NIST Technical Note XXXX

**NIST Virtual Factory Testbed based on CMSD**

John Michaloski

Fred Proctor

John Horst

Brian Weiss

Thursday, August 08, 2013

DRAFT



NIST Technical Note XXXX

**NIST Virtual Factory Testbed based on CMSD**

John Michaloski

Fred Proctor

John Horst

Brian Weiss

Engineering Laboratory (EL)

Month and Year of Publication Date



U.S. Department of Commerce

*xxxx, Secretary*

National Institute of Standards and Technology

*Patrick D. Gallagher, Under Secretary of Commerce for Standards and Technology and Director*

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

**National Institute of Standards and Technology Technical Note XXXX**

**Natl. Inst. Stand. Technol. Tech. Note XXXX, NNN Pages (Month and Year)**

**CODEN: NTNOEF**

Table of Contents

[1 Introduction 5](#_Toc362454865)

[2 Purpose 5](#_Toc362454866)

[3 NIST VFT Requirements 5](#_Toc362454867)

[3.1 Overview 5](#_Toc362454868)

[3.2 Key Concepts 5](#_Toc362454869)

[3.2.1 MOM Manufacturing Operations Management 5](#_Toc362454870)

[3.2.2 KPI Key Performance Indicators 6](#_Toc362454871)

[3.2.3 DES - Discrete Event Simulation 7](#_Toc362454872)

[3.2.4 MTConnect 7](#_Toc362454873)

[4 CMSD Overview 9](#_Toc362454874)

[5 NIST Virtual Testbed 11](#_Toc362454875)

[5.1 Identifiable Measurement Science Challenges 11](#_Toc362454876)

[5.2 Rockwell Automation Arena 13](#_Toc362454877)

[5.2.1 MTConnect 17](#_Toc362454878)

[6 CMSD VFT Simulation of GM Casting Facility 24](#_Toc362454879)

[6.1 CMSD Manufacturing Use 24](#_Toc362454880)

[7 References 29](#_Toc362454881)

[Appendix I 30](#_Toc362454882)

[Appendix II CMSD Builder 31](#_Toc362454883)

[A. C++ Reflection 35](#_Toc362454884)

[Appendix III 37](#_Toc362454885)

[A. Cost Functions 37](#_Toc362454886)

[B. Obstacles 38](#_Toc362454887)

# Introduction

This document describes the purpose, scope and requirements of NIST Virtual Factory Testbed using CMSD as its core information model.

# Purpose

The purpose of NIST Virtual Factory Testbed is to develop a system methodology and software framework in which to conduct analysis and compute optimizations of factory performance. The long-term strategic goal is to develop an automated approach to optimization of production systems for more efficient operation. Given that NIST itself only has a small manufacturing job shop, the Virtual Factory Testbed will be used in the study of a broader range of production and would greatly assist in the ability to measure and optimize a multitude of manufacturing scenarios. Ultimately, the ability to study a broader range of production would be of greater use to NIST’s industrial manufacturing partners.

# NIST VFT Requirements

## Overview

Production modeling must consider all aspects of the manufacturing operation, i.e., plant, process and personnel as well as design, production, and maintenance.

Although CMSD provides a neutral framework that facilitates the creation of collections of related manufacturing information suitable for use in the creation or enhancement of manufacturing simulations and other manufacturing applications, CMSD would be better served if supplemented by a 1) more incremental approach to file development, 2) more feedback and separation of manufacturing operation and 3) intrinsic language to describe optimality in the system.

## Key Concepts

### KPI - Key Performance Indicators

KPI can serve as a basis for decisions (problem identification) and for performance benchmarks (target / actual comparison) to describe and understand important production elements. Since KPI can characterize the behavior of their production, this can help manufacturer's benchmark performance. Benchmarking can be used to compare current performance against historical performance, as well as a comparison against others in industry. Depending on the perceived KPI value, companies can choose to initiate subsequent remediation action to improve efficiency or not.

Key Performance Indicators (KPI) are metrics that provide a means to characterize productivity. To help understand and implement KPI, the International Organization for Standardization (ISO) has developed a draft standard ISO 22400, “Automation systems and integration Key performance indicators for manufacturing operations management”, that offers guidance with the definition, usage, application and benefits of production KPI and associated metrics. KPIs computed for benchmarks, comparisons, estimations and forecasts are described in terms of data pertaining to the process, the machinery and equipment, the product manufactured and its quality, the manufacturing personnel and other related manufacturing resources.

Measured KPI are necessary, but not sufficient, in order to enact improvements. For example, throughput is a KPI to measure the performance of a process, i.e. the quantity per unit time is produced and is an important indicator of production efficiency. Although, throughput KPI forms a necessary basis for improvements by better production information, unsatisfactory KPI values implies the need for more detailed data analysis, that most likely is not be directly discernible from the KPI itself. Instead, production data of finer granularity is required to understand the contributing factors that lead to the unsatisfactory KPI. For example, in one scenario the machine often sits idle blocked, waiting for the arrival of materials or tooling. In contrast, excessive machine or process faults cause prolonged unproductive periods of operations. In either case, understanding the type and severity of delays or faults within production are required to remediate process problems and improve OEE. Often the underlying analysis is more difficult to undertake and the data is more difficult to categorize and remediate.

Our research goals regarding KPI are to quickly synthesize KPI and then understand the underlying root causes of poorly performing KPI.

### DES - Discrete Event Simulation

To understand the complicated relationships in modeling production, simulation modeling can be extremely beneficial as it is well-suited to handling the complexity of large scale interaction of humans, machines, and processes. Simulation can be effective to model real or proposed production and evaluate concepts, identify problem areas, and quantify and optimize system performance. Simulation is especially popular when the intricacy of real world makes analytical closed-form solutions difficult. Simulation that models a system as a chronological sequence of discrete events is known as Discrete Event Simulation (DES). DES is useful for modeling and analysis of manufacturing systems, such as machinery operation, shop workflow and scheduling, and production lines.

The major drawback to DES is that it is costly, error-prone, and time-consuming, especially the data acquisition, filtering and statistically modeling which can take up to 31 % of the effort involved. A goal of our project will be to enable “Push–button” DES that would provide seamless end-to-end DES integration minimizing manual operations. In this case, DES could be used continually and cost-effectively in order to evaluate machine operations. Push–button DES requires automating the data collection as the statistical characterization of the machining.

Our research goals include using MTConnect data and improving MTConnect based on requirements as determined by automating DES. Due to the costly nature of the DES data processing, the goal of automating DES is undeniably critical and considerable effort will be applied in this area.

### MTConnect

In order to reduce costs, increase interoperability, and maximize enterprise-level integration, the MTConnect specification has been developed for the manufacturing industry. MTConnect is a specification based upon prevalent Web technology including XML and HTTP. MTConnect uses the Web “REST”model interface, basically a “connectionless” interface in which an Agent only services single requests, and it is the responsibility of the client application to maintain any session information. Using prevailing technology and providing free software development kits minimize the technical and economic barriers to MTConnect adoption.



**Client** - is typically a factory application, such as shop floor dashboard visualization, OEE, and data mining of asset and process knowledge.

**Agent** - acts much like a Web Server that acts as an intermediary between a Device and a Client. The MTConnect Agent receives and stores single or a time series of data samples or events from the device. Clients use HTTP to communicate four basic requests to the MTConnect Agent:

* “Probe” provides the configuration of the device data items,
* “Current” gives a snapshot of the device's data items' most recent values, or
* “Sample” provides historical range of values for samples, events, or conditions stored within the device.
* “Asset” provides data about more unvarying items associated with the device for a limited period of time, such as, parts, tools, and fixtures. Such assets would not generally have any controller of their own, and would be managed by another device.

**Device** - is a piece of factory equipment organized as a set of components that provide data.

**Physical Device Data Model** - provides the Device(s) description that the world will see, which will be typically a subset of the total possible data from a CNC. It is expressed in an XSD provided by MTConnect. Clients use a “probe” command to read this device configuration.

**Information Model** - is an XSD information model that describes the entirety of permissible device “Data items” and mobile “Assets”. The information is flexible in that some new data items can be established. This capability is used to prototype the quality measurement data.

MTConnect models a device as a set of components with constituent data items. Initially the MTConnect specification is targeted at machine tools and their constituent components - axes, power, controller, and control sequencing. In this information model structure, one or more devices contain a series of components, of some Component type: controller, linear axis, rotary axis, etc. Each component then has event or sample Data Item definitions. MTConnect further provides XML attributes in which to help refine the Device information models. Such XML attributes include Category, Name, Type, Subtype, and Units.

Overall, an MTConnect Device model is not hardwired; rather users assemble an XML information model to match their devices. MTConnect allows independent development of versions, with new extensions coexisting with legacy functionality. “MTConnect data items are self-describing and messages carry a protocol version number, and extensions can be added to MTConnect without jeopardizing backwards compatibility; principals that do not understand the extensions can safely ignore them.” For example, in previous work, we were able to add a “PartCount” DataItem, which at the time was not explicitly part of the MTConnect specification, without any trouble.

# CMSD Overview

The Enterprise domain is responsible for handling process customer orders and deciding whether to make or buy parts. The processing of customer orders triggers the creation of a unique part order (or workorder) within the Manufacturing Execution System (or production system). The creation of a part order (e.g., 10 front bumpers, 12 side panels) is incumbent on the knowledge on how to build the parts on some set of equipment (abstract process plan), the part definition(revision, part and quality) and part programs (how to make the part). All these are combined into jobs to make the parts (and may be scheduled). Of course, a job could describe not only production, but quality, inventory, or maintenance tasks. But in our case, we want a complete model of production operation so we will initially focus on production.

In Figure 2, a part is assumed to be a finished product that was produced or that can be used in production activities as raw material or a work in-progress component. Many different kinds of information can be specified for a part, such as, information about the production status of a part, the named category of parts that a specific part belongs to, the sub-component parts used to create this part, the process that can be used to create the part, and some basic characteristics of this Part. For a part, there is also a Planned Process Plan, which holds a list of Process objects in some sequence. This sequence can be used to describe the routing of a Part object. Each Process class holds a list of the Part Type that it requires for processing, as well as the products it creates. The Process class also holds a list of the resources required for processing. These resources can include physical locations (e.g. machines) and mobile resources (e.g. laborers or work fixtures).

In the scenario, a customer orders enters the system and triggers a part order. To match our case study discussed later, we will assume that the part order will contain only one type of part, a fixed number of parts will be made per day, and that this part will be made based on a abstract process plan, a part definition (revision and associated resources and part programs) and part programs (or recipes) to make part on a series of equipment. After all this information is collated, a job will be generated containing the part and a process plan to describe the potential sequence through a set of resource types. This process plan will serve as the scheduled routing of the raw material to become a part. The alternate routing serves in case to reroute if a broken equipment arises - however, in our case study, this is not a concern as the primary equipment will be used and fixed if broken. Using shop-floor data, the simulation will then make the part and buffer the finished part based on the data.

In Figure 2, the concept of resource breaking down is based on production task - even though maintenance information is necessary to differentiate the equipment and the mechanic must replace or fix the broken equipment even though from a production standpoint, the piece of equipment and time to break and time to repair are important performance indicators.



Figure 1 Job Life Cycle

CMSD is a freely available standard specification that would allow the translation from numerous related domains into a manufacturing domain-specific representation suitable for analysis. The primary use of CMSD is to generate the simulation model by using a suitable model representation of the physical system. In this approach, all CMSD information required must be acquired at the point of creating the simulation model.



Figure CMSD Flow (missing jobs box)

Figure 3 addresses the problem using the CMSD specification to address issues related to information management and manufacturing simulation development. Please note, shaded CMSD boxes are out of scope. The CMSD entities defined in this framework represent a core set of the manufacturing entities and relationships needed for manufacturing simulation that CMSD offers representations for many categories of manufacturing information, but in our case we were most interested in:

* **Resource** - describes equipment that performs manufacturing activities. Resources in the CMSD are used to represent stations, machines, cranes, employees, tools, and fixtures. (for this iteration we assumed no trained personnel was required.)
* **Part** - provides a means to specify the characteristics of the materials and subcomponents that are used to make end products.
* **Process plan** - specifies the set of production activities needed to transform materials and subcomponents into finished products. Each process plan is built of process steps (with associated resource(s)) that must be executed for the part to be finished.
* **Process** - defines a manufacturing activity or group or manufacturing activities that encompass a detailed strategy for creating a part. The process will most likely contain information that describes the resources that will be used, the parts that will be consumed and produced, the sequence in which resources will be used, and the sequence of activities within a group of activities.
* **Job** - defines normal, maintenance or repair operation, but in our case the job represents normal manufacturing and is the central construct of the system. Each job (assuming it came from customer order as described earlier), would generate an appropriate number of parts into “spawned” jobs (type of job) and under each spawned job contains the part knowledge exhibited within the job e.g., process plan, the resources, etc. The spawned job would contain a copy of the initial job that described all the parts and quantities.

Jobs typically define complex production work items can involve activities at multiple stations that ultimately produce parts. Processes are lower level work items that are typically performed at a single workstation or area within the shop. The basic fulfillment of a “spawned” job is to know its process plan, it current process step within the process plan (at what process) and its processing status.

Although the goal of CMSD is to provide a neutral framework that facilitates the creation of collections of related manufacturing information suitable for use in the creation or enhancement of manufacturing simulations and other manufacturing applications, we found that CMSD would be better served if supplemented by a 1) more incremental approach to file development, 2) more feedback and separation of manufacturing operation and 3) intrinsic language to describe optimality in the system.

# NIST Virtual Testbed

Analysis of factory activity based on current technology is always open to new and better ideas, and in our case we hoped to develop the NIST Virtual Testbed will improve the insights and judgments of potential decisions. (It is not the job of NIST to enforce or overcome any factory human factors that may make a better process or lead to potential improvement for consideration by outside industrial partners.) In the long run, it can be found that inadequate knowledge can lead to ill-informed decisions making changes difficult and thus prevent or thwart potential improvements.

Generally, FMC is operated under three different situations such as single-lot production, flow production, and produce-to-order production. In single-lot production, the known sets of parts are to be produced in a given period of time. Whereas in flow production, the parts are produced continuously in fixed and known proportions. During produce-to-order situation, orders with different processing requirements are received randomly with different inter-arrival times and due dates.

## Identifiable Measurement Science Challenges

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:

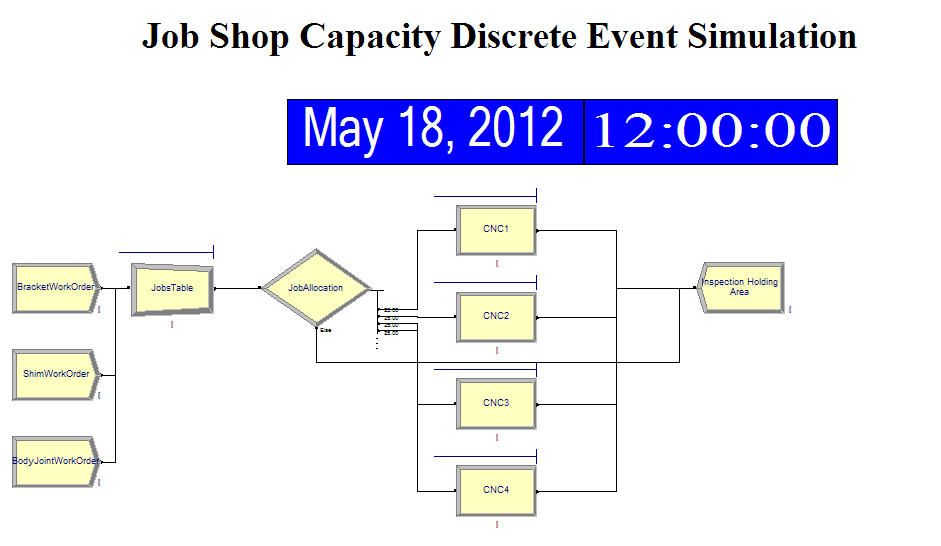


As mentioned, the Arena simulation is used to determine the time to produce a mix of 3 parts types, each visiting a different sequence of stations. The first implementation of the example was to study a job shop capacity to determine that given a set of resources, how long would it take to build a part 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. Using Rockwell Arena, we were able to develop a display as well as simulation in order to understand the expected time to achieve total N parts of given part mix, given M resources, O operators and F factory schedule. Rockwell Arena has a COM back end which we used:

## Rockwell Automation Arena

The NIST Virtual Testbed investigated Rockwell Automation’s Arena software application. Arena is a visual DES software package that uses the SIMAN processor and simulation language for its simulation kernel. Arena can be integrated with other applications using the Microsoft COM model and COM interfaces so that Visual Basic for Applications can be integrated if specific functionality is needed.

Rockwell Arena has certain quirks that must be accommodated, however, NIST was able to translate a CMSD description of the job, part, process plan and process, and cells/resource and automatically generate a runnable Rockwell Arena DES model. Below show the output from the NIST Virtual Testbed interpretation of the example capacity CMSD description into a Rockwell Arena DES system.



A “part” in Arena is called an “Entity”. Arena has the Station concept which contains the name of the current station location of the entity. The Arena Sequence concept contains the name of the sequence of station visitations the entity (i.e., part) will follow. This maps into CMSD so that we define a

To hook into the Arena COM functionality, we first need to import all the Arena interfaces which can be done using the Microsoft COM import functionality in Visual C++, and we defined a class to hold the functionality: CComArenaHook.

#import "C:\\Program Files\\Rockwell Software\\Arena\\Arena.exe"

class CComArenaHook

We will assume we have the functionality to read the CMSD and then map this functionality into a list of C++ variables. Next we need to define a list of all the Arena COM types. COM supplied the CComPtr type to manage COM pointer (release the pointer when done), so that garbage collection does not consume the source code.

CComPtr<Arena::IArenaApp> \_appdispatch;

CComPtr<Arena::IModels> \_arenamodels;

CComPtr<Arena::IModules> \_Modules;

CComPtr<Arena::IModel> \_mymodel;

CComPtr<Arena::IConnections> \_myconnections;

CComPtr<Arena::IModule> \_curmodule;

Using the Microsoft COM QueryInterface we can query the interfaces (on a running Arena system) and get pointers to these interfaces. Of note, bstr\_t is a Microsoft COM string type that easily converts between ANSII strings and UNICODE string. All COM communication is done with UNICODE (With 2 bytes or 16 bits per typed character). Using these COM pointers we can then add modules to the Arena model. So first, we use COM to cocreate and then activate an interface:

\_appdispatch.CoCreateInstance(L"Arena.Application");

\_appdispatch->Activate();

\_arenamodels=\_appdispatch->Models;

\_mymodel=\_arenamodels->Add(); // create new arena model

\_Modules=\_mymodel->Modules; // add new arena model to collection of modules

\_myconnections =\_mymodel->Connections; // get connections

And similarly for exiting Arena (but one can use the manual Arena GUI if desired):

void quit() { if(\_appdispatch !=NULL ) \_appdispatch->Quit(); }

Next we create 3 parts: shims, body joints and brackets. We will only show how to create in Arena COM the "Bracket" part. A part is defined as a BasicProcess so we can define a Workorder for the number of parts (100) to Sequence through the Cells.

// Define Parts as "Entites"

CComPtr<Arena::IModule> entity=

\_Modules->Create(\_bstr\_t(L"BasicProcess"), \_bstr\_t(L"Entity"), 100, 400);

entity->PutData ( \_bstr\_t(L"Name"), \_bstr\_t(L"Bracket"));

// Create Parts

CComPtr<Arena::IModule> create=

\_Modules->Create(\_bstr\_t(L"BasicProcess"), \_bstr\_t(L"Create"), 100, 400);

create->PutData ( \_bstr\_t(L"Name"), \_bstr\_t(L"BracketWorkOrder"));

create->PutData ( \_bstr\_t(L"Max Batches"), \_bstr\_t(L"100"));

create->PutData ( \_bstr\_t(L"Entity Type"), \_bstr\_t(L"Bracket"));

We follow the development of an shim, body joint and bracket entity with a JobsTable creation that is a "Basic Process" that can hold the parts (batch size 1).

CComPtr<Arena::IModule> batch=

\_Modules->Create(\_bstr\_t(L"BasicProcess"), \_bstr\_t(L"Batch"), 900, 400);

batch->PutData ( \_bstr\_t(L"Name"), \_bstr\_t(L"JobsTable"));

batch->PutData ( \_bstr\_t(L"Batch Size"), \_bstr\_t(L"1"));

So far, we have defined via Microsoft COM the entities, the initial JobsTable where parts where will be batched, and now we define the Basic Process from which all parts will be disposed when done. We will call this the Inspection Holding Area to signify that the parts could undergo quality control.

// Create ExitSystem Station and Dispose of Part

CComPtr<Arena::IModule> exitStation=

\_Modules->Create(\_bstr\_t(L"AdvancedTransfer"), \_bstr\_t(L"Station"), 5800, 0);

exitStation->PutData ( \_bstr\_t(L"Name"), \_bstr\_t(L"ExitSystem"));

exitStation->PutData ( \_bstr\_t(L"Statn"), \_bstr\_t(L"ExitSystem"));

CComPtr<Arena::IModule> dispose=

\_Modules->Create(\_bstr\_t(L"BasicProcess"), \_bstr\_t(L"Dispose"), 6600, 0);

dispose->PutData ( \_bstr\_t(L"Name"), \_bstr\_t(L"Inspection Holding Area"));

\_myconnections->Create(exitStation,dispose);

After defining the creation/deletion of the Entities, we now create an Arena decision tree to decide where the next part will go.

CComPtr<Arena::IModule> decide=

\_Modules->Create(\_bstr\_t(L"BasicProcess"), \_bstr\_t(L"Decide"), 1800, 400);

decide->PutData ( \_bstr\_t(L"Name"), \_bstr\_t(L"JobAllocation"));

decide->PutData ( \_bstr\_t(L"Type"), \_bstr\_t(L"NWith"));

Now an Advanced Transfer module is created which will handle routing of the "entities" through the factory cells:

CComPtr<Arena::IModule> route=

\_Modules->Create(\_bstr\_t(L"AdvancedTransfer"), \_bstr\_t(L"Route"), 2600,0);

route->PutData ( \_bstr\_t(L"SG"), \_bstr\_t(L"Sequential"));

\_myconnections->Create(batch,route);

Next we define the Resources and performance of the resources with two states busy or idle within Arena:

// Create resources

for(int i=0; i< xml.resources.size(); i++)

{

CComPtr<Arena::IModule> resource=

\_Modules->Create(\_bstr\_t(L"BasicProcess"), \_bstr\_t(L"Resource"), 700, i \* 700);

resource->PutData ( \_bstr\_t("Name"), xml.resources[i].name);

resource->PutData ( \_bstr\_t(L"Capacity"), \_bstr\_t("1"));

if(DataExists(xml.resources[i].hourlyRate))

resource->PutData ( \_bstr\_t(L"Busy"), xml.resources[i].hourlyRate);

if(DataExists(xml.resources[i].hourlyRate))

resource->PutData ( \_bstr\_t(L"Idle"), xml.resources[i].hourlyRate);

}

Now the resources are grouped into Cells. Again each type is an AdvancedTransfer but the cells are a station type. Both the AdvancedTransfer and the Station (cell) require a name. In the Cell station definition the SDR is an acronym for the "seize delay release" resource handling. The SDR resources require parameterization for the Units, DelayType, Expression, Resource Name, and Quantity handled at one time by the resource. Many of these parameters are hard-coded, but conceptually it would be straightforward to turn them into variables.

for(int i=0; i< xml.cells.size(); i++)

{

bstr\_t n = StdStringFormat("%d",i+1).c\_str();

// Create Station

CComPtr<Arena::IModule> station=

\_Modules->Create(\_bstr\_t(L"AdvancedTransfer"), \_bstr\_t(L"Station"), 3400, 0+i\*450);

station->PutData ( \_bstr\_t(L"Name"), xml.cells[i].name);

station->PutData ( \_bstr\_t(L"Statn"), xml.cells[i].name);

// Create Process

CComPtr<Arena::IModule> process=

\_Modules->Create(\_bstr\_t(L"BasicProcess"), \_bstr\_t(L"Process"), 4200, 0+i\*450);

process->PutData ( \_bstr\_t(L"Action"), \_bstr\_t(L"SDR"));

process->PutData ( \_bstr\_t("Units"), \_bstr\_t("Minutes"));

process->PutData ( \_bstr\_t(L"DelayType"), \_bstr\_t(L"Expression"));

process->PutData ( \_bstr\_t(L"Expression"), \_bstr\_t(L"CapacityFactor \* ProcessTime"));

if(xml.cells[i].resourceIds.size() == 1)

{

Resource r = xml.FindResourceById(xml.cells[i].resourceIds[0]);

process->PutData ( \_bstr\_t(L"Resource Type(1)"), \_bstr\_t(L"Resource"));

process->PutData ( \_bstr\_t(L"Resource Name(1)"), r.name);

process->PutData ( \_bstr\_t(L"Quantity(1)"), \_bstr\_t("1"));

}

else

{

// create set

CComPtr<Arena::IModule> set=

\_Modules->Create(\_bstr\_t(L"BasicProcess"), \_bstr\_t(L"Set"), 4200, 0+i\*450);

set->PutData ( \_bstr\_t("Name"), \_bstr\_t(xml.cells[i].name+"ResourceSet"));

set->PutData ( \_bstr\_t("Type"), \_bstr\_t(L"Resource"));

for(int j=0; j< xml.cells[i].resourceIds.size(); j++)

{

Resource r = xml.FindResourceById(xml.cells[i].resourceIds[j]);

bstr\_t n = StdStringFormat("%d", j+1).c\_str();

set->PutData ( \_bstr\_t("Resource Name("+n+")"), \_bstr\_t(r.name));

}

// set of resources...

process->PutData ( \_bstr\_t(L"Resource Type(1)"), \_bstr\_t(L"Set"));

process->PutData ( \_bstr\_t("Set Name(1)"), xml.cells[i].name+"ResourceSet");

process->PutData ( \_bstr\_t(L"Quantity(1)"), \_bstr\_t("1"));

}

// Create route when done

route=\_Modules->Create(\_bstr\_t(L"AdvancedTransfer"), \_bstr\_t(L"Route"), 5000, 0+i\*450);

route->PutData ( \_bstr\_t(L"SG"), \_bstr\_t(L"Sequential"));

\_myconnections->Create(station,process);

\_myconnections->Create(process,route);

}

In Arena, we use Advanced Transfer to define a Sequence of operations for an Entity (i.e., Part). At the end the entity is moved to the ExitSystem station.

for(int i=0; i< xml.processplans.size(); i++)

{

// CMSD process plan to define part. Each process plan has steps of process to

// make part.

ProcessPlan plan = xml.processplans[i];

// Create advanced transfer and name it

CComPtr<Arena::IModule> var=

\_Modules->Create(\_bstr\_t(L"AdvancedTransfer"), \_bstr\_t(L"Sequence"), 00, 0);

var->PutData ( \_bstr\_t("Name"), xml.processplans[i].identifier);

for(int m=0; m < seq.steps.size(); m++)

{

// This is the processplan process name, map into cell name

Process step = plan.FindProcess(seq.steps[m]);

bstr\_t station;

Cell cell =xml.FindCellById(step.resourcesRequired[0]);

\_bstr\_t timings[] = {

\_bstr\_t(L"TRIA(6,8,10)"),

\_bstr\_t("TRIA(5,8,10)"),

\_bstr\_t("TRIA(5,8,10)"),

\_bstr\_t("TRIA(15,20,25)")

};

bstr\_t n = StdStringFormat("%d",m+1).c\_str();

var->PutData ( \_bstr\_t("Station("+n+")"), cell.name);

var->PutData ( \_bstr\_t("SG(1,"+n+")"), \_bstr\_t("Attribute"));

var->PutData ( \_bstr\_t("Att(1,"+n+")"), \_bstr\_t("ProcessTime"));

var->PutData ( \_bstr\_t("Value(1,"+n+")"), step.operationTime.time);

}

bstr\_t n = StdStringFormat("%d",seq.steps.size()+1).c\_str();

var->PutData ( \_bstr\_t("Station("+n+")"), \_bstr\_t("ExitSystem"));

We mentioned that within Cell3, the two resources had different capabilities. To distinguish this in Arena/COM we need to add performance factor array to differentiate CNC3 new versus old:

// Add performance factor array to differentiate CNC3 new vs old

{

CComPtr<Arena::IModule> var=

\_Modules->Create(\_bstr\_t(L"BasicProcess"), \_bstr\_t(L"Variable"), 00, 0);

var->PutData ( \_bstr\_t("Name"), \_bstr\_t("Factor"));

var->PutData ( \_bstr\_t("Rows"), \_bstr\_t(L"2"));

var->PutData ( \_bstr\_t("Initial Value(1)"), \_bstr\_t(L"0.8"));

var->PutData ( \_bstr\_t("Initial Value(2)"), \_bstr\_t(L"1.0"));

var=\_Modules->Create(\_bstr\_t(L"BasicProcess"), \_bstr\_t(L"Variable"), 00, 0);

var->PutData ( \_bstr\_t("Name"), \_bstr\_t("TransferTime"));

var->PutData ( \_bstr\_t("Initial Value"), \_bstr\_t(L"2"));

}

In Arena, based on the current station and Jobstep, chooses the next station, sets all the appropriate attributes, and moves the part to the next station. Next we add some Arena descriptors, variables, and clock to make the demonstration better.

\_mymodel->ProjectDescription=\_bstr\_t(L"Simple Job Shop contains 4 machines and 3 parts. 24/7 shifts");

\_mymodel->ProjectTitle=\_bstr\_t(L"Estimating Job Shop Capacity");;

\_mymodel->HoursPerDay=\_bstr\_t(L"24");

The clock is added as an Arena type with data supplied by Windows (COleDateTime::GetCurrentTime();). Formatting the display clock for Arena can be difficult as the actual documentation is sparse and programmers often rely on the COM utilities.

// Add clock

COleDateTime t1 = COleDateTime::GetCurrentTime();

CComPtr<Arena::IStatusDates> dates = \_mymodel->StatusDates;

CComPtr<Arena::IStatusClocks> clocks = \_mymodel->StatusClocks;

CComPtr<Arena::IStatusDate> date = dates->Create (

2000,-900,4050, -450, // where

Arena::smDateDisplayText, // enum smDateDisplayType dateType,

Arena::smDateFormatMonthDayYear, // enum smDateFormatType DateFormat,

Arena::smDateUnitSecond, // enum smDateUnitType dateUnits,

t1.GetMonth(), t1.GetDay(),t1.GetYear(),t1.GetHour(), t1.GetMinute(), t1.GetSecond(),

0, 0xFF0000 /\* background\*/, 0xFFFFFF/\*font color\*/ , \_bstr\_t(L"Microsoft Sans Serif" ));

clocks->Create(4050, -900,5800, -450,VARIANT\_TRUE,VARIANT\_FALSE,

60.0,0, 0,0, 0,

0xFF0000, // blue background

0xFFFFFF );

### MTConnect

Descibed earlier, MTConnect is a factory communication and integration technology aimed at integrating CNC machines, tool, assets, sensors, etc. MTConnect is based on passing XML streams between communicating parties using HTTP that in turn are specified using the XSD W3 standard. MTConnect uses the http REST Client/Server model – so that in general synchronous polling of a Web server using the http communication protocol and decorated URLs for specializing the http get is the standard operating procedure.

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 Read-Read communicating MTConnect Agents

One test performed with the NIST Virtual Factory testbed was evaluation of the MTConnect "Read-read" technology in order to communicate and acknowledge new commands and parameters from "cell" controllers to simulation controllers. In the figure below, low level communication between the cell and simulated controller is done using MTConnect Adapters that transmit SHDR data to agents. In our case, the simulated controllers were called simusers, could run under either a Linux or Windows platform, and were based on the NIST Go Motion software/simulator.



Figure NIST Virtual Factory Testbed Deployment

In Figure 4, the NIST Virtual Factory testbed shows the initial deployment which was based on CMSD resource/layout architecture description. The CMSD had a job with N Shims, M Body Joints and P brackets to make (in this case, 10,20, 30). For each part, there was a CMSD processplan that described how the part was made, and a CMSD process(es) that described the sequence of operations, the process part program (e.g., SHIM.NC) and the expected time to mill the part (in the SHIM.NC case 1 minute). Using SSH, 2 agents and 1 adapter was spawned in order to control MTConnect devices, i.e., Simuser devices.

The Virtual Factory featured Simuser MTConnect “devices” and each of these Simuser devices had an associated MTConnect adapters. These Simuser devices could be either 1) machine tools, 2) robots, and 3) conveyers. There is a startup script that runs that remotely and automatically starts up a full Go Motion controller and associated MTConnect adapter. Once started, the Simuser device waits for a Cell level command. These Simuser devices run on Linux or Windows (you all have only seen Linux ones running).

Figure 5 shows an expanded Command Cell MTConnect Agent snapshot. There are five Devices bundled together into one XML stream that is broken down into Samples, Events, and Conditions to make it more readable. Of not, the three Tags “Command”, “CmdNum” and “Program” are critical in commanding a Simuser device. In this case Simuser devices are spawned for CNC1\_RESOURCE, CNC2\_RESOURCE, CNC3New\_RESOURCE, CNC3Old\_RESOURCE, and CNC4\_RESOURCE. Each of these factory resources, as represented by the Simuser simulations, knows its name, and read the tags in order to command the Sinuser device. Note, CNC4\_RESOURCE may be active but has no commands yet as the Shim, Body Joint and Bracket are still on earlier CNCs (1-3), so only a RESET command and -1 cmdnum have been directed to this Simuser device.

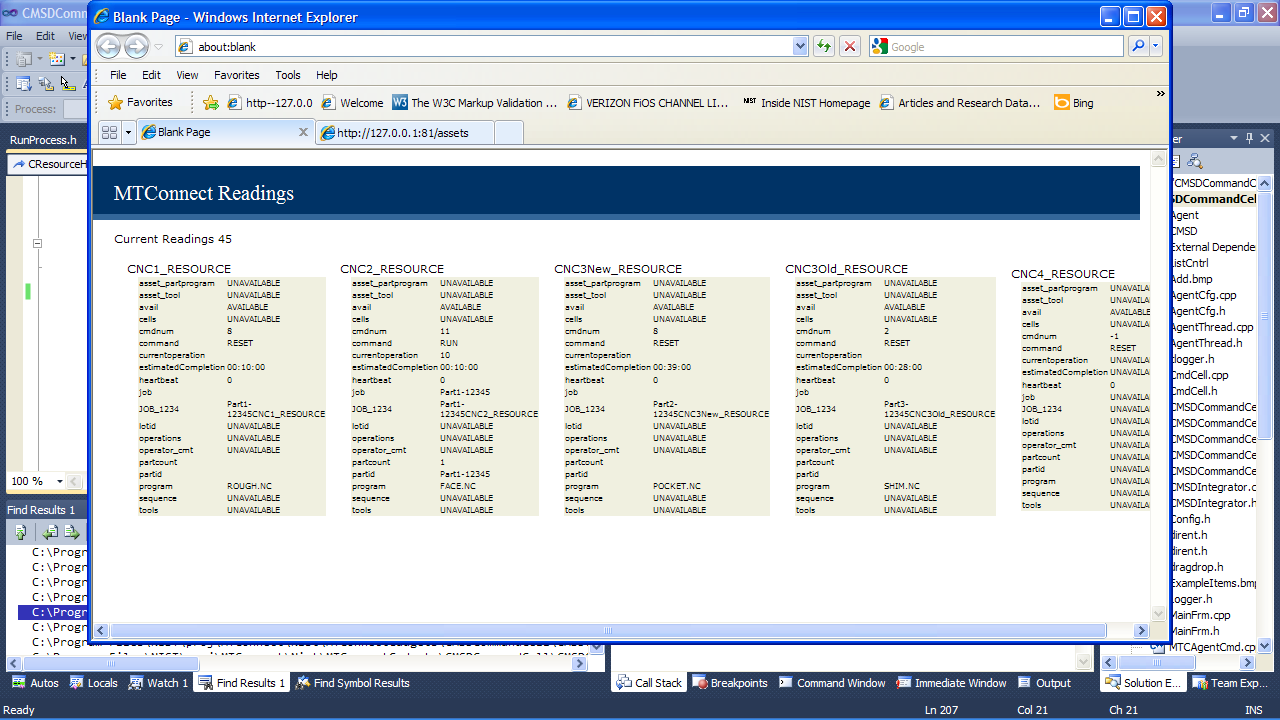


Figure NIST Virtual Factory Testbed Command Cell Snapshot

For command and control, a CMSD file was read and used to spawn all the simusers on individual or jointly used computers using Secure Shell (SSH). SSH is a cryptographic network protocol for secure data communication, remote command-line login and command execution between two networked computers. Cygwin offers a Windows ssh was used to remotely spawn (spin up) a Simuser, requiring the SSH arguments (location of encrypted key in Windows file system – assuming this computer is allowed to have remote access), the ip of the Simuser platform, the SHDR socket port (MTConnect specific port number used for text based communication between MTConnect Adapter and MTConnect Agent), device name, so that in the case of multiple devices embedded in a Command Cell Agent XML the Simuser would know which device it is, and an ip address for the Command Cell agent to remotely read the Simuser SHDR output. Below is the spin up code in C++, with plain text password (not real password):

void StartSpinUp()

{

boost::mutex::scoped\_lock lock(\_access);

\_cond->wait(lock);

std::string cmd, ssh;

for(int i=0; i< shdrPorts.size(); i++)

{

remoteProcess.push\_back(CFindRemoteProcess());

ssh = StdStringFormat("ssh simuser@%s -i \"/cygdrive/c/Documents and Settings/simuser/My Documents/.ssh/id\_rsa\" ",

ipMachines[i].c\_str());

cmd = StdStringFormat(" /usr/local/proj/gomotion/bin/spinup -s %s -p %d -m %s -c %s",

ipDeviceAgents[i].c\_str(),

shdrPorts[i],

devicenames[i].c\_str(),

ipCmdAgents[i].c\_str()

);

remoteProcess.back().ExeSsh(ssh + cmd, "password1\n");

}

}

The Simusers emulations come up to the “ready” state – but do not do anything until a command (with an incremented command number) is received. After all the Simusers are spawned, the Cell controller then reads the CMSD Jobs containing a part mix. For each CMSD part, the Cell must look up the CMSD part process plan, and then send a command (CMD, CMD NUM, PROGRAM) to the appropriate Simuser resource via MTConnect Agent, which in the “Read-Read” paradigm, the device is continually monitoring to see if there is a new command. (Note that although we could have used the MTConnect sequence numbers in the XML for the Command "Tag" but we used a command number tag instead to indicate a “new command.”) Upon reading a new command from the Cell agent the Simuser acts on this command and echoes a response through its agent – either go/no go based on detection of any errors.

The Simuser Command client is written in python and reads the Cell Agent XML and extracts all the Event/Sample/Condition information. For example, a command would contain the following information:

RUN, 1, Face.NC

In the interim, we used a pre-existing set of part programs installed on the Simuser platform that should be “Run”. The Cell controllers has a Status Client that reads the CMSD and translates the CMSD into a Simuser "run program" commands, monitors the Simuser for completion, and then issues a new command when the Simuser device is done.

The NIST Virtual Testbed can have a large collection of devices and associated MTConnect adapters. Devices can be machine tools, robots, or conveyers. We have a program that starts up a full Go Motion controller and associated MTConnect adapter. We use a SSH command for “spinning this up”. Simusers simulations can run on Linux or Windows. There is also a Factory Agent that is a single “large” MTConnect agent running that rolls up all the distributed Simuser devices and the current states of the Cell Controller, and turns it into a dashboard display.

A Cell Controller is in control of a single large MTConnect agent running on a Windows computer that rolls up all the distributed Simuser devices as well as the Cell Controller, takes their SHDR adapter output, turns it into MTConnect, and serves it up at some well-known URL/current. We call this the Factory Agent. There will be many MTConnect client applications that poll the Factory Agent and looks for state changes that trigger events, such as running NC programs, moving robots, and activating conveyers. We call these the Device Clients. For now, we envision one Device Client per device, but we may be able to aggregate some, maybe one per simulation computer.

The Device Client associated with a particular device, say a machine tool, would poll the Factory Agent and look for Cell Controller state changes that signify that the Cell Controller wants that particular machine to run an NC program. The Device Client then sends a command to the device to do this. This means the Device Client includes HTTP connecting, XML parsing, and controller shared memory reading. The Device Client can be quite complicated.

The Device Clients don’t send out any status – the status that shows that a program is complete is generated by the MTConnect adapter for that device, and is rolled up by the Factory Agent. The Cell Controller would poll the factory agent to see when the execution state of a machine is done, and then move on to the next thing in its CMSD list of things.

Summarizing, to give an idea of the functionality, a step by step analysis of the process will be given:

1. We start with a CMSD description of two machines and the routing of one part. This would look something like this:

Machine #1, MTBF 17 hours, MTTR 20 minutes

Machine #2, MTBF 30 hours, MTTR 1 hour

Part: 40 minutes machining on #1, 15 minutes transport time, 70 minutes machining on #2

2. We compare the simulation against the Arena simulation.

3. Next, we load this CMSD configuration into a DeviceSpawn application which spins up two machine simulations. Those simulations also include failures and recoveries according to their MTBF and MTTR. I sort-of have that now.

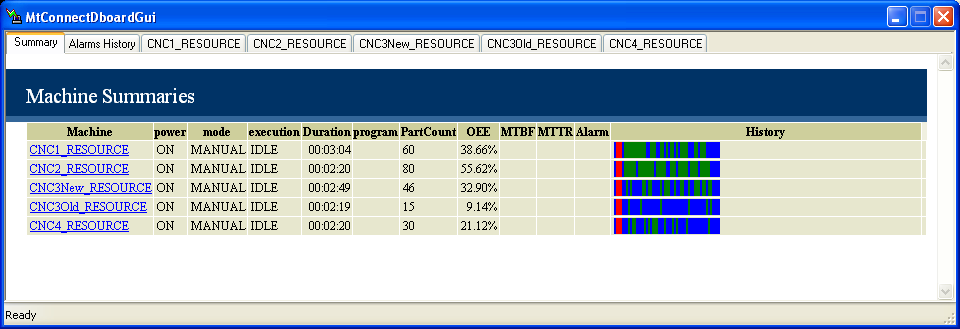
4. We code up a new Cell Controller that takes the CMSD “schedule” and runs all the parts on the machines. This happens by automatically creating NC files that take 40 minutes and 70 minutes, getting them to run on #1 and #2, waiting the transport time, checking for errors, re-running on failures after waiting according to the MTTR. This is a somewhat complicated piece of code, but Python is good for this functionality.

5. Next, we run an MTConnect client that reads each machine’s MTConnect output and collects statistics over time, building a CMSD description of what he saw. We compare this to the original and see how well it correlates.

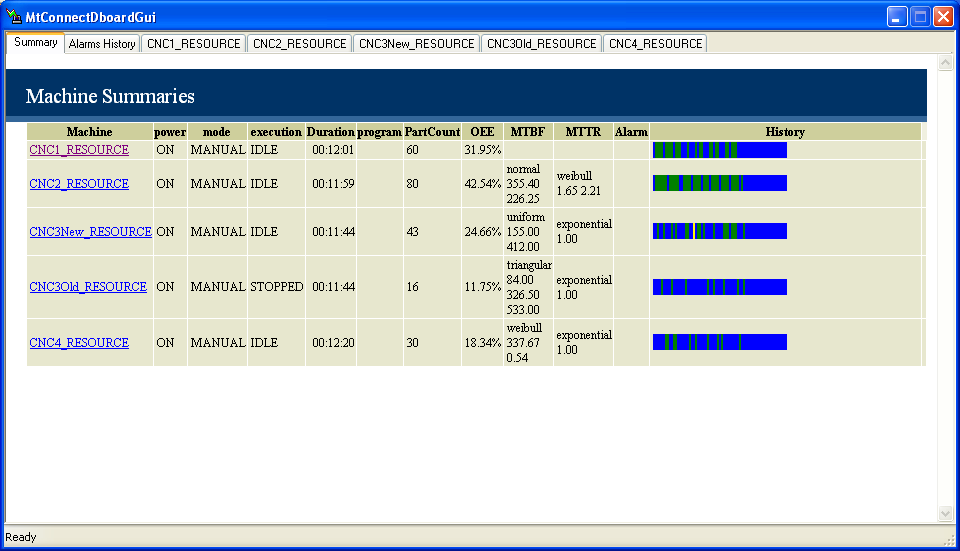
We could then supplement this functionality by adding:

1. Robot loading and unloading via ROS Industrial, and handle the machine control using MTConnect Read-Read.
2. Enumerate potential devices to emulate using MTConnect – controller and communication back-end. We would then articulate expected behavior coverage – level of functionality compliance, faults, time periods of operation, interaction, etc.
3. Developing a computer architecture of MTConnect factory testbed using CMSD XML specification. Evaluate and report on CMSD for ability to express factory configuration and capacity planning.
4. Programming simple device emulation functional behavior, completely compliant device communication back-ends.
5. Programming factory configuration utility to import various CMSD factory configurations to be easily developed.
6. Programming factory capacity and workflow and fault simulation using CMSD as configuration model.
7. Developing repeatable testing scenarios for regression analysis. Tests should include good, bad, longevity, loading, bandwidth, hardware disruption, error recovery, memory leaks, etc. when analyzing behavior.
8. Specifying performance metrics - acceptable metrics for system response to test scenarios. Develop software coverage strategy and incorporate into metrics.
9. Developing software to automate testing and archive results. Increase the level of interface capability and functionality.
10. Develop randomized tests for more stochastic model of coverage and software quality Developing. (See 4.)
11. Evaluate outcomes of tests, fix problems, document results – reliability and performance, and adopt issues/suggestions.

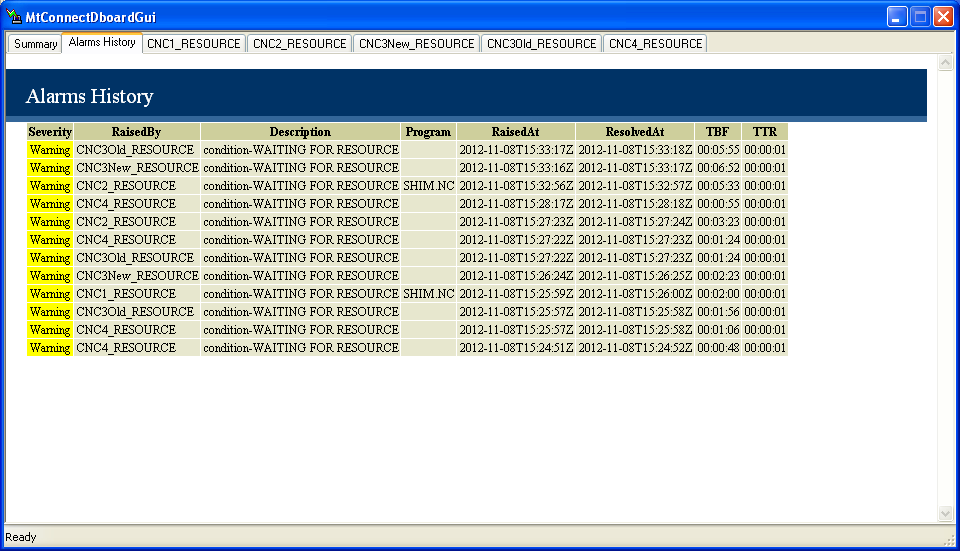
The NIST Virtual Factory has a Factory Dashboard to monitor simulation activity so that at any instant plant floor supervisors can see a snapshot of the factory performance. The Factory Dashboard (knowing where the Command Cell and Device agents are) monitors each device for activity and keeps track of its state machine history (busy, idle, faulted, etc.), part count that each machine has completed , provides a corresponding OEE (with no quality information) based on the state machine history and calculates a MTBF and MTTR based on the performance of each device. In the figure below there are no fault to statistically characterize.



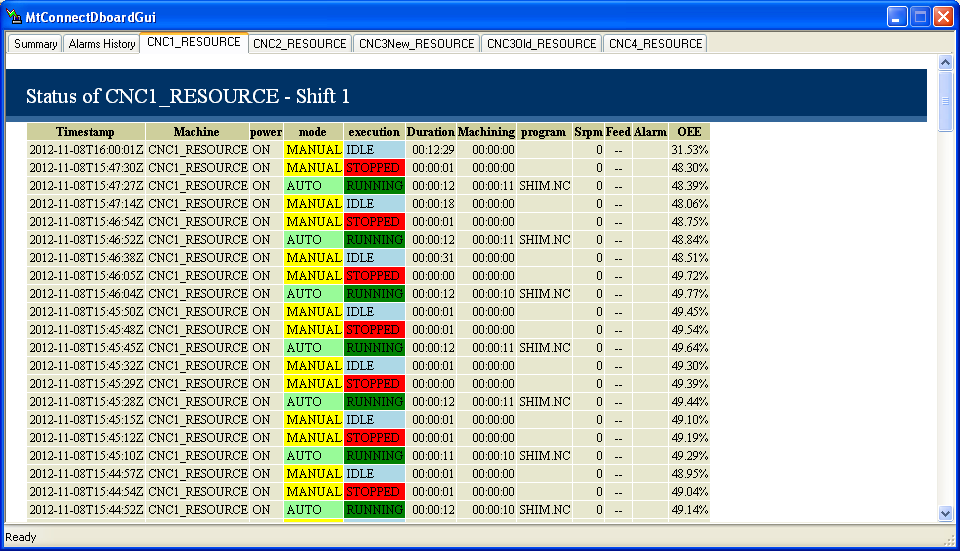
The Simuser device simulations are programmed to act like a real resource, and thus has faults and recoveries. The Factory Dashboard monitors the MTConnect agent for conditions that not normal, and thus are faults. The Factory Dashboard keeps track of the faults and then using mathematical analysis computes the Mean Time Between Failure (MTBF) and the Mean Time To Repair (MTTR) statistical distributions. Since the first occurrence of a fault can be infinite it is omitted from the calculation.



Included in the fault monitoring, is an Alarms History which gives the progression, occurrence, duration, resolution time and TBF’/TTR of each alarm. The Alarm History is a separate tab in the Dashboard, and supplies a severity for each alarm. For this release, the Dashboard monitored conditions that registered as Warning as well as Faults in the Alarm History.



Likewise, each resource (machine tool) had its MTConnect XML status monitored and presented to the user to give a snapshot of the performance of the resource. Each line in the Status tab, indicates that a new program was execution on the resource. Included in the snapshot was a history, as well as a timestamp when the new program event occurred, the mode the machine is in, whether the program is idle, running or stopped, current duration that the program is active as well as duration the program has actually been machining (computed by a RPM>0 as well X,Y,Z motion), the program (if one), and the machine rpm and feed. Included in the snapshot are any active alarms, and the overall OEE of the machine performance. OEE is a KPI that measures “Availability \* Effectiveness \* Quality Rate.” Since there is no inspection or quality results in the calculation, the quality rate if assumed to be 1.0.



# CMSD VFT Simulation of GM Casting Facility

Next, NIST studied the data from a GM Precision Casting Facility in Saginaw Michigan that makes V8 engines for more automotive vehicles. For this case we had extensive data, including process data in Excel tables from SQL queries on performance data, utilities data, and HVAC data. Some of the data was found to be insufficient to analyze sustainability of the building but future plants would incorporate ideas from the analysis.

In summary, precision sand molds are built from sand and alloys; and then filled with molten aluminum, and then the sand is removed, with only the cast remaining, in this case a V8 engine. It is a complicated process, but can be broken into individual factory lines for study.

We initially looked at the casting portion of the precision sand casting, so in this phase, complete sand molds (without tops which were attached) were brought into the line via conveyors, and then the tops were inserted by robots onto the top of the casts. Once the casts with cooling element and tops were completed, a robot inserted the cast upside down into the area where molten aluminum was pumped into the mold, and then a robot removed the cast and placed it on a conveyor for chilling plate to be removed, and then the cast was headed for a finishing line, where as much of the molded sand as possible was removed from the cast from the engine cast.

## CMSD Manufacturing Use

The integration of vast amounts of production knowledge that often comes from different sources with different formats poses a great challenge to industry. NIST will use a common production knowledge repository format with one unifying and neutral information model, CMSD. CMSD is an XML standard to facilitate the exchange of information between models and the production software applications, such as used in factory layout, process planning, scheduling, inventory management, production management, or supply chain management. CMSD has been covered earlier, and the reader is encouraged to review this material.

It became clear that the incremental loading of CMSD information model is a more preferable way to incrementally grow the information models and then to understand and simulate the manufacturing operation. For example, we used this incremental functionality to separate the production measurement from the production operation. Thus, one CMSD file was used for describing a part and its process plan. Another CMSD file was developed to describe the resource operation with KPI to describe the length of buffers, the failure rate of the equipment, and the time per processing a unit. By separating out the different CMSD elements, parts of the model could be added and reassembled in pieces – assuming the basic factory, part and process descriptions were in place. Figure 6 shows the basic CMSD files that were assembled into a factory model for further study.



Figure Manufacturing Operations represented with incremental CMSD

The goal of the studying the combination of CMSD was to replicate the original data output as read by the PLC on the shop floor (for which actual data and performance were retrieved via SQL queries on a factory performance database.) NIST was provided with prodigious amounts of factory data, some in daily portions, but the data was not necessarily (as is common in today's environment) conducive to study. The state machine for the equipment which was continually operating for about 10 hours a day was delineated into the states: processing, starved, blocked, faulted and did not consider idle. Unfortunately, some data such as starved and blocked were given as a lump sum of operation per day, and not as a statistical distribution accumulated as a daily observation. Thus, 100 hours of starved could mean 1 starved state at 100 minutes starvation, or 100 starved periods at 1 minute per starvation. Faults, processing and repair were provided with statistical distributions so were easier to replicate. However, we were determined to match the factory given state timings as best as possible.

As pointed out, we modified CMSD to allow multiple file merging. Figure 6 shows that with the use of a CMSD resource referenced to an existing CMSD reference, which could be extended to merge the CMSD manufacturing model, and allow modularization of data. Thus, manufacturing operations, manufacturing data and the job could all be separated and then input simultaneously to create a Factory Model in an incremental mode.

Within our ProcessPlan we included CMSD Resource to describe equipment or groups of equipment that performs manufacturing activities. A CSMD resource may be processed on a particular layout for one manufacturing configuration for a certain amount of time, and then used in a different layout for another manufacturing configuration.

<CMSDDocument>

<DataSection>

<Resource>

<Identifier>SMCO:LINE1\_PS\_CAST1\_ELV1</Identifier>

<Name>LINE1\_PS\_CAST1\_ELV1</Name>

<ResourceType>elevator:</ResourceType>

<Description>Elevator1</Description>

</Resource>...

Although a CMSD job has the ability to reprogram the sequence of operations of the manufacturing equipment, this reconfigurability requires a different CMSD job strategy and a more dynamic layout of the resources in the manufacturing operation. Before delving into CMSD optimization of resource allocation, we will assume that part/jobs define a static layout of resource. Each resource can then add or subtract parameters to attempt to optimize the manufacturing operation.

<CMSDDocument>

<DataSection>

<Resource>

<Identifier>SMCO:LINE1\_PS\_CAST1\_ELV1</Identifier>

<Property><Name>InQueue</Name><Value>1</Value></Property>

<Property><Name>Mtbf</Name> <Value>394</Value></Property>

<Property><Name>Mttr</Name<Value>85.8</Value></Property>

<Property><Name>Mttp</Name> <Value>64.3</Value></Property>

</Resource>

</DataSection>

</CMSDDocument>

The development of a DES model is a large undertaking but with the incremental CMSD approach, deployment can be handled in phases so that one can incorporate increasingly detailed parameterization. At first, DES manufacturing operation can start with the basic manufacturing operations to build parts, process plans, processes, and resource. Next, a CMSD file (possibly generate from live data sources) can add key performance indicators (KPI) such as, cycle time, breakdown, and buffer sizes. Later we will discuss an approach to add optimization criteria as part of the CMSD framework.

Figure 7 shows a simulation snapshot of the casting facility. The assumptions include that the robots have sufficient automated carts with casting tops for the cover delivery,

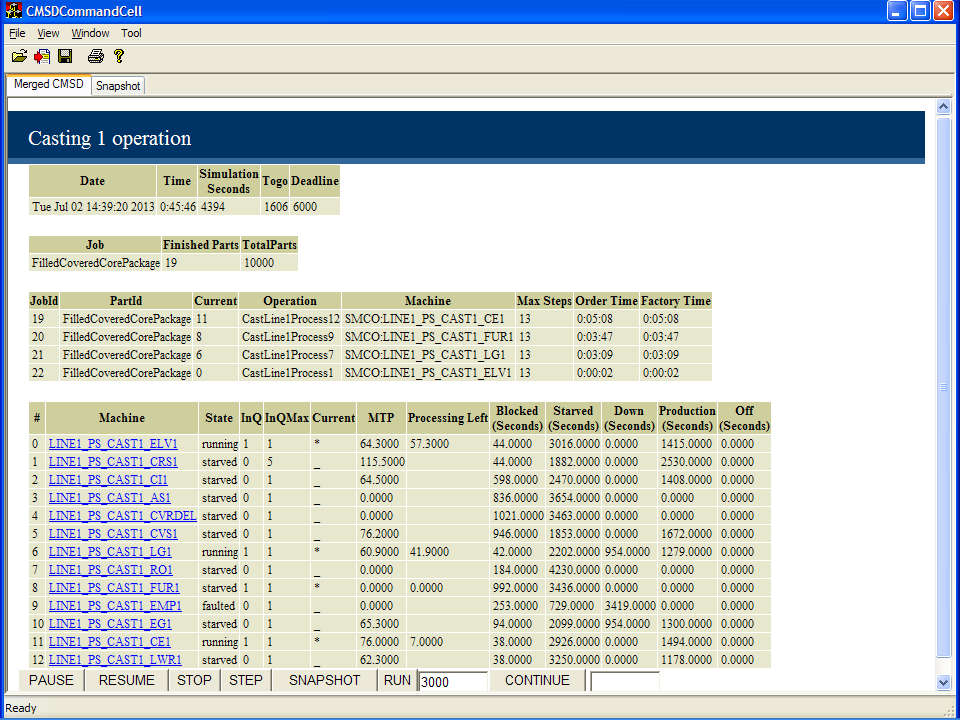


Figure Casting Simulation Snapshot based on CMSD

Energy Benchmarks were done as costs which is outlined in Appendix III Miscellaneous. After each resource in the production line is finished, the utility costs are then computed. Although these utility costs may not be correct, they use actual costs associated with each resource and are described in the CMSD for each resource to represent the cost of these utilities. These costs may be wrong since they are currently group all states (busy, faulted, etc.) into one group (active and with an associated cost of $0.05 per kilowatt hour) when calculating the utility cost.

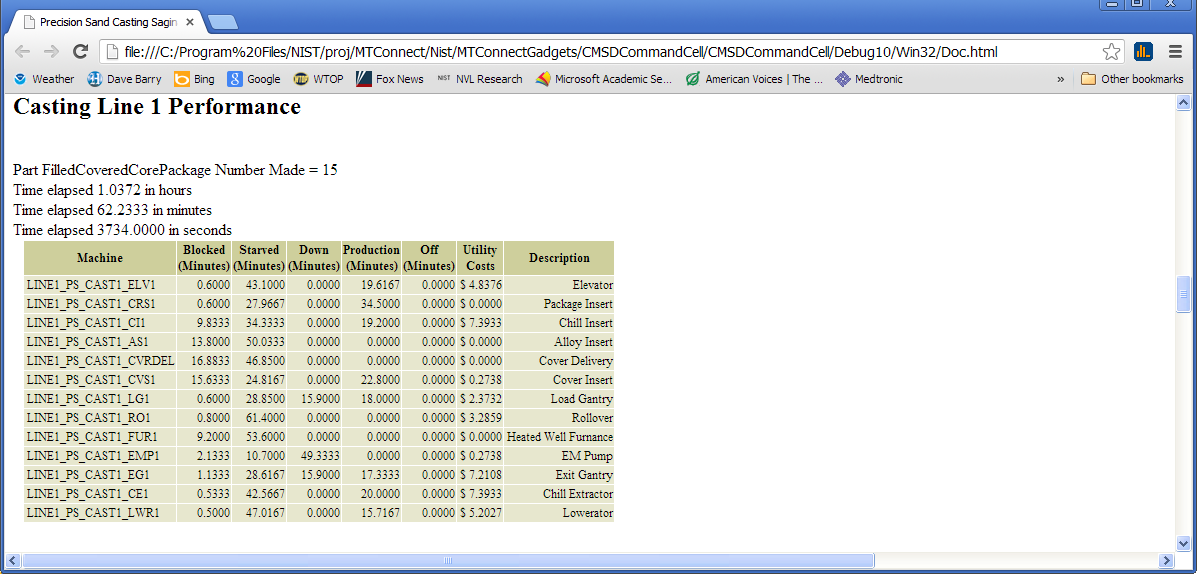


Figure Utility Costs snapshot

Below provides a more detailed look at each utilitity used in calculating the associated cost. In Figure 9 one sees a detailed look at the utility activity, but only based on the “any” state.



Figure Detailed look at utility costs

# References

# Appendix I Definitions

|  |  |
| --- | --- |
| Term | Description |
| Bill of materials | A description of the hierarchical relationships between a part and its subcomponents. |
| Calendar | A long term focused collection of shift and holiday information that, taken together, specify the time periods during which production is and is not expected to take place. |
| CMSD document | An aggregation of information that is organized based on the CMSD specification and that is suitable for exchange or archival. |
| Cost | The expense incurred by an enterprise due to its performing some manufacturing activity. |
| Distribution | The name and parameter values that define a statistical distribution. A statistical distribution is a mathematical function where: 1) the range of possible values of the function is known, and; 2) the probability that a random input to the domain of the function will produce an output value in a subset of the range is also known. |
| Inventory item | A part or (non-employee) resource for which information about its availability for production activities is tracked. |
| Inventory item type | Information about a kind of part or kind of (non-employee) resource that can be an inventory item. |
| Job | A request for production-related activities to take place, originating from a person or organization internal to the manufacturing enterprise. |
| Layout | A representation of the spatially-relevant characteristics of, and relationships between, the manufacturing resources that are a part of a manufacturing facility. |
| Lot | Information about a group of parts that were manufactured together or obtained together, and that have some important characteristic. |
| Machine program | A set of instructions that allow a computer-controlled machine tool to perform a specific manufacturing function. |
| Maintenance plan | A collection of maintenance processes that provide the necessary instructions for maintaining a (non-employee) manufacturing resource. |
| Maintenance process | A manufacturing activity or group of manufacturing activities that perform a corrective or preventive maintenance operation on a resource. |
| Metadata | Information about the format and value space that is allowable for a given property attribute. |
| Order | A request for products or services originating from a person or organization external to the manufacturing enterprise. |
| Parent entity | An entity that has other entities nested within it. A parent entity defines a scope for determining the uniqueness of multiple instances of the same kind of entity that may be nested within it. |
| Part | A raw material or sub-component used in or produced by some stage of production, or an end product that is the final objective of production. |
| Part type | Information about the characteristics of a specific kind of part. |
| Process | A manufacturing activity or group of manufacturing activities that either: 1) transforms a part from a known state/condition to another known state/condition; 2) transports a part from a known location to another location; 3) verifies that a part is in a known state state/condition or location, or; 4) all of the above. |
| Process Plan | A collection of processes that provide the necessary instructions for producing a part. |
| Property | A means for extending the information that can be associated with an entity by allowing name and value information for a noteworthy characteristic of the entity to be associated with that entity. |
| Property attribute | A characteristic of an entity that was specified using property information. |
| Resource | A piece of equipment or an employee that is performing or is to perform a manufacturing activity. |
| Resource class | Information about the characteristics of a specific kind of resource. |
| Schedule | A plan containing a time-ordered collection of production activities, and/or the results obtained by carrying out such a plan. |
| Shift | A time period during a day of the week when production activities are to take place and a specification of the days on which this time period is applicable. |
| Shift schedule | An intermediate term collection of shift and holiday information specifying when production is and is not expected to take place. |

# Appendix II CMSD Builder

Although CMSD can be hand coded, and was done for many CMSD related projects, NIST internally developed a programming application. One alternative was to use Alta Nova XMLSpy to generate C++ CMSD code as this could contribute to other manufacturing related projects associated with CMSD. The C++ produced by XMLSpy from CMSD was straightforward with some minor problems associated with low-level non-string object definitions.



Figure CMSD Archiving

Figure 7 shows the sequence of operations to turn the CMSD information model into an archival application. First, although designed in UML, CMSD has a C# or .Net Framework mapping built separately at NIST in which to read and parse CMSD files. Using this .Net EXE and the xsd.exe software tool provided freely by Microsoft, an XSD was generated. This XSD gave a (traditional) XML schema for the CMSD information model (although CMSD had Schematron and other representations, no pure XSD defined in XML was available.) Next, the tool from Altanova - XMLSpy - was used to load XSD documents, validate the XSD files, and then generate C++ archival code (read and writing from files) code based on a XML parser. For the XMLSpy approach, we generated code for XML reading and validation using Microsoft MSXML technology, although Xerces a freeware XML toolkit for Linux or Windows was available, and has been used by NIST.

The initial tool was built that incorporated C++ reflection (which was built ground up and not native to C++). Using the C++ archiver, the C++ reflection and a WTL list control, a C++ CMSD builder was built. The XMLSpy archival code easily read and wrote CMSD as it was tested on all the test files by the CMSD standards group. The XMLSpy code to parse a CMSD XML file is given by:

ParseCMSD(std::string filename)

{

CMSD::CCMSD doc = CMSD::CCMSD::LoadFromFile(filename);

doc.SaveToFile((::ExeDirectory() + "Test1.xml").c\_str(), true);

CMSD::CwhiteSpaceType root = doc.whiteSpace.first();

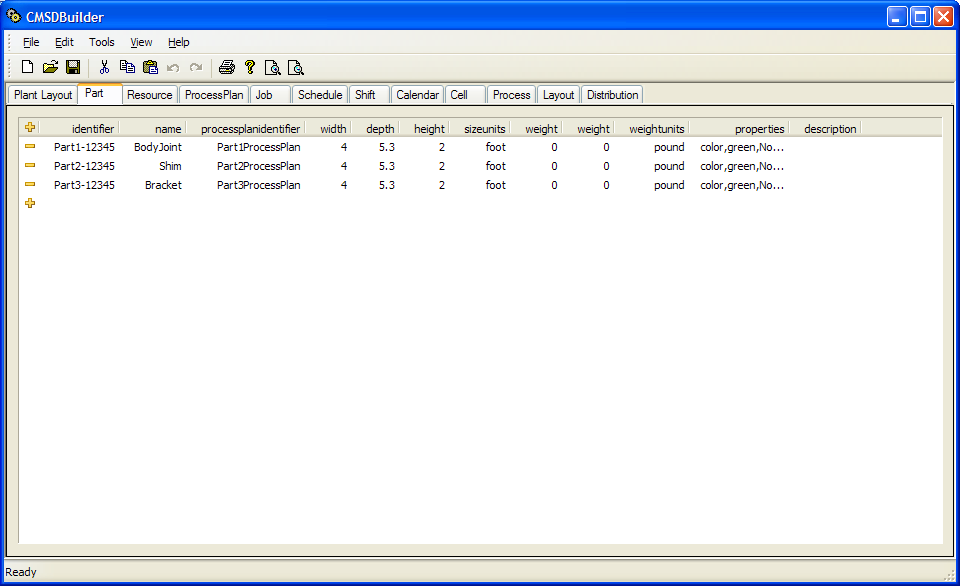
CMSD::CCMSDDocument cmsddocument = doc.CMSDDocument[0];

CMSD::CDataSectionType2 data = cmsddocument.DataSection[0];

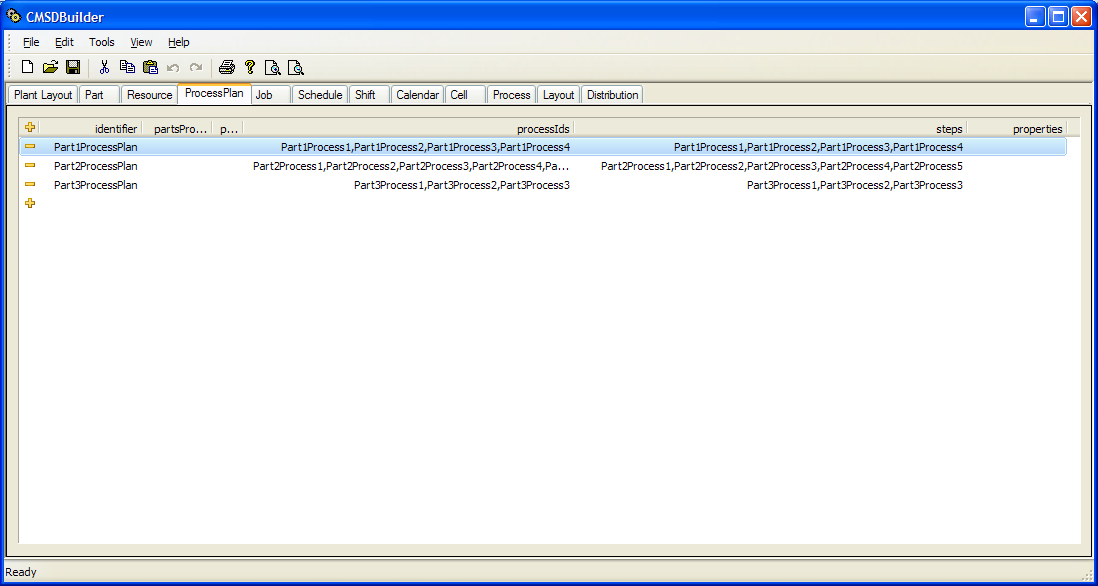
The application was embedded in a CCMSDIntegrator class that relied on the XMLSpy code and the C++ code reflection. This code could read and parse each CMSD type relevant to this project and created a mirrored class to simplify access to the CMSD. It is not clear that a simple mirror of CMSD is valuable, although a GUI to build and validate CMSD files is valuable.

The application uses the Microsoft GUI – Windows Template Library – which is a free and template based solution to building windows applications. In this case, CMSDBuilder was an SDI application in windows parlance, or a single document interface (SDI) which is a method of organizing graphical user interface (GUI) applications. In the SDI application, each window contains its own menu or tool bar to control a single window. Of note, SDI applications allow only one open document frame window at a time, so that only one CMSD file can be edited at a time.

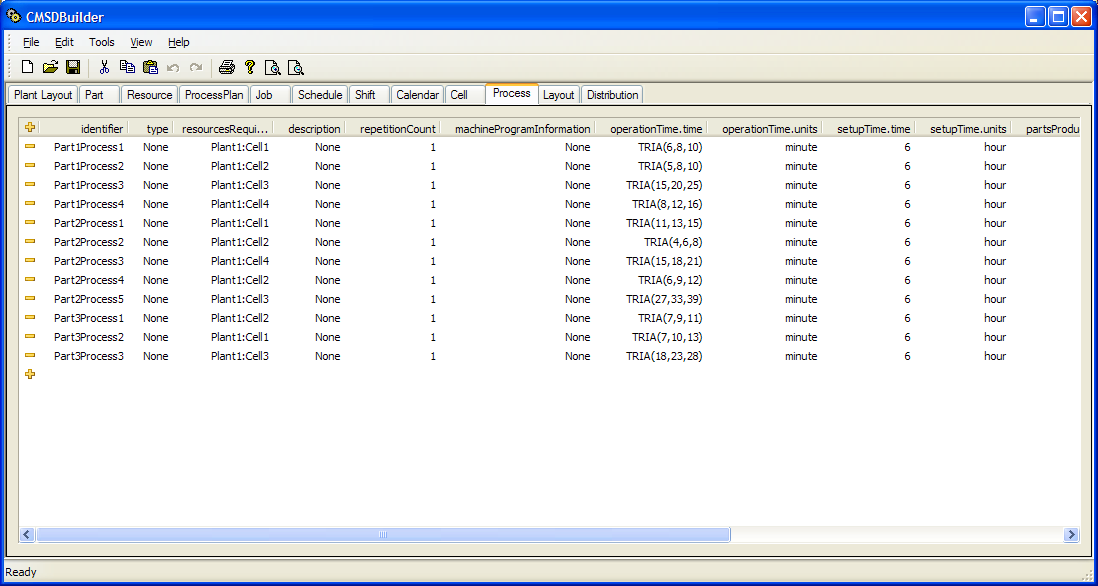
Thus, in the CMSD builder, for the initial NIST Virtual Factory demonstration, the file has three parts, whose name (but not unique identifiers) are BodyJoint, Shim, and Bracket. The figure below shows the GUI and three parts. In the diagram, the minus sign signals the removal of a part, and the positive sign signals the addition of a unique part.



Regarding the other parameters: processplanidentifier, width, height, size units, weight, height, weightunit, properties only the processplanidentifier is critical to implementing the flow through a factory. In the case below, Part1ProcessPlan, Part2ProcessPlan and Part3ProcessPlan are the process plans that are used to sequence the part through the shops. Below are the ProcessPlans for the different parts defined in CMSDBuilder.

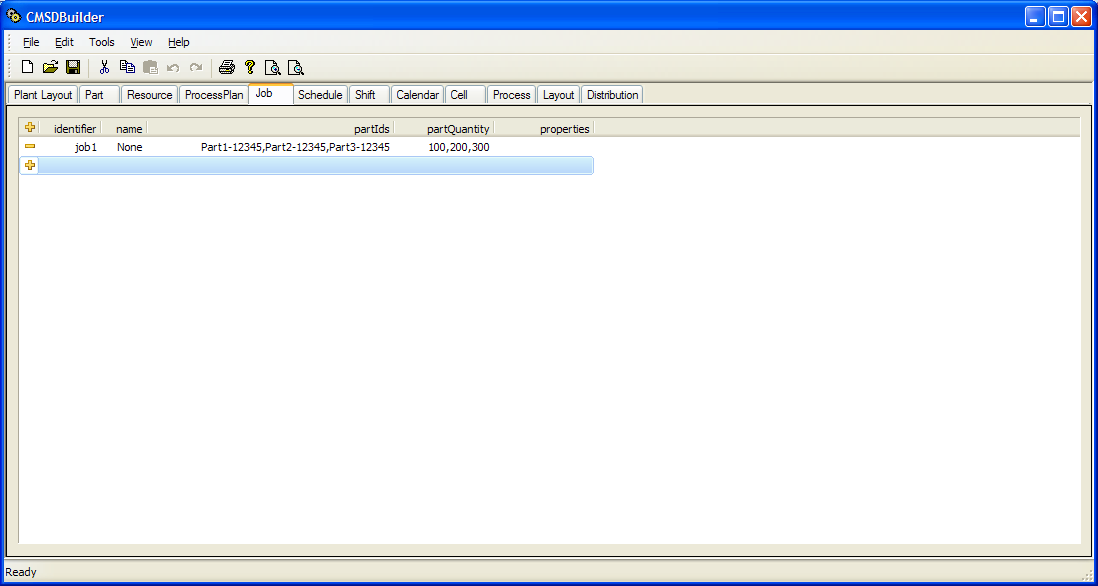


As can be seen the ProcessPlan most importantly contains an processIds for each ProcessPlan as well as the steps (processes) through the factory that would correspond to cells in the factory. Note, that each part has different Steps. Thus, given the steps one has to look at (or define) the Processes as the appear in the factory:



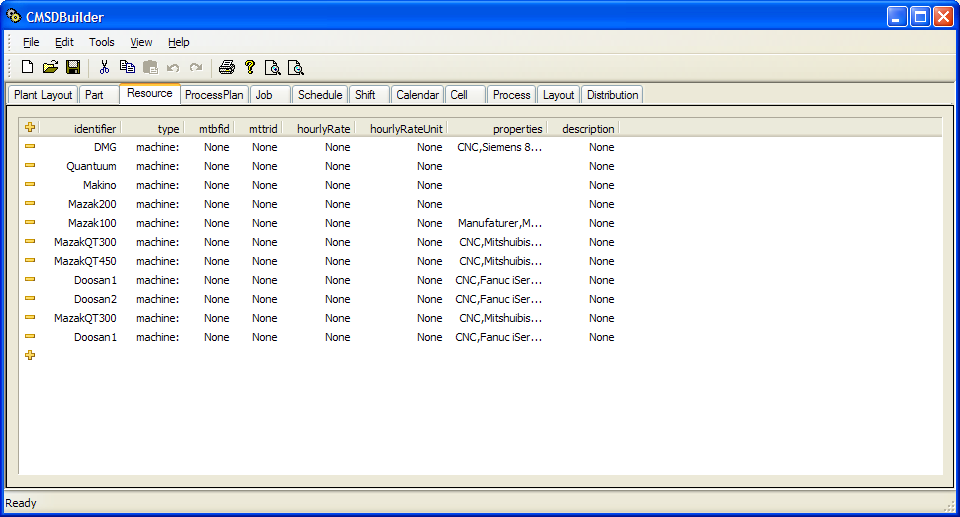
Each process is defined, and in this case, is uniquely specified to the part being built. Of interest, and can be found in the CMSD standard, is the breaking up of a Process into operation and setup time. Each part of the Process must be defined and have units assigned to the parameters. The CMSD Builder application can take a multitude of statistics for describing the length of the operation or setup, but must be defined as a 5-tuple: X(a,b,c,d) or less depending on the statistical entity. For example, gamma only needs X (gamma) and a mean (Gamma (1.1) which corresponds to the gamma name and the gamma mean).

The job itself defines the amount of parts for the Shim, Body Joint and Bracket in CMSD and is given by the following definition. (In this example 100 Part1s, 200 Part2s and 300 Part3s are defined in the job.)

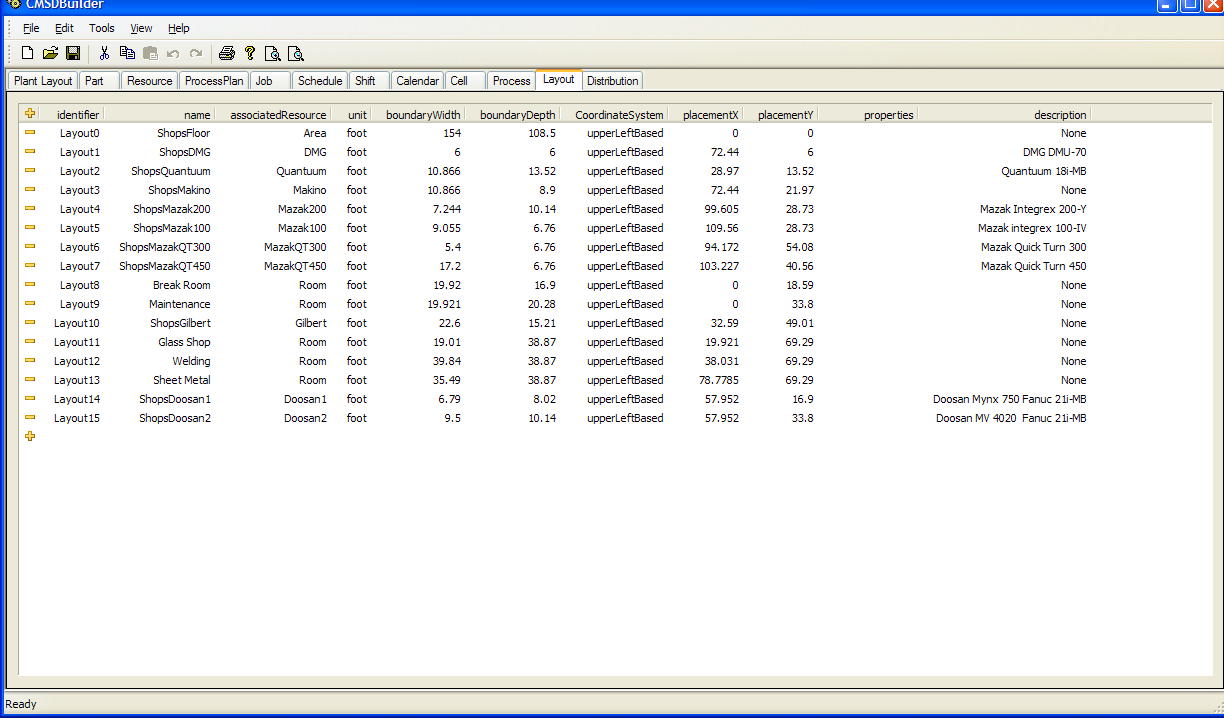


Other, but not all, CMSD elements are defined. Of interest, was the layout functionality of CMSD so that by reading the MTConnect Agent output one could define with little effort the layout of a factory.

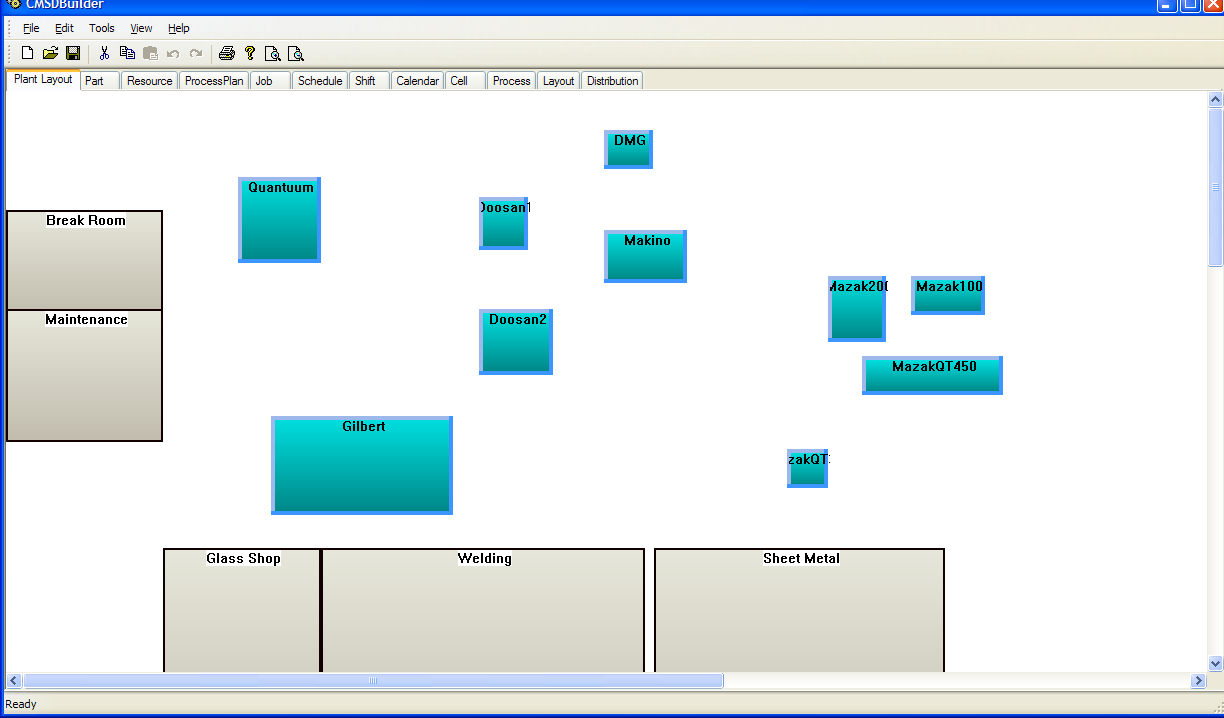
For the NIST job shop, the following CMSD was hard coded into the GUI. First the resources were defined with properties holding the specialized vendor specific information.



Left blank, (None in the case of the CMSD builder) are he mtbdid, mttrid, hourRate, hourlyRateUnit and description – since these parameters are not valued added to understanding our shops. Some properties for each was defined but was out of scope of this document. Then, layout of each resource was defined in the CMSD layout field, containing a coordinate system mapping, placement location, boundary width and depth, and associated resource the layout was defining.



From the CMSD resource and CSMD layout definition, the GUI was able to develop an application specific Plant Layout of the resources and layout, which corresponds to the following overview for the NIST shops layout.



It was determined that although the GUI is a nice feature, hand coding of CMSD is possible and that layout was as important when compared to the value of modeling and simulation. Of note, WTL has a freeware List control which simplified (but did not eliminate!) programming.

## C++ Reflection

One area that was troublesome is the mating of XML to some C++ internal representation. To this end, we maintained CMSD definitions in a simple reflection C++ list that maintained the relationship between XMLSchemas and the model for CMSD and MySQL archiving.

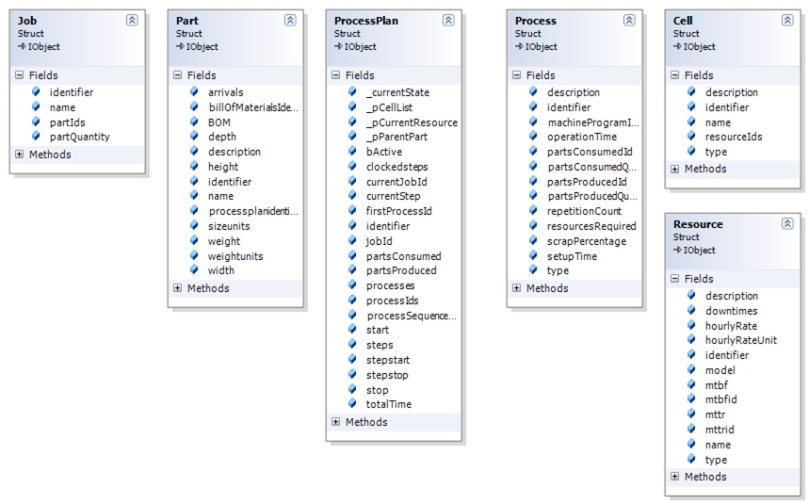


Figure CMSDIntegrator

The CMSD reads or merges from multiple files and reads into a CMSD integrator class. The CMSD integrator class is a class that mirrors the CMSD representation, but is somewhat easier to navigate because all the NULL pointers are removed and most variables are wrapper classes that works like a smart pointer or STL library data collections. After XMLSpy parses the entire CMSD file, all new parses go into the CMSD integrator relevant item (Jobs, Parts, ProcessPlans, Resources, etc.) and fills out the entries based on the MSXML nodes derived from CMSD file. For the resource, the CMSD integrator parses the CMSD and creates either a Cell or a Resource based on the type of CMSD information. CMSD merges are similar in that the entire CMSD file is read by the XMLSpy generated code is parsed into a MSXML structure, and then each item (Jobs, Parts, ProcessPlans, Process, Resource, Cell, Schedule, Calendars, etc.)

#define ASSIGN(X,Y,Z) try { X=Y; } catch(...) { X=Z;};

#define CREATEIF(X,Y) try { if(Y != bstr\_t(L"None") && Y != bstr\_t(L"") ) \

X=std::string((LPCSTR)Y); } catch(...) { };

#define CREATEIFBSTR(X,Y) try { if(Y != bstr\_t(L"None") && Y != bstr\_t(L""))\

X=Y; } catch(...) { };

ASSIGN(name ,((std::string) resource.Name[0]).c\_str(), name);

The MACRO ASSIGN is used pervasively throughout the code to assist in NULL pointer and default assignments. The ASSIGN macro takes three parameters (X,X,Z) and assign Y to X is possible. If Y is a non-existant or NULL then the default value Y is assigned to X. In CMSD assignment code, the macro ASSIGN is used liberally to assign values to the flat CMSD data structures from the given MSXML node. Should the MSXML node be NULL or have an exception value, the default value will be assigned, which in most part is the originally assigned value to each flat CMSD data item "None". Unfortunately, unless the XSD explicitly points out the size of a data structure, it is usually assumed to be a array of elements, and so the MACRO (and other users) must search/find/assign the size of one array spot, and only one array spot. Exception would occur if the array item 1 was selected and the array only had one item 0.

void Resource::Load(MSXML2::IXMLDOMNodePtr node)

{

CMSD::CResource resource = node;

ASSIGN(name ,((std::string) resource.Name[0]).c\_str(), name);

ASSIGN(identifier ,((std::string) resource.Identifier[0]).c\_str(), identifier);

ASSIGN(type , ((std::string) resource.ResourceType[0]).c\_str(), type);

ASSIGN(description ,((std::string) resource.Description[0]).c\_str(), description);

ASSIGN(hourlyRate , resource.HourlyRate[0].Value2[0].GetNode()->text, hourlyRate);

ASSIGN(hourlyRateUnit ,((std::string) resource.HourlyRate[0].Unit[0]).c\_str(),hourlyRateUnit);

PropertyElement().LoadProperties<CMSD::CResource>(resource, properties, distributions);

}

...

for(int i=0; i< data.Resource.count() ; i++)

{

if(Cell::IsResourceCell(data.Resource[i].GetNode()))

{

Cell \* acell( (Cell \*) IObject::CreateSave<Cell>());

acell->Load(data.Resource[i].GetNode());

}

else

{

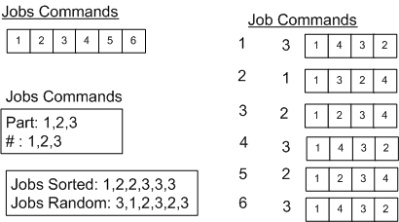
Resource \* aresource ((Resource \*) IObject::CreateSave<Resource>());

aresource->Zero();

aresource->Load(data.Resource[i].GetNode());

}

}



The state machine has the following states: Off, ready, running, stopped, interrupted, faulted, exit, blocked, and starved. The state machine is only concerned with states for timing and sequencing computations. Thus, running is the primary state and calls an internal function Running() where it determines whether to dequeue and queued input job or enqueue a done processing job. If there is no job, and no job is on the Inqueue, then the Resource is “starved” and goes into that state.

All CMSD resources are handled similarly as shown in Figure~\ref{fg:StateMachine}. In this case, a inqueue can buffer awaiting jobs, and an outqueue can buffer finished jobs. The size of these queues can determine the performance of the Resource (with larger queue sizes behaving better but at cost). Thus, instead of separating the Resource from the input/output conveyor, they are bundled under the same state machine and allow easier CMSD parameterization, modification and ultimately optimization. The size of the inqueue and outqueue can be adjusted by the CMSD resource definition that is merged. Some Resources are static in that they do not have a conveyor associated with the Resource, so their in/out buffers wil be one. By default, the size of the in/out buffers is one. Within the CMSD optimization extension, the in/out buffer sizes can be changed up/down within a range to see the most reasonable cost--effective performance.

Within the main thread, a timer is maintained to all the Resources in the system, new jobs are added to the JobHandler pool as necessary, each in/out queue to a Resource is updated, and then each CMSD Resource state machine (any events processed) is updated. The main thread breaks when all CMSD job (in the jobs handler) are done processing. A Delay is done based on the one second minus the length of time required to service each Resource. It is assumed this Delay is always greater than zero, otherwise the loop does not delay.

while(m\_bRunning)

{

ControlThread::threadtimer.restart();

Newworkorder(); // done with job - start new work order

Update(\_resourceHandlers); // update queues to resource

UpdateResourceHandlers(\_resourceHandlers); // update state machine

ControlThread::\_totalTime+=ControlThread::threadtimer.elapsed();

if(AllFinished())

break;

ControlThread::threadtimer.Delay(dLoopTime-ControlThread::threadtimer.elapsed());

}

The boost function double elapsed() const; returns the elapsed time in seconds (as a double) so that dLoopTime is a double in seconds also. The maximum measurable elapsed time may be as low as 596.5 hours (or even less) and this timer cannot be depended upon to be robust.

# Appendix III Miscellaneous

## Cost Functions

A cost function is a mathematical formula used to predict the cost associated with a certain action or a certain level of output. In our case, we can use cost functions to forecast the utility expenses associated with equipment and therefore production, in order to determine what pricing strategies achieve desired profit margins.

The GM case study data had electric, gas, water, hydraulic, and other utility measures associated with the precision sand casting equipment and process. In a sustainable world, measuring utility costs to associate the utilities to particular equipment and processes is obligatory for assessing performance costs. CMSD has costs defined, but these costs are assigned to Processes. In reality we may just want to add up the seconds in a given state and at the end determine the costs - by calculating the total cost as given by time x UtilityCost x EquipmentCost , be it kWh for electrical usage for a piece of equipment, etc. This approach lends itself to an easier global summation of costs for a final cost determination, without hand noting every cost. But breaking down the costs can help in pinpointing excessive process costs. Because cost functions are associated to states, then if monitoring on versus off in a cost, time will only be accumulated when the cost is active.

Ultimately, it was decided as a test to compute cost functions as their occurred but we were constrained by the name/value/unit/description simple properties that were assigned to resources. For example, in the listing below the resource SMCO:LINE1\_PS\_CAST1\_ELV1 (or lift elevator) has the cost function “Upper Conveyor” and Value 0.08 / 3600.0 since we want to change seconds to hour dimensions (60\*60) for kWh and multiply by the scaled rated kWh power use of the resource's equipment i.e., 0.8.

<Resource>

<Identifier>SMCO:LINE1\_PS\_CAST1\_ELV1</Identifier>

<Property>

<Name>Upper Conveyor</Name>

<Unit> KWHz:any </Unit>

<Value>0.8/3600</Value>

<Description>COST</Description>

</Property>

</Resource>

Clearly we could keep the duration in time of the “Upper Conveyor” in the ON state and then compute the cost of kWh at the end based on this value. Instead, we are accumulating the sum of the kWh used the Resource equipment after each interval of simulation and at the end will multiply this value by a global cost of kWh to determine the cost. Then all COST functions (as described as such in the description) will be added up to provide the overall resource including the power cost of the used equipment.

Thus, we used KWHz:any where the any or another state such as faulted could be used to measure all states or an explicit state as necessary.

## Obstacles

The core filling process describes when a sand core arrives and is filled with molten aluminum. In this process, a reusable chiller is inserted into the bottom of the core and then a cover is inserted and glued to the top of the core. After these actions have been performed, the core is then ready to be filled with molten aluminum.

The sand cores are already built and a inventory of the premade cores exists, and the premade cores are stored on an automated storage shelf, where 12 or so cores are stored at a time sans reusable chiller or top. The top of a core is attached separately and storage is an automated rack. Elevators recycle the chiller after it is done to be inserted into the core. Robots lift premade cores onto a series of conveyors to have chill inserted, top attached, and then inverted for core to be filled. These conveyors move the core along the filling process, and robots insert, attach, and lift and invert the cores when the automation is called upon.

Of question within the filling operator are the performance of the conveyors, the energy used to keep the molten liquid, the speed at which the entire operation takes, how many cores are filled how often before being sent to the cooling phase, the typical breakdown of the commensurate equipment, and the best and worst performance as based upon the saved filling data.

The core filing process entails equipment that uses the facility utilities. These utility costs were assigned to resources within as updating costs and these utilities are represented as costs per resource in CMSD that is merged into the DES model.

* AGV moves 2x3 core packages from buffercore.
* Core package removed from buffercore by robot from 2x3 input AGV buffer. Question: where does the robot place the core package (minus top) -- on base plate for core package that allows chiller to be inserted.
* Chill Insert - sand core is place on top of iron casting base plate, and the casting is lifted up and then inserted with the chill insert underneath. (Thus, robot places core package on top of iron casting base plate which is lifted up and inserted with chill insert at same time. - How does robot sense putting object on conveyor - dead reckoning?)
* Alloying element insertion -- before cover put on alloy composition inserted.
* Robot inserts cover on core package -- core covers arrive on an upper conveyor where it is picked by the robot and placed on top of existing core package
* Overhead carrier (gantry) which moves core package from the inbound line to the outbound line.
* Aluminum is pumped from the heated well into the core package
* Core package with cooling casting moving towards chill extract station. Core package and base plate are lifted upwards and the chill is extracted (and sent to the basement for cooling also).
* Core package\& Base Plate with Casting (goes on lowerator to basement for cooling and shakeout)

Assumptions:

* The FMC can process a variety of part types from a large but finite and known population.
* The system is designed in such a way that it has to assign the jobs in a given time.
* Processing times are assumed to be deterministic.
* Transportation time is considered to be negligible.
* The material handling system does not impose constraints (blocking and starving).
* The operation times and setup activities are included in operation times, and
* Machines are subjected to random failures.