Information from 2024-2025 work:

Background information about EMRALD

EMRALD is an open-source, time-based PRA software. Instructions for installation can be found on its [github](https://github.com/idaholab/EMRALD), and it also has a [web-based UI](http://emrald3app.inl.gov) on which users can build and edit models. When accessing the web page, the UI is equipped with a “default model” which is useful to play around with to understand how the models are set up. To open an empty model, click “Project/New” in the top ribbon. For further instruction, if you click “help” in the top ribbon, you will be directed to tutorial documentation for EMRALD. A model’s information is coded in json (the files use .emrald extensions), and the simulation engine is coded in C#.

At the core of an EMRALD model’s function are diagrams and logic trees.

* Diagrams: these can represent the components of your plant, subsystems of the plant, or high-level plant response. A diagram is composed of several “states,” each of which has “events” and “actions” associated with it. It can be useful to think about events as causes and actions as effects. For example, a user can create a “timer” event such that after a defined amount of time, the event would *cause* its diagram to transition from a “startup” state to a “normal operations” state. That transition would be the *effect*, the action. There also exist “immediate actions,” which execute immediately when the simulation enters the action’s respective state.
  + Users can set diagrams to be either “multi-state” – capable of occupying many states at once but cannot be evaluated by logic trees – or “single state- evaluation” – capable of being evaluated by logic trees, but only occupying one state at a time. You may find that most diagrams end up being single-state, especially those that represent lower-level components. For single-state components, each state is assigned a boolean value (or “unknown”), and there must be at least one “true” state and one “false” state.
* Logic trees: these are effectively a tool to simultaneously track several diagrams’ failure conditions in a streamlined manner. Logic trees are made of nodes; at the lowest level are component nodes, and higher-level nodes are called gates. Component nodes adopt the boolean value of whatever state that component is in, and this value propagates up the parent gates until it reaches the top. Gates may be OR, AND, or NOT. To incorporate these logic trees into the model’s diagrams, create an event and select from the dropdown menu “component logic.” You will choose to evaluate a “success tree” or a “failure tree.” For success trees, a component in a True state has a value of True in the logic tree, but for failure trees, a component in a False state has a value of True in the logic tree.

Another useful power of EMRALD’s is its variables. Users can create variables whose values will be tracked by the simulation, and they can reassign variable values using actions. These actions are programmed in C#, and whatever value the action’s program returns will be stored in the specified variable.

Background information about our model

Our model is one of a nuclear power cycle coupled with a zero liquid discharge (ZLD) system and a thermal energy storage (TES) system. On the low level, each component has a failure probability modeled with a piecewise distribution. We use a Weibull distribution for the startup phase, an exponential distribution for the useful life phase– both of which are standard practice for reliability analysis– and a Gompertz distribution for the degradation/wear-out phase– which is a less common choice, but the literature seems to support it, and the precursor model from previous work used it. Each piece of the distribution is modeled with its own state in its respective component’s diagram, and each such state has a failure event that triggers according to the distribution that we specify. These are the main input parameters to the model: the parameters of the failure distributions.

On a high level, the power cycle enters a planned refueling outage every 17520 hours (which is stored in the variable TimeToRefuel). When this happens, the TES system also enters an outage state, which then causes the ZLD to enter an outage state. Additionally, each plant has a logic tree that evaluates whether the plant as a whole has failed, i.e. entered an unplanned outage due to component failures. If the power cycle or ZLD fail, the TES enters an outage state. If the TES fails, the ZLD enters an outage, and the power cycle is interpreted to be in a low-power state. Each of these failure and outage states have variables to track the number of times the state is entered and the amount of time spent in the state (referred to as downtime in the variable names). The power cycle also has a low flow state that tracks when two out of three condenser or feedwater pumps have failed. These are the main outputs of the model: the number of failures for each plant and their downtimes.

As a side note, many (if not all) of the diagrams have a state called “dummy.” The dummy states are not connected to anything, so if a component is in it, then it is effectively inactive. Since EMRALD models are in json, they cannot use comments, so to “comment out” a diagram, make the dummy state the initial state, and make the actual initial state a standard state.

When the base model is complete, we can copy it and make changes to the system design, maintenance strategy, etc. and compare the output variables to gain insight into how well different systems perform against each other. One design choice of greater interest is whether to include TES in the coupled plant because it may provide a “buffer” between the power cycle and ZLD, i.e. if the ZLD fails, the TES can hypothetically support the power cycle running for longer with less risk of failure, improving its economic performance.

Tasks to accomplish:

1. Finish debugging the model
   1. The software has a bug such that when two states are evaluated at the same time, e.g. as a state-change condition to trigger an event, a number of problems can arise. These can include errors in accruing variables, failing to execute actions, or even halting the simulation run. Errors do not occur every time multiple states are evaluated at once, so there are likely additional conditions for the errors that are not clear. That said, avoid making the simulation evaluate multiple states at once. Rather, make one event for each state that needs evaluating if appropriate; it accomplishes the same thing in most cases.
   2. The big problem: the simulation halts spontaneously (usually after several hundred runs) when the events “Start\_ZLD\_Outage” or “Start\_TES\_Outage” are active (they can be made inactive by making them “var condition” events and programming the code to always return false).
      1. The issue does not arise every time multiple components are in the “degradation to preventative maintenance” transition, but it only happens during transitions like these, according to the debug log.
      2. The last line in the debug log is always “Set variable value: Maint\_Duration = {some integer value}.” This line prints when the action “Set\_Maint\_Duration” executes.
2. Input parameters we need:
   1. Maintenance times for: pressure vessel, nanofiltration, crystallizer flash chamber and slurry pump, electrodialysis
   2. Failure curves for: crystallizer flash chamber and slurry pump, electrodialysis
   3. To find parameters for electrodialysis: I have all the subcomponents in json format (all file names begin with “ElDi”). Upload them all into an empty EMRALD model and find the average failure rate and maintenance time of the system by running a simulation of the coupled components. The “ElDi membranes” component does not have its proper failure and maintenance parameters; we need to find those in literature before performing this test.
   4. For any component, when we find an overall failure rate, we can determine a failure distribution with the following process:
      1. Where MTF is the mean time to failure (found from literature), T is the mean time to degradation (hard coded as 24000 hours in the model). 0.1043 is the assumed shape parameter of the Weibull distribution, and 0.070413 is a function of the assumed shape parameter of the Gompertz distribution (0.020795). is the scale of the exponential piece of the distribution. Solve iteratively to find .
      2. Where is the scale of the Gompertz distribution and is the shape of the Gompertz distribution, 0.020795.
      3. Where is the scale of the Weibull distribution and is the shape of the Weibull distribution, 0.1043.
3. Other things to consider:
   1. When we perform maintenance on a component, it effectively returns to a “good as new” condition. Can we make a variable mean time to degradation to reflect maintenance leading to a “good as old” condition?
   2. How might we capture low flow conditions into the ZLD plant? Is that necessary?
   3. Could we design an MED system that employs bypasses between MED stages so that if one stage fails, the whole system would not necessarily fail?

Common issues with easy (but seldom obvious) fixes

1. If you delete a component, you may see some events automatically deleted when they shouldn’t be. The easiest way to fix this that I’ve found is to go into the json code and use CTRL+F to find the places where the names need to be restored, and you can copy-paste the event itself from a previous version.
2. If the simulation fails to begin running (the engine says “run 0 of n”), check if the debug log is open. If it’s open, the simulation won’t run.
3. If there is a compiling error for code within the model (e.g. in an action that reassigns a variable value), first check that none of the variables have illegal names. Then, check that each variable used in the code is marked on the right sidebar in the UI when editing the code. Having these variables marked is essentially how the C# in the model recognizes the variables as existing objects.
4. If a logic tree seems to disappear in the UI, then its top node may not be recognized as a top node. Check that its top node’s json object contains the key and attribute “isRoot”: true.
   1. Likewise, if you want to use a logic node in a logic evaluation event, that node must be recognized as a root. Fortunately, any node can be a root, even if it is not at the top of its logic tree, but you must set this by directly editing the json.