To check for outlier data point, use GESD. Source: NIST Math whiz



The architecture assumes three major components. First, there is a factory simulator that fulfills job orders using a MTConnect “read-read” command interface as its communication backbone. Second there is an MTConnect archiver program that reads MTConnect XMl streams and saves this output into a MySQL data base. Finally, there is a dashboard that reads the MySQL output and displays a graphical user interface to watch factory activity.

Cmd:CNC1\_RESOURCE\_CM State:ready QSize: 0 Step#:-1 Part: Program: Program#:-99

Cmd:CNC2\_RESOURCE\_CM State:ready QSize: 0 Step#:-1 Part: Program: Program#:-99

Cmd:CNC3Old\_RESOURCE State:running QSize: 1 Step#: 4 Part:shim Program:Setup3 Program#:0

Cmd:CNC3New\_RESOURCE State:ready QSize: 0 Step#:-1 Part: Program: Program#:-99

Cmd:CNC4\_RESOURCE\_CM State:ready QSize: 0 Step#:-1 Part: Program: Program#:-99

Resource r3old

Current Job Step 4

Current JobId 1

Active 1

PartID shim

Program Machine3O\_2-5

Program# 1

Parts bracket shim bodyjoint

Numbers to Make 100 75 50

Numbers Started 0 54 0

Numbers Finished 0 54 0

pp1, pp1p1, c1, r1, (Setup1\_1-1,Machine1\_1-1)

pp1, pp1p2, c2, r2, (Setup2\_1-2,Machine2\_1-2)

pp1, pp1p3, c3, r3old, (Setup3O\_1-3,Machine3O\_1-3)

pp1, pp1p3, c3, r3new, (Setup3N\_1-3,Machine13N\_1-3)

pp1, pp1p4, c4, r4, (Setup4\_1-4,Machine4\_1-4,Inspect4\_1-4)





[ProcessPlan][Process][Cell][Resource][ProgramList][EstimatedTime][ActualTime]

Send reset to all resources. First all the command resource handlers (which contain a Resource, State Model, and MTConnect interface) send the command “reset” to every resource in the “factory”. At this point there is no checking that all resources are not down or faulted, so that a loop updates each resource handler and then sleeps to allow other threads to execute. Without the thread sleeping, the MTConnect actual resource simulator could not respond to the command.

CResourceHandler::SendAllResourceHandlers("reset");

while(!CResourceHandlers::AllResourcesInState("ready"))

{

CResourceHandlers::UpdateResourceHandlers();

::Sleep(\_nServerRate);

}

## Random Numbers Using Boost

Boost has a math library that can generate random numbers based on a distribution. Generally, a uniform distribution, which gives an equal probability of getting number n<(0..m) where m is the maximum number in the distribution.

#include <random>

#include <functional>

typedef std::mt19937 engine\_type; // a Mersenne twister engine

std::uniform\_int\_distribution<engine\_type::result\_type> udist(0, 200);

engine\_type engine;

int main()

{

// seed rng first:

engine\_type::result\_type const seedval = get\_seed();

engine.seed(seedval);

// bind the engine and the distribution

auto rng = std::bind(udist, engine);

// generate a random number

auto random\_number = rng();

return random\_number;

}

## Resource Definition

A resource is a \_\_\_ within the manufacturing environment. A resource accepts commands from a MTConnect agent that is specified by the URL field within the configuration ini file. Next, the resource can fail, and the distribution and fault rate is defined by the FAULT field, which can take the following distributions: constant, beta, exponential, erlang, gamma, laplace, lognormal, normal, poisson, student, triangular, uniform, weibull. Below, the FAULT distribution is defined as triangular, with range of 90..100, with equal probability of getting a random number between 90 and 100. Assuming a resource, has broken down, it needs as follow up to be repaired. We again assign a distribution to the REPAIR field which defines the time quotient it takes for a repairman to show up and then repair the resource. Eventually, the field, RESPONSE which defines the response time of the service repair man will be added, as this is a productivity loss that is more easily rectified in a wireless age of instantaneous communication.

[CNC1\_RESOURCE]

URL=127.0.0.1:5000/CNC1\_RESOURCE\_CMD

FAULT=uniform,90,100

REPAIR=triangular,10,20,30

## Program Configuration

The Program field in the ini configuration file defines the length in seconds (this is a simulation) that a program or setup will take to complete. Within the simulation, the resource can expect a variety of programs to run based on the “process plan” for a part. We assume a setup, mill, and inspect simulation trio, to be done on all parts. Generally, inspection on a CNC milling machine is not done unless the part is very large and would suffer distortion and effect the inspection.

[Programs]

Setup1=10

Setup2=30

Mill1=50

Ream=10

Drill=5

Inspect1=10

Inspect2=10

The types of programs are defined in the GLOBALS section of the ini configuration file, to define the functionality of the different programs.

[GLOBALS]

Machining=Mill1,Ream,Drill

Inspection=Inspect1,Inspect2

Setup=Setup1,Setup2

## Yield Configuration

The YIELD field in the ini file defines the probability that a part will return a “good part” after inspection assuming a uniform distribution from (0..100). As such, the yields below define a 90% and 95% good part probability that will be used to simulate the outcome of an inspection .

[YIELD]

Inspect1=90

Inspect2=95

Job configuration is based on producing a number of parts based on the job description. We will assume that the

[GLOBALS]

Job=JOB1

Under a section titled JOB1 is the set of parts and the amount of parts to be made. A job assumes that each part has a section defining the sequence of resource steps (called Plan) and a Process Plan for each step ( a series of programs to run).

[JOB1]

Bracket=100

Shim=50

BodyJoint=75

Each Part (Bracket, Shim,BodyJoint) has an associated sequence of ordered resources (Plan) in a comma separated list and Process Plan (set of program to run) for each part.

The experiments run on the NIST Virtual Factory testbed were done to identify key measurement science challenges that must or should be overcome before automated model generation, remediated data integration and optimization analysis can become accurate, timely and cost effective. We started with an existing Arena based model, as it was available on the internet and dovetailed into our efforts to map the Virtual Factory Testbed into DES systems using CMSD. The example with parts steps and initial process times is based on the example as given by <http://www.actsolutions.it/File/Arena/Arena%20User's%20Guide.pdf> from Ch. 6 of “Simulation with Arena”.

The example shows a system modeled as four manufacturing cells and where we designated three part types – shims, body joints, and brackets. In our example, cells 1, 2, and 4 each have a single machine, while Cell 3 has 2 machines a newer faster model and an older one:





As mentioned, the Arena simulation is used to determine the time to produce a mix of 3 parts types, each visiting a different sequence of stations. The first implementation of the example was to study a job shop capacity to determine that given a set of resources, how long would it take to build a part mix. For example, assume we had 250 shims, 250 body joints and 500 brackets, how long would it take to make this part mix given a set of 4 cells and 5 resources.



The factory is defined by using a basic skeleton of a resources, cells, part, process plans, process and job definitions. CMSD defines more factory and simulation elements, but for understanding purposes,

First, we define the resources in the resources,

Resource \* r1 = Resource::Create<Resource>("r1");

Resource \* r2 = Resource::Create<Resource>("r2");

Resource \* r3old = Resource::Create<Resource>("r3old");

Resource \* r3new = Resource::Create<Resource>("r3new");

Resource \* r4 = Resource::Create<Resource>("r4");

Next, we define the cells in the factory, which is a collection of resource. The cell collection can contain only one resource, or in the case of cell 3, can contain 2 resources from which to use, r3old and r3new. Thus, the C++ statements

c3->resourceIds.push\_back("r3old");

c3->resourceIds.push\_back("r3new");

used the bstr collection of resourceIds to add the ids of r3old and r3new to the cell c3. The overall code for defining the cells is as follows:

Cell \* c1 = Cell::Create<Cell>("c1");

c1->resourceIds.push\_back("r1");

Cell \* c2 = Cell::Create<Cell>("c2");

c2->resourceIds.push\_back("r2");

Cell \* c3 = Cell::Create<Cell>("c3");

c3->resourceIds.push\_back("r3old");

c3->resourceIds.push\_back("r3new");

Cell \* c4 = Cell::Create<Cell>("c4");

c4->resourceIds.push\_back("r4");

Next, the processes that are used to produce parts is defined. Each process is part of process plan, so the naming convention is pp#p# where pp# is the process plan number, and the p# is the plan number. Thus, pp2p1 is process plan 2 process 1 definition. Of importance in a process plan step is the resourcesRequired, and the programs used in the process plan.

Process \* pp1p1= Process::Create<Process>("pp1p1");

Process \* pp1p2= Process::Create<Process>("pp1p2");

Process \* pp1p3= Process::Create<Process>("pp1p3");

Process \* pp1p4= Process::Create<Process>("pp1p4");

pp1p1->resourcesRequired.push\_back("c1");

pp1p1->AddProperty("Program","Setup1","","Setup pp1 for process 1");

pp1p1->AddProperty("Program","Machine1","","Machining pp1 in process 1");

pp1p2->resourcesRequired.push\_back("c2");

pp1p2->AddProperty("Program","Setup2","","Setup for pp1 process 2");

pp1p2->AddProperty("Program","Machine2","","Machining pp1 in process 2");

pp1p3->resourcesRequired.push\_back("c3");

pp1p3->AddProperty("Program","Setup1","","Setup for pp1 process 3");

pp1p3->AddProperty("Program","Machine1","","Machining pp1 in process 3");

pp1p3->resourcesRequired.push\_back("c4");

pp1p3->AddProperty("Program","Setup2","","Setup for pp1 process 4");

pp1p3->AddProperty("Program","Machine2","","Machining pp1 in process 4");

Process \* pp2p1= Process::Create<Process>("pp2p1");

Process \* pp2p2= Process::Create<Process>("pp2p2");

Process \* pp2p3= Process::Create<Process>("pp2p3");

Process \* pp2p4= Process::Create<Process>("pp2p4");

Process \* pp2p5= Process::Create<Process>("pp2p5");

pp2p1->resourcesRequired.push\_back("c1");

pp2p1->AddProperty("Program","Setup1","","Setup pp1 for process 1");

pp2p1->AddProperty("Program","Machine1","","Machining pp1 in process 1");

pp2p2->resourcesRequired.push\_back("c2");

pp2p2->AddProperty("Program","Setup2","","Setup for pp1 process 2");

pp2p2->AddProperty("Program","Machine2","","Machining pp1 in process 2");

pp2p3->resourcesRequired.push\_back("c4");

pp2p3->AddProperty("Program","Setup1","","Setup for pp1 process 3");

pp2p3->AddProperty("Program","Machine1","","Machining pp1 in process 3");

pp2p4->resourcesRequired.push\_back("c2");

pp2p4->AddProperty("Program","Setup1","","Setup for pp1 process 3");

pp2p4->AddProperty("Program","Machine1","","Machining pp1 in process 3");

pp2p5->resourcesRequired.push\_back("c3");

pp2p5->AddProperty("Program","Setup3","","Setup for pp1 process 3");

pp2p5->AddProperty("Program","Machine3","","Machining pp1 in process 3");

Process \* pp3p1= Process::Create<Process>("pp2p1");

Process \* pp3p2= Process::Create<Process>("pp2p2");

Process \* pp3p3= Process::Create<Process>("pp2p3");

pp3p1->resourcesRequired.push\_back("c2");

pp3p1->AddProperty("Program","Setup2","","Setup for pp1 process 3");

pp3p1->AddProperty("Program","Machine2","","Machining pp1 in process 3");

pp3p2->resourcesRequired.push\_back("c1");

pp3p2->AddProperty("Program","Setup2","","Setup for pp1 process 2");

pp3p2->AddProperty("Program","Machine2","","Machining pp1 in process 2");

pp3p3->resourcesRequired.push\_back("c3");

pp3p3->AddProperty("Program","Setup1","","Setup for pp1 process 3");

pp3p3->AddProperty("Program","Machine1","","Machining pp1 in process 3");

Each Process or step is then assigned to a ProcessPlan as part of a sequence of flow through the factory. The important parameters of the ProcessPlan are the identifier (e.g., “pp1”) and the processes required for each process plan ( for pp1: pp1p1, pp1p2, pp1p3 and pp1p4). Below the code defines the process plans by adding processes and saves the process identifiers in lists.

///////////////////////////////////////////////////

ProcessPlan \* pp1= ProcessPlan::Create<ProcessPlan>("pp1");

ProcessPlan \* pp2= ProcessPlan::Create<ProcessPlan>("pp2");

ProcessPlan \* pp3= ProcessPlan::Create<ProcessPlan>("pp3");

pp1->processes.push\_back(pp1p1); pp1->processIds.push\_back(pp1p1->identifier);

pp1->processes.push\_back(pp1p2); pp1->processIds.push\_back(pp1p2->identifier);

pp1->processes.push\_back(pp1p3); pp1->processIds.push\_back(pp1p3->identifier);

pp1->processes.push\_back(pp1p4); pp1->processIds.push\_back(pp1p4->identifier);

pp2->processes.push\_back(pp2p1); pp2->processIds.push\_back(pp2p1->identifier);

pp2->processes.push\_back(pp2p2); pp2->processIds.push\_back(pp2p2->identifier);

pp2->processes.push\_back(pp2p3); pp2->processIds.push\_back(pp2p3->identifier);

pp2->processes.push\_back(pp2p4); pp2->processIds.push\_back(pp2p4->identifier);

pp2->processes.push\_back(pp2p4); pp2->processIds.push\_back(pp2p5->identifier);

pp3->processes.push\_back(pp2p2); pp2->processIds.push\_back(pp2p2->identifier);

pp3->processes.push\_back(pp2p1); pp2->processIds.push\_back(pp2p1->identifier);

pp3->processes.push\_back(pp2p3); pp2->processIds.push\_back(pp2p3->identifier);

Each part that can made in the factory is defined. In our case, we have 3 parts: bracket, shim and bodyjoint with associated ids that are used in the Create function for identification and lookup. For each part, we then associate the text of a processplanidentifier for lookup of the process plan to make this part.

Part \* bracket = Part::Create<Part>("bracket");

Part \* shim = Part::Create<Part>("shim");

Part \* bodyjoint = Part::Create<Part>("bodyjoint");

bracket->processplanidentifier="pp1";

shim->processplanidentifier="pp2";

bodyjoint->processplanidentifier="pp3";

A job defines the part mix (which parts) and the part quantities associated with each part. A job has an id for identification, (e.g., “job1”) and then has a list of Part identifiers\* and the quantity for each part to be made. Below is the code to define Job1:

Job \* job1 = Job::Create<Job>("job1");

job1->partIds.push\_back("bracket"); job1->partQuantity.push\_back("100");

job1->partIds.push\_back("shim"); job1->partQuantity.push\_back("75");

job1->partIds.push\_back("bodyjoint"); job1->partQuantity.push\_back("50");

The Factory integration class uses basic C++ class reflection to offer some basic lookup by identifier or name services.

CFactoryIntegrator factory;

//std::vector<Resource \* > shimCells=factory.GetJobResources("shim") ;

Resource\* resource = factory.FindResourceByName("M2");

Cell\* cell = factory.FindCellById("c2");

Resource \* res1= factory.FindResourceById("r1");

Part \* shim1= factory.FindPartById("shim");

ProcessPlan\* pp = factory.FindProcessPlanById("pp1");

//CostAllocation \* factory.FindCostByName(bstr\_t name) ;

Job \* job = factory.FindJobById("job1") ;

}



void CMonitor::Setup()

{

stats.nBlockedTime=0.0;

stats.nStarvedTime=0.0;

stats.nDownTime=0.0;

stats.nProductionTime=0.0;

stats.nOffTime=0.0;

stats.nRepairTime=0.0;

stats.nIdleTime=0.0;

\_mtbf=MTBF;

\_mttr=MTTR;

stats.nTotalParts=stats.nGoodParts=0;

}

## State model

Factories have to be flexible, adaptable and committed to shorter product life-cycles and varying demand in order to be competitive. Clearly, integration, flexibility and efficiency requirements and the ability to simulate the production of a factory play a crucial role in meeting the competitive demands. Thus, our goal is to build a Factory framework suitable for simulation, testing and production by creating a common, modular, flexible, and integrated Framework. Manufacturers see the need for analytics that would point them to the Factory variables that actually impact the production.



The Factory framework is based on the functionality defined by CMSD, B2MML and other production schemas. Thus, with a back-end XML parser CMSD or B2MML can be parsed and inserted into the Factory framework and production can simulate using the same framework. In fact, a CMSD importer was developed as a front-end to the factory framework and was used to validate the functionality of CMSD[[1]](#footnote-1).

The Factory framework involves simulation that is defined by a Job which is a mix of parts and quantities. This simulated job is expanded into individual work orders that maps into one part per job (as currently defined). Basically a sequence is created to service all the parts and quantities. In the case study, there is C++ code that is the Factory framework which acts as a bridge between jobs and plant simulation.

Key to an efficient factory is ensuring that there is no down time and that everything and everyone is busy.

While more than 50% of organizations have installed factory scheduling applications, less than 5% have been able to achieve internal goals. The difference between success and failure boils down to three considerations: clear expectations, goal alignment, and system selection.

The Factory framework does not in itself define the communication mechanism between components. In order for the Cell Command Resource to interact with a resource, it needs a communication mechanism in order to achieve this communication. MTConnect was used as the mechanism for the command/status components to communicate. Thus, in the MTConnect communication paradigm there was a command agent and a status agent to send/receive messages. In effect, the MTConnect communication, which is called the “read-read” paradigm, behaves similarly to a communication mailbox, allowing one writer and many readers.



MTConnect provides streams of real-time data as well as intermittent asset updates, in which assets contain 3rd party XSD schema data in which XML data is to be communicated. As mentioned, MTConnect is strongly biased toward the "Read-only" http get and retrieve of XML data. As such, status and monitoring are strongly emphasized within the initial MTConnect client applications – such that, factory dashboards and the development of similar passive integration technology is the focus of MTConnect technology development. New to the MTConnect paradigm is the "Read-read" technology which enables controllers to command and control other controllers.

Below, the basic concept of MTConnect "Read-read" technology is shown, so that two agents communicate, and one agent sends a command through its XML interface which the other agent reads using the HTTP get shown in the Read-only case. Once the command is read the second agent can echo a response through its XML interface interface, again with by a Read–only agent http get. The resemblance to the long-standing communication mailbox is quite striking, so that, it is well-established control technique for communication and control of factory devices. It has been established through further study that the MTConnect "Read-read" technology is quite efficient and timely, with latencies in the 10s of millisecond ranges, if required.



Figure 1 Read-Read communicating MTConnect Agents

In order to communicate and acknowledge new commands and parameters from "cell" controllers to simulation controllers, there must be a factory communication mechanism in place. It is important that this communication scheme can allot for both real factory communication and simulated communication. Simulated communication with communication is important in MTConnect because it allows for reproducible data and testing procedures. MTConnect already provides a single machine on the internet (i.e., agent.mtconnect.org) that is running MTConnect for clients to test against. This testing mechanism provides both samples of the latest improvements to MTConnect and a stable platform upon which to test the communication and client functionality.

The read-read communication paradigm is similar to command/status communication mailboxes. In this scenario, each cell (and thus underlying simulated resources) is associated with both a command and a status mailbox that is the destination for MTConnect formatted XML messages. The MTConnect Agent writer first uses HTTP to post MTConnect XML to the mailbox,. A mailbox is defined as a combination of IP address and TCP port. A client then uses a HTTP pull operation to retrieve the XML from the MTConnect TCP/IP mailbox. The XML contents of the MTConnect mailbox do not change until the writer updates the XML.

NIST was evaluated the MTConnect "Read-read" technology to see if it is feasible for a factory communication scheme with efficient operation and reproducible results.

The ResourceHandler handled simulation using a Factory Resource, a State Machine Model and a MTConnect communication mechanism.

The ResourceHandler employs a State Machine Model in order to handle the coordination of the resource. State machines use event transition to trigger changes in the states. For the resource, events include start, run programs and stop/restart. These events leave a contiguous state trace that is easy to handle. In this case, off->init->ready->running->done->ready->running->etc. without interruptions or spurious events is easily handled and in fact, with higher reliability demonstrated by modern machines would be sufficient, if the state model also tracked state timing for KPI calculations.

However, the existence of random events was crucial to advanced functionality required for reality and simulated testing, so that event handling for faults, resets, and estops were vital. Such events require a state machine that can remember where it was. Sadly, such create spaghetti of diagrams depicting the course of potential state change activity.

Adding additional requirements added complexity to the state machine. Requirements for simplicity and the ability to add key performance indications, left existing state machines technology such as UML, Microsoft .Net state machine, boost C++ state machine as deficient for our requirements. Further,

|  |  |  |
| --- | --- | --- |
| State | Event | Transition State |
| any | quit | myExit |
| any | fail | faulted |
| any | estop | stopped |
| any | reset | resetting |
| any | off | off |
| any | run | running |
| faulted | run | running |
| faulted | softreset | ready |
| finished | next | ready |
| off | init | ready |
| ready | run | running |
| resetting | next | ready |
| running | stop | stopped |
| running | done | finished |
| starved | run | running |
| stopped | run | running |

The mailbox scheme between communicating MTConnect agents used an incremented command number to signal a new command. Thus, upon startup each machine is “reset” so that its command number is cleared and new commands can be inserted into the command mailbox, and the resource will respond to such commands

One of the most important results regarding flows in production systems is the well-known formula L = A W, commonly referred to as "Little's Law". A primary principle in the Factory Physics framework is Little's Law: WIP=THxCT. It shows a relationship between work-in-process – WIP, throughput – TH, and cycle time – CT.

.

1. J. Michaloski, F. Proctor, J. Arinez, and J. Berglund, “Toward the ideal of automating

   production optimization,” in Proceedings of the 15th ASME 2013 International Mechanical

   Engineering Congress & Exposition, (San Diego, CA, USA), ASME, Nov. 2013. [↑](#footnote-ref-1)