Toward Smart Manufacturing Using Decision Analytics

Alexander Brodsky, Mohan Krishnamoorthy, Daniel A. Menascé
Department of Computer Science, George Mason University, Fairfax VA, USA, {brodsky,mkrishn4,menasce}@gmu.edu

Guodong Shao, Sudarsan Rachuri
Systems Integration Division, National Institute of Standards and Technology, Gaithersburg MD, USA
{gshao,sudarsan.rachuri}@nist.gov

***Abstract*—This paper is focused on decision analytics for smart manufacturing. We consider temporal manufacturing processes with stochastic throughput and inventories. We demonstrate the use of the recently proposed concept of the decision guidance analytics language to perform monitoring, analysis, planning, and execution tasks. To support these tasks we define the structure of and develop modular reusable process component models, which represent data, decision/control variables, computation of functions, constraints, and uncertainty. The tasks are then implemented by posing declarative queries of the decision guidance analytics language for data manipulation, what-if prediction analysis, decision optimization, and machine learning.**

***Keywords*-smart manufacturing; decision support; decision guidance; optimization; data analytics**

I. Introduction

Making decisions is prevalent in various domains including manufacturing, supply chain and logistics, and sustainability to name a few. To support decision making, enterprises turned to Decision Support Systems (DSS) [1] and, more recently, to Decision Guidance Systems (DGS) [2], which is a class of DSS that are geared toward producing actionable recommendations to decision makers. In this paper we are concerned with decision guidance for manufacturing processes.

In the past few years, there has been significant technological advancements in different areas of manufacturing process analysis and optimization. These processes often involve physical or virtual inventories of products, parts and materials that are used to anticipate uncertainties in supply and throughput of machines. Over time, the state of the machines, inventories and the entire process changes until process completion. We use the term Buffered Temporal Flow Processes (BTFP) [3] to describe these types of processes. BTFP can be found in many areas of manufacturing including automobiles, furniture, smartphones, airplanes and toys. To perform tasks on BTFP, there is a need to accurately model machines, systems and processes. These models need to represent (a) metrics of machines (such as cost, energy consumption, and emission) as a function of control variables, (b) process routing that describes the flow of materials through the manufacturing floor, and (c) work-in-progress inventories. In BTFP, this needs to be modeled over a temporal sequence and include uncertainty of machines' throughput, inventories and supply.

Consider an example of a DGS for satisfying demand for certain products on a discrete manufacturing floor. The tasks that should be allowed in Smart Manufacturing can be broadly categorized as: (1) monitoring, (2) analysis, (3) planning, and (4) execution decision analytics. The monitoring tasks resemble those of database management systems, dealing with data manipulation and transformation of data (especially temporal sequences) from multiple sources. For example, the production operator may want to monitor the number of items produced, number of items in the inventories, and energy consumed by each machine. DGS will need to generate different aggregated views of this information, continuously, over time.

Analysis tasks may use the techniques of stochastic simulation and statistical learning for regression, classification, and estimation [4]. For example, given the current speed of the machines and inventory details, the process operator may want to estimate the average maximum energy consumed by the manufacturing floor over the next month and identify faults and risks. Prediction and estimation of uncertain outcomes, in turn, may involve regression analysis of functions [5] such as for cost, time, risk, or building classifiers for different categories of outcomes.

Planning tasks involve optimization and sensitivity analysis [6]. The production planner may ask an optimization query, e.g., which machine should run at what speeds and what should be the capacity of the inventory so that the business rules are satisfied and the demand is met at a minimum cost or the planner could do sensitivity analysis such as if noise was introduced in the speed of the machines, how would the cost of production and satisfaction of demand be affected. Planning tasks typically correspond to a deterministic or stochastic optimization problem, possibly using multiple criteria and under various business assumptions. Finally, execution tasks involve making use of the results obtained in the other tasks in a sensible and meaningful way. There is a need for a modular, task-independent, reusable and easy-to-use model to perform the monitor, analysis, planning, and execution tasks on the BTFP class of problems.

As we discuss in Section II (Related Work), today's state-of-the-art technologies have significant limitations for the development of manufacturing DGS. To overcome these limitations, the concept, syntax and semantics of the Decision Guidance Analytics Language (DGAL) and the Analytical Knowledge-Base (AKB) were recently proposed in [7]. The key idea behind DGAL is that of modular reusable and composable models, which are called analytical objects (AOs), are created and stored in the AKB independently of the tasks and the tools that may use these models. DGAL is intended to (1) express models that represent data, decision/control variables, computation of functions, constraints, and uncertainty, and (2) pose declarative queries for data manipulation, what-if prediction analysis, decision optimization and machine learning.

This paper focuses on using DGAL to support the performance analysis of Smart Manufacturing systems. More specifically, we consider Buffered Temporal Flow Processes (BTFP) [3], which are composite temporal manufacturing processes with stochastic throughput and inventories. The research contributions reported in this paper are twofold. First, we define the structure of and develop DGAL modular reusable process components for BTFP, including (1) the generic module for process flows, inventory aggregators, base and composite processes, (2) domain-specific module for a simple case study on wood processing, and (3) a domain-specific module for analytical viewpoints for monitoring, predicting, learning, and optimizing. Second, we demonstrate how to use the DGAL components in AKB to perform monitoring, analysis, planning and execution tasks.

The rest of this paper is organized as follows. Section II discussed related work and its limitations. Section III gives an overview of the model, monitoring, analysis, planning and execution tasks on the manufacturing floor. Section IV gives an overview of DGAL. Section V describes the construction and composition of the AOs for smart manufacturing. Sections VI, VII, VIII, and IX describe the monitoring, analysis, planning and execution tasks. Finally, section X concludes and identifies some directions for future research

II. Related Work And Its Limitations

We borrow from [7] to describe the current state-of-the-art and its limitations. Consider the six classes of tools/languages relevant to modeling analytical tasks in decision guidance systems:

1) Closed domain-specific end user oriented tools, e.g., strategic sourcing optimization modules within procurement applications [8].
2) Data manipulation languages, such as Structured Query Language (SQL), XQuery [9] and JSONiq [10].
3) Simulation modeling languages, such as Modelica [11]
4) Simulation languages, such as Jmodelica and Simulink [12].

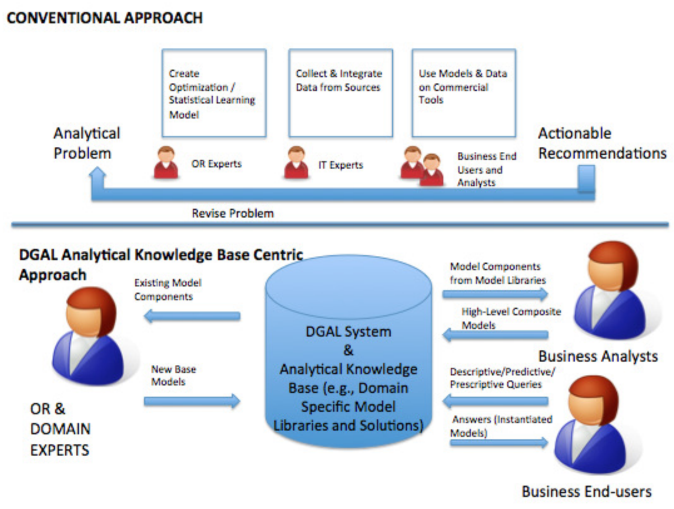
CONVENTIONAL APPROACH
Create Optimization / Statistical Learning Model
Collect & Integrate Data from Sources
Use Models & Data on Commercial Tools
Analytical Problem
OR Experts
IT Experts
Business End Users and Analysts
Actionable Recommendations
Revise Problem
DGAL Analytical Knowledge Base Centric Approach
Existing Model Components
New Base Models
OR & DOMAIN EXPERTS
DGAL System & Analytical Knowledge Base (e.g., Domain Specific Model Libraries and Solutions)
Model Components from Model Libraries
High-Level Composite Models
Business Analysts
Descriptive/Predictive/ Prescriptive Queries
Answers (Instantiated Models)
Business End-users

Figure 1. Conventional Approach vs DGAL Approach

5) Optimization modeling languages, such as A Modeling Language for Mathematical Programming (AMPL) [13], General Algebraic Modeling System (GAMS) [14], and Optimization Programming Language (OPL) [15] for Mathematical Programming (MP) and Constraint Programming (CP).
6) Statistical learning languages/interfaces, such as Predictive Model Markup Language (PMML) [16].
7) Multi-Agent reasoning tools such as the heterogeneous temporal probabilistic (HTP) agent [17] and Interactive Maryland Platform for Agents Collaborating Together (IMPACT) agent platform [18][19].

Domain-specific tools may be easy to use for a particular well-defined task, but are not extensible to reflect the diversity of emerging descriptive, predictive and prescriptive analytical tasks. Nor do they support compositionality, i.e., the ability to compose their (white-box) models to achieve system-wide- optimal predictions and/or prescriptions (e.g., actionable recommendations), rather than system-component-optimal predictions and prescriptions.

Simulation languages and tools have the advantage of their modeling expressivity, flexibility, and OO modularity, which support reusability and interoperability of (black-box) simulation models. However, performing optimization using simulation models/tools is based on (heuristically-guided) trial and error. Because of that, simulation-based optimization is significantly inferior, in terms of optimality of results and handling computational complexity, to MP and CP tools/algorithms, for problems expressible in supported analytical forms (e.g., MILP) [20]. Also, while sufficiently expressive, simulation languages were not designed for declarative and easy data manipulation provided by data manipulation languages such as SQL, XQuery, and JSONiq.

Because optimization modeling languages such as AMPL, GAMS or OPL are used with MP and CP solvers, which use a range of sophisticated algorithms that leverage the mathematical structure of optimization problems, they significantly outperform simulation-based optimization, in terms of optimality and running time. However, optimization modeling languages are not modular, extensible, reusable, or support compositionality; nor do they support low-level granularity of simulation models. Statistical learning languages/tools have similar limitations and advantages, because most are based on optimization.

Because of such diversity of computational tools, each designed for a different task (such as data manipulation, predictive what-if analysis, decision optimization or statistical learning), modeling typically requires the use of different mathematical abstractions/languages. Essentially, the same underlying reality must often be modeled multiple times using different mathematical abstractions. Furthermore, the modeling expertise required for these abstractions/languages is typically not within the realm of business analysts and business end users.

Perhaps most problematic in decision guidance modeling today is the fact that it is task-centric: every analytical task is typically implemented from scratch, following a linear, non-reusable methodology of gathering requirements, identifying data sources, developing a model/algorithm using a range of modeling languages and tools, performing analysis (see the top of Fig. 1, the Conventional Approach). Using the conventional task-centric approach, models and algorithms are difficult to develop, modify, and extend. Furthermore, they typically are not modular or reusable, nor do they support compositionality.

The Modelica simulation modeling language was designed to reuse knowledge. It allows a detailed level of abstraction, including OO code and differential equations [11]. Modelica by itself is not a language for performing optimization, learning, or prediction. But there are tools such as JModelica for simulation, and Optimica for simulation-based optimization [12]. However, because of the low level of abstraction allowed in Modelica, general Modelica models cannot be automatically reduced to MP/CP models and solved by MP/CP solvers.

The Sustainable Process Analytics Formalism (SPAF), developed at National Institute of Standards and Technology (NIST), was designed for the creation of modular reusable model components [21]. However, SPAF was mainly proposed for optimization and was not designed to support data manipulation, stochastic predictions, and machine learning.

To overcome these limitations, the concept, syntax and semantics of the Decision Guidance Analytics Language (DGAL) and the Analytical Knowledge-Base (AKB) were recently proposed in [7], which we use in this paper to support process tasks described in the next section.

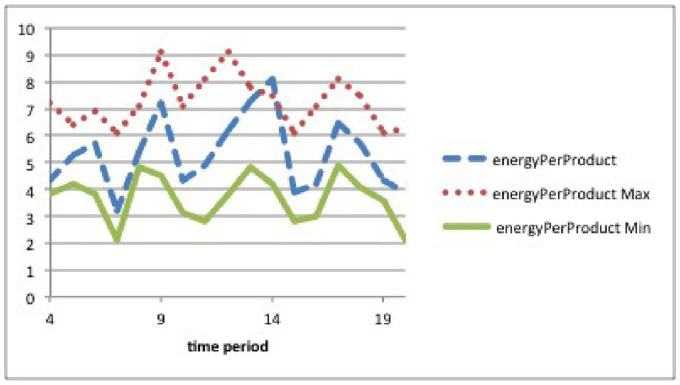

Figure 2. Monitored total *energyPerProduct* and predicted total *energyPerProduct Max*, *energyPerProduct Min* for *currentTime* = 20 and *timeWindow* = 4

III. Process Modeling, Monitoring Analysis, Planning and Execution

The process modeling will be performed by initializing and composing smaller components into larger ones so that the manufacturing floor can be mathematically represented as a whole. The models make use of reusable objects that map directly to the components on the manufacturing floor that are easy to compose and initialize. The composed models support the tasks of monitoring, analysis, planning and execution. The process models also support stochastic metrics that allows the tasks to be run on components that emulate the manufacturing floor more accurately.

The monitor phase is used to collect information, through sensors in an automated or semi-automated way, on individual machines as well as on the entire environment. Examples of data collected by sensors include the quantity of items used in the buffer, the speed and the energy of the machines running on the manufacturing floor. Data is collected by a variety of sensors. The monitoring phase is also used for preprocessing, cleaning and filtering the raw data collected by sensors. This phase also uses the sensor data for predicting the minimum and maximum bounds for metrics such as total energy per product. An example of the minimum and maximum predictions of the total energy per product over a time window is given in Fig. 2.

The analysis phase is used for model/parameter calibration, prediction and diagnostics. A process operator may want to learn a functional relationship between energy consumed or cost incurred and the throughput of the manufacturing process. Diagnostics is the process of detecting faults (i.e., any deviation from what is considered to be normal behavior) in the manufacturing process so that the best corrective actions can be determined. For instance, in Fig. 2, the process operator can see that the actual energy consumed per product by the manufacturing floor is more than the predicted maximum at the 14^{th} time period. So here, the operator can detect and plan for such faults.

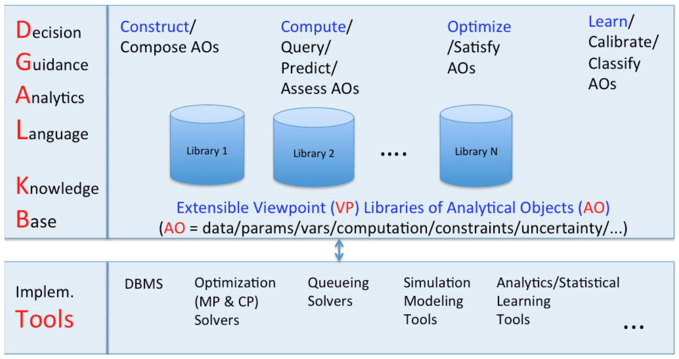

Figure 3. Smart manufacturing MAPE-K framework

The plan phase uses the results of the analysis phase in order to plan the optimal actions necessary for prognostics, i.e., to fix faults or to improve the efficiency of the manufacturing process as a whole. For instance, upon discovering the spike in energy consumed per product in the 14^{th} period in Fig. 2, the process planner may want to redistribute the load among the machines or along the time horizon so as to mitigate the problem. The plan phase can also be used to determine the expected throughput so that the cost or energy consumed is minimized.

IV. Overview of Decision Guidance Analytics Language

In this section we borrow from [7] to overview the concept of the Decision Guidance Analytics Language. The following are the design principles and features of DGAL:

- Reusable-KB-centric modeling approach: DGAL must support the paradigm shift from non-reusable task-centric approach to reusable KB-centric approach.
- Task-independent representation of analytical knowledge: DGAL AOs must represent analytical knowledge (including data and its structure, parameters, control/decision variables, constraints and uncertainty) uniformly, regardless of the tasks (e.g., computation, prediction, optimization or learning) that may be using it.
- Unified language for analytical knowledge manipulation: DGAL must uniformly support (1) data manipulation (with the ease of data manipulation languages like SQL and JSONiq), (2) deterministic and stochastic computation/prediction, (3) decision optimization based on MP/CP, (4) statistical/machine learning, and (5) construction/composition of AOs.
- Flexible construction of analytical knowledge: DGAL must support modular AO composition, generalization, specialization and reuse. Algebra over analytical KB: DGAL operators must form an algebra over the set of well-defined AOs, that is, operators applied to AOs (in the AKB) must return an AO. Thus, the resulting AOs can be put back into the AKB, and then used by other operators. Note, this is analogous to data manipulation languages (such as SQL, XQuery and JSONiq), which are algebras over the corresponding data model (relational, XML or JSON).
- Declarative high-level language: DGAL analytical knowledge manipulation operators (compute, optimize, learn) must be declarative and simple for end users.
- Compact language core: It is desirable for DGAL to have a compact core, and allow additional functionality through built-in and user-developed libraries (in the knowledgebase).
- Ease of use by modelers: DGAL should be easy to use by mathematical modelers and software/DB developers.
- Ease of use by end users: DGAL should enable built in KB libraries of AOs to raise the level of abstraction (obscure mathematical detail, etc.), which would make it easy to use by end users (such as business analysts and managers).

The high-level DGAL framework and functionality is depicted in Fig. 3. Central to the framework is the Analytical Knowledge Base, which is a collection of Analytical Objects (AO). AO is the base component of analytical knowledge. Each AO can represent, uniformly:

- Data and typing: we adopted the Java Script Object Notation (JSON) as the data model, a language-independent format.
- Within the data, decision/control variables and parameters over reals, integers and other domains
- Computation of functions represented via JSON data manipulation language, JSONiq, extended with indexed access, and equation syntax of OPL.
- Constraints, borrowing from the equation syntax of OPL, but using JSONiq sequences for index sets (over summation, universal quantification, etc.)
- Uncertainty, by adding distribution functions to expressions (in functions and constraints), which implicitly define random variables.

All DGAL operators are applied to AO and return an AO, and so DGAL constitutes an algebra (like data manipulation languages SQL, XQuery and JSONiq). The DGAL operators are of four key types (see upper part of Fig. 3):

- Construct: this class of operators allows to construct an AO from scratch, from another AO by specialization/generalization, or by composing an AO from previously defined AOs.
- Compute: this class of operators instantiates an AO by perform computation of functions (may involve uncertainty quantification).
- Optimize: this class of operators instantiate an AO by finding values of decision variables that optimize an objective, and then compute it with the optimal values
- Learn: this class of operators instantiate an AO by

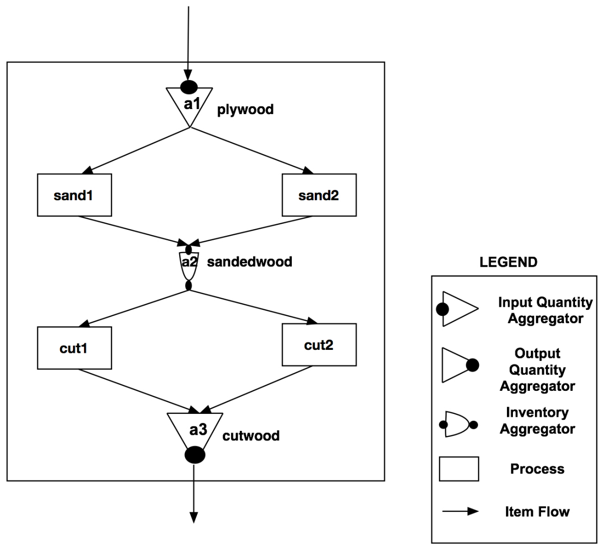

Figure 4. A graphical notation for the sand and cut manufacturing floor

finding values of its parameters, as to minimize an estimation error against a learning set.

The (deterministic) optimization and learning operators are performed by creating a formal MP/CP model and solving it using an appropriate solver (see lower part of Fig. 3). The AO, AKB, and DGAL operators are designed according to the designed principles outlined in this section. We further explain DGAL through the manufacturing examples in the following sections.

V. Modeling Manufacturing Process in DGAL

Manufacturing floors can be complicated to model and may contain very tricky structures. It also requires reusable components that are easy to compose to perform various monitoring, analysis, planning, and execution tasks. DGAL is intended to be a declarative high level language that provides for analytical knowledge manipulation and reusability. DGAL is also easy to use for modelers and end users and provides for task-independent representation of analytical knowledge that may include structures, parameters, decision variables, constraints and uncertainty. Therefore, in this section, DGAL is used to model the complex analytical knowledge of the manufacturing floor. In order to describe this model, first, a composition of a small manufacturing example in DGAL is described. Then, DGAL is used to model the components used in this composition in a compact and reusable way. These components map directly to the machines and inventories on the manufacturing floor.

Consider the sand and cut manufacturing processes shown in Fig. 4 that takes plywood as its input. In this example, a part of the plywood goes to the sand1 machine and the remaining goes to the sand2 machine. The sanded plywood is then buffered and redistributed among the cut1 and cut2 machines. The sanded plywood is cut in these machines and finally, the cut plywood is collected and provided as output from the sand and cut manufacturing processes. We assume that time is divided into time intervals of duration Δt and that time intervals start and end at time points. We assume without loss of generality that Δt = 1. A time interval (also known as a period) is denoted by $p_{i+1} = (t_i; t_{i+1})$.

In order to compose the sand and cut manufacturing processes in DGAL, six modular components will be initialized to map them to the physical entities of the floor. The first component is the input quantity aggregators (IQA) such as *a1*, that get the items from the input and distributes them among the *sand1* and *sand2* processes. The second component is the base process that maps to the machines on the floor. This process has controls for the speed of the machines and the metrics for energy consumed and cost of running the machine. In our example, base processes are the *sand1*,

```
declare variable sandCutProcess := $smc:compositeProcess {
let
      $plywood := $smc:itemFlow{ $mt := “plywood” },
      $cutwood := $smc:itemFlow{ $mt := “cutwood” },
      $a1 := $smc:IQA{
            ... // Inputs and outputs are initialized and installed similarly to the IA, a2
      },
      $a2 := $smc:IA{
            $mt := “sandedwood”,
            $inputIds := [“inputSandedwood1”, “inputSandedwood2”],
            $outputIds := [“outputSandedwood1”,”outputSandedwood2”],
            $inputid…Values := ({id: inputSandedwood1,
                  inputFlow: $sand1.$outputFlow, inAllocRatio: 0.5},
                  {id: inputSandedwood2, inputFlow: $sand2.$outputFlow,inAllocRatio: 0.5}),
            $outputId…Values := ({id: outputSandedwood1,
                  outputFlow: $cut1.$inputFlow, outAllocRatio: 0.4},
                  {id: outputSandedwood2, outputFlow: $cut2.$inputFlow, outAllocRatio: 0.6}),
            $capacity := 10, $initInv := 1
      },
      $a3 := $smc:OQA{
            ... // Inputs and outputs are initialized and installed similarly to the IA, a2
      },
replace
      $inputIds := [“inputPlywood”],
      $outputIds := [“outputCutwood”],
      $inputid…Values := ({id: inputPlywood, inputFlow: $plywood}),
      $outputId…Values := ({id: outputCutwood, outputFlow: $cutwood}),
      $itemFlowIds :=[“plywoodToSand1”,”plywoodToSand2”,...],
      $itemFlowId…Values := ({id: plywoodToSand1, ifvar: $sand1.$inputFlow}...},
      $subProcessIds := [“sand1”,”sand2”,”cut1”,”cut2”],
      $subProcessId…Values := ({id: sand1, pvar: $sand1}, {id: sand2, pvar: $sand2},...),
      $inputQtyAggrIds := [“a1”], $inputQtyAggrId…Values := ({id:a1, iqavar:$a1}),
      $inventoryAggrIds := [“a2”], $inventoryAggrId…Values := ({id:a2, iavar:$a2}),
      $outputQtyAggrIds := [“a3”], $outputQtyAggrId…Values := ({id:a3, oqavar:$a3}),
}
```

Figure 5. Composition of the sand and cut manufacturing floor analytical object in DGAL

```
declare variable $sand1 := $smc:baseProcess{
      let         $plywoodToSand1 := $smc:itemFlow{ $mt := “plywood” },
                  $plywoodFromSand1 := $smc:itemFlow{ $mt := “plywood” },
      replace     $inputIds := [“plywoodToSand1”],
                  $outputIds := “plywoodFromSand1”,
                  $inputId…Values :=({id:”plywoodToSand1”,inputFlow:$plywoodToSand1,$inputPerOutput: 2}),
                  $theOutputFlow := $plywoodFromSand1,
                  $capacity := 5.5,
                  $coef…values := ( {coef: "slope1", energyPWLvalue: 2.5, costPWLvalue: 5.0}, ...),
                  $machineSigma := 0.3,
                  $initLeftOver = 0
},
```

Figure 6. Composition of the *sand1* analytical object in DGAL

sand2, *cut1* and *cut2* components. The third component is the inventory aggregators (IA) such as *a2* that provide the analytical knowledge for a work-in-progress inventory. The fourth component is the output quantity aggregators (OQA) such as *a3*, which collect the items from *cut1* and *cut2* machines and dispense them from the floor. The fifth component is the item flows that carry items from the input or output or between the processes and aggregators. Finally, the sixth component is that of the composite process that contains one or more of the six components mentioned here to represent the manufacturing processes as a whole. For instance, let's assume that sand1 process actually consisted of two baseProcesses and their outputs were considered to be the output of the sand1 process. Then this situation can be handled by creating a composite model of these two baseProcesses along with their flows and aggregators and the sand1 process would be replaced by this composite model consisting of two baseProcesses. Note that BTFP processes are applicable in domains where allocation of resources to processes is fixed, e.g., in mass production. Therefore, we do not distinguish, for the purpose of decision analytics, between atomic processes and machines (i.e., resources) that support them. The composite process also provides the global metrics such as total cost and total energy consumed by the entire floor.

Each component has a base version in the library (AKB) that can be reused repeatedly until a process modeler can represent the entire manufacturing operation as a composite process. The DGAL for the parameters, variables, and constraints of the base version of these AOs are discussed later in this section.

The process modeler can perform the composition of the sand and cut manufacturing processes in DGAL by just instantiating the inputs, outputs and metrics in the base version of the AO. All base processes will first construct and install their input and output flows. This procedure is shown later in this section as the construction for *sand1* process (Fig. 6). Then, the modeler will use the base composite process and initialize its inputs and outputs as those AOs that map to the respective flows on the manufacturing floor. Then, the modeler maps the inputs and outputs of the IA, IQA, and OQA aggregators to those previously defined by the base processes, thus connecting the aggregators to these processes. Finally, the metrics in the base composite process are replaced to reflect the composite view of our example as the AO *sandCutProcess*. The DGAL code for this construction is shown in Fig. 5. All the base components are installed by initializing the parameters of the index sets in the composite process. For instance, the parameter values of the index *subProcessId* are initialized with the ids from *subProcessIds* array and the AO of the previously created baseProcesses (*sand1*). In this way, previously created AOs can be reused. The composed *sandCutProcess* for our example is also stored in the AKB.

The AOs for the six components mentioned above are DGAL encoding of the tMQL component metrics and constraints discussed in [3]. These AOs capture the entire process, aggregator and flow analytical knowledge into reusable modules that are independent of the task performed on them. Hence the same individual or composed module can be used to perform many different tasks as shown in the sections below. In this section, these base AOs are discussed briefly. The code for these AOs can be found in appendix A.

The AO for the baseProcess captures the analytical knowledge of a single stochastic machine. Thus, this AO contains the inputs and output item flows indexed on their ids. The three dots (...) as values mean that these variables are parameters, the values for which will be provided before any tasks are run on these AOs. The modeler can set the *machineSigma* parameter to specify the level of noise in the speed of the machine. Each baseProcess has the piecewise linear function (PWL) for cost and energy whose parameters are the values of the coefficient index sets of *energyPWLvalue* and *costPWLvalue* respectively. The expectation of the speed of the machine is given by the *throuputExp* variable for each period and has a value for a decision variable. These variables will be determined when an optimization query is run on these AOs. In order to handle the stochastic nature of the machine speeds on a manufacturing floor, the *throughputControl* variable is computed as a function of the *throuputExp* and noise that has a gaussian distribution. This random speed of the machine is used in the computation of the metrics of the AO to better emulate the randomness among the components metrics on the floor. The *capacity* of the machine is the maximum producing power of the machine. Finally, the baseProcess contains the metrics for the total cost (*totalCost*) and total energy (*totalEnergy*) that is a cumulative of the cost and energy variables calculated per period. It is also possible to return the values of the variables and constraints as a JSON collection after a task is run on this AO.

The AO for the aggregators (IA, IQA and OQA) also contain inputs and outputs indexed on their ids. In addition, the IA contains metrics like *invQty* that gives the number of items in the buffer, *initInv* that gives the number of items at the start of the time horizon and *capacity* that is the total capacity of the IA. The AO for itemFlow acts as an interface between the aggregators and processes and contains metrics like *periodQty* that is the number of items in the flow and *tpAlloc* that bounds the number of items in the flow. From the baseProcess, it is now possible to initialize a new AO that maps to the *sand1* machine. This process first creates and installs the itemFlow for its inputs and outputs as shown in the declaration of the *plywoodToSand1* and *plywoodFromSand1* variables respectively in Fig. 6. Then the parameters such as *capacity*, *initLeftOver* and *machineSigma* parameters of the *sand1* process can be initialized as per the process design. The *sand1* process is added to the AKB and

```
collection{"machineMeters"}:
        {machine: "sand1", tp: 0, kwh: 502.5},{machine: "sand1", tp:1, kwh: 507.2},...
collection{"machineSetting"}
        {machine: "sand1", period: 1, thru: 4.5},{machine: "sand1", period: 2, thru: 5.0}, ...
collection{"inventoryQty"}
        {inventory: "a2", tp: 0, qty: 17},\ {inventory: "a2", tp: 1, qty: 19},...
collection{"cumInput"} {tp:0, qty:14},{tp:1, qty:13},...
collection{"cumOutput"} {tp:0, qty: 4},{tp:1, qty: 3},...
```

Figure 7. Raw process and aggregator data for the sand and cut manufacturing floor

```
declare variable $ns:sandCutPredictingView as aObject := { let
$currentTime := …,  // must be greater than $timeWindow
$timeWindow := …, $probability := …, $machineSetting := …, $inventoryQty := …,
for $i in range($timeWindow,$currentTime) let (
        $tp as index := {tp:$i},
        $globalSetting($tp) := $smg:globalSetting { replace $noPeriods:=$timeWindow, $periodLength := 1.0 },
        $predictedPocess($tp) as  := $smp:sandCutProcess { let
                $globalSetting := $globalSetting($tp),
                for $ma in $sc:sandCutProcess.$subProcessIds,
                        $currP in range($tp-$timeWindow, $tp),
                        $ms in $machineSetting
                where $ma = $ms.machine && $currP = $ms.period replace (
                        $pvar({id:$ma}).$throughputExp({period:$currP}) :=  $ms.thru
                )
                for    $ia in $sc:sandCutProcess.$inventoryAggrIds, $iq in $inventoryQty
                where $ia = $iq.inventoy && $tp-$timeWindow = $iq.tp let ( $iavar({id:$ia}).$initInv := $iq.qty)
                $averageEnergyPerProduct := sum( for $ma in $sandCutProcessIn.$subProcessIds,
                        $currP in range($tp-$timeWindow, $tp))
                        $sandCutProcessIn.$pvar({id:$ma}).$energy({period:$currP})) /
                        sum(for $currP in range($tp-$timeWindow, $tp)
                        $sandCutProcessIn.$outputFlow({id:cutwood}).$periodQty({period:$currP}),
                $predictedMax as float := ?
                $belowPredMax as constraint:=
                                prob(averageEnergyPerProduct >= $predictedMax) <= $probability,
        }
        $optimizedEnergyMax($tp) := minimize({aObject: $predictedProcess($tp), objective: "predictedMax"}))
        return{tp: $tp.tp, maxKwhPerProduct: $optimizedEnergyMax.$predictedMax
}
compute ($ns:sandCutPredictingView { let                                        //usage example
                $timeWindow := 6, $currentTime := 30, $probability := 0.05,
                $machineSetting := collection("machineSetting"), $inventoryQty := collection("inventoryQty")
        }
)
```

(a)

```
declare variable SandCutMonitoringView as aObject := { let
$timeWindow := …,
$currentTime := …,  // must be greater than $timeWindow
$machineMeters := collection("machineMeters"),
$cumOutput := collection("cumOutput"),
for $tp in range($timeWindow,$currentTime) let (
$kwhPerTimeWindow :=
        sum ( for $ma in $sc:sandCutProcess.$subProcessIds, $me in $machineMeters
                  where $ma = $me.machine && $me.tp = $tp) $me.kwh) -
        sum ( for $ma in $sc:sandCutProcess.$subProcessIds   , $me in $machineMeters
                  where $ma = $me.machine && $me.tp = $tp - $timeWindow) $me.kwh,
$prodsPerTimeWindow := sum (for $o in $cumOutput where $o.tp = $tp) $o.qty) -
                sum (for $o in $cumOutput where $o.tp = $tp - $timeWindow) $o.qty,
$kwhPerProduct := $kwhPerTimeWindow /  $prodsPerTimeWindow,
return {
        tp: $tp, kwhPerTimeWindow: $kwhPerTimeWindow,  prodsPerTimeWindow: $prodsPerTimeWindow,
        kwhPerProduct: $kwhPerProduct}
}}
```

(b)

Figure 8. Code for (a) extracting the observed total *energyPerProduct* from sensor data, (b) extracting the observed total *energyPerProduct*

can be reused to query about itself or can be composed into a larger composite process as shown in Fig. 6 to be used in a more global query.

VI. Monitoring Tasks

Once the manufacturing problem has been modeled using the composite process AO, a process operator may decide to run tasks on the composed AOs. The manufacturing floor may generally have sensors on machine and aggregator components. These sensors collect useful data on the floor components over a time horizon. The collected data have their own JSON schema, usually stored in a sensor data repository. The sensor data repository could be stored in a database or may use log files that are stored in the file system. Also, raw sensor data may be processed on the fly, without being persistently stored. Assume that we have four such sensors in our example (Fig. 7). The first sensor could be at the IA *a2*, which collects the *invQty* metric or the number of items in the *a2* at each time point (*inventoryQty* collection in Fig. 7). The second sensor could be at the machines that collect the throughput and the power meter reading in kWh (*machineSetting* and *inventoryQty* collections in Fig. 7). The third and fourth sensors could be at the input and output that collect the number of items coming in and the number of items leaving the sand cut process (*cumInput* and *cumOutput* collection in Fig. 7).

```
{tp: 6, kwhPerTimeWindow: 25.8, prodsPerTimeWindow: 6, kwhPerProduct: 4.34.},
{tp: 7, kwhPerTimeWindow: 42.4, prodsPerTimeWindow: 8, kwhPerProduct: 5.3}, …
```

(a)

```
{tp: 6, minKwhPerProduct: 7.5, maxKwhPerProduct: 3.3},
{tp: 7, minKwhPerProduct: 8.1, maxKwhPerProduct: 4.1}, ...
```

(b)

Figure 9. JSON collections for (a) observed total *energyPerProduct* from sensor data, (b) predicted min and max total *energyPerProduct*

Since DGAL uses JSON at its foundation, it is possible for a process operator to analyze the total *energyPerProduct* of the sand cut manufacturing floor. To do so, the operator writes code (see Fig. 8) to produce JSON collections (Fig. 9) that are used for a chart drawing (for example see Fig. 2).

The code that extracts the sensor values of the observed total *energyPerProduct* is shown in Fig. 9(a). The computation is performed as a sliding window starting from the time window (*timeWindow*) on the left and ending at the current time on the right (*currTime*). For each time window interval, the energy consumed (*kwhPerTimeWindow*) is calculated from the sensor data for all processes as the difference of the power meter reading for the interval. The number of output items produced (*prodsPerTimeWIndow*) in the time window interval is the sum of all items produced in each period of the interval. The total energyPerProduct of the time window interval is then calculated from the above two values. Finally, the computed values are returned to give the observed total *energyPerProduct* in a JSON collection (see Fig. 9(a)).

The code to predict the maximum total *energyPerProduct* is shown in Fig. 8(b). The computation is performed as a sliding window as described above. DGAL allows the reuse of the sand cut composed AO (*sandCutProcess*) by creating a copy and initializing it with the monitored values (*sandCutProcessIn*). This enables DGAL's AKB to have a compact knowledge core of base AOs, composed AOs and task-ready AOs. First, the throughputs for the time window interval are initialized from the sensor data as *throutputExp* for each machine. Second, the inventory quantities from the sensor data at the start of the interval are initialized

```
declare variable $ns:sand1LearningView as aObject := { let
    $ns:globalSetting as $smg:globalSetting := …,
    for $i in range(1, $ns:globalSetting.$noPeriods) let (
        $period as index := {period: $i},
        $ns:cost($period) := …,  $ns:energy($period) := …,  $ns:throughputControl($period) := …,
    ),
    $error as float := sum($period in $periodIndex) (
        $energy($period) - $sc:sand1.$energyPWL($throughputControl($period))) ** 2 +
        $cost($period) - $sc:sand1.$costPWL($throughputControl($period))) ** 2
    ),
    $learnedSand1 as $smg:process:=learn ({aObject: $sc:sand1, params:["coef…values"],
                                            error: $error}),
    }}
collection("learnedSand1input"): {period:1, throughput: 4, cost:54, energy: 19},
                                 {period:2, throughput: 5, cost:62, energy: 21}, ...   // usage example
$ns:period…values := collection("learnedSand1"),
return compute(learningView{$period…values := $ns:period…values}.$learnedSand1)
```

Figure 10. Code for learning the cost and total energy PWL coefficients in *sand1*

```
$sc:sand1LearningView { replace $coef…values := collection("coefIndex")} where:
collection("coefIndex") is:
    {coef : slope1, energyPWLvalue: 2, costPWLvalue: 5},
     {coef: bound1, energyPWLvalue: 10, costPWLvalue: 10},
    {coef: slope2, energyPWLvalue: 2.5, costPWLvalue: 5.5},
    {coef: bound2, energyPWLvalue: 20, costPWLvalue: 20},
    {startX: slope1, energyPWLvalue: 1, costPWLvalue: 1},
    {coef: startY, energyPWLvalue: 2, costPWLvalue: 5}
```

Figure 11. Analytical object sand1 returned by the learn operator

as *initInv* in the new AO. Then, we define a new decision variable called *predictedMax* and also define a constraint of the probability that *averageEnergyPerProdcut* will be greater than the predicted maximum is less than some threshold probability (*belowPredictedMax*). This constraint signifies that the likelihood of the total *energyPerProduct* for the stochastic process to be greater than the maximum value is less than the probability value used in the constraint. For each of the time window intervals, the corresponding AO is used to minimize the *predictedMax* value, so that the maximum total *energyPerProduct* can be found for that interval. Finally, the computed values are returned to give the predicted maximum total *energyPerProduct* in a JSON collection (see Fig. 9(b)).

The minimum total *energyPerProduct* prediction code is complementary to the maximum prediction code and has been omitted here. Using the JSON collections in Fig. 9, the operator can now build a chart as shown in Fig. 2. The process operator can now make an accurate assessment of whether the actual readings were within the bounds of the predicted maximum and minimum values with certain probability. For instance, in Fig. 2, the operator can see that the actual energy consumed is more than the predicted maximum at the 14^{th} period. This approach provides a good diagnostic tool to assess the performance of a component on the manufacturing floor. This approach can also be used by a process planner to plan for any spikes in consumed energy.

VII. Analysis tasks: calibration, prediction and model-based diagnostics

In addition to monitoring, it is also possible to use the AOs for analysis tasks. The energy and cost PWL functions require PWL coefficients as parameters. It may not always be possible for the process operator to determine their values. Also, most manufacturing floors have logs that record machine speeds, cost of running a machine and the energy it consumes for the time horizon. It is then possible to learn the cost and energy coefficients from these logs. Since the AOs created for the baseProcess are independent of the tasks performed, we can reuse the *sand1* base process initialized in section V to learn the energy and cost PWL coefficients.

Fig. 10 shows the code for learning the PWL coefficients from the logs. Assume that the logs are stored as a JSON collection. We can then store the collection of *cost*, *energy* and *throughputControl* into the index variable that is indexed on periods of the time horizon. The error is given as the sum of the squares of errors of *energy* and *cost*. We use the least squares method to learn the coefficients of cost and energy PWL that would give approximately the same cost and energy as the historical data. The output of the learn function is the *sand1* AO with the energy and cost PWL coefficients replaced (Fig. 11).

Other analysis tasks that can be performed include prediction and model-based diagnostics. These tasks overlap with the prediction of the minimum and maximum total *energyPerProduct* (Fig. 2). This provides a good diagnostic tool to assess the running of the component on the manufacturing floor. The process operator can use this tool to monitor the current metrics coming from the sensors and if these metrics go above the maximum or below the minimum, then the operator can diagnose the component that is at fault with certain probability and take the necessary steps to mitigate the fault.

VIII. Planning tasks: process optimization and what-if analysis

Planning and what-if analysis tasks can be performed on the AOs. A process planner may want to optimize certain metrics like minimizing total cost of production. The planner is interested in knowing what machines should be on or off and at what expected speed should these machines run at, such that the demand is met and all other constraints of the composite AOs are satisfied with certain probability. This task can be easily performed by the process planner by just initializing the parameters of the composite AO *sandCutProcess* and then running the minimize task with the objective as the expected cost of production:

$optimizedProcess := minimize ({
aObject: $ns:sandCutSchedulingView,
objective: Exp (sandCutProcess: totalCost}))

The output of such an optimize query is the *sandCutOptimizingView* AO with the expected throughput (*throughputExp*) values for each period replaced with the ones found by the optimizer.

It is also possible to perform what-if analysis tasks on the same composed AO. The planner may be interested to know:

given a particular planned machines' throughput, and load distribution among the machines, what would be production output, work-in-progress inventories, for each time interval over the time horizon, as well as overall manufacturing key performance indicators (KPIs) such as cost and total energy. This task can also be easily performed by the process planner by just initializing the parameters and variables of the composite AO *sandCutProcess* and then running the nondeterministic compute:

$computeProcess := nd-compute ($sandCutProcess)

The output of the compute query is the JSON collection with all the variables instantiated to the computed values that satisfy the constraints with certain probability.

IX. Execution Tasks

As indicated in previous sections, the output of the monitoring, analysis and planning tasks can be JSON collections or AOs. This depends upon the query being used to execute these tasks. The learn query returns the input AO with the learned metrics replaced, the optimize (minimize/maximize) query returns the input AO with the decision variables replaced and the compute (nd-compute) query returns the JSON collection computed from the input that satisfies all of its constraints. All the returned AOs can further be computed to obtain the JSON collection that corresponds to the learned or optimized initializations.

The JSON schema returned by these queries may not always match what the end-user expects or parses. Hence, it may be required that before the results of these tasks are returned back to the end-user, the result be put in an accurate JSON schema. This can be easily performed by using any off-the-shelf JSON parser. Once the results are packaged in the right schema, it is possible for the end-user to parse, understand and make use of them in a chart drawing software or to initialize other AOs.

X. Conclusion and Future Work

In this paper, we (1) consider composite manufacturing temporal processes with inventories and uncertainty, (2) report on the developed DGAL library of modular reusable process components for buffered temporal flow processes and (3) demonstrate how to use the DGAL library to perform monitoring, analysis, planning and execution tasks. The proposed approach is unique in that it allows modeling reusable manufacturing system components once, and uses them uniformly and easily, through high-level declarative queries, for diverse tasks.

The practical contribution of this research is enabling a significantly faster and less expensive development of decision guidance applications for smart manufacturing, which utilizes reusability of component models, yet allows to use the best technology/algorithm to support each analytical task. The novelty and the challenge to enable this approach involves the development of formal semantics of the underlying language (DGAL) and sound and complete reduction procedures and specialized algorithms, which is the subject of further research.

Many research questions remain open. They include (1) developing systematic manufacturing machine and process libraries for various domain-specific viewpoints, (2) implementation of the DGAL language via reduction algorithms from DGAL queries and analytical objects to specialized formal models of optimization and statistical learning; (3) developing graphical user interfaces for domain-specific languages based on DGAL, (4) developing efficient algorithms for stochastic predictions and optimization based on deterministic model approximations, and (5) development of specialized algorithms that can utilize preprocessing of stored (and therefore, static) AO's to speed up optimization, generalizing the results in [22].

Acknowledgment

The work represented here was partially funded through cooperative agreement #70NANB12H277 between George Mason University and the National Institute of Standards and Technology (NIST). The authors thank Kevin Lyons and Sharon Kemmerer from NIST for their valuable comments and suggestions which helped improve the paper.

References

[1] J. P. Shim, M. Warkentin, J. F. Courtney, D. J. Power, R. Sharda, and C. Carlsson, "Past, present, and future of decision support technology," *Decis. Support Syst.*, vol. 33, no. 2, pp. 111–126, Jun. 2002.

[2] A. Brodsky and X. Wang, "Decision-guidance management systems (dgms): Seamless integration of data acquisition, learning, prediction and optimization," in *Hawaii Intl. Conf. System Sciences, Proc. 41st Annual*, Jan 2008, pp. 71–71.

[3] M. Krishnamoorthy, A. Brodsky, and D. A. Menascé, "Temporal manufacturing query language (tmql) for domain specific composition, what-if analysis, and optimization of manufacturing processes with inventories," Department of Computer Science, George Mason University, Tech. Rep. GMU-CS-TR-2014-3, 2014. [Online]. Available: http://cs.gmu.edu/~tr-admin/papers/GMU-CS-TR-2014-3.pdf

[4] G. Shmueli and O. R. Koppius, "Predictive analytics in information systems research," *MIS Q.*, vol. 35, no. 3, pp. 553–572, Sep. 2011.

[5] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to Linear Regression Analysis (4th ed.).* Wiley & Sons, Jul. 2012.

[6] P. J. Haas, P. P. Maglio, P. G. Selinger, and W. chiew Tan, "Data is dead without what-if models," *PVLDB*, vol. 4, no. 12, pp. 1486–1489, 2011.

[7] A. Brodsky and J. Luo, "Decision guidance analytics language (dgal): Toward reusable knowledge base centric modeling," Department of Computer Science, George Mason University, Tech. Rep. GMU-CS-TR-2014-6, 2014. [Online]. Available: http://cs.gmu.edu/~tr-admin/papers/GMU-CS-TR-2014-6.pdf

[8] S. Katz, Y. Labrou, M. Kanthanathan, and K. Rudin, "Method for managing a workflow process that assists users in procurement, sourcing, and decision-support for strategic sourcing," Nov. 21 2002, uS Patent App. 09/858,122.

[9] M. Rys, D. Chamberlin, and D. Florescu, "Xml and relational database management systems: The inside story," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 945–947.

[10] D. Florescu and G. Fourny, "Jsoniq: The history of a query language," *Internet Computing, IEEE*, vol. 17, no. 5, pp. 86–90, Sept 2013.

[11] P. Fritzson and V. Engelson, "Modelica a unified object-oriented language for system modeling and simulation," in *IProc. European Conf. Object-oriented Programming (ECOOP98*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, vol. 1445, pp. 67–90.

[12] J. Äkesson, K.-E. Ärzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit, "Modeling and optimization with optimica and jmodelica.orglanguages and tools for solving large-scale dynamic optimization problems," *Computers & Chemical Engineering*, vol. 34, no. 11, pp. 1737 – 1749, 2010.

[13] R. Fourer, D. M. Gay, and B. W. Kernighan, "Ampl: A mathematical programming language," AT&T Bell Laboratories, Murray Hill, NJ 07974, Tech. Rep., 1987.

[14] A. Brook, D. Kendrick, and A. Meeraus, "Gams, a user's guide," *SIGNUM Newsl.*, vol. 23, no. 3-4, pp. 10–11, Dec. 1988.

[15] P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Rgin, "Constraint programming in opl," in *Principles and Practice of Declarative Programming*, ser. Lecture Notes in Computer Science, G. Nadathur, Ed. Springer Berlin Heidelberg, 1999, vol. 1702, pp. 98–116.

[16] W.-C. L. Alex Guazzelli and G. Williams, "PMML: An Open Standard for Sharing Models," *The R Journal*, vol. 1, no. 1, pp. 60–65, May 2009.

[17] J. Dix, S. Kraus, and V. S. Subrahmanian, "Heterogeneous temporal probabilistic agents," *ACM Trans. Comput. Logic*, vol. 7, no. 1, pp. 151–198, Jan. 2006.

[18] T. Eiter, V. Subrahmanian, and G. Pick, "Heterogeneous active agents, i: Semantics," *Artificial Intelligence*, vol. 108, no. 1-2, pp. 179 – 255, 1999.

[19] T. Eiter and V. Subrahmanian, "Heterogeneous active agents, ii: Algorithms and complexity," *Artificial Intelligence*, vol. 108, no. 1-2, pp. 257 – 307, 1999.

[20] V. Jain and I. E. Grossmann, "Algorithms for hybrid milp/cp models for a class of optimization problems," *INFORMS Journal on Computing*, vol. 13, no. 4, pp. 258–276, 2001.

[21] A. Brodsky, G. Shao, and F. Riddick, "Process analytics formalism for decision guidance in sustainable manufacturing," *Journal of Intelligent Manufacturing*, pp. 1–20, 2013.

[22] N. Egge, A. Brodsky, and I. Griva, "An efficient preprocessing algorithm to speed-up multistage production decision optimization problems," in *System Sciences (HICSS), 2013 46th Hawaii Intl. Conf.*, Jan 2013, pp. 1124–1133.

Appendix
DGAL code for smart manufacturing components

```
declare variable $ns:globalSetting as aObject := { let
        $noPeriods as integer := …,
        $lastTP :=  $noPeriods,
        $periodLength as float := …,
        $epsilon as float := …
        return {noPeriods: $noPeriods, periodLength: $periodLength, epsilon:$epsilon}
};
declare variable $ns:itemFlow as aObject := { let
        $mt as string := …,
        for $i in range(0, $ns:globalSetting.$lastTP) let ($tp as index := {tp : $i}, $tpAlloc($tp) as integer := ?),
        for $i in range(1, $ns:globalSetting.$noPeriods) let ($period as index := {period: $i}, $periodQty($period) as integer := ? ),
        $qtyWithinAllocRule as constraint := every ($period in $periodIndex) satisfies  $periodQty($period) <= $tpAlloc({tp: $period.period-1}),
        return {matchType: $mt, tpAllocIndex :  $tpAllocIndex,   periodQtyIndex:  $periodQtyIndex, qtyWithinAlloc:  $qtyWithinAllocRule}
};
```

Figure 12. DGAL code for global metrics and *itemFlow* analytical object

```
declare variable $ns:IA as aObject := { let
        $mt as string := …,
        $inputIds as [ string ] := …,
        $outputIds as [string] := …,

        for $i in $inputIds[] let (
                $inputId as index:= {id : $i},
                $inputFlow($inputId) as $ns:itemFlow := …,
                $inAllocRatio($inputId) as float := …
        ),

        for $o in $outputIds[] let (
                $outputId as index := {id: $o},
                $outputFlow($outputId) as $ns:itemFlow := …,
                $outAllocRatio($outputId) as float := …
        ),

        $capacity as integer := …,
        $totalQty as integer:= ?,
        for $i in range(0, $ns:globalSetting.$lastTP) let (
                $tp as index := {tp : $i},
                $invQty($tp) as integer := ?,
        ),
        $initInv as integer := …,
        $invQty({tp: 0}) := $initInv,
        for $tp in $tpIndex where $tp.tp >= 1 let (
                $invQty($tp) := $invQty({tp: $tp-1}) +
                        sum (for $i in $inputIdIndex)   $inputFlow($i).$periodQty({period:$tp.tp}) -
                        sum (for $o in $outputIdIndex)    $outputFlow($o).$periodQty({period:$tp.tp}),
        ),
        for $tp in $tpIndex, $i in $inputIdIndex let (
                $inputFlow($i).$tpAlloc($tp) := $inAllocRatio($i) * ($totalQty - $invQty($tp))
        ),
        for $tp in $tpIndex, $o in $outputIdIndex let (
                $outputFlow($o).$tpAlloc($tp) := $outAllocRatio($o) * $invQty($tp)
        ),
        $capacityRule as constraint := ($totalQty <= $capacity),
        return {matchType: $mt, inputs: $inputId…?Values, outputId: $outputId…?Values, capacity: $capacity, totalQty: $totalQty,
                invQtyIndex: $invQtyIndex, capacityRule: $capacityRule,
        }
};
```

Figure 13. DGAL code for *IA* analytical object

```
declare variable $ns:IQA as aObject := { let
        $mt as string := …,
        $inputId as string := …,
        $inputIds := [ $inputId],
        $outputIds as [string] := …,

        for $i in $inputIds[] let
                $inputId as index:= {id : $i},
                $inputFlow($inputId) as $ns:itemFlow := …,

        for $o in $outputIds[] let
                $outputId as index := {id: $o},
                $outputFlow($outputId) as $ns:itemFlow := …,
                $outAllocRatio($outputId) as float := …,

        for $i in range(0, $ns:globalSetting.$lastTP) let (
                $tp as index := {tp : $i},
                $totalTPAlloc($tp) as integer :=
                        sum (for $i in $inputIdIndex) $inputFlow($i).$tpAlloc($tp),
                for $o in $outputIndex let
                        $outputFlow($o).$tpAlloc($tp) := $outAllocRatio($o) *
                                $totalTPAlloc($tp)
        ),
        for $p in range(1, $ns:globalSetting.$noPeriods) let (
                $inputFlows($inputId).$periodQty({period:$p}) =
                        sum (for $o in $outputIdIndex)
                        $outputFlow($o}.$periodQty({period:$p})
        ),

        return {input: $inputId…?Values,
                output: $output…?Values,
                totalTPAllocIndex: $totalTPAllocIndex,
        }
};
```

(a)

```
declare variable $ns:OQA as aObject := { let
        $mt as string := …,
        $inputIds as [ string ] := …,
        $outputId as string := …,
        $outputIds := [ $outputId ],

        for $i in $inputIds[] let )
                $inputId as index:= {id : $i},
                $inputFlow($inputId) as $ns:itemFlow := …,
                $inAllocRatio($inputId) as float := …,
        ),
        for $o in $outputIds[] let (
                $outputId as index := {id: $o},
                $outputFlow($outputId) as $ns:itemFlow := …,
        ),
        for $t in range(0, $ns:globalSetting.$lastTP) let (
                $tp as index := {tp : $t},
                $totalTPAlloc($tp) := sum (for $o in $outputIdIndex)
                        $outputFlow($o).$tpAlloc($tp)
                for $i in $inputIndex let (
                        $inputFlow($i).$tpAlloc($tp) := $inAllocRatio($i) *
                                $totalTPAlloc($tp)
                )
        ),
        for $p in range(1, $ns:globalSetting.$noPeriods) let (
                $outputFlow($outputId).$periodQty({period:$p}) :=
                        sum (for $i in $inputIdIndex)
                                $inputFlow($i).$periodQty({period:$p})
        ),
        return {input: $inputId…?Values,
                output:$output…?Values,
                totalTPAllocIndex: $totalTPAllocIndex,
        }
};
```

(b)

Figure 14. DGAL code for analytical objects: (a) *IQA* and (b) *OQA*

```
declare variable $ns:compositeProcess as $ns:process := { let
        $inputIds as [ string ] := …,
        $outputIds as [string] := …,

        for $i in $inputIds[] let
                $inputId as index:= {id : $i},
                $inputFlow($inputId) as $ns:itemFlow := …,

        for $o in $outputIds[] let
                $outputId as index := {id: $o},
                $outputFlow($outputId) as $ns:itemFlow := …,

        $itemFlowIds as [ string ] := …,
        for $i in $itemFlowIds[] let
                $itemFlowId as index:= {id : $i},
                $ifvar($itemFlowId) as $ns:itemFlow := …,

        $subProcessIds as [ string ] := …,
        for $i in $subProcessIds[] let
                $subProcessId as index:= {id : $i},
                $pvar($subProcessId) as $ns:process := …,

        $inventoryAggrIds as [ string ] := …,
        for $i in $inventoryAggrIds[] let
                $inventoryAggrId as index:= {id : $i},
                $iavar($inventoryAggrId) as $ns:IA := …,

        $inputQtyAggrIds as [ string ] := …,
        for $i in $inputQtyAggrIds[] let
                $inputQtyAggrId as index:= {id : $i},
                $iqavar($inputQtyAggrId) as $ns:IQA := …,

        $outputQtyAggrIds as [ string ] := …,
        for $i in $outputQtyAggrIds[] let
                $outputQtyAggrId as index:= {id : $i},
                $oqavar($outputQtyAggrId) as $ns:OQA := …,

        $metrics as [string] = [for
                $p in $subProcessIds[],
                $m in $p.metrics[],
                return $m
        ]

        for $m in $metrics let
                $metric as index := {metric: $m},
                $mValues($metric) as float := sum ($p in $subProcessIds[],
                        $spm in $p.metrics[] where $spm = $m)  $p.mValues({metric: $m})
        return {
                input: $inputIds…?values, output: $outputIds…?values, itemFlows:
                        $itemFlowId…?values, subProcesses: $subProcessId…?values,
                        inventoryAggrs: $inventoryAggrId…?values, inputQtyAggrs:
                        $inputQtyAggrId…?values,outputQtyAggrs: $outputQtyAggrId…?values,
                        metrics: $metric…?values,
        }
}
```

(a)

```
declare variable $ns:process as aObject := {
        $inputIds as [ string ] := …,
        $outputIds as [string] := …,
        for $i in $inputIds[] let (
                $inputId as index:= {id : $i},
                $inputFlow($inputId) as $ns:itemFlow := …,
        ),
        for $o in $outputIds[] let (
                $outputId as index := {id: $o},
                $outputFlow($outputId) as $ns:itemFlow := …,
        ),
        $metrics as [string] = …,
        for $m in $metrics let (
                $metric as index := {metric: $m},
                $mValues($metric) as float := ?
        )
};
```

(b)

Figure 15. DGAL code for analytical objects: (a)*compositeProcess*, (b) *process*

```
declare variable $ns:baseProcess as $ns:process := { let
        $machineSigma as float := …,
        $inputIds as [string] := …,
        $initLeftOver as float := …,
        $capacity as float := …,
        for $i in $inputIds[] let (
                $inputId as index := {id:$i},
                $inputFlow($inputId) as $ns:itemFlow = …,
                $inputPerOutput($inputId) as integer := …,
        ),
        $outputId as string := …,
        $outputIds := [$outputId],
        for $o in $outputIds[] let (
                $out as index := {id: $o},
                $outputFlow($out) as $ns:itemFlow := $ns:itemFlow{}
        ),
        $PWLcoefs := ["slope1", "bound1",  "slope2", "bound2", "startX", "startY "},
        for ($c in $PWLcoefs) let (
                $coef as index := {coef: $c},
                $energyPWLvalue($coef) as float := …,
                $costPWLvalue($coef) as float := …
        ),
        $energyPWL as PWLfunction = pwl($energyPWLvalueIndex),
        $costPWL as PWLfunction = pwl($costPWLvalueIndex),
        for $i in range(0, $ns:globalSetting.$lastTP) let (
                $tp as index := {tp : $i},
                $leftOver($tp) as float := ?
        )
        $leftOver({tp:0} := $initLeftOver,
        for $i in range(1, $ns:globalSetting.$noPeriods) let
        $period as index := {period: $i},
                $throughputExp($period) as float := ?,
                $throughputControl($period) as float :=
                        $throuputExp($period) + Gaussian({exp: 0.0, sigma: $machineSigma}),
                $accumAmt($period) as float :=
                        $leftOver({tp: $period.period - 1}) + $throughputControl($period) *
                        $ns:globalSetting.$periodLength,
                $leftOver({tp:$period.period}) := $accumAmt($period) -
                        $theOutputFlow.$periodQty($period),
                $energy($period) as float := $energyPWL($throughputControl($period)),
                $cost($period) as float := $costPWL($throughputControl($period)),
                for $i in $inputIndex let (
                        $inputFlow($i).$periodQty($period) = $theOutputFlow.$periodQty($period) *
                                $inputPerOutput($i),
        ),
        $throughputRule as constraint :=
                every ($period in $periodIndex),
                satisfies $throughputControl($period) <= $capacity,
        $outputQtyBoundsRule as constraint :=
                every ($period in $periodIndex)
                satisfies $accumAmt($period) - 1 + $ns:globalSetting.epsilon <=
                        $theOutputFlow.$periodQty($period) <= $accumAmt($period),
        $totalEnergy as float := sum($period in $periodIndex) $energy($period),
        $totalCost as float := sum($period in $periodIndex) $cost($period),
        $metrics := [ "energy", "cost"],
        for $m in $metrics let (
                $metric as index := {metric: $m},
                $mValue($metric) as float := ?
        )
        $mValue({metric: "energy"}) := $totalEnergy,
        $mValue({metric: "cost"}) := $totalCost,
        return {input: $inputIdIndex…?Values, output: $outputIdIndex…?Values, throughputControlIndex: $throughputControlIndex,
                energyCoef: $ecoef…?Values, costCosef: $ccoef…?Values, leftOverIndex: $leftOverIndex, metricValues: $metric…?Values,
                throuputRule: $throughputRule, outputQtyBounds: $outputQtyBoundsRule
        }
};
```

Figure 16. DGAL code for *baseProcess* analytical object