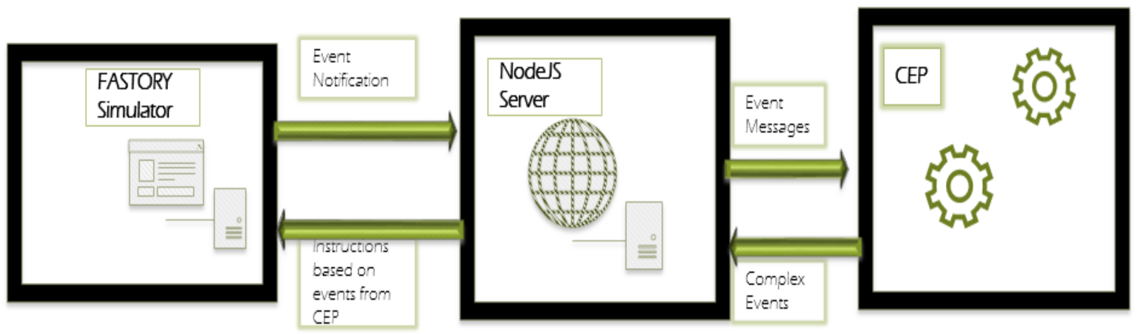
Runtime Orchestration of Products using Complex Event Processing

Architecture:



Problem definition:

Generally, when a certain number of products are ordered by the end user, those products are manufactured using the same resources due to limited number of resources available on the factory floor.

This sequential nature of production already increases the time needed for the production. Normally, the sequence of the products to be manufactured is hardcoded, and cannot be changed at run time. Therefore, for some reason if a production of a particular product is halted, it makes the situation even worse, as it affects the overall production time drastically.

Since, other products might use the same resource to get manufactured, dysfunction or malfunction of a resource will result in the accumulation of the products in the assembly line. Although, in a well-designed system there might be some bypass functionalities to avoid such circumstances, but what if even those bypasses are also jammed due to bulk number of orders. Hence, there might be a possibility of a possible queueing in the bypass system as well.

Work Description:

The main objective of this work is to create a system which processes the products in manufacturing state and determine whether the new product to be pushed in to the assembly line uses the same resources that are already in use.

If the product to be pushed uses the already devoted resources, then such a product is avoided from being sent in to the production line and another product is selected from an array of products still to be manufactured. The same procedure is repeated before feeding a product in to the assembly line.

Therefore, avoiding/reducing the probability of queueing in the bypass systems.

What is a **RECIPE**?

It is a set of instructions for manufacturing a product. It contains the necessary commands that are needed to be executed in order to perform relevant work on the product.

So, each product needs to implement a recipe in order to get manufactured.

Hence when a customer orders that it needs 3 number of products with some specifications X, and 2 number of products with some other specifications Y, what actually will be implemented would be an array of this sort:

[recipe_X, recipe_X, recipe_X, ……., recipe_Y, recipe_Y, ………].

id = 1 id = 2 id = 4 id = 10 id = 11

Procedure:

Two events are registered in to the CEP, EventA and EventB respectively.

EventA:

It represents the data associated with each product. It contains the following information:

RecipeId, ProductId, PercentageCompletion, TimeOfExecution.

Note: ProductId will be unique for every product.

EventB:

It contains the data regarding the next product (recipe ID) to be fed in to the production line.

It contains the following information:

RecipeId.

- ⇨ As operations are performed on a specific product, this information is converted to EventA and is fed up in to the data stream.
- ⇨ Hence, as the product is moving along the production line, it is sending information regarding its completion percentage.
- ⇨ EventB is sent to the data stream, before a new product could be sent in to the production line.
- ⇨ As soon as the Event Processor encounters EventB, it runs an **algorithm** to check whether the requested product (recipe ID of the product is used for analysis) could be sent for production or not, and reply with a BOOLEAN value of true or false. TRUE indicating a green signal, whereas FALSE specifying that this product must not be pushed in to the assembly line.

- ⇨ **Variables used in the Algorithm:**
- ➢ Congestion on the assembly line. (specifying whether the assembly line has enough space to handle a new product).
- ➢ How many products having the same Recipe ID are already there in assembly line?
- ➢ What is the percentage completion of the products having the same Recipe ID?
- ➢ Distance of the latest product having the same Recipe ID from the start point of the assembly line.

Use Cases:

FASTORY simulator is used to implement the given functionality.

http://escop.rd.tut.fi:3000/fmw

An array of recipes is constructed as follow:

[recipie#1, recipie#1, recipie#1, recipie#1, recipie#1, recipie#2, recipie#2, recipie#2, recipie#3, recipie#3, recipie#3, recipie#3, recipie#4, recipie#4, recipie#4, recipie#4, recipie#4];

CASE 1 (SIMPLE CASE):

1) Create an Array that contains different recipes. e.g: 5 recipie#1, 3 recipe#2, 4 recipe#3 and 5 recipe#4.

Array = [recipie#1, recipie#1, recipie#1, recipie#1, recipie#1, recipie#2, recipie#2, recipie#2, recipie#3, recipie#3, recipie#3, recipie#3, recipie#4, recipie#4, recipie#4, recipie#4, recipie#4];

2) Now sequentially run all these recipes.

3) Calculate the time it takes for all the products to finish.

CASE 2 (RANDOMIZE THE EXECUTION ARRAY):

1) Create an Array that contains different recipes. e.g: 5 recipie#1, 3 recipe#2, 4 recipe#3 and 5 recipe#4.

Array = [recipie#1, recipie#1, recipie#1, recipie#1, recipie#1, recipie#2, recipie#2, recipie#2, recipie#3, recipie#3, recipie#3, recipie#3, recipie#4, recipie#4, recipie#4, recipie#4, recipie#4];

2) Now randomize the Array (shuffle all its elements). Use Fisher–Yates shuffle algorithm to achieve this.

3) Now sequentially run all these recipes (from the new array).

4) Calculate the time it takes for all the products to finish.

CASE 3 (CEP CASE):

1) Create an Array that contains different recipes. e.g: 5 recipie#1, 3 recipe#2, 4 recipe#3 and 5 recipe#4.

Array = [recipie#1, recipie#1, recipie#1, recipie#1, recipie#1, recipie#2, recipie#2, recipie#2, recipie#3, recipie#3, recipie#3, recipie#3, recipie#4, recipie#4, recipie#4, recipie#4, recipie#4];

2) Now take the first recipe in the array and run it. (Now all the data events are being passed to the CEP).

IMPORTANT: POP the specific element out of the array that has been sent for execution.

3) Before executing the recipe at the 2nd index of the array, pass the recipeId to CEP to see if it can be executed or not.

If an event is generated allowing to execute it, pop the element and execute the recipe.

If not, move to the other element and repeat the same procedure.

4) Calculate the time it takes for all the products to finish.

RESULTS:

A fixed array is used for all the three test cases:

NOTE:

‘1’, ‘2’, ‘3’ and ‘4’ are the recipe Ids.

TEST # 1:

Array = ['2', '2', '2', '2', '1', '1', '1', '1', '4', '4', '4', '4', '4'];

Use Cases	Time of Execution (ms)
CASE 1	124084
CASE 2	112695
CASE 3	96653

TEST # 2:

Array = ["2","2","2","2","1","1","1","1","1","1","4","4","4","4","4","4","4","4","4","4"];

Use Cases	Time of Execution (ms)
CASE 1	167684

CASE 2	125386
CASE 3	119795

TEST # 3:

Array = ["2","2","2","2","2","2","2","2","2","2","2","1","1","1","1","1","1","1","1",
"1","1","1","1","1","1","1","4","4","4","4","4","4","4","4","4","4","4","4","4","4","4","4"];

Use Cases	Time of Execution (ms)
CASE 1	291506
CASE 2	236490
CASE 3	214803

Description of Results:

From the results obtained, it can be clearly seen that as the order of the recipes are changed the manufacturing time is reduced drastically.

TODOs:

1) make improvements in code. (make JSON array for CEP case) {“recipe1”:5, “recipe2”:3}
2) Make graphs, and do comparison.

Problems Faced:

IMPORTANT:

EPL improvements:

In a given window, pattern captures relations among (at least 2) events:

•e.g. A => B - an A event after a B event

•e.g. every A =>B - every A event after a B event http://www.dis.uniroma1.it/~baldoni/semSD2012slides/Esper.pdf