$\text{maxTime} \geq 1.2 \times \text{OBSERVATION_TIME}$	Prevents premature cutoff before the Transducer completes
8	$\text{maxIterations} \geq 2 \times \text{GEN_MAX_COUNT} \times 10$	Sufficient headroom for all stimulus-response cycles
9	No dead-code paths bypass the stop checks	All stop checks execute on every iteration
10	stopReason is printed after the loop	System.out.println("Stop Reason: " + stopReason);

If any row fails, EF Builder corrects the generated code and re-verifies all ten rows from the start — a fix to one row is never assumed safe for the others.

VI. Output Files Reference

Every EF Builder run writes into a per-project folder named after the system, inside a fixed output root, with generated documentation and generated Java code kept in separate subfolders:

```
output/
  [SystemName]/
    docs/
      [SystemName]_EF_Analysis.md
      [SystemName]_EFSUT_Table.md
    java/
      [SystemName]Model.java
      [SystemName]_[FrameName]_EFGenerator.java
      [SystemName]_[FrameName]_EFTransducer.java
      [SystemName]_[FrameName]_EF.java
```

Figure 5. The output directory structure for a system named [SystemName], after the Conceptual and Developmental Phases have both completed.

A. File-by-File Guide

File Type	Output Path	Produced After
EF Analysis Report	output/[SystemName]/docs/[SystemName]_EF_Analysis.md	Conceptual (C5)
Stakeholder Requirements EFSUT Table	output/[SystemName]/docs/[SystemName]_EFSUT_Table.md	Conceptual (C5)
System Model Java (SUT)	output/[SystemName]/java/[SystemName]Model.java	Developmental (D5)
EF Generator Java	output/[SystemName]/java/[SystemName]_[FrameName]_EFGenerator.java	Developmental (D5)
EF Transducer Java	output/[SystemName]/java/[SystemName]_[FrameName]_EFTransducer.java	Developmental (D5)
EF Coupled Model Java	output/[SystemName]/java/[SystemName]_[FrameName]_EF.java	Developmental (D5)

The EF Analysis Report is the single most complete artifact EF Builder produces — it is worth reading in full for any non-trivial system, since it captures the reasoning trail, not just the conclusions. Its ten sections walk from the system overview through to the transition into the Developmental Phase:

1. SYSTEM OVERVIEW
2. STAKEHOLDER QUESTIONS
3. CANDIDATE EXPERIMENTAL FRAMES
4. EF PARTIAL ORDER
5. FORMAL DERIVABILITY RELATION
6. BEST-MODEL TABLE
7. APPLICABILITY MATRIX

- 8. SELECTED FRAMES
- 9. STAKEHOLDER REQUIREMENTS EFSUT TABLE
- 10. TRANSITION TO DEVELOPMENTAL PHASE

If you selected multiple frames in C5, remember that only one System Model file is produced — it is shared across all of them — while the Generator, Transducer, and Coupled EF files are produced fresh per frame. A system with two selected frames therefore yields $1 + (3 \times 2) = 7$ Java files in total.

VII. Troubleshooting

Most problems you will encounter trace back to one of two things: a port mismatch between the system description and a frame, or the two implementation patterns from Section V. This section lists the failure modes documented in the ef-builder reference material, grouped roughly in the order you are likely to hit them — conceptual issues first, then code-generation issues.

Candidate frame specifies ports not present on the system

Cause: Frame observation or stimulus ports reference ports the system does not have.

Solution: Re-check the system interface (Step 0). Either narrow the frame's port scope or update the system description to include the missing ports.

Partial order has no derivability edges

Cause: All candidate frames are independent (no frame's data is derivable from another's).

Solution: This is valid. Independent frames address orthogonal questions. No action needed unless frames should be related.

Best-model table maps all frames to the same model

Cause: All frames require the same level of model fidelity.

Solution: This can occur when frames differ only in scenarios, not in observed ports. Consider whether coarser models could serve some frames.

Generator output ports do not match System Model input ports

Cause: Port names declared in the Generator's `addOutputPort()` calls do not match the corresponding `addInputPort()` names on the System Model.

Solution: Verify that every Generator output port name exactly matches a System Model input port name. Port names are case-sensitive. Re-check the system interface (Step 0), the frame's stimulus port list (C2), and the port union in D3B.

System Model missing accessor method called by an EF

Cause: A Coupled EF's `runTestCaseSimulation()` calls `sys.getXxx()` but the System Model class does not have a corresponding `getXxx()` method.

Solution: Add the missing accessor method to the System Model. Re-run D3B Step D3B.8 to scan all EF references and ensure every accessor is present.

Pending emit values lost in internalTransition

Cause: `clearEmit()` is called before saving `pendingPhase/pendingSigma` to local variables in `internalTransition()`, destroying the values before they can be used.

Solution: Save `pendingPhase` and `pendingSigma` to local variables BEFORE calling `clearEmit()`. See Section V.A for the correct pattern.

Observation window too short for scenario count

Cause: Constraint violated: $\text{DEFAULT_OBS_TIME} < \text{DEFAULT_GEN_PERIOD} * \text{DEFAULT_GEN_MAX_COUNT}$.

Solution: Increase `DEFAULT_OBS_TIME` or decrease `DEFAULT_GEN_PERIOD` / `DEFAULT_GEN_MAX_COUNT` in the Coupled EF defaults.

Missing DEVS methods in Generator or Transducer

Cause: One or more of the 6 required DEVS methods (`initialize`, `getTimeAdvance`, `getOutput`, `internalTransition`, `externalTransition`, `confluentTransition`) is absent from the generated Java file.

Solution: Re-run D2 (Generator) or D3 (Transducer) generation. All 6 methods are mandatory for `PhaseAtomic` subclasses.

SCENARIO_NAMES length does not match SCENARIOS row count

Cause: The `SCENARIO_NAMES` String array has a different length than the `SCENARIOS` `double[][]` array.

Solution: Ensure both arrays have exactly the same number of entries. Each scenario row must have a corresponding name.

Coupling count mismatch in Coupled EF constructor

Cause: The number of `addCoupling()` calls does not equal N (stimulus) + M (observation) + 1 (stop).

Solution: Count the Generator output ports (N), Transducer input ports excluding stop (M), and verify exactly $N + M + 1$ `addCoupling()` calls exist in the Coupled EF constructor.

runTestCaseSimulation() has fewer than 5 verification checks

Cause: Insufficient test coverage in the generated Coupled EF.

Solution: Add domain-specific verification checks. At minimum: (1) SUT received stimuli, (2) Transducer received reports on each port, (3) Generator produced expected count, (4) state variables in valid ranges, (5) system exited initial state.

runTestCaseSimulation() runs infinitely / never terminates

Cause: The stop condition is either missing, unreachable, or depends on a domain event that never fires. Common sub-causes:

- The while loop lacks a `maxIterations` guard (loops on `while (true)` or `while (!stopConditionMet)` with no iteration cap).
- `maxTime` is checked but never reaches because the model enters a livelock (alternating states with zero-time advances).
- The logical completion check (Transducer `DONE` && Generator `PASSIVE`) is absent or uses the wrong phase string (e.g., "Done" vs "DONE").
- `stopConditionMet` is never set to true because the only stop condition is a domain-specific event that the current scenario does not trigger.

Solution: Apply the full Stop Condition Verification Checklist from Section V.B. Specifically:

- Ensure the while loop has BOTH guards: `iteration < maxIterations` AND `!stopConditionMet`.
- Ensure `maxTime` is checked inside the loop with a break statement, and that `MAX_SIMULATION_TIME >= 1.2 * OBSERVATION_TIME`.
- Ensure `MAX_ITERATIONS >= 2 * GEN_MAX_COUNT * 10`.
- Ensure the logical completion condition uses exact phase strings matching the Transducer and Generator implementations (case-sensitive).
- Ensure at least one stop condition is unconditionally satisfiable (the iteration cap and max time bounds guarantee this).
- After fixing, re-verify all 10 rows of the checklist before writing the file.

VIII. Reference Section

A. Glossary

Term	Meaning
EF (Experimental Frame)	A specification of admissible inputs, observation conditions, and control conditions under which a system is studied.
SUT (System Under Test)	The DEVS model — atomic or coupled — being experimented upon.
EFSUT	Experimental Frame–System Under Test: the methodology (Zeigler, 2026) formalizing the relationship between stakeholder questions, EFs, and models.
Generator	The atomic model that injects scenario-based stimuli into the SUT's input ports.
Transducer	The atomic model that observes SUT outputs, collects statistics, and signals when the experiment should stop.
Derivability relation ($\leq$)	$EF1 \leq EF2$ if $EF1$'s data can be obtained from $EF2$'s data by selection, aggregation, or coarsening.
Selection (σ)	Restricting the time interval or choosing a subset of input/output trajectories.
Aggregation (α)	Combining multiple observations into summary statistics, e.g. averaging over time.
Coarsening (γ)	Reducing the resolution of an observation, e.g. discretizing continuous outputs into categories.
EF Partial Order	The hierarchy formed by the derivability relation among a family of frames, from finest to coarsest.
Best-Model Table	The mapping from each EF to the simplest model that can accommodate it.
Applicability	An EF is applicable to a model if the model's ports and behaviors satisfy the frame's requirements.
Maximal EF	The synthetic union of all ports and scenarios across every candidate frame, used as the top of the lattice for verification.
Mission Thread	A sequence of EF-Model pairs that together support end-to-end decision making.
Pending-Emit Pattern	The mechanism by which a PhaseAtomic model produces output in direct response to an external event, using a zero time-advance.
Stop Condition Enforcement	The requirement that every test harness has three independent, mandatory, satisfiable termination conditions.
EF_CONTEXT	The shared object that persists system, frame, verification, and output data across every step of both phases.

B. Related Skills

- `devs-builder` — Generate DEVS models from system descriptions.
- `ses-builder` — Build System Entity Structure (SES) hierarchies.
- `pwsa-expert` — Expert on PWSA mission engineering and DEVS-PWSA framework.

C. References

Zeigler, B.P. (2026). “Experimental Frame–System Under Test (EFSUT): A Principled Foundation for Model Choice and Lifecycle Management in Digital Twins.” *Digital Engineering and Digital Twin*, 4. DOI: 10.32604/dedt.2026.082492

Zeigler, B.P., Muzy, A., & Kofman, E. (2019). *Theory of Modeling and Simulation*, 3rd ed. Academic Press.

Zeigler, B.P. (2000). *Theory of Modeling and Simulation*, 2nd ed. Academic Press.