

# ***Expressway***<sup>™</sup> **User Manual**

2008-06-30

Cliff Berg

Expressway Solutions LLC

2003 Cutwater Court

Reston VA 20191

703-994-4932

[Cliff.Berg@ExpresswaySolutions.com](mailto:Cliff.Berg@ExpresswaySolutions.com)

## Table of Contents

Installing and Running Expressway.....	5
Increasing Java Memory.....	5
Making Simulations Repeatable.....	5
Why Command Line?.....	5
Why Not Open Source?.....	5
Maturity of the Implementation.....	6
Public Forums.....	6
Overview of Ways to Use Expressway™.....	7
As a Technical Strategy Valuation Tool.....	7
Defining a Probabilistic Simulation Model.....	7
As a Design Decision Modeling Tool.....	7
As a Design Tool.....	8
Key Modeling Concepts.....	9
Understanding Model Domains.....	10
State.....	10
Events.....	10
Ports.....	10
Conduits.....	10
Activities.....	10
Function State Model.....	12
Closure of Native Implementations.....	12
Structure is Statically Determinable.....	13
More on the Event Model .....	13
Activation of Model Components.....	13
Pre-Defined Events.....	14

Initial State Values.....	14
What Constitutes an “Event”.....	14
Attributes.....	15
Attribute Value Determination.....	15
Attribute-State Bindings.....	15
Simulation and Simulation Runs.....	16
Watching for Race Conditions.....	16
Pre-Defined Activities and Functions.....	16
Terminals .....	16
Generators.....	17
Startup of Generators.....	18
Dynamically Changing a Generator's Statistical Parameters.....	19
Generation of Repeated Events.....	19
What a Generator Generates.....	19
Pulse Generators.....	19
Tallies.....	20
Delays.....	21
Sustains.....	21
Switches.....	21
Modulator.....	21
Discriminator.....	22
Reflections on the Event Model.....	22
Understanding Decision Domains.....	23
Decision Points.....	23
Decisions.....	23
Choices.....	23
Variables.....	23
Variable-Attribute Bindings.....	23

Decision Scenarios.....	23
Decision Propagation.....	23
Expressway Input Syntax.....	24
Using Expressway.....	34
Making a Business Case for an IT Strategy.....	34
Refinement Of Strategies Into Designs.....	34
Facilitating Traceability Between Actions, Strategies, and Business Value.....	34
Providing Input to Portfolio Planning.....	34
Open Enterprise Repository.....	35
OER Core Schema.....	36
Enterprise Meta Model.....	36
Security Model.....	36
Architecture Meta Model.....	36

## Installing and Running Expressway

The current version of *Expressway*<sup>™</sup> is a Java application. It requires Java 5 or later.

This version is run from the command line. (On a Mac, just open a bash shell window.) There is no “installation”. Just specify the jar file when you run *Expressway*<sup>™</sup>. Here is an example:

```
java -jar expressway.jar my_model.xml
```

Here is the complete command line syntax:

```
java -jar <expressway_jar_path> [-random_seed=<number>] <model_file_path>
```

To run *Expressway*<sup>™</sup> you will also need the following components in your classpath:

1. Version 1.1 (commons-math-1.1.jar) of the Apache Commons math package, available from <http://commons.apache.org/math/>
2. Version 1.4.4 of the Xerces XML package (xerces.jar), available from <http://xerces.apache.org/xerces-j/>

### Increasing Java Memory

For large models, you might want to specify more OS virtual memory for the Java VM, using the `-Xmx` extended option (for Sun's Java implementation). Here is an example that specifies 256Mbytes of virtual memory for the Java VM:

```
java -Xmx256m -jar expressway.jar my_model.xml
```

### Making Simulations Repeatable

It is often useful to be able to have repeatable simulations. This can be achieved by using the same pseudo-random number seed each time. This allows you to debug your model because you will get the same (right or wrong) results time you run it. Here is an example:

```
java -jar expressway.jar -random_seed=1 my_model.xml
```

### Why Command Line?

This version of *Expressway*<sup>™</sup> is a command line tool in order to make the capability available to users for free. Developing a full GUI-driven version with enterprise features is a major undertaking and requires a business model – i.e., such a version will not be free. However, it is likely that we will make a GUI-driven version freely available, but without many enterprise features. If we do that, it will be a fully usable version – not a crippled version. It merely will not have some features that are useful for broad usage within a large organization. Thus, rest assured that if you as an individual start to use *Expressway*<sup>™</sup> now that you will not be cornered into paying for it later.

## **Why Not Open Source?**

It is possible – and hopeful – that an open source version of this type of tool will evolve, and we might even release *Expressway*<sup>™</sup> as open source at some point, if we can establish a viable business model for that. At present we are using the tool to explore the kinds of features and usage models that are desirable, and we want to maintain control of our implementation. In the meantime, we are making the binary available free for use.

## **Maturity of the Implementation**

Please note that the current implementation (0.1) is an *alpha implementation*. We are actually using it in our consulting work, and it works, but it might have bugs that we don't know about.

We have created a very cursory validation test suite, consisting of 33 tests. However, a full validation suite (which we are developing) will have hundreds – may be thousands – of tests.

We consider reliability and correctness to be more important than features, so our effort is slanted toward enhancing the validation suite, so you can expect the product to become more reliable over time. At present, we are not aware of any bugs or incorrect behavior. However, it should be pointed out that the behavior of Functions has not been validated, so we encourage you to use Activities exclusively until we can validate Functions. Functions are a convenience, and are not required to build usable models.

If you should find a bug, please email me (Cliff) at [cliff.berg@expresswaysolutions.com](mailto:cliff.berg@expresswaysolutions.com)

## **Public Forums**

There are two public forums that we have established for *Expressway*<sup>™</sup>. One site, <http://ValueDrivenIT.com>, is unabashedly commercial, and its purpose is to promote awareness of the underlying concepts, through the book Value-Driven IT. This site contains material that is prepared with care and published on the site.

The other site, <http://ValueDrivenIT.org>, is strictly non-commercial, and is intended for the user community to share models and insights. The site is currently un-moderated, and this will continue if the majority of postings are courteous, on-topic, and of value.

Finally, there is the home website of the company Expressway Solutions (<http://expresswaysolutions.com>) that develops and sponsors *Expressway*<sup>™</sup>. It takes money to do this, and Expressway Solutions is the money-making enterprise.

## Overview of Ways to Use *Expressway*<sup>TM</sup>

*Expressway*<sup>TM</sup> is fundamentally a decision support tool. That said, decision-making is perhaps the most important aspect of any planning process, and of any system design process. Therefore, while *Expressway*<sup>TM</sup> supports decision-making, its purpose is to support the planning and design of IT solutions.

### ***As a Technical Strategy Valuation Tool***

The essential steps are:

1. Define a probabilistic simulation model.
2. Define one or more scenarios, each of which embodies different assumptions or strategies that are to be compared.
3. Stochastically simulate each of the scenarios many (hundreds of) times and compare the aggregate results for each scenario.
4. Examine the results to try to understand the reason behind the differences in each outcome.
5. Make small modifications to select scenarios to assess the impact on the business performance of the scenario.
6. Perform steps 3-5 several times until an understand of overall system cause and effect is achieved, and how various assumptions or strategies contribute or detract from overall business value once all interactions are accounted for.

### **Defining a Probabilistic Simulation Model**

- a) Identify the Things that Can Happen
- b) Identify the Rules of Behavior
- c) Assign Probability Distributions

### ***As a Design Decision Modeling Tool***

The essential steps are:

1. Define a decision model, to define the issues that determine the available design choices.
2. Link the decision parameters to simulation models; these decision models determine the value of each design choice.
3. Explore various decisions to determine the optimum choices.

## ***As a Design Tool***

The Expressway semantic model is effective for design because of these reasons:

1. It directly defines robust patterns for concurrency and closure, two extremely important concepts for reliable systems.
2. The behavior model is functional, which lends itself to translation into highly concurrent and provably correct implementations.
3. It is simple and unambiguous.
4. It is hierarchical and component-oriented, enabling use for the design of large and complex systems.
5. The design can be directly simulated, enabling incremental design and validation.

Code generation is a natural expectation for a design tool, and code generation from an Expressway model should be straightforward: all that is needed is to build a generator that compiles each behavioral model for the runtime environment, and link each output program with a (highly optimized) library that implements the Expressway event model.

## Key Modeling Concepts

A model domain is a model of a system in terms of cause-and-effect relationships. Strictly speaking, it is a discrete event model, but it is flexible enough to allow one to model behavior as well as the design of real-world systems, including information technology systems. One can also model the economic value of IT systems, and that is the intended primary use of the Expressway modeling environment.

A decision domain is a model of the interdependence of the many decisions that one must make when planning an IT system, or the migration of one IT environment to another. For example, planning for the improvement of an IT system from a “current state” to a “future state” is often complicated by many interdependencies among the decisions. The decision model capability is designed to help in such situations. Further, because decision domains can be linked to model domains, one can track which models results from which decisions, and how models depend on those decisions. For example, if a decision results in the re-evaluation of subordinate decisions, and some of those provide inputs to a model domain, the model domain can be re-simulated automatically, and the inter-dependence is transparent.

## Understanding Model Domains

A model domain defines a system with discrete event semantics. The core model is conceptually very simple: Model Components contain State, and whenever a state changes, an Event describing that change is generated and propagates along Conduits to other Model Components.

While this model is simple in concept, there are many subtleties. These are discussed below.

### State

Any component may define any number of States. A State has a value associated with it, and that value is guaranteed to remain constant during the duration of a simulation epoch. When a State changes value, an Event occurs, by definition. (See next section.) Such Event may only propagate through a Port (discussed below) that is *bound* to the State. The bound Port must belong to the component that owns the State. If a Port is bound to a State, then any Events pertaining to the State are dispatched from that Port, and travel along Conduits (discussed below) to other Ports.

### Events

An Event is, by definition, a change in the value of a State. Events manifest as Event objects that are in effect notices that are sent to other Model Components.

### Ports

A Port is the *only means* by which information can enter or leave a Model Component.

By default, a Port does not govern the direction of flow of information, or how many Conduits may be connected to it. However, if a Port is behavioral (this is discussed shortly), then it can be defined to be “input” or “output”, in which case Events may only propagate in or out of the Port (relative to the Component that owns the Port), respectively.

Ports are typeless: they do not constrain the value type of Events that pass through them.

### Conduits

A Conduit is an explicit connection from one Port to another. Information (as Events) may only travel along Conduits – there is no other means for dynamically communicating information.

### Activities

An Activity is a kind of Model Component. Conceptually, an Activity generates *future* State value changes, i.e., Events. That is, when an Activity creates an Event, it schedules the Event (change of State value) to occur at a future time. If that time is the current simulation time, then the Event occurs in the next simulation epoch, which will be zero seconds (in simulation time) from the current epoch. It is not possible for an Activity to cause an Event to occur in the current simulation epoch.

Illustration 1 shows a birds-eye view of how an Activity contains a State, and when the Activity's State changes its value, the change is propagated as an Event through Ports and along Conduits to other Model Components.

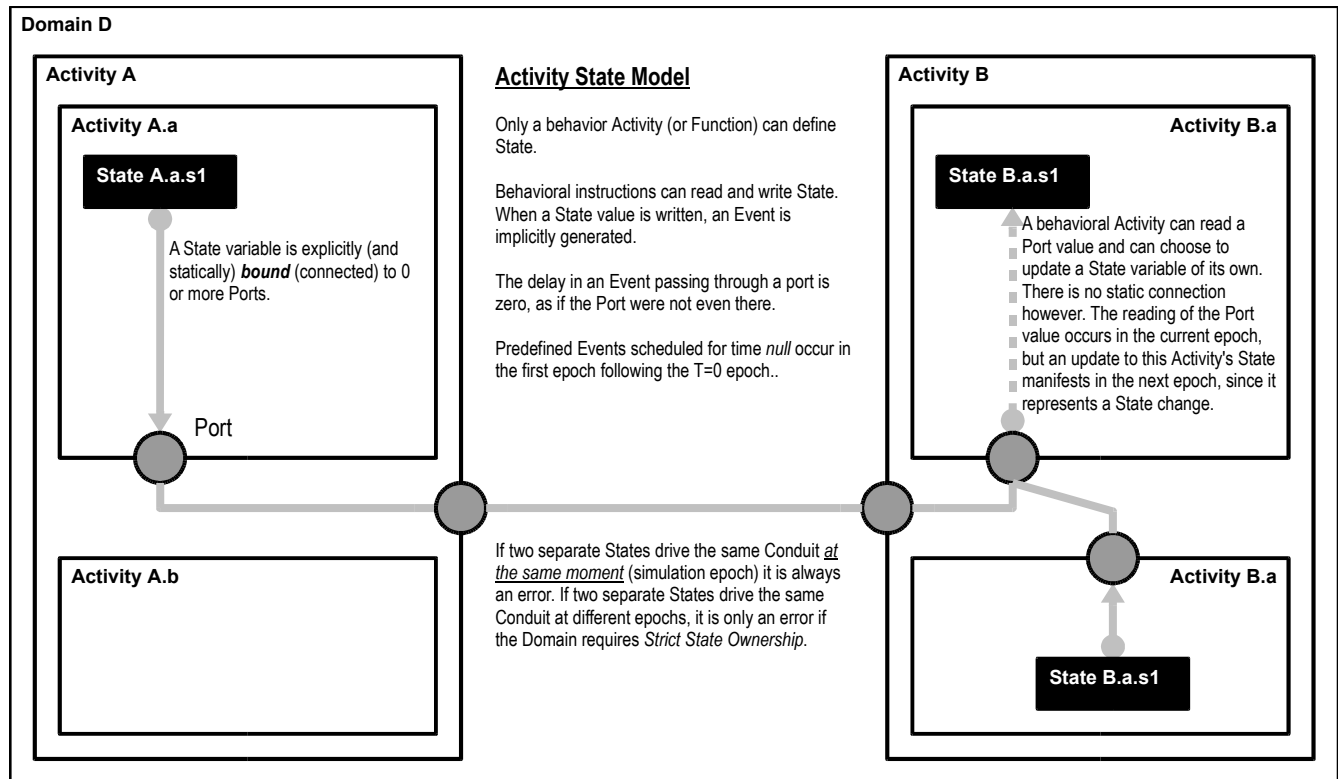


Illustration 1: The invocation and event propagation model for Activities.

Activities can be **structural** or **behavioral**. A structural Activity may contain other Activities or Functions, and it derives its behavior from those. A behavioral Activity may not contain other Activities or Functions: rather, it derives its behavior from a “native implementation” class that is specified for the Activity, or from an internal state with predefined events. A behavioral Activity also may not contain internal inter-connections (conduits). [Verify detection]

Behavioral components may not contain Conduits.

Both structural and behavioral components may define State, and those States may bind to Ports of the component.

Only a behavioral Activity (or Function) can define State. [Verify detection] Behavioral instructions can read and write State. When a State value is written, an Event is implicitly generated. The delay in an Event passing through a port is zero, as if the Port were not even there. Predefined Events scheduled for time null occur in the first epoch following the T=0 epoch. [Verify]

If two separate States drive the same Conduit *at the same moment* (simulation epoch) it is always an

error. If two separate States drive the same Conduit at different epochs, it is only an error if the Domain requires Strict State Ownership.

## **Function State Model**

A Function is a kind of Model Component. Conceptually, a Function generates current Events: that is, all Events generated by a Function occur in the current simulation epoch.

A Function may not define internal Activities.

A Function cannot query which Ports have received events: its design must determine which Ports it should read, in a deterministic manner.

A Function may have internal State: in fact, that is the only way that a Function can generate Events, since Events are changes of State. The State must be owned by the Function (as for an Activity). The State must be bound to a Port of the Function in order for it to send Events outside the Function.

A Function may not read any of its own State values: a Function's States exist only so that the Function can generate Events that propagate outward. Thus, a Function's outputs are based solely on its inputs and not based on its internal state.

If a Function generates an Event, the Event propagates to other Functions in the same simulation epoch. That is, Functions introduce no simulation-time delay.

Functions may not generate *future* events. A Function may only generate an Event that takes effect in the same epoch as the Event that triggered the Function – that is, the current epoch.

Functions should be designed so that they are idempotent: that is, so that a function can be evaluated any number of times with the same inputs with no ill effects. Thus, it is inadvisable to model stateful processes such as counters using a Function; rather, use an Activity. The reason is that it is possible that a Function will be evaluated more than once during an epoch, and so the behavior could be undefined if the Function is not idempotent.

If a Function generates an Event during an epoch, and then the Function is invoked again *during that same epoch*, and generates another Event that conflicts with the prior Event, the later Event replaces the prior one such that Activities only receive the later Function Event.

This model makes it possible for race conditions to occur. *Expressway*<sup>TM</sup> does not currently detect race conditions, but allows the user to prevent them by setting a parameter that controls the maximum number of times that a Function may be invoked during any single epoch. This parameter may be set independently for each Function.

## **Closure of Native Implementations**

Model Components (both Activities and Functions) imply a level of closure<sup>1</sup> around information, and

---

<sup>1</sup> The term “closure” is used in the traditional sense, as defined in mathematics, and should not be confused with the

this closure should be respected by native implementations of Model Components. The dynamic (simulation-time) flow of information (contained in Events) may only occur through Ports, and along Conduits. There is no other way. A native implementation of an Activity or Function should not circumvent this intention.

## ***Structure is Statically Determinable***

The desire to retain closure in terms of communication into and out of a component means that we must be careful about allowing the dynamic (simulation time) creation of communication paths. Therefore, at this time, no dynamic creation of components or Conduits is supported. It is likely that at some point the model will be extended to allow the dynamic creation of statically defined structures, in a template-like fashion, so that the communication paths can still be statically determined.

## ***More on the Event Model***

An Activity or Function will never receive more than one Event for the same State during an epoch. [Verify prevention]

Two Events may drive the same State only if their values agree. [Verify detection]

Events that do not actually change a State's value may be pared by the simulator, but this is not guaranteed.

Two different States may drive the same Port (e.g., one from the outside and one from the inside) as long as they are not in the same epoch or are compatible. [Verify detection]

An Event created with time = null should be scheduled for the very next simulation epoch.

## **Activation of Model Components**

A Model Component is “activated” when its behavioral implementation is executed. Thus, only behavioral components can be activated.

Activation of a component occurs in response to an *external* event on a Port of the component. Feedback is allowed: that is, if a component generates an Event, and that Event travels outside the component and then propagates back to the component and arrives at a *different* Port of the component, the component may be activated in response.

Another way to think of this is that a behavioral component C will be activated in response to a change in State value if (and only if) there is a path (via Conduits) from an output-capable (output or bi) bound Port of the State owner to an input-capable (input or bi) Port of C. In the case in which C is the State owner, the two Ports must not be the same.

---

“closure” constructs that are defined by some programming languages.

## **Pre-Defined Events**

### **Initial State Values**

A State's value is undefined until an Event occurs on the State. Once an Event occurs on a State, the State is said to be driven.

At simulation startup each EventProducer is activated, as if it had received an Event, in order to ensure that each EventProducer has an opportunity to start driving its States. Thus, each Function and Activity can expect to be activated once in response to no Events. When an Activity is activated with no Events, it can respond or not respond, depending on what is appropriate to the behavior of the Activity.

Predefined Events also occur during the startup Epoch.

### **What Constitutes an “Event”**

An Event has two meanings: (1) a logical event, as defined by the behavior model; and (2) an object type used to implement a logical event.

A logical Event is only considered to exist when the value driving a State changes. However, an Event will only be propagated to other components if the value on one or more Ports will change as a result. Otherwise, the Event is effectively undetectable anyway: nothing changes, from the perspective of other components.

Let's consider some scenarios. If two Event objects are created for the same State, and they each specify the same value for the same epoch, there is only one logical Event. Similarly, if an Event object specifies a value for a State, and the new value is the same as the prior value of the State, then there is no new logical Event, and the Event object can (and will) be safely deleted and not processed. Finally, consider a case in which two *different* States are bound to Ports that are connected via a Conduit, and one of the States changes its value to 10. If the other State later changes its value to 10 also, the new Event will be propagated between the Ports, because there is no detectable change; but if the other State had changed its value to 11 (i.e., different than 10), then the new Event would have been propagated.

In other words, an Event only occurs when a change to a State's value changes, and an Event is only propagated when it is detectable by other Activities or Functions. If the value driving a Conduit does not change, then there is no Event, even if the Activity or Function driving the Conduit has changed.

Note that it is possible for a State to have a value that is different from the value that currently drives a Port. This is because incoming Events do not automatically update State: it is up to the behavior of an Activity (or Function) to detect the incoming Event and update its State if wants to. Conceptually, a State drives its bound ports only until another State “seizes” those Ports by driving them from the outside. The first State may or may not update its own value: this is up to the designer of the Activity (or Function).

## **Attributes**

Any element in a model domain may have attributes. (Attributes may even have attributes.)

An Attribute is not updated during a simulation. An Attribute value may be set prior to a simulation, or it may be set as a result of a simulation of a *different* Model Domain.

### **Attribute Value Determination**

An attribute's value does not change during simulation. However, its value (determined at the start of simulation) can be set in several ways. A model Attribute value is determined as follows, in decreasing order of precedence:

1. A constant value in a model, or
2. The value of a bound Variable in a Decision Domain, or
3. The value of a bound State in another model, or
4. A default value.

An attribute can be bound to a Decision Domain variable so that Decisions that affect the Variable are used to determine the attribute's value.

An attribute can also be bound to a State, so that when the model that defines the State is simulated, the attribute's value is set. However, the attribute may not belong to the same model domain as the State that it sets. Otherwise, there might be inconsistency if the State's value changes during simulation.

At the start of a model simulation,

1. All Attributes that are bound to a State are identified, and their value is obtained from the applicable ModelScenario and used to update the corresponding bound value in the ModelScenario being simulated.
2. All Decisions that impact the model (i.e., that have Variables bound to an Attribute in the model, and for which the Variable has a Decision that is part of a current DecisionScenario) are identified, and the corresponding Decision value is propagated to the Attribute value in the ModelScenario. Some of these value updates may overwrite (take precedence over) values obtained from State bindings.

At the successful completion of simulation, all model-calculated Attributes are updated in the ModelScenario.

### **Attribute-State Bindings**

The value of an Attribute in a Model Domain can be determined by the simulation results of another Model Domain. Specifically, the Attribute can be bound to a State in another Domain such that when the other Domain is simulated, the final value of that State is used as the value of the Attribute when the Attribute's Domain is simulated.

There is a problem with this simple approach though: the other Domain can be simulated many times, so there will be many final values for the State, one for each run of each Model Scenario: which should be used? The simulator uses the “current” Model Scenario for the State's Model Domain. The problem still remains, which simulation run to use? There are two options: (a) the State's final value from the most recent simulation run, or (b) the statistical average of the final values of each existing run of the most recent scenario.

## ***Simulation and Simulation Runs***

### ***Watching for Race Conditions***

A race condition is when the model, or some portion of it, gets into a pattern of state changes that repeats indefinitely. Race conditions are analogous to an “infinite loop” situation that is often encountered in procedural programming. Race conditions are more difficult to detect, however, because they involve multiple concurrent activities, each with its own state. It is the combination of states, as they affect each other over time, that can become a race condition.

Since Expressway is an event-oriented simulator, race conditions can only manifest as a discrete number of iterations over which a state of the model (or some portion of it) repeats. A race condition of this kind is easiest to detect if the number of iterations between repetitions is small, ideally one, which corresponds to a case of direct feedback from one port of a component to another port of the same component.

The general problem of detecting – let alone diagnosing – race conditions is a difficult one. However, certain types of race condition can be prevented, by having the simulator detect them, and also by designing components to detect them. For example, components can be designed to detect and prevent unintended feedback by excluding events that originate from the component itself, even if the event travels a circuitous route outside the component to arrive back at it. The built-in components are designed to detect and prevent feedback.

*Expressway*<sup>™</sup> will detect direct feedback, but it only provides a warning and does not prevent it since sometimes it is desired. For example, a Generator (discussed below) often feeds back the output of its Count port into the Input port, in order to trigger the Generator again. The intended semantics of the component determine whether infinite repetition of some type is intended.

Since a race condition occurs unexpectedly, it is a good idea to always set a limit on the number of iterations in a simulation, in case the simulation “gets stuck” in a repeating set of states and does not progress.

## ***Pre-Defined Activities and Functions***

### **Terminals**

A Terminal is a pre-defined Function that, upon activation from an external event, merely reads all of

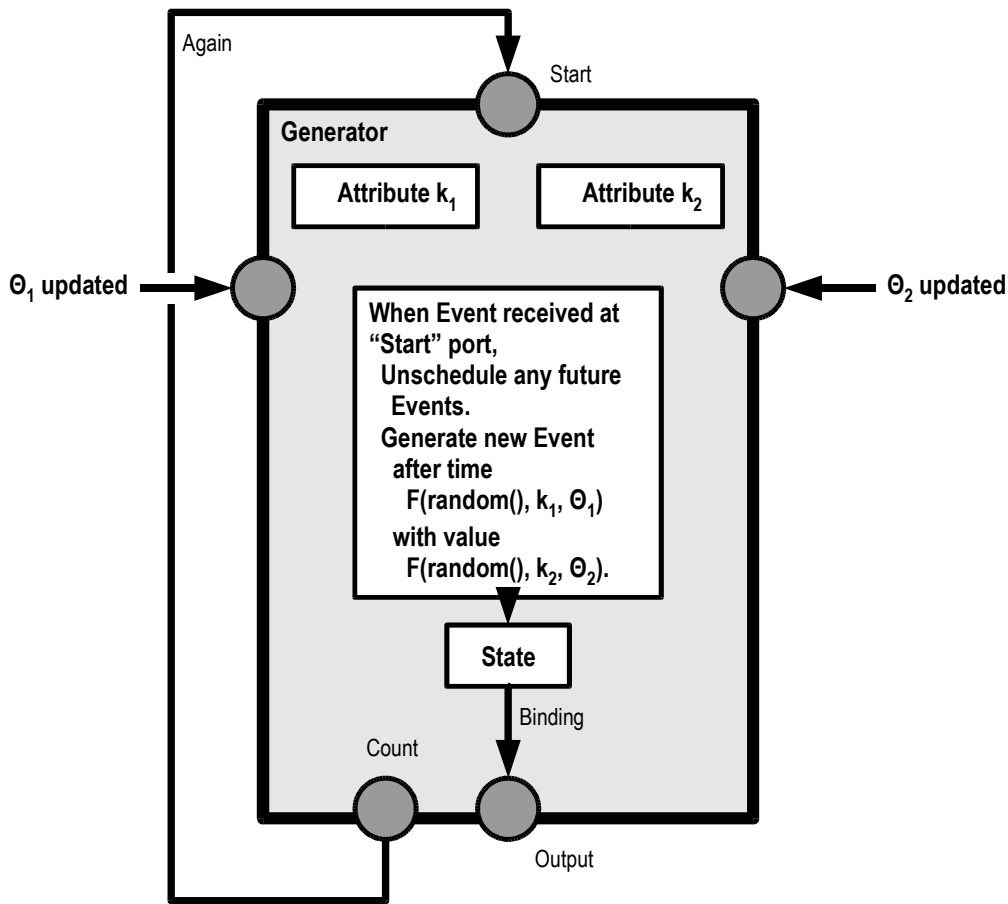
its Ports, selects the one that is currently driven, and then broadcasts that value to all of its Ports. More precisely, upon activation, a Terminal does the following things:

1. Selects the Port that is currently driven (it is an error if multiple Ports are driven with different values during the same epoch).
2. Updates its internal state with the value of that Port.
3. Since all of the Terminal's Ports are bound to the internal state, all then receive events (in the current epoch) containing the new state value.

## **Generators**

A Generator is a pre-defined Activity that generates Events according to a parameterized statistical distribution. A Generator maintains an internal State to which its Events apply. There are Generators pre-defined for various types of distribution, and for both discrete and non-discrete State values.

For any Generator, there are actually two distributions involved: (1) the distribution of time between changes in value, and (2) the distribution of values.



*Illustration 2: A Generator is a pre-defined Activity that produces events according to (1) a statistical distribution  $\theta_1$  for the period between each event, and (2) a statistical distribution  $\theta_2$  for the value of each event. A Variable Generator (shown) allows the distribution parameters to change during simulation. (The “Count” port updating is not shown for simplicity. It sends an event in the same epoch as the out put event, with the value that was sent to the “Start” port, plus one.)*

### **Startup of Generators**

Like all Activities, Generators are activated at startup to get them going. However, there are many circumstances in which you might want a Generator to remain dormant until a particular situation arises, at which point you would want to wake it up. Therefore, you can configure a Generator to ignore simulation startup. If a Generator is configured to ignore startup, the Generator will not actually generate an Event unless it is told to do so, by sending an Event to its “Start” Port. The value of this Event should be integral (e.g., 0), and must be numeric, because the output Port “Count” is given this value plus one (1). Further, a Generator will respond to the first non-null valued Event, and after that only to input Events that are *rising* – that are an *increase in value* over the most recent input Event.

## **Dynamically Changing a Generator's Statistical Parameters**

It is very useful to be able to modify the statistical distribution of a Generator in response to a change in some condition. This allows one to model situations that have changing behavior, such as when a change in the security of a system reduces the rate at which security breaches occur.

If a Generator is declared as “variable”, one may modify its distribution parameters through two special Ports for that purpose: one for changing the time distribution scale, and the other for changing the value distribution scale. One may not dynamically change the shape parameter of a distribution.

If one changes a distribution parameter during simulation, future Events for that Generator instance are cancelled, since they represent the prior distribution. A new future Event will immediately be scheduled if one has been cancelled, or if a rising value is presented to the Generator's input Port.

## **Generation of Repeated Events**

To cause the Generator to generate more than one output Event (this is what you normally want), you must connect (by using a Conduit) the Generator's Count port to the Generator's Start port. This causes the Generator to start again after an output Event actually occurs. This is the normal configuration. Also, make sure that you do not continue to send other Events to the Start Port, as these will also cause the Generator to be invoked.

In the rare circumstance that you only want the Generator to generate a single Event, do not connect its Count Port to its Start Port.

If an Generator has received an Event on its Start port and has not yet generated an Event, it is an error for the Generator to receive another Event on its Start port. However, at present the Expressway simulator cannot detect this condition and the outcome is undefined.<sup>2</sup>

## **What a Generator Generates**

A Generator, like any Activity, drives the value of a State variable. In an Expressway simulation, it is the value of the State variables that matter. Events are merely changes to those values. For example, if an Activity generates an Event that does not actually change a State's value, the Event is not considered to be a true Event and might not be propagated by the simulator.

Conceptually, what a Generator generates, then, is not Events, but changes to the value of the Generator's internal State. Mathematically, the State can be thought of as a series of step changes. Each step change endures until the next step change.

## **Pulse Generators**

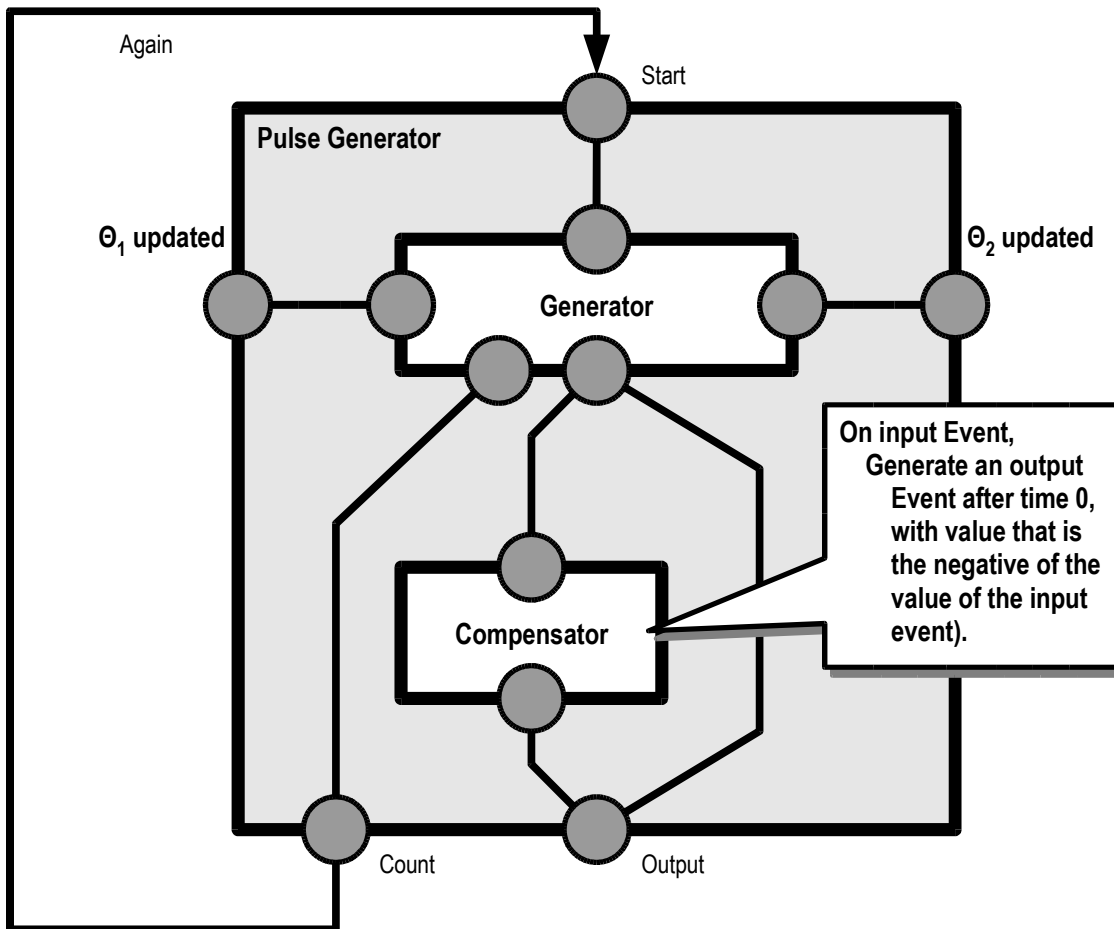
In contrast to a Generator, which generates a step function, a Pulse Generator generates a step change that endure, a Pulse Generator generates a series of infinitely short step changes. Each change is immediately followed (in the very next epoch, after zero time units) by a compensating change in the

---

<sup>2</sup> What actually happens in the current implementation is that the Generator will start to behave as if it were two Generators, each generating Events.

opposite direction, that drives the Pulse Generator's State back to zero (0). Thus, a Pulse Generator generates a series of pulses in the State's value, each of zero duration, and of height determined by the Pulse Generator's value distribution.

These pulses can be very roughly thought of as analogous to “delta functions”. However, the analogy is a weak one, since true delta functions are infinite in height and it is their *integral* that would be equal to the pulse value.



*Illustration 3: Conceptual model of a Pulse Generator.*

Pulse Generators are useful if one wants to model a series of events using Expressway's inherently state-oriented model. If a Pulse Generator is used to generate a series of Events, those Event values can be tallied by a Tally component. If a standard Generator were used, successive Events of the same value would be missed.

### Tallies

A Tally adds the values presented it to it and maintains a sum, or “tally”.

A Tally requires a separate input Port for each source of inputs. Otherwise, if a single input Port were used, then all sources of input would have to be connected to that Port, and identical-valued inputs would be treated as redundant by the simulator and discarded, and different-valued inputs would be treated as a conflict. Note also that each source of input should only generate one input to a Tally during any simulation epoch: to do otherwise is to violate the simulator's Event model.

Another thing to note is that a Tally sums the *unique* Events that enter each of its input Ports. Thus, if a single Event enters through more than one input Port, the Tally will only tally the Event once, so that it avoids double-counting. If this behavior is not desired (i.e., if double-counting should be allowed), then one should set the “double\_count” attribute to true.

## Delays

## Sustains

A Sustain is a pre-defined type of Activity. A Sustain responds to a rising change to its input, and generates that as an output for a period of time, which is a random variable according to a specified distribution. After the time has passed, the Sustain's output drops to zero (or false, if the input is Boolean). The input to a Sustain must be Numeric or Boolean.

Note that a Sustain's output can be over-ridden if another component starts to drive any connected Conduits before the Sustain's period has expired.

## Switches

A switch is a pre-defined Activity that acts just as a switch in the physical world: if it is closed (“conducting”), it conveys something, but if open (“non-conducting”), it conveys nothing. In our case, a closed switch conveys the driven state of a conduit. To open or close the switch, one sends a binary event to the switch's Close port: true to close, and false to open.

The purpose of a switch is to open or block the effect of one or more other components. This is useful if there is a condition that must be met for a source of value to be usable, and so the component that generates that value can be blocked by a switch until the condition is met.

A Switch is an Activity, so it conveys an Event in the next simulation cycle, if at all. An important case is when a Switch receives an Event, but the Switch is open. Since the Switch is open, it blocks the Event. However, if the Switch is later closed, it then determines the current state of its input port and generates a new Event on its output port to match the value driving the input port. Thus, Events are not “lost” when a Switch is open.

A switch starts out in an open (non-conducting) state, unless its *start\_conducting* option is true.

## Modulator

A Modulator is a pre-defined Activity that simply multiplies its input by a specified factor and

generates an Event on its output with that new value. The factor can be pre-set, or it can be dynamically changed. A Modulator is useful if you need to set the scale of an input, or if you need to adjust the effect of something dynamically.

## **Discriminator**

A Discriminator is a pre-defined Activity that generates a true or false output, depending on whether its input is greater or less than a specified threshold value, respectively. The threshold value can be preset, or it can be changed dynamically. A Discriminator is useful if you need to determine if a value is greater or less than a certain amount.

## ***Reflections on the Event Model***

Some readers will notice that the event model is very similar to the model used by electrical engineering simulation tools. This is not deliberate, and I have experience in both the electrical engineering and system modeling domains. It has turned out this way because the electrical engineering model supports concurrency, and continuously changing real values, and that is what we tend to have in business value models, since value flows over time, just like electricity.

For example, there was a choice of whether to allow an event to propagate even if the source value is unchanged, and that is what many system simulation approaches would allow. That approach is more event-oriented than value oriented; but it has the disadvantage that continuously changing values are harder to represent. Consider the case of a flow of value that is blocked by some gating condition: when the condition is eventually such that the flow opens, an event-oriented model would have to process the open event received from the gate and adjust the accumulated value accordingly, whereas a value-oriented model would receive notice that the flow was beginning, and this notice would be received from the source of the flow rather than from the gate. Thus, it is more natural for modeling flows.

On the other hand, there are situations, even in a value-oriented model, when one wants to represent events. This can be achieved in a number of ways, including the generation of pulse values (“delta functions, if you will). Some components therefore allow the modeler to choose whether to generate a continuous value or a pulse.

For those who are interested, an event model is the mathematical “dual” of a flow model. In other words, they are logical equivalents, and convertible into each other.

# Understanding Decision Domains

*Decision Points*

*Decisions*

*Choices*

*Variables*

*Variable-Attribute Bindings*

*Decision Scenarios*

*Decision Propagation*

## Expressway Input Syntax<sup>3</sup>

Names may not contain embedded periods, parentheses, or dollar signs (\$), but they may contain any other character. The simulator is case-sensitive.

**<expressway>**

Sub-elements:

- <model\_domain>** (optional)
- <define>** (optional)
- <decision\_domain>** (optional)
- <model\_scenario>** (optional)
- <decision\_scenario>** (optional)
- <simulate>** (optional)
- <update\_variable>** (optional)
- <export\_events>** (optional)
- <export\_final\_states>** ( optional)
- <export\_stats>** (optional)
- <export\_histograms>** (optional)
- <export\_correlations>** (optional)

**<define>** – Define a lexical symbol. If any attributes reference the symbol, by providing the symbol's name in parentheses prefixed with a dollar sign (\$), the reference is replaced by the symbol value.

Attributes:

- name –
- value – May not contain special characters dollar sign (\$) or parentheses, but may contain periods.

**<model\_domain>** –

Attributes:

- name –
- strict\_state\_ownership (optional) – Boolean, indicates if the model domain requires “strict state ownership”.  
Default: false. (This option is not implemented yet: current behavior is false.)

Sub-elements:

- <attribute>** (optional)
- <activity>** (optional)
- <function>** (optional)
- <terminal>** (optional)
- <generator>** (optional)
- <delay>** (optional)
- <tally>** (optional)

**<attribute>** –

Attributes:

- name –
- default (optional) –

---

<sup>3</sup> The XML syntax used by the current release of Expressway will eventually be replaced by a design and modeling language. The language will compile to the core Expressway simulation model that is represented by this XML syntax.

Sub-elements:

- <attribute> (optional)
- <state\_binding> (optional)

<state\_binding> –

Attributes:

- state –
- kind – one of “average” or “most\_recent”.

<activity> –

Attributes:

- name –
- native (optional) – The fully qualified name of the native implementation class.
- native\_path (optional) – A classpath for locating the native class file. Each path must be a file location. Paths are separated by colons.

Sub-elements:

- <attribute> (optional)
- <state> (optional)
- <activity> (optional)
- <function> (optional)
- <terminal> (optional)
- <generator> (optional)
- <delay> (optional)
- <tally> (optional)
- <port> (optional)
- <conduit> (optional)

<function> –

Attributes:

- name –
- native (optional) – The fully qualified name of the native implementation class.
- native\_path (optional) – A classpath for locating the native class file. Each path must be a file location. Paths are separated by colons.
- max\_evals (optional) – The maximum number of times that the function will be evaluated during any epoch. Setting this prevents race conditions. Must be greater than 0. Default: 5.

Sub-elements:

- <attribute> (optional)
- <state> (optional)
- <function> (optional)
- <terminal> (optional)
- <port> (optional)
- <conduit> (optional)

<terminal> –

Attributes:

- name –

**Sub-elements:**

<attribute> (optional)

<generator> – An activity that generates output events according to a gamma distribution. The time between events is characterized by the time\_shape and time\_scale parameters, and the (double) value of each event is characterized by the value\_shape and value\_scale parameters, respectively. A generator has two implied ports: “input\_port” and “output\_port”. If “repeating” is true, these are connected together via an implied conduit, so that the generator is re-triggered each time it produces an event. A generator also has an internal state called “GeneratorState”.

Attributes:

name –

variable (optional) – boolean to indicate whether the generator has ports to allow the time scale and the value scale to change dynamically during simulation. Default is false. If true, the generator has implied ports called “time\_scale” and “value\_scale”.

repeating (optional) – boolean to indicate if a conduit should be generated to connect the output port to the input port. Default is false;

time – The time interval between generated events, starting from when the generator is started. Must be specified if and only if time\_shape and time\_scale are not specified. If the generator is variable, the generator's implied “time\_scale” port adjusts the time attribute.

time\_shape –

time\_scale – In days.

value – The value of each generated event. Must be specified if and only if value\_shape and value\_scale are not specified. If value is specified, then “pulse” must be true. If the generator is variable, the generator's implied “value\_scale” port adjusts the value attribute.

value\_shape –

value\_scale –

ignore\_startup (optional) – If true, ignore simulation startup. Default: false.

pulse (optional) – boolean. If true, then a second event is generated immediately following the first by zero time. Thus, the first event becomes a “pulse”, or “delta function”, of zero duration. The second event returns the generator's state to the value that it had immediately prior to the first event, or if it had none, then its default value; if there is no default value or prior value (they are both null), then the second pulse is not generated. That situation will occur when the generator is activated at simulation startup. If pulse is false, a second event is not generated. Default is false.

Note: If a generator is declared to be “pulse”, then the generator is actually modeled as a generator nested within another generator. Each nested generator has the same name. Therefore, if an export statement (or any other statement) needs to refer to the generator's internal state (“GeneratorState”), the generator name must be repeated twice within the path name. For example, if a generator called “myGenerator” is “pulse”, then its internal state is identified by the path (relative to the enclosing component) of “myGenerator.myGenerator.GeneratorState”. If the generator is not “pulse”, then its relative path is merely “myGenerator.GeneratorState”.

<delay> – Generate an output event, equal in value to any input event received. The output event is scheduled for “time\_delay” later.

Attributes:

name –

time\_delay – The delay inserted between the input signal and the output signal.

**Sub-elements:**

<attribute> (optional)

<sustain> – Generate an output event, equal in value to any input event received, for a random. A Sustain has ports input\_port and output\_port, and an internal State called “SustainState”.

Attributes:

name –  
time\_shape –  
time\_scale –  
time – If the time to sustain the output is constant, use this attribute instead of the time\_shape and time\_scale attributes.

**Sub-elements:**

<attribute> (optional)

<tally> – Each time an input event occurs, add its value to the current tally. The tally starts at the initial\_value. Each tally contains an implicitly defined state called “TallyState”.

Attributes:

name –  
initial\_value (optional) – Default: 0.  
double\_count (optional) – Default is false. If true, an event that enters through more than one port will be counted separately for each port.

**Sub-elements:**

<attribute> (optional)

<input\_port> – An input port must be specified for each source of input values to be tallied. The attributes and sub-elements of a input\_port are the same as for a port.

<switch> – If the switch's control port (“control\_port”) is true (the “conducting”, or “close” state), then the value driving the input port propagates to the output port (“output\_port”). If the control port is false (non-conducting), then the switch does not drive its output port, and its value corresponds to the most recent event that reached the port from any component. A switch introduces a delay of one simulation cycle, of time duration 0.

Attributes:

name –  
start\_conducting (optional) – either “true” or “false”. Default is “false”.

**Sub-elements:**

<attribute> (optional)

<modulator> – When a Modulator receives an Event on its input\_port, it produces an Event on its output\_port that is the product of the input\_port value and the value on the control\_port. These values must be numeric. The Modulator's internal state, which drives its output, is called "state".

Attributes:

name –  
factor (optional) – The factor times which to multiply the input to compute the output. If this is not specified, the factor is determined by a control port.

**Sub-elements:**

<attribute> (optional)

<discriminator> – Whenever the value driving a Discriminator's input\_port rises to a value that is equal to or greater than the value on its control\_port, the Discriminator generates an Event of value true. Whenever the value driving the input\_port falls below the value on the control\_port, the Discriminator generates an Event of value false. The Discriminator's internal state, which drives its output, is called "state".

Attributes:

name –  
threshold (optional) – The threshold for discrimination. If this is specified, then a control port is not generated.

**Sub-elements:**

<attribute> (optional)

<expression> – An Activity that evaluates a specified expression. At present, only numeric expressions are supported. The expression may contain references to any input ports that are defined, as well as numeric values. The result of the expression is placed on the special port “output\_port”. If any of the ports have null values, the result will be null. At this time expressions are currently limited to the arithmetic operators.

Attributes:

name –  
expr –

**Sub-elements:**

<attribute> (optional)  
<port> –

<not> – An Activity that logically a logical input that is presented on the pre-defined port “input\_port”. The result of the expression is placed on the special port “output\_port”.

Attributes:

name –

**Sub-elements:**

<attribute> (optional)

<state> –

Attributes:

name –

**Sub-elements:**

<attribute> (optional)  
<pre\_event> (optional)  
<port\_binding> (optional)

<pre\_event> –

Attributes:

time (optional) –  
value | attribute –

Sub-elements:

<attribute> (optional)

<port\_binding> –

Attributes:

port –

Sub-elements:

<attribute> (optional)

**<conduit>** –

Attributes:

name (optional) –

portA – Two possible forms: (1) “compABC.port123”, where “compABC” is a sub-component of the container that owns the conduit; or (2) “port123”, where “port123” belongs to the enclosing ModelContainer.

portB –

Sub-elements:

**<attribute>** (optional)

**<port>** –

Attributes:

name –

direction (optional) – one of “input”, “output”, or “bi”. Default: bi. A port direction may only be input or output if the component has a native implementation.

Sub-elements:

**<attribute>** (optional)

**<decision\_domain>** –

Attributes:

name –

Sub-elements:

**<decis\_point>** (optional)

**<precursor>** (optional)

**<dependency>** (optional)

**<decis\_point>** –

Attributes:

name –

native –

Sub-elements:

**<variable>**

**<variable>** –

Attributes:

name –

Sub-elements:

**<attr\_binding>** (optional)

**<attr\_binding>** –

Attributes:

attribute –

**<precursor>** –

Attributes:

source –  
target –

<dependency> –

Attributes:

source –  
target –  
kind – One of “precludes”, “requires”, “modifies”, or “co-dependson”. (At present, only “precludes” is supported.)

<model\_scenario> –

Attributes:

name –  
model\_domain –

Sub-elements:

<attribute\_value> – Specify the value of an attribute, for this scenario.  
<attribute\_distribution> – Specify a distribution for the value of an attribute; for each run, the simulator will select a random value for the attribute, according to the distribution.  
<attribute>

<attribute\_value> –

Attributes:

path –  
value –

<attribute\_distribution> (not supported yet)

Attributes:

kind – the type of distribution, e.g. “gamma”. (Only “gamma” is supported at present.)  
parameters – a comma-separated set of double values, defining the distribution. E.g., for a gamma distribution, two values must be specified, as in “1, 2.3”.

<decision\_scenario> –

Attributes:

name –  
decision\_domain –

Sub-elements:

<choice> (optional)

<choice> –

Attributes:

decis\_point –  
variable –  
value –

<simulate> –

Attributes:

- name (optional) – the name of this simulation task. (This is used by other tasks that must wait for a simulation to complete.)
- model\_domain – the model domain to simulate.
- model\_scenario – the scenario of the model domain to simulate.
- max\_iterations (optional) – the maximum number of times to iterate during a simulation.
- max\_sim\_time (optional) – the maximum duration (days), in *simulation time*.
- initial\_epoch (optional) – the date/time at which the simulation time should start.
- final\_epoch (optional) – the date/time past which the simulation time should not progress.
- runs – the number of times to run the simulation. Each run is a separate, complete simulation, starting from the first epoch. Default is 1.

Sub-elements:

- <set\_current\_scenario> (optional)
- <watch> ( optional)

<watch> – When a change is detected in the specified element, information about the change is reported (to standard output), including the event causing the change, the activity or function that created the event, and the prior value.

Attributes:

- path – The path of a state or a conduit.

<set\_current\_scenario> –

Attributes:

- model\_domain – The model domain for which to set the current scenario.
- model\_scenario – The model scenario to set as the “current scenario” for the domain. The simulator will use the current scenario when interpreting attribute/state bindings.

<update\_variable>

Attributes:

- decision\_scenario –
- propagate –

<export\_events> –

Attributes:

- file – The filesystem path of the output file to generate. An extension is automatically added based on the format of the output.
- waitfor (optional) – The name of a task (e.g. a <simulate> task) for which this task should wait.
- format (optional) – A comma-separated list of “html”, “tabbed”, or “archive”. Default: “tabbed”. If archive is specified and if the archive file exists and is the file specified by the val\_archive\_file attribute, it will not be over-written.
- model\_domain – The model domain that owns the model to which the states belong.
- model\_scenario – The model scenario that was simulated to produce the events.
- max\_fraction\_digits – The maximum number of digits to be displayed to the right of the decimal for any event value written.
- epoch\_format – One of “date” or “number”. If “date”, then the epoch date/time is written; if “number”, then the epoch sequence number is written.
- val\_archive\_file (optional) – The path of a serialized file to compare the results against.

Sub-elements:

<path> – There may be any number of these. The path may specify a state, an **activity**, a **port**, or a **conduit**.<sup>4</sup> The value of the specified node is reported for each epoch. If the node is an activity, the value reported is merely “true” or “false”, indicating whether the activity was activated (or not) during each epoch.

<path> –

Attributes:

path – The fully qualified name of a state, activity, function, port, or conduit.

<export\_final\_states> –

Attributes:

file – The filesystem path of the output file to generate. An extension is automatically added based on the format of the output.

model\_domain – The model domain that owns the specified model scenario.

model\_scenario – The model scenario for which all existing simulation runs should be analyzed statistically.

format (optional) – A comma-separated list of “html”, “tabbed”. Default: “tabbed”.

waitfor (optional) – The name of a task (e.g. a <simulate> task) for which this task should wait.

max\_fraction\_digits – The maximum number of digits to be displayed to the right of the decimal for any value written.

Sub-elements:

<path> – There may be any number of these. Must be a state.

<export\_stats> – Write a statistical summary table of attribute, variable, or state values to a file, based on all scenarios.

Each row is a node, and each column is a scenario. Each table entry is a set of values determined by the options below. The file format is tab-delimited.

Attributes:

file – The filesystem path of the output file to generate. An extension is automatically added based on the format of the output.

model\_domain – The model domain that owns the specified model scenario.

model\_scenario – The model scenario for which all existing simulation runs should be analyzed statistically.

format (optional) – A comma-separated list of “html”, “tabbed”. Default: “tabbed”.

options – Any or all of “mean”, “geomean”, “max”, “min”, “sd”, “gamma”. (Note: gamma is not supported yet.) For example, one might specify options=“mean, sd”. If one includes “gamma”, then the parameters of a gamma distribution are written, based on a best fit of the data. The confidence level of the gamma fit (based on a t-distribution analysis) is also written. It is an error if no options are specified.

waitfor (optional) – The name of a task (e.g. a <simulate> task) for which this task should wait.

max\_fraction\_digits – The maximum number of digits to be displayed to the right of the decimal for any value written.

Sub-elements:

<path> – There may be any number of these. The path specifies the fully qualified name of the state to analyze statistically.

<export\_histograms> – Export histograms of the final value (across all runs) of each of a specified list of States.

Attributes:

file – The filesystem path of the output file to generate. An extension is automatically added based on the format of the output.

model\_domain – The model domain that owns the specified model scenario.

model\_scenario – The model scenario for which all existing simulation runs should be analyzed statistically.

---

4 At present, only state paths are supported.

format (optional) – A comma-separated list of “html”, “tabbed”. Default: “tabbed”.  
bucket\_size – The width of each bucket of each histogram, in the implied units of each state's range of values.  
Must be numeric.  
bucket\_start – Must be numeric.  
no\_of\_buckets – Must be numeric.  
all\_values (optional) – If true, values that fall outside the buckets will be treated as an error. Default is true.  
waitfor (optional) – The name of a task (e.g. a <simulate> task) for which this task should wait.  
max\_fraction\_digits – The maximum number of digits to be displayed to the right of the decimal for any value written.

Sub-elements:

<path> – There may be any number of these. Must be a state.

<export\_correlations> – Compute the correlation coefficient or co-variance of pairs of states, based on the results of all simulation runs for the specified scenario.

Attributes:

file – The filesystem path of the output file to generate. An extension is automatically added based on the format of the output.  
model\_domain – The model domain that owns the specified model scenario.  
model\_scenario – The model scenario for which all existing simulation runs should be analyzed statistically.  
format (optional) – A comma-separated list of “html”, “tabbed”. Default: “tabbed”.  
options (optional) – One or both of of “cv” (for co-variance) or “cc” (for correlation coefficient). Default is “cc”.  
waitfor (optional) – The name of a task (e.g. a <simulate> task) for which this task should wait.  
max\_fraction\_digits – The maximum number of digits to be displayed to the right of the decimal for any value written.

Sub-elements:

<correlated\_states> –

<correlated\_states> –

Attributes:

path1 – The fully qualified name of a state.  
path2 – The fully qualified name of a state.

## **Using Expressway**

***Making a Business Case for an IT Strategy***

***Refinement Of Strategies Into Designs***

***Facilitating Traceability Between Actions, Strategies, and Business Value***

***Providing Input to Portfolio Planning***

# Open Enterprise Repository

For too long organizations have been held hostage to the companies that sell them products that rest on a proprietary repository of some kind. Proprietary repositories become a “data jail” of sorts, in that to get data in and out, one must hope that the vendor provides the interfaces needed, and those interfaces are generally proprietary so that one's application code is then dependent on the repository's API. The importance of an enterprise repository as a strategic platform for *operations* is becoming clear, and should not be based on proprietary interfaces. The time has come to define an open repository, designed with enterprise requirements in mind. This is the purpose of the Open Enterprise Repository (OER).

The mission of the OER is therefore *to enable and control access to all enterprise operational data according to an enterprise data model, in an open manner.*

The features of the OER are:

**Enterprise-class.** It must provide operational assurance, lifecycle stability, security, and scalability from the outset, and distributable. Highly durable, highly recoverable, highly reliable. Guarantee of backward support if any internal data formats change. High performance. It is hoped that some OER implementations will become security certified via Common Criteria.

**Open source design.** To avoid dependence on any vendor's APIs.

**Simple.** The APIs have been kept simple and streamlined, so that the learning curve is minimal. Also designed to be lean: and avoid the “bloat” phenomenon in software today. Achieved through keeping the repository focused on its mission.

**Enterprise object model.** A standard (“core”) enterprise object model, that can be extended, and is defined in XML.

**Enterprise security model.** Addresses security as a foundational enterprise requirement head-on. Built-in rule- and role-based security model, with hierarchical realms, defined separately from the enterprise object model.

**Portlets.** A portlet interface for easily creating Web 2.0-style UI applications, using any Web server.

**Standard SOA interface.** A *simple* SOA interface based on JSON, for getting data in and out.

**Addresses varying data consistency needs.** Some applications require a “snapshot” view of data, e.g., all data as it existed at a certain time, or the very most recent data. The repository provides these views natively so that applications do not need to build them.

**Any database.** Allows plug in of any underlying database model, including object-oriented, relational, and column-oriented. Applications use an object model or SOA services.

**Manageable.** Adapters for policy management systems, and a secure SOA interface for management functions. Designed for low administration with regard to the underlying database. The ability to make schema enhancements without performing ETL.

**Migratable.** If an organization changes its preferred underlying database, it is possible to migrate

the repository to the new database without writing special ETL applications. This is because the repository provides the ETL features needed, even to migrate to an entirely different kind of underlying database (e.g., from relational to OO, or relational to column-oriented).

**Any file system.** Can leverage underlying journaling file systems or content management systems to recover snapshots of file-based objects such as documents and other file objects in the repository.

**Can feed warehouses.** Built-in system for replicating or sending data to an external relational system, so that ad-hoc applications can be run without impacting the operational repository.

**Can be provisioned.** Data center-ready: provisioning systems from Tivoli, VMWare, and others can be used to create repository server instances, accounts, and applications that use these.

**URIs for objects.** Defines a standard URI syntax for repository objects.

**Document publishing.** Built-in document publishing model, for exporting views of repository data using XML-based (“multi-channel”) document publishing products or open source tools. That is, you can control these tools through the repository interface.

*Expressway™* provides a fully supported OER, as an optional component of its products.

## **OER Core Schema**

### **Enterprise Meta Model**

### **Security Model**

### **Architecture Meta Model**

Decision tree or framework (taxonomy, choices).

Decisions and intent (about design or approach, structural and behavioral). (Strategies are highest-level decisions.)

Performance predictions.

Patterns (to ensure intent, and for reuse of strategies and designs).

