Dependency Language

by Sven Nilsen

Introduction

A dependency language encodes information about dependency relationships between objects, tasks, algorithms, people etc. The motivation for constructing a such language is to automate the process of self diagnostics in computer systems.

Logical AND and OR

A logical AND gate requires all inputs to be positive to give a positive result.

A	**B**	**A*B**
0	0	0
0	1	0
1	0	0
1	1	1

A logical OR gate requires only one input to be positive to give a positive result.

A	**B**	**A+B**
0	0	0
0	1	1
1	0	1
1	1	1

When we construct a dependency language, we will use these as logical building blocks. The goal is really simple: To detect whether a system is working or broken.

In a dependency language the inputs to a gate are dependencies. There might be additional information tied to the dependency information. A system that encodes dependency information is called an “expert system”.

Spreading of Risk

The logical OR gate is the key to risk management. Since this gate only relies one of its dependencies, we can have multiple inputs to reduce the risk of that gate failing. If each dependency is associated with a probability of failing, the overall probability of failure for an OR gate is:

$$P(\text{failure}\,|\,\text{OR}^n) = \prod_i^n P_i$$

For example, if there is a 90% chance of failure per input and we have 10 inputs, the probability of failure for all the inputs is approximately 35%.

The logical AND gate on the other hand is a collector of risk, because if just one of its dependencies fails it will fail too. The overall probability of failure for an AND gate is:

$$P(\text{failure}\,|\,\text{AND}^n) = 1 - \prod_i^n (1 - P_i)$$

For example, if there is a 10% chance of failure per input and we have 10 inputs, the probability of failure for any input is approximately 65%.

If we know the probability per time of failure for each dependency in a hierarchical structure we can compute the probability per time of failure for the whole structure and simulate failures without having to simulate the failure of each component. We can also compute the expected frequency of when the structure will fail and possibly correct probabilities by observing the system over time.

Interference

In Information Algebra there is an inference rule that goes as following:

$$(X \circ Z, Y \circ Z) \rightarrow (X, Y) \circ Z = X \circ Z \wedge Y \circ Z$$

A dependency language that supports inference is capable of composing two hierarchical structures together and return a new hierarchical structure that can be used to answer questions concerning both of the structures it is created from.

If both share no common dependencies, the return structure created by interference is empty. This leads to a possible invalid question when we want to know the probability of failure on an empty structure:

$$A \wedge A \rightarrow (() + A)$$

What is the probability of failure on an empty structure? If we look at the equations for AND and OR gates, the natural answer is 100% for OR gates and 0% for AND gates. The reason is the way you code a product look for probability. A mutable rational number is set to 1 and multiplied with each probability.

$$P(\text{failure} \mid \text{OR}^0) = \prod_i^0 P_i = 0$$

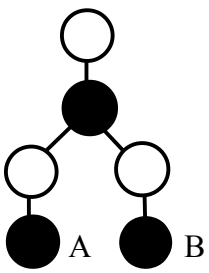
$$P(\text{failure} \mid \text{AND}^0) = 1 - \prod_i^0 (1 - P_i) = 1$$

We choose to start all structures with OR gate at the top and pick 100% probability for failure on an empty structure. This avoids introducing a new gate type. An empty is simply an OR gate with no inputs.

$$A \wedge A \rightarrow A$$

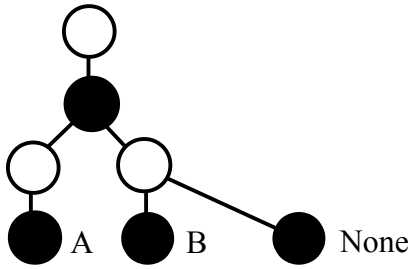
This leads to another decision: In a nested hierarchical structure, the leafs that are OR gates will fail, so all leafs that does not fail should be AND gates. The simplest structure we can make that does not fail is one with one OR gate (white) at the top and one AND gate (black):

When we read a Boolean algebraic expression “A*B” we can interpret it as a dependency graph:



Optional Dependency

If A is required and B is optional, we can create a dependency graph like the following:



A data structure with types as dependencies would look like this:

$$(A, (B+()))$$