



Elements of System Dynamics

Antoine B. Rauzy

PART 4. SYSTEMIC DIGITAL TWINS

LECTURE 11. SCRIPTING AND OPTIMIZATION

LECTURE 11. SCRIPTING AND OPTIMIZATION

Key notions:

- Domain Specific Languages
- Scripting
- Combinatorial Optimization
- Meta-Heuristics
- WorldLabToolbox

Rational

When designing large systemic digital twins, it is often the case that the system under study contains a variable number of similar components. The objective of the study could be precisely to determine the optimal number of these components.

Designing by hand a model for each configuration of the system is both tedious and error prone.

It would have been possible to introduce constructs in Σ^{TM} to do some kind of "meta-modeling", i.e. to write parametric models that could then be compiled into actual models. This choice has been however rejected for three main reasons:

1. This would have complexify considerably the language and the compilation process, without being certain to implement the right features.
2. Scripting languages, especially Python, provide all the required features to implement at relatively low cost this compilation idea.
3. Moreover, Python can also be used to implement optimization algorithms.

"Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp." Greenspun Tenth's Rule 1993.

Learning Objectives

This lecture aims at presenting:

1. Domain Specific Languages and their application to Σ^{TM} scripting.
2. Some optimization algorithms and (meta-)heuristics applied to system dynamics.
3. The WorldLabTM Toolbox, which is a Python package dedicated to the implementation of domain specific languages on top of Σ^{TM} and of optimization algorithms applying on configuration spaces of Σ^{TM} models.

Agenda

Section 1. Case Study: Cascading Failures in Power Grid

Section 2. Domain Specific Languages

Section 3. The WorldLab Toolbox

Section 4. Optimization Algorithms and Metaheuristics

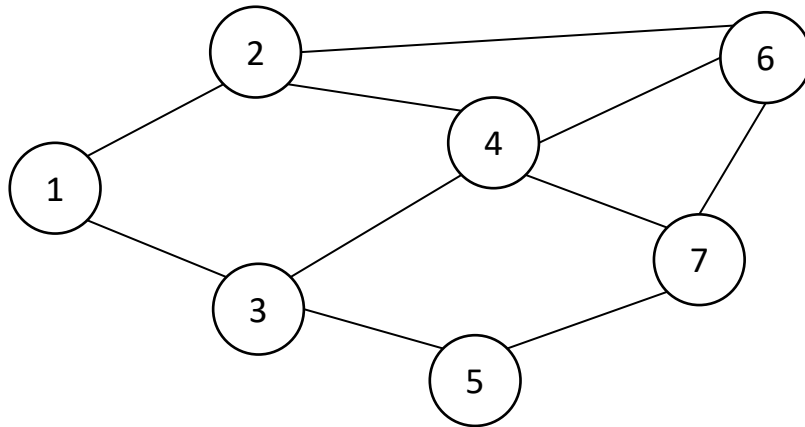
Section 5. Wrap-up

LECTURE 11. SCRIPTING AND OPTIMIZATION

SECTION 1. CASE STUDY: CASCADING FAILURES IN POWER GRID

Cascading Failures in Power Grids

We consider an abstraction notion of power grid:



The **power grid** is seen as a set of connected **nodes**.

Each node has:

- A **capacity** C , e.g. $C = 1$
- A **stress ratio** θ , e.g. $\theta = 0.8$
- A **current load** L .

Load balancing:

In normal operation, when the load L of a node exceeds its stress threshold θC , it transfers part of its load to its working neighbors. This transfer can only take place if it does not make target neighbors exceed their stress threshold.

Failures

Each node has two failure modes:

- **Intrinsic failure**: the node fails for some internal reason.
- **Overloading**: the node fails when its load exceeds its capacity.

Intrinsic failures

- exponentially distributed
- Base rate λ_{base} , e.g. $\lambda_{\text{base}} = 10^{-4} \text{ h}^{-1}$
- Additional rate λ_{stress} when L exceeds θ :

$$\lambda_{\text{stress}} = k \left(\frac{L - \theta C}{C - \theta C} \right)^n$$

k : stress sensitivity (scale parameter), e.g. $k = 0.1 \text{ h}^{-1}$

n : fatigue acceleration (shape parameter), e.g. $n = 3$

When a node fails, its load is equally split on its working neighbors, up to their capacity. What remains of its load is kept as a "debt" and is reintroduced when the node is repaired.

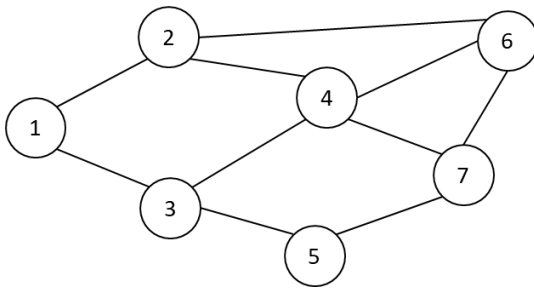
Repairs

When failed, nodes are repaired. The time to repair is exponentially distributed, with a known Mean Time To Failure, e.g. 48 hours.

When a node is repaired, its load is set to its "load debt".

Key Issue

The activity that describes the operation of a node depends on the neighbors of this node.



It would be both tedious and error prone to write the model "by hand".

```
activity Node.Operate
trigger:
  return state==WORKING and not operating;
start: {
  operating = true;
}
completion: {
  if load>capacity then
    state = FAILED;
  else {
    float failureRate, failureProbability, z;
    failureRate = baseFailureRate;
    if load>stressThreshold then
      failureRate += stressSensitivity*pow(
        (load - stressThreshold)
        /(capacity - stressThreshold),
        fatigueAcceleration);
    failureProbability = 1 - exp(-failureRate*dt);
    z = uniformDeviate(0, 1);
    if z<failureProbability then
      state = FAILED;
    }
    if state==FAILED then {
      float loadTransfer;
      // here split the load among the working neighbors
      loadDebt = load;
      load = 0;
    }
    else if load>stressThreshold then {
      float loadTransfer;
      // here transfer the load to the non-stressed,
      // working neighbors
    }
    operating = false;
  }
duration:
  return dt;
end
```

LECTURE 11. SCRIPTING AND OPTIMIZATION

SECTION 3. DOMAIN SPECIFIC LANGUAGES

Domain Specific Languages

A **Domain-Specific Language** (DSL) is a specialized computer language designed for a particular problem area or domain. This is in contrast to a general-purpose language (GPL), such as C++ or Python, which is broadly applicable across domains. The frontier between DSL and GPL is not that clear, but GPL are in general **Turing-complete**, i.e. they can emulate any computer calculation, if given enough time and memory.

DSLs focus on essential features for a **specific purpose**, making them more efficient, easier to learn for domain experts, and better for clear thinking within that field, with examples including SQL (databases), HTML (web pages), CSS (styling of web pages), and of course Σ^{TM} , WIDL and Tau.

DSLs are widely used in **Model-Based Systems Engineering**, with expected benefits:

- **Increased Productivity**: Faster development for domain-specific tasks.
- **Lower Barrier to Entry**: Enables domain experts to write code without deep programming knowledge.
- **Improved Maintainability**: Code is more readable and understandable within its context

Implementation

The implementation of most of DSLs, and especially those designed on top of Σ^{TM} , consists of three parts:

- Its **data structure** that reflects the **ontology** of the considered domain.
- Its **front end**, i.e. a way to describe models. It can be:
 - A full textual grammars, like those of GPL: rather heavy to implement and to learn.
 - A XML and Json grammar. These data formats are powerful, machine friendly but not very human friendly.
 - A spreadsheet, typically Microsoft Excel: well-mastered by practitioners but could suffer from lack expressive power.
 - A dedicated graphical user interface: user friendly but heavy to implement.
- Its **back end**, i.e. the calculation(s) performed onto the model/program, in our case the compilation of the DSL into Σ^{TM} and/or WIDL.

Ontology

In our example, the ontology consists of the following elements.

Nodes:

Nodes are the core elements of the power grid. They have a number of characteristics such as their capacity, their stress threshold, their base failure rate, their mean time to repair...

Edges:

Edges link two nodes. They define the neighborhood of a node. They make it possible to transfer the load of a node to its neighbors (when stressed or failed).

Power grid:

The power grid consists of a set of nodes and edges linking these nodes.

Excel Front-End

AutoSave Off PowerGrid2.xlsx No Label Saved to this PC Search

File Home Insert Draw Page Layout Formulas Data Review View Automate Help

Clipboard Font Alignment Number Styles

General Conditional Formatting Format as Table Cell Styles

Cells Editing Sensitivity Add-ins Function...

N10

	A	B	C	D	E	F	G	H	I	J	K	L
1	Code	Capacity	StressRatio	initialLoad	BaseFailureRate	StressSensitivity	FatigueAcceleration	MeanTimeToRepair	Neighbors			
2	N1	1	0.8	0.82	1.00E-04	0.1	3	48	N2, N3			
3	N2	1	0.8	0.71	1.00E-04	0.1	3	48	N1, N4, N6			
4	N3	1	0.8	0.78	1.00E-04	0.1	3	48	N1, N4, N5			
5	N4	1	0.8	0.79	1.00E-04	0.1	3	48	N2, N3, N6, N7			
6	N5	1	0.8	0.81	1.00E-04	0.1	3	48	N3, N7			
7	N6	1	0.8	0.78	1.00E-04	0.1	3	48	N2, N4, N7			
8	N7	1	0.8	0.77	1.00E-04	0.1	3	48	N4, N5, N6			
9												
10												
11												
12												
13												
14												

PowerGrid

Ready Accessibility: Good to go

Workflow

- 1 Initialize the "PowerGrid" data structure
- 2 Read the Excel spreadsheet and populate the "PowerGrid" data structure
- 3 Parse the base Σ^{TM} elements.
- 4 Create the Σ^{TM} model from the "PowerGrid" data structure and the base Σ^{TM} elements.
- 5 Export the Σ^{TM} model.
- 6 Assess the Σ^{TM} model.

Note: the WIDL model can be created along the same line.

LECTURE 11. SCRIPTING AND OPTIMIZATION

SECTION 3. THE WORLDBLAB™ TOOLBOX

What it is?

The WorldLab™ Toolbox is a Python package that implements classes and methods to:

- Import, export, create and modify Σ^{TM} models;
- Import, export, create and modify WIDL models;
- Call Σ^{TM} assessment tools (Σ^{TM} compiler, Σ^{TM} stochastic simulator, Σ^{TM} player, Σ^{TM} calculator);
- Perform statistics and other calculations on time series;
- Perform optimization procedures

The development of the toolbox is very incremental and still on going...

Installing the package:

- unzip the package in a folder
- open a command line
- change directory to the folder where the package is located
- `pip install -e .`
- Restart you Python interpreter

Designing a Σ^{TM} Model (1)

```
import WorldLabToolbox

core = WorldLabToolbox.Core()
designer = WorldLabToolbox.Sigma.Designer(core)

model = WorldLabToolbox.Sigma.Model()
designer.setModel(model)

designer.importFile("baseElements.sigma")

mainSystem = designer.newSystem("PowerGrid")
designer.push(mainSystem)

for node in powerGrid.getNodes():
    nodeInstance = designer.newInstance(node.identifier, "Node")
    designer.push(nodeInstance)
    designer.newParameter("float", "capacity", node.capacity)
    ...
    designer.pop()

designer.pop()

designer.exportModel("targetModel.sigma")
```

Designing a Σ^{TM} Model (2)

```
targetActivityName = "PowerGrid.{0:s}.Operate".format(node.identifier)
targetActivity = designer.newActivity(targetActivityName)
designer.push(targetActivity)
targetCompletionBlock = sourceCompletionBlock.clone()
ifFailedInstruction = targetCompletionBlock.getInstruction(1)
ifFailedBlock = ifFailedInstruction.getThenInstruction()
index = 0
for neighbor in node.neighbors:
    ifWorkingInstruction = designer.newInstruction("""
if main.{0:s}.state==WORKING then {{
    loadTransfer = min(main.{0:s}.capacity - main.{0:s}.load, load);
    if loadTransfer>0 then {{
        load -= loadTransfer;
        main.{0:s}.load += loadTransfer;
    }}
}}
""".format(neighbor.identifier))
    index += 1
    ifFailedBlock.insertInstruction(index, ifWorkingInstruction)
targetCompletionClause = designer.newCompletionClause(targetCompletionBlock)
designer.pop()
```

PowerGrid

