PetriNetSimulate Documentation

Chris Dunne, MRes, MEng

December 9, 2024

PhD Student - CDT Future Propulsion and Power
NCCAT, Loughborough University
c.dunne@lboro.ac.uk

Contents

1 README **2**

2 Component Nets and Phased Mission Logic **2**

3 Main Algorithm **3**

4 Adaptation for Repairable systems **4**

Version History

Version 2.0 - 2024/12/02

- Simplified subnet definition with separate import matrices.
- Compressed A-matrices to support any numbering convention without increasing matrix size or computational cost.
- Enhanced parallelization handling for improved performance.

Version 1.0 - 2023/11/23

- Initial full release.

Version 1.0 - 2020/01

- First developed for Loughborough University Module TTD100 - Advanced Risk and Reliability Assessment.

---

1 README

V2: 2024-Dec-2nd Chris Dunne MEng, MRes

c.dunne@lboro.ac.uk

The following is a parallelised MATLAB code for simulating petri nets for a phased mission. Please note results have not been verfied but appeared to be within expected limits and logic has been tested using the optional "opts.debugNetByPlotting = true" statement, which shows how tokens move through each net on a graph.

Connectivity between places is specified in 'A' matrices, and times to failure are generated from exponential/weibull/normal distributions. Data for these distributions is read in as stored as excel spreadsheets which are read in and assembled to a using the "AssembleAGlobal" function. All other parameters are specifed at the start of the 'initialiser' scripts. Assumes all components active (could fail) during any phase (but will not cause phase failure if they are not in the phase petri net)

Numbering:
Components' working places are always the first places to be defined (e.g.1:10). Components' failure places must then be the next integers following the working places (e.g. 11:19). Subseqeuent places in the net should be ordered methodically but order does not matter so long as A-matrix correctly programmed.

Uses innovative method of keeping componenet failure nets (2 places 1 transition each), seperate from the main petri nets, allowing the failed nature of each component to carry across to all new phases.

Also built for subnets, which allow for additional repeated subnets to be created and used in multiple phases. These are imported a separate matrices and built into the global A-matrix automatically.

2 Component Nets and Phased Mission Logic

The code has three main inputs to simplify the creation of the 'global' A matrix. Phased missions require knowledge about which components have already failed to be stored when moving to the next phase. To handle this, this code has separated the 'component nets' from the rest of the A-matrix which is solved for. Component nets have two places and one transition per component.

If the phase times being too short creates an error, there is likely an issue with the "small_" constant being too big for your phase duration. Be defauly its defined to be SimTime/1e5 but this may be too big if phases are considerably shorter than mission duration. Try making it smaller.

3 Main Algorithm

Algorithm 1 Phased Mission Petri Net Simulation

Initialize simulation parameters:
Load connectivity matrices, failure data, and configuration options.
Setup global matrices for places and transitions.
Initialize the marking vector for places ($M_{0,Global}$) and failure tracking.
Setup parallel processing environment:
if ParallelSim **then**
    Initialize a parallel pool with specified number of processors.
end if
Simulation Loop:
for each simulation run (from 1 to $MaxNSims$) **do**
    Initialize phase parameters:
    Set initial phase $P = 1$, system time $t_{sys} = 0$.
    Load phase-specific connectivity matrix and determine phase failure places.
    Generate failure times for components:
    Calculate failure times using specified distributions or random values.
    Loop until simulation end (endtime or sys failed)
    while mission not failed or successful **do**
        Check for failure
        if tokens reach failure places in current phase **then**
            Register system failure and exit loop.
        end if
        if t_{sys} exceeds current phase end time **then**
            if final phase **then**
                Mark simulation as successful and exit loop.
            else
                Increment phase:
                Update phase end time, reload connectivity matrices.
                Reset phase-specific variables.
            end if
        end if
        Identify enabled transitions:
        Evaluate tokens in input places of each transition to find all enabled transitions
        Find all transitions to fire by comparing enabled transitions with their remaining time
        Calculate time progression:
        Determine time increment Δt from minimum firing time for enabled transitions and .
        Increment system time and remaining transition times by Δt.
        Fire transitions:
        Update markings of places based on global phase connectivity matrix A and T_{fire}.
        Copy component net failed place tokens to corresponding phase net places
        Compute InsertionVector from component net to phase net connectivity
        Copy tokens
        Ensure each failed place token is only copied once per phase:
        Adjust AllowNetCopying to prohibit future copying of tokens that were just copied
        if debugging enabled **then**
            Plot the current marking of the Petri net for debugging.
        end if
    end while
end for
Post-simulation analysis:
Compute failure probabilities and component failure statistics.
Save results and generate visualizations.

4 Adaptation for Repairable systems

The code for a phased-petri net simulator is in general not too hard to adapt for repairability. Unlike a phased mission, which ends after the mission either fails or all phases succeed, repairability requires that after mission phases, a repair phase is added. The main principle is that you will be simulating just 1 'simulation' but with a very large number of phases(working/maintenance/working...) ($\sim$ 1e6). Repair phases must be treated differently and hence must first be identified. E.g. if P=1:1e6, and there are 2 different phases (working and maintenance), use $R = mod(P, 2)$ which will be 1 for repair phases, and 0 for normal phases. You could then declare only 2 A matrices, saving memory, one for failure and one for repair, and when loading APhase, load A.Amod(P,2) instead. If you have 2 working phases and 1 repair phase, you could use $R = (mod(P, 3) == 0)$ and a differnt A-matrix index.

The repair phase behaves quite differently from other phases and requires some modification in the code. It depends whether the repairability is modeled with revealed or unrevealed failures, and whether the pool of maintenance engineers is limited. The simplest case is where all failures are revealed at the end of the main mission, and all are repaired before the start of the next mission. Where some component failures are unrevealed, or only certain components are inspected if the mission didn't fail, additional modifications must be considered.

You can then define transition durations to be your randomly sampled? Or constant repair times in the tRemainTransitions matrix, using a function like is done for the repair times.

You also need to reinitialise and reset the "component time remaining to failure" and the "repair time remaining" vectors using a new randomly sampled number set, so that means running the functions again, based on the current phase (mod p...) Do that within the next phase update loop rather than at the start of the script.

All the output logging data will have to be changed to averaged over all the phases, rather than just taking the final mission marking after all the phases are complete, as that is only representative of the state of the current mission/repair rather than any of the ones that happened before it.

4.1 Approaches:

4.1.1 Component nets

These are used to keep track of component failure even after moving between phases. They could be useful if implemented properly for repair. Normally at the end of a phase we copy tokens from component net output places to phase net input places (*componentNetsToOutputPlaces*) and then prevent further copying for the rest of that phase. For repair, this still needs to happen at the start of the phase, but at the end of the phase, the component nets need to have their tokens moved from failed to working. This could be treated as a 'move' operation rather than a 'copy', since tokens are already in the component failed place. One must be taken out of that place and put into the working place.

One way to achieve this would be to reverse the direction of component net transitions in the repair phase A-matrix, set the component transition times to inf, and then set them to zero once a token is in the component's 'repaired' place in the phase net (perhaps overcomplicated).

If you assume all components are repaired before the next mission, there is a very simple option. Just reset the marking to the all working state at the end of a repair phase.

If you're modelling the case of operate until failure, and then repair all components before next operation, you can simply model the repair phase by making tRemainTransitions, phase dependent, and adding repair times to the transitions between places in your repair phase net.

If not all components are repaired or inspected during the repair phase, you will need to consider

the transitions you inspect for time remaining before firing them. Create a boolean inspection vector of size (1,NGlobalPlaces), true in all component places which are inspected during the repair phase, and multiply by this when computing T_fire for a repair phase only.

4.1.2 No component nets

The corresponding places in the A-matrix used directly, with tokens moving from failed to working places, through a 'in-repair' place in the repair phase net. This would however require a lot of modification to the code as it was always written around component nets but is very possible for simple cases. This is most easy if you do not need to know which components failed, just the total number. Otherwise, component nets may be easier.

4.2 Other important advice

In all cases you must remember that during a repair phase, the component times to failure must not be allowed to continue to tick down as if it were a normal phase. You can solve this by indexing 1:Ncomponents where tTimeRemaining is updated, and not updating those transitions if R (R is true, where R=mod(P,2)==0).

Moreover, currently transition times are only initialised for a new simulation. This must be done after every repair phase in the loop now, using the generateTimesToFailure function. I recommend using a similar function to generate repair times even if they're all constants, for consistency and readability.

The main difference will be the condition you use for moving on to the next phase. That will now no longer be dependent on just phase end time but also on an 'all components repaired/inspected' condition if and only if phase is a repair phase.

You'll also need your phase times to be infinite if operating until failure, or repairing until all repaired.

If you have any pool of maintenance engineers, you'll have to declare that as a place and add the appropriate links in your A matrix, and set the place's marking to the number of engineers available at the start of every repair phase.