Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

Temporal Manufacturing Query Language (tMQL) for Domain Specific Composition, What-if Analysis, and Optimization of Manufacturing Processes With Inventories

Mohan Krishnamoorthy mkrishn4@gmu.edu
Alexander Brodsky brodsky@gmu.edu
Daniel A. Menascé menasce@gmu.edu

Technical Report GMU-CS-TR-2014-3
Created: May 2014; Modified: May 2015

Abstract

Smart manufacturing requires streamlining operations and optimizing processes at a global and local level. This paper considers manufacturing processes that involve physical or virtual inventories of products, parts and materials, that move from machine to machine. The inventory levels vary with time and are a function of the configuration settings of the machines involved in the process. These environments require analysis, e.g., answering what-if questions, and optimization to determine optimal operating settings for the entire process. The modeling complexities in performing these tasks are not always within the grasp of production engineers. To address this problem, the paper proposes the temporal Manufacturing Query Language (tMQL) that allows the composition of modular process models for what-if analysis and decision optimization queries. tQML supports an extensible and reusable model knowledge base against which declarative queries can be posed. Additionally, the paper describes the steps to translate the components of a tMQL model to input data files used by a commercial optimization solver.

1 Introduction

In the past few years, there has been significant technological advancements in different areas of process analysis and optimization. Examples of processes include manufacturing processes, such as assembly lines, and supply chain. These processes often involve physical or virtual inventories of products, parts and materials that are used to anticipate uncertainties on supply or throughputs of machines. Over time, the state of the machines, inventories and the whole process changes until process completion. We use the term Buffered Temporal Flow Processes (BTFP) to describe them. BTFP can be found in many different areas of manufacturing and supply chain. A particularly important BTFP is in the area of discrete manufacturing such as in automotive, furniture, smartphones, airplanes and toys. The BTFP formalism presented in this paper can be used to model problems that deal with inventories whose state varies at discrete time intervals.

Due to increased global competition, manufacturing companies look toward ways to reduce their cost and increase efficiency of operations. This results in a greater need for analysis and optimization of the operation results on the manufacturing floor while taking into account sustainability metrics. To support analysis and optimization of BTFP, there is a need to accurately model machines, systems and processes. These models need to capture (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 thorough the manufacturing floor, and (c) work-in-progress for inventories. In BTFP, this needs to be modeled over a temporal sequences and include stochasticity of machines throughput and supply.

Using these models, it is desirable to allow manufacturing and process engineers to perform a variety of analysis and optimization tasks including what-if prediction and optimization. For example, as a prediction question, a process engineer may ask: 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, efficiency and carbon emissions. Or, as an optimization question, a process engineer may ask: given the process design

(which includes the flow of work pieces through various stages of processing), which machines should be on and off, and how to set-up controls of every operational machine, and distribute processing load among the machines, as to minimize the total production cost, while satisfying the demand for every time interval over a planning horizon, and within a limitation on the capacity of work-in-progress inventories? Given the diversity of manufacturing processes, it is highly desirable to have a system that would allow the flexible specification of manufacturing processes, and able to answer declarative analysis and optimization queries to end users.

There has been extensive research on analysis and optimization of BTFP like processes (e.g., see [1], [2], [3] for overview). Broadly, the work can be classified into three broad categories: (1) customized domain-specific solutions for optimization of manufacturing processes, (2) simulation-based systems, and (3) optimization solvers and modeling languages based on mathematical programming (MP) and constraint programming (CP). Customized domain-specific solutions for BTFP are designed for specific, limited setting of a manufacturing process, and would typically provide a graphical user interface that is easy to use by the end users. Examples include [4] and [5]. The implementation of domain specific solutions may use optimization tools based on mathematical programming, and integrate them with other systems such as Enterprise Resource Planning (ERP). However, while these solutions may be both efficient (in terms of optimality of results and computational time), they are (1) typically not extensible to additional aspects of machines, processes and metrics, and (2) perform a "silo" optimization, which would not achieve the global optimum if an extended underlying system needs to be optimized.

Simulation-based systems allow to accurately model a system and its inner workings. It is object-oriented, modular, extensible, and reusable. Furthermore, many simulation tools provide an easy-to-use graphical user interface. Tools like SIMULINK [6] and Modelica-based ones [7] like JModelica [8], Dymola [9], and MapleSim [10] allow users to model complex systems in mechanical, hydraulic, thermal, control, and electrical power. For example, Modelica comes with over 1000 generic model components that can all reused. However, simulation-based optimization is significantly inferior to optimization solutions based on MP/CP in terms of optimality of results and computational complexity. This is because simulation-based optimization amounts to a heuristically-guided trial and error search, which does not utilize the mathematical structure of the underlying problem the way MP/CP methods do.

Optimization solvers and modeling languages based on MP and CP are often the technology of choice, when optimality and computational complexity are the priority. Many classes of MP, such as linear programming (LP), mixed integer linear programming (MILP), and non-linear programming (NLP), have been very successful in solving real-world large-scale optimization problems. CP, on the other hand, has been broadly used for combinatorial optimization problems like scheduling and planning. To use these tools, one would have to use an algebraic modeling language such as AMPL [11], OPL [12], GAMS [13], or AIMMS [14]. However, MP and CP modeling present a significant challenge for engineers and business analysts to model. It would require an OR expert to model a problem and express it in an algebraic modeling language like the ones mentioned. Additionally, these formal models are typically difficult to modify, extend, or reuse. This is comparable to "spaghetti" code versus an object-oriented approach.

Sustainable Process Analytics Formalism (SPAF) [15] was recently proposed to target modularity and reusability of optimization models, yet designed to use MP/CP algorithms via a reduction of an SPAF model and a declarative optimization query to formal MP/CP models and solving them using commercial solvers. In turn, SPAF is based on the ideas of modular representation of constraints and reductions to formal optimization models from [16], [17], [18], [19], [20], and [21]. However, being a general modeling language, SPAF did not address the problem of building a high-level domain specific language for BTFP processes, i.e., over a temporal domain, and involving product and work-in-progress inventories. This is exactly the focus of this paper.

In this paper, to address the limitation of current approaches, we propose tMQL - the temporal manufacturing query language and framework - for modular composition, manipulation, what-if analysis and optimization of BTFP processes, machines and work-in-progress inventories. More specifically, the contributions of this paper are as follows:

- We propose the tMQL language, define its syntax and semantics
- We describe the tMQL framework for the management of a centralized knowledge-base of reusable tMQL components, and performing what-if analysis and decision optimization tasks using a declarative end-user language.
- We present a graphical notation to model the components of the BTFP problem as a process interaction diagram.
- We report on and describe the implementation of tMQL using the Optimization Programming Language (OPL).

The rest of this paper is organized as follows: an example motivating the *tMQL* framework is given in section 2. The *tMQL* framework is explained in section 3 followed by the tMQL language in section 4. Section 5 gives and informal description of the syntax and semantics of the tMQL language. Section 6 describes the implementation of deterministic tMQL, and finally, section 7 concludes.

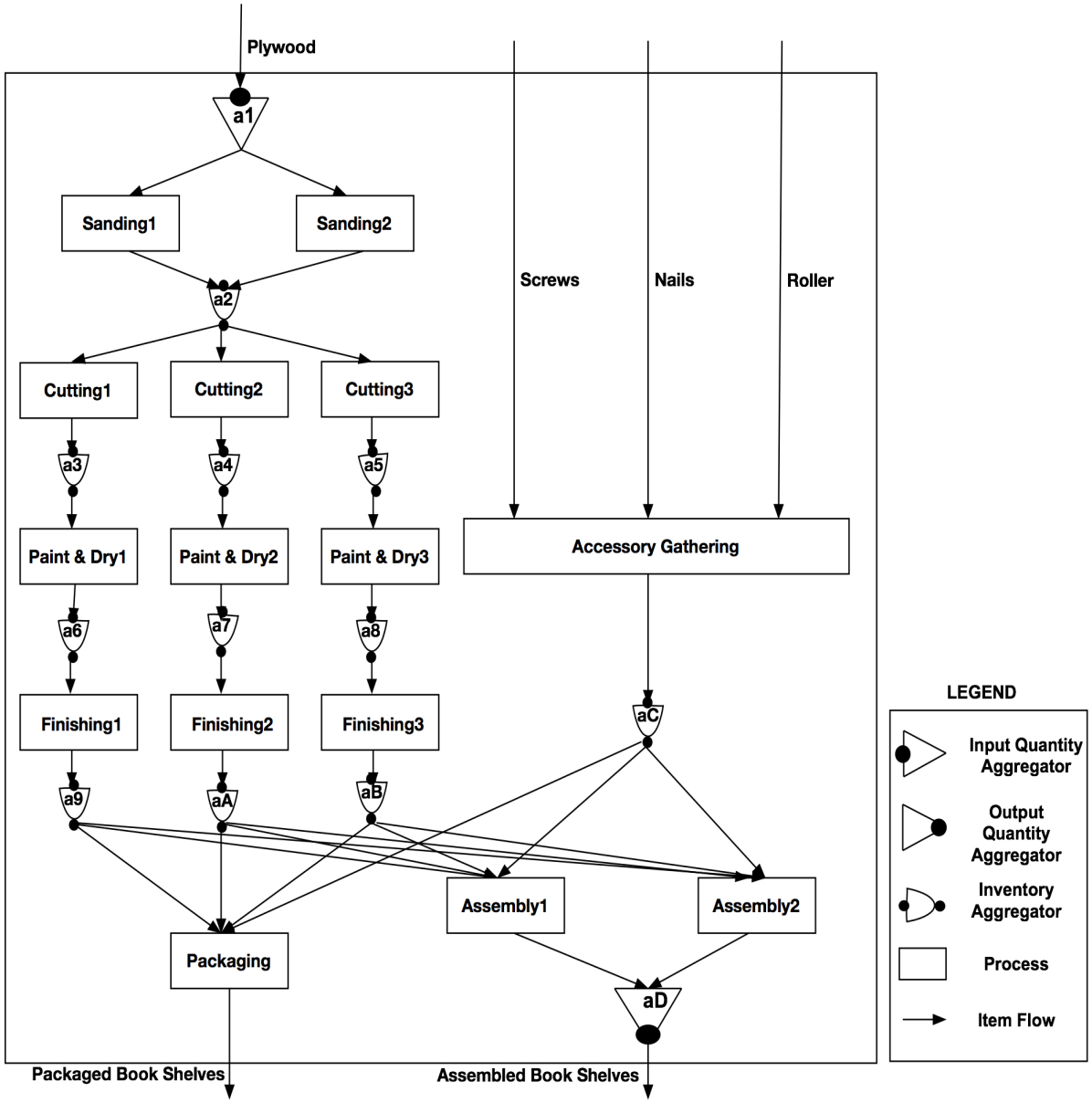

Figure 1: A graphical notation for the bookshelf manufacturing floor

2 Motivating example

This section introduces an example BTFP problem to describe the challenges motivating the *tMQL* framework. The example consists of a simplified bookshelf packaging and assembly system. For simplicity, we consider that the bookshelf consists of a left plank, a right plank, and three shelves, is supported through screws and nails, and is provided with rollers for easy movement.

A graphical notation for the process interaction is shown in Fig. 1. The machines, shown as rectangular boxes, are interleaved with buffers used to store and/or distribute the items produced by one machine to another. The concept of buffer is explained in greater detail in the section 4. Raw materials for the bookshelf manufacturing floor consist of plywood logs and accessories such as nails, screws, and rollers. The plywood goes first through two sanding machines where the plywood logs entering the floor are sanded to smoothen the wood. Next, the left plank, right plank and the shelves are cut by their own respective machines. This stage involves the wood first undergoing the rough cutting of appropriate size from the log of plywood, precise cutting of the edges, and sanding of the edges. After this, the left plank, right plank and shelves are painted and dried by their respective machines. After the drying is complete, a finishing touch is given to the left plank, right plank, and the shelves by their respective machines. This involves drilling holes and cleaning the respective pieces of wood so that they are ready for use. In parallel, the ac-

cessories are gathered by a machine. Then, the finished left plank, right plank, shelves and the accessories are either packaged in a box to be sent to a warehouse or they are assembled into a bookshelf by a machine to be put on display.

The system could include controllable and non-controllable parameters from all the machines. Examples of controllable parameters include the speed of the sanding machine and the temperature of the painting and drying machines. In addition, all machines along with the heating and cooling auxiliary devices consume energy.

Machines output parts to be used by succeeding machines, which may not be able to immediately work on these parts. Hence, all machines are interleaved with physical *buffers* that can hold parts until the succeeding machine(s) are ready to take these parts as input. These buffers are constrained by their capacity. Also, there may be multiple machines working at different speeds and efficiencies, at the same stage of production. Finally, the flow of items between the machines and buffers is bounded by the number of items that the machines can handle and/or by the number of items that can be stored in the buffers.

tMQL allows various queries to be asked against the process interaction. Optimization queries can be of the form maximize the number of bookshelves packaged or minimize the energy consumed or minimize the time taken to assemble one bookshelf. What-if analysis queries can be of the form what is the throughput of the manufacturing floor, in packaged bookshelves/hour, for a given setting of the speed of the various machines?

3 *tMQL* Framework

This section provides a high level description of *tMQL* which allows one to model the components of a manufacturing floor and perform queries on this model (see Fig. 2). The framework consists of the tMQL query language, the tMQL knowledge base (KB), tMQL roles, and the tMQL engine. A Buffered Temporal Flow Processes system can be described as a set of components, which can be processes, aggregators, and the flows that connect them. These components are stored in the tMQL KB. For instance, in the bookshelf example, the tMQL KB will have all the sanding, cutting, paint & dry, finishing, packaging, assembly and accessory gathering processes as well as the aggregators a1 - aC. In addition, the KB will also contain the flows shown as one direction arrows in Fig. 1. The tMQL KB acts as a repository of components. The KB can contain compositions of more than one component as well as components with their metrics evaluated so that they can be composed into other components and used in a other queries.

The tMQL queries are: a) *compose*: used to compose two or more components from the KB, b) *specialize*: used to specialize the value of the metric of a component, c) *compute*: used to compute metrics of a component and d) *optimize*: used to optimize a numeric metric of a component subject to constraints, and e) *new*: used to create a copy of a component from the KB. The *new* query can be used along with the *specialize* query to compose components.

There are two tMQL roles: process modeler and process operator. The process modeler is responsible for composing components to support the needs of the BTFP problem. The process modeler stores these composed components in the KB. The process operator can specialize the components and perform computation or optimization queries on the components from the KB. The process operator can store the results of tMQL queries back in the KB for future use. Although the process modeler and process operator are described as two separate roles, it is possible for a single person to play both roles.

The tMQL engine takes existing components from the tMQL KB and the queries on these components and applies efficient techniques to generate answers for these queries. An optimization solver such as IBM's CPLEX optimizer may be used to solve deterministic optimization queries. Simulation solvers may be used to compute metric values, perform metric calibration and what-if analysis. Some other techniques such as embedded optimization solvers may be used to perform optimization in a stochastic environment.

4 Query Language for *tMQL* framework: tMQL

tMQL is a modular query language for the *tMQL* framework. The language has both a graphical notation, exemplified in section 2, and a textual notation. tMQL allows for any discrete, buffered, manufacturing process to be modeled, analyzed, and optimized. Because tMQL allows for process refinement, complex manufacturing processes can be easily modeled. This section describes tMQL by example. The formal syntax and semantics of the language are provided in appendices A and B.

Consider the example in Fig. 3, which is a smaller version of the bookshelf example described in Fig. 2. 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 floor. 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 $\Delta t = 1$. A time interval (also known as a period) is denoted by $p_{i+1} = (t_i, t_{i+1})$. For this example, we consider that there are three time points and therefore two time intervals.

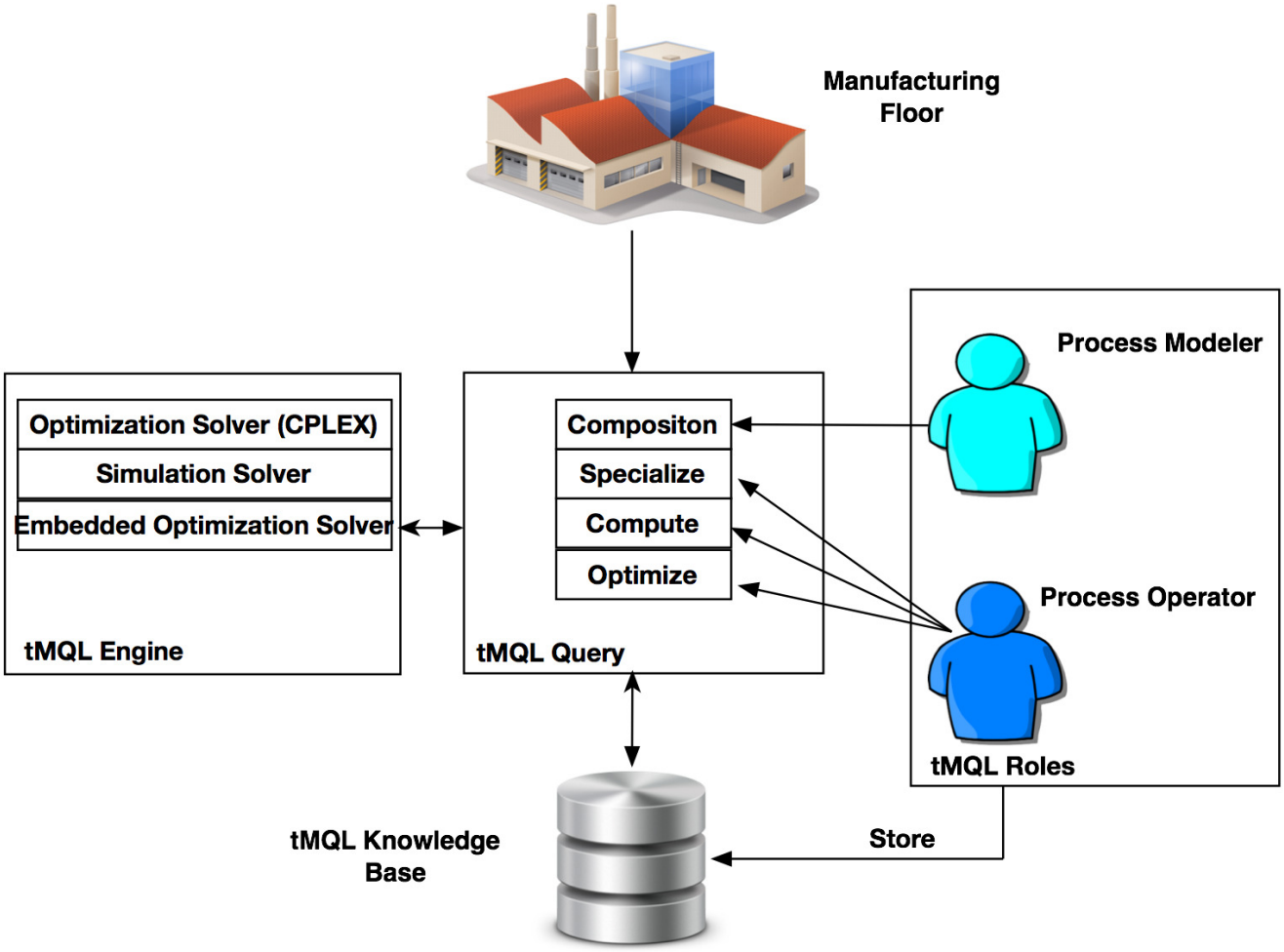

Figure 2: *tMQL* Framework

The tMQL KB contains built-in components that do not have any values associated to metrics. The KB also contains user-defined components that can be generated from built-in components by specializing them with characteristics and data from the manufacturing floor. User-defined components include process components that either model individual machines, a subset of the manufacturing floor, or the entire manufacturing floor. These components can be stored back into the KB. For instance, the component used to model the *sand*1 machine provides metrics such as the speed of the machine, number of inputs required per output produced and to compute the cost of production. The component used to model the sand and cut manufacturing floor encapsulates all the process, flow and aggregator components of the floor. The components in the KB also include aggregators such as input quantity aggregator (IQA), inventory aggregator (IA), and output quantity aggregator (OQA).

The IQA (e.g., *a*1) allows for the input plywood to be distributed among the sand processes. The IA (e.g., *a*2) is the buffer that allows the sanded wood from the two sand processes to be stored and/or distributed among the cut processes. The OQA (e.g., *a*3) allows for the cut-wood from the cut processes to be collected and output as single collection. Finally, the KB has flow component such as the flows going out of IQA *a*1 that contains information of how the items flow among *a*1 and sand process. Assume that the KB already contains all the process components for the *sand1*, *sand2*, *cut1*, *cut2* processes. These processes have their in and out flows specialized to the respective machine's inputs and outputs on the floor. In addition, these processes also have some of their metrics specialized. A snapshot of this specialized *sand1* process is described in subsection 4.2.

4.1 Composition of tMQL components

A process modeler can now compose the manufacturing floor model. This can be done by invoking the specialize query on the built-in component of the manufacturing floor called *compositeProcess* to update its inputs, outputs,

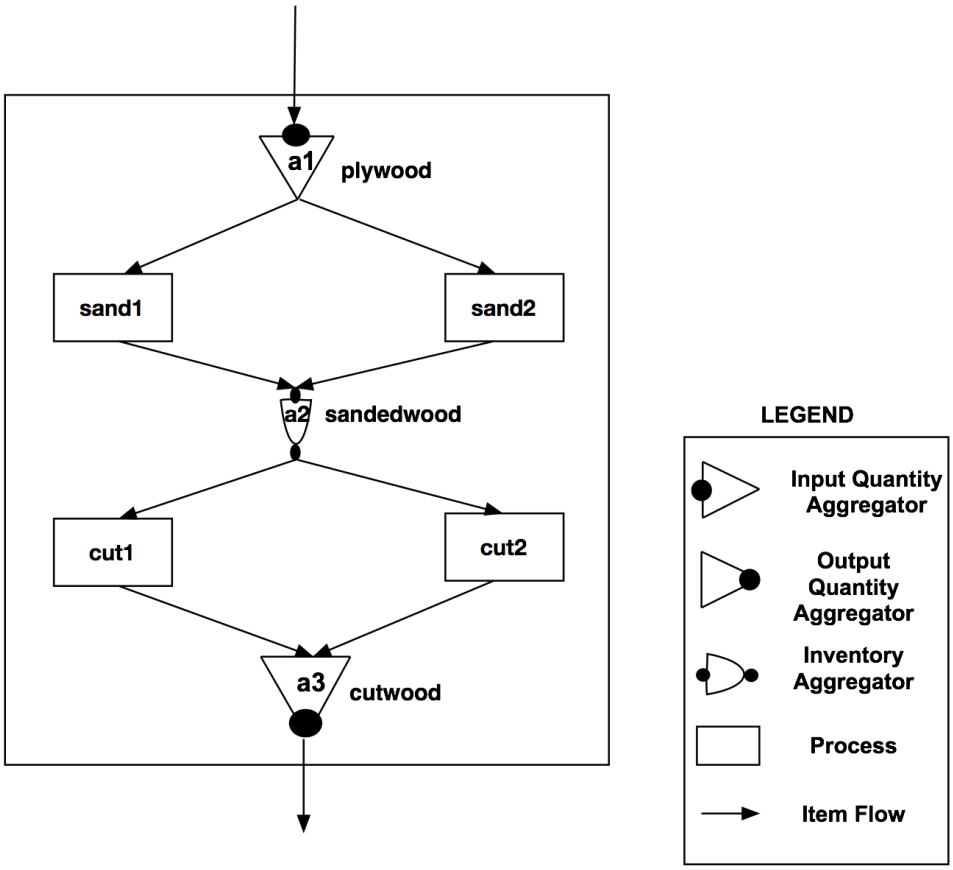

Figure 3: A graphical notation for the sand and cut manufacturing floor

flows, processes, aggregators, and metrics. An example of the specialize query for the sand and cut floor (Fig. 3) is shown in Fig. 4. The input and outputs of the floor are the *plywood* and *cutwood* flows. They are first created by calling the new query on the built-in itemFlow component and then the input, I, and output, O, of the compositeProcess component is specialized. Then, the individual sand and cut machines are installed using the new query on the respective process components in the KB. Similarly, the IQA, IA and OQA are installed as a1, a2, and a3, respectively, with their inputs and outputs specialized. Finally, the composite process contains the cost metric defined as the sum of the cost incurred by each base process. In this way, different components from within the KB are used to compose the sand and cut composite process. This specialized composite process is stored in the KB as *SandCutProcess* using the *insert* command.

Intuitively, the above model captures the metrics and constraints of the sand and cut process, the aggregators used to store and/or distribute the wood, and the flows that enable the processes and aggregators to interact with each other. The metrics specific to the sand and cut manufacturing floor are incorporated into each component that is copied (via the *new* query) or specialized from the KB. In addition, the constraints in these components enable the composite process to restrain the decision metrics within the metrics specific to the sand and cut manufacturing floor.

4.2 tMQL components

Before showing the queries that can be run on the composite model created above, this subsection describes the mathematical definitions of the tMQL components used to model individual machines, aggregators and flows with respect to the sand and cut manufacturing floor shown in Fig. 3. The formal syntax of the components can be found in appendix A. The mathematical definition of the constraints associated to components is discussed in section 5.

Metrics initialized by "..." are parameters of the model and will need to be specialized later. The metrics with value "..." in the flows and aggregators of Fig. 5 can be specialized as constants or expressions specific to the respective component or be left to be specialized later. A specialization of this kind for the sand and cut manufacturing floor was shown in Fig. 4. Metrics initialized by a "?" are decision variables (dvar) that can be specialized as constants, expressions, be specialized later, or be used as decision variables for optimization queries.

```
insert SandCutProcess = specialize compositeProcess {
                        // Global inputs and outputs
                        itemFlow plywood = new itemFlow;
                        itermFlow cutwood = new itemFlow;
                        {itemFlow} I = {plywood};
                        {itemFlow} O = {cutwood};

                        // Base Processes
                        Process sand1 = new sand1;
                        Process sand2 = new sand2;
                        Process cut1 = new cut1;
                        Process cut2 = new cut2;

                        {Process} pvars = {sand1, sand2, cut1, cut2};

                        // Aggregators
                        IQA a1 = specialize IQA{
                                        {itemFlow} I = {plywood};
                                        {itemFlow} O = {sand1.plywood, sand2.plywood};
                        };
                        IA a2 = specialize IA{
                                {itemFlow} I = {sand1.sandedwood, sand2.sandedwood};
                                {itemFlow} O = {cut1.sandedwood, cut2.sandedwood};
                        };
                        OQA a3 = specialize OQA{
                                        {itemFlow} I = {cut1.cutwood, cut2.cutwood};
                                        {itemFlow} O = {cutwood};
                        };
                        {IA} iavars = {a2};
                        {IQA} iqavars = {a1};
                        {OQA} oqavars = {a3};

                        //Metircs
                        {string} metrics = {cost};
                        float mValues[cost] = sum(p in pvars) p.cost;
};
```

Figure 4: Specialize query on compositeProcess to create a sand and cut composite process

We first describe the mathematical definitions of flows and aggregators. A flow is a tMQL component of type itemFlow. An example in the sand and cut process is the *sand1.plywood* flow that carries plywood from IQA $a1$ to *sand*1 process. The mathematical definition of a built-in flow component is shown in Fig. 5(a). The number of items that the flow contains in each period is given by the *periodQty* metric of this flow. These items are constrained by the $tpAlloc$ metric. This constraint ensures that the number of items flowing through this component are within the bounds of what can be handled by the aggregators and the processes at either end of the flows.

An aggregator of type IA is a tMQL component that stores and distributes items on the floor (see e.g., $a2$ in our example and Fig. 5(b) for its mathematical definition). The IA stores and distributes items coming in from processes via its input flows and going out to processes via its output flows. The IA has a storage $capacity$ that is the maximum number of items that the IA can physically store. This IA may not require to use this $capacity$ entirely because either the inputs are flowing slowly or there is a high demand on the output flows or the demand on the manufacturing floor is low. Hence the total space used during the time the IA is active is captured by the $totalQty$ metric and is upper bounded by the $capacity$ metric. An IA uses the input allocation ratio ($inAllocRatio$) and output allocation ratio ($outAllocRatio$) to determine the upper bound on the number of input items to take and the amount of these items to be distributed among the output processes. The number of items stored in the IA at each time point is captured by the inventory quantity metric ($invQty$) calculated as the difference between the number of items brought in by the input flows and taken out through the output flows.

An aggregator of type IQA is a tMQL component that distributes inputs to two or more processes (see e.g., $a1$ in our example and Fig. 5(c) for its mathematical definition). The IQA takes all the items entering via its input flow and distributes them among all its output flows in a way that maintains an output allocation ratio ($outAllocRatio$) for all its outputs. Based on the distribution scheme used, the sum of all input allocations ($totalTPAlloc$) is distributed among its outputs. Because the IQA does not have storage, all incoming items into the IQA need to be distributed among processes.

An aggregator of type OQA is a tMQL component

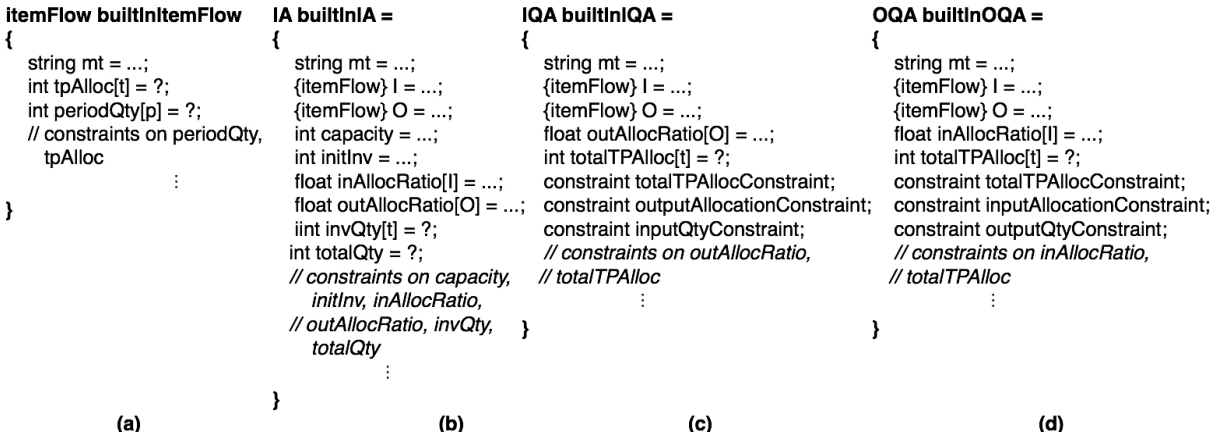

Figure 5: Mathematical Definitions for the built-in components of (a) item flow (b) inventory aggregator, (c) input quantity aggregator, and (d) output quantity aggregator

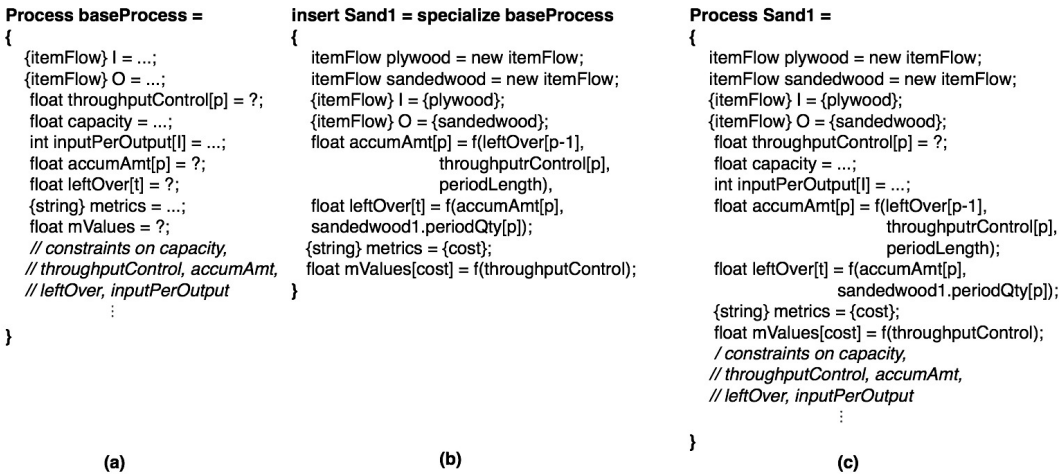

Figure 6: Defining Process components for machines: (a) mathematical definitions for the built-in components of baseProcess, (b) specialize query on the baseProcess to map the Sand1 machine (c) the result of the specialize query

that collects outputs from two or more processes (see e.g., *a*3 in our example and Fig. 5(d) for its mathematical definition). The OQA collects items from two or more processes via its input flows in a way that maintains the input allocation ratio (*inAllocRatio*) for all its inputs. Based on the distribution scheme used, the sum of all output allocations (*totalTPAlloc*) is distributed among its inputs. The processes will use this as a guide to produce and output the items to the OQA. Because the OQA does not have storage, all items coming into the OQA must leave the floor.

A base component, known as the *baseProcess*, is used to map the machines on the manufacturing floor by capturing their metrics and constraints and their interaction with aggregators via flows. The mathematical definition of the *baseProcess* component is described in Fig. 6(a). The *baseProcess* component can be reused to conceive the user-defined components for different machines. For example, instead of defining the *Sand*1 process directly, it can be defined as a specialization of the *baseProcess* (see Fig. 6(b)).

Figure 6(c) shows the *Sand*1 process, which is an output of the specialize query on the *baseProcess*. The *Sand*1 process is used for sanding the plywood coming in as input and providing sanded plywood as output. As stated previously, the *Sand*1 process is specialized from the *baseProcess* with new objects of the input and output itemFlows. The flows are installed as the inputs, *I* and the outputs, *O* using the new query. The speed of sanding the plywood is controlled by the *throughputControl* metric at each time period. This metric is initialized by a "?", which makes it a decision variable (dvar). The speed of the machine is constrained by its maximum speed called the *capacity*. This *capacity* could be the maximum speed possible for the process or may be constrained by some sustainability metric. The sand process also has the input per output metric (*inputPerOutput*) that captures the amount of plywood that the sand process requires per sanded wood. This restraints the amount of plywood required per period. The values of the *capacity*

and the *inputPerOutput* metrics are initialized as ..., which means that they will need to be specialized later. The amount of sanded wood accumulated at each time period is captured by the *accumAmt* metric. The amount of leftover sanded wood at each time point is captured by the *leftOver* metric. Finally, *metrics* and *mValues* are used to capture the cost of the sand1 process. The total cost incurred by the *Sand*1 process is a function of the *throughputControl* value at each time period. Other sand and cut processes can be specialized similarly.

4.3 Queries on the composite process

The previous subsection described the *specialize* and the *new* queries. This subsection introduces two other queries supported by tMQL: *compute* and *optimize*. Once the manufacturing floor has been composed into the composite model shown in subsection 4.1, a process operator may decide to specialize some of the parameters and/or decision variables of the processes and aggregators discussed in subsection 4.2. See Fig. 7(a) for an example of using the *specialize* query on the *SandCutProcess*. Also, the syntax and semantics of the queries are provided in appendices A and B.

The sand1 machine will sand at a speed of 2.2 and 3.2 plywoods in the two time periods. Hence, the operator will specialize the *throughputControl* metric of the *sand*1 process with these values. Also, the maximum amount of plywood that can be sanded in each time period by the sand1 machine is 5.5 and one plywood is required to produce one sanded wood. Thus, the operator specializes the *capacity* and *inputPerOutput* metric with these values, respectively. The other metrics of the *sand*1 process are not specialized here and hence their values remain the same as shown in Fig. 6. Similarly, other sand and cut process metrics are also updated by their respective values. In addition, the operator may update the metrics of the aggregators. For the IA, *a*2, its maximum capacity (*capacity*) and initial inventory quantity (*initInv*) metrics are specialized as 10 and 1, respectively. Also, the proportion of sanded wood taken as input at each time point by *a*2 from both sand1 and sand 2 machines is 0.5. Similarly, the proportion of sanded wood given as output at each time point by *a*2 to the cut1 machine is 0.4 and to cut2 machine is 0.6. These values are specialized by the operator in the *inAllocRatio* and *outAllocRatio* metrics of IA *a*2. For the IQA, *a*1, the distribution of plywood to sand1 and sand2 machines are 40% and 60% whereas the OQA *a*3 accepts 50% of the cut wood from the cut1 machine and 50% of the cut wood from the cut2 machine at each time point. These values are specialized by the operator in the *a*1 and *a*3 aggregators, respectively. Finally, the result of this specialized metric composite process is stored as *SandCutProcessDvarless* into the KB.

The specialized composite process stored as *SandCutProcessDvarless* in the KB captures the current state of the sand and cut manufacturing floor. If all the metrics are initialized as constants or expressions in terms of other metrics, we say that the component is *dvarless*. Due to the specializations performed on the process components in subsection 4.2 and in this subsection, the process components are *dvarless*. Also, because of the specializations performed in subsection 4.1 and in this subsections, the flow and aggregator components are dvarless. It is now possible to compute the expressions and variables of the components for the three time points. In order to do this, the operator issues the following tMQL query called *compute* on the composite process *SandCutProcessDvarless*:

```
Process computedProcess = compute
SandCutProcessDvarless
```

For the configuration of the processes and the aggregators in Fig. 7(a), the *compute* query gives the output shown in Fig. 7(b). The result of the *compute* query is a *grounded* component where all expressions and constraints are evaluated to respective constants. Since the computation does not change any constants in the input component, the constants that were specialized in this subsection are not shown. The *mValues* show the total cost incurred due to production in the two time periods. The IA *a*2 has one sanded wood that was initially present in the inventory for the first time point. For the two subsequent time points, five and three sanded wood were collectively produced by the two sanding machines and thus stored in the intermediary buffer. This computed value is reflected in the *invQty* metric of *a*2. Also, the *totalQty* metric in *a*2 reflects the maximum size of the inventory that was used during these three time points, which in this case, was five. The *totalTPAlloc* metric computed for the IQA *a*1 signifies the maximum number of plywoods available to be distributed among the two sanding machines according to the *outAllocRatio* metric in each time point. Analogously, the *totalTPAlloc* metric computed for the OQA *a*3 signifies the maximum amount of cut woods that can be output from the floor. The OQA *a*3 fetches the cut wood from the two cut machines distributed according to the *inAllocRatio* metric, the total of which is upper bounded by the *totalTPAlloc* metric of *a*3. Finally, as shown in the formula in Fig. 4, the *mValues* metric of the composite process is just the total cost of both the cut and sand processes.

Another way to query the composed model is to ask optimization queries against it. In order to do so, the process operator will have to first specialize the original composed sand and cut composite process (*SandCutProcess*) as shown in Fig. 8 (a). The composite model is specialized in a similar way as *SandCutProcessDvarless* shown in Fig. 7(a) with two differences. The first is that the speed of the machines here remain as decision variables (*throughputControl* metric has the value of "?"). The second difference is that the composite process *metrics* and *mValues* metrics are further specialized to include the demand value. The specialization here indicates that at least five cut woods should be produced by this man-

```
insert SandCutProcessDvarless =
        specialize SandCutProcess {
            //Process Metrics
            float sand1.throughputControl = [2.2,3.2];
            float sand2.throuputControl = [4.2,1.1];
            float cut1.througputControl = [2.1,1.5];
            float cut2.throughputControl = [1.2,3.3];
            float sand1.capacity = 5.5;
            float sand2.capacity = 4.2;
            float cut1.capacity = 3.1;
            float cut2.capacity = 4.1;
            int sand1.inputPerOutput = [1];
            int sand2.inputPerOutput = [1];
            int cut1.inputPerOutput = [1];
            int cut2.inputPerOutput = [1];

            //IQA Metrics
            string mt = "plywood";
            float a1.outAllocRatio = [0.4,0.6];
            int totalTPAlloc[t] = sand1.plywood.tpAlloc[t] +
                                  sand2.plywood.tpAlloc[t];

            //IA Metrics
            string mt = "sandedwood";
            int a2.capacity = 10;
            int a2.initInv = 1;
            float a2.inAllocRatio = [0.5,0.5];
            float a2.outAllocRatio = [0.4,0.6];
            int invQty[t] = invQty[t-1] +
                        sand1.plywood.periodQty[p] +
                        sand2.sandedwood.periodQty[p] -
                        cut1.cutwood.periodQty[p] -
                        cut2.cutwood.periodQty[p];
            int totalQty = max (t in 1..3) invQty[t];

            //OQA Metrics
            string plywood = "cutwood";
            float a3.inAllocRatio = [0.5,0.5];
            int totalTPAlloc[t] = cut1.cutwood.tpAlloc[t] +
                                  cut2.cutwood.tpAlloc[t]
};
```

(a)

```
Process computedProcess  {
    // Base Processes
    Process sand1 = {
        float mValues[cost] = 32.4;
                ⋮
    };
    Process sand2 = {
        float mValues[cost] = 45.6;
                ⋮
    };
    Process cut1 = {
        float mValues[cost] = 61.2;
                ⋮
    };
    Process cut2 = {
        float mValues[cost] = 54.2;
                ⋮
    };

    // Aggregators
    IQA a1 = {
        int totalTPAlloc = [2,5,1];
                ⋮
    };
    IA a2 = {
        int invQty = [1,5,3]
        int totalQty = 5
                ⋮
    };
    OQA a3 = {
        int totalTPAlloc = [0,4,3];
                ⋮
    };

    //Metircs
        float mValues[cost] = 193.4
};
```

(b)

Figure 7: Compute query: (a) specialize query on the sand and cut composite process to update process and aggregator metrics, (b) partial output of the compute query on the *SandCutProcessDvarless* component

ufacturing floor during the three time points. The result of this specialized metric composite process is stored as *SandCutProcessParameterless* into the KB.

If all the metrics of the composite process are specialized as constants or expressions in terms of other metrics or as decision variables, we say that the component is *parameterless*. Optimization queries can only be issued on parameterless components. Due to specializations performed on the components in subsection 4.1, subsection 4.2 and this subsection, all components for this example are parameterless. The process operator can now issue the following optimization query called *min* on the composite process, *SandCutProcessParameterless* to minimize the total production cost subject to constraints:

```
Process optmizedProcess = min
SandCutProcessParameterless.mValues[cost]
```

The output of the min optimization query is shown in Fig. 8 (b). This query finds the speeds for the sanding and cut machines such that all the constraints are satisfied. If many such speeds are found by the optimizer, then the one that computes to the least cost is returned as the result. In this case, the result is a *grounded* component where all expressions are evaluated to constants and the constraints are evaluated to true. If no such speeds can be found for which all the constraints of the components are satisfied, then the original component is returned with an *infeasible* or *unbounded* status. In our example, a feasible solution is found for the speeds of the machines shown in Fig. 8 (b). In this case, all the constraints of the components have a true value (not shown here). For the demand of five cut woods, the minimum cost incurred of running the two sanding and two cut machines is $163.5.

```
insert SandCutProcessParameterless =
        specialize SandCutProcess {
            //Process Metrics
            float sand1.capacity = 5.5,
            float sand2.capacity = 4.2,
            float cut1.capacity = 3.1,
            float cut2.capacity = 4.1,
            int sand1.inputPerOutput = [1],
            int sand2.inputPerOutput = [1],
            int cut1.inputPerOutput = [1],
            int cut2.inputPerOutput = [1],

            //IA Metrics
            int a2.capacity = 10,
            int a2.initInv = 1,
            float a2.inAllocRatio = [0.5,0.5],
            float a2.outAllocRatio = [0.4,0.6],

            //IQA Metrics
            float a1.outAllocRatio = [0.4,0.6],

            //OQA Metrics
            float a3.inAllocRatio = [0.5,0.5],

            //SanCutProcess Metrics
            {string} metrics = {cost, demand},
            float mValues[cost] =
                    sum(p in Process) p.cost
            float mValues[demand] = 5

}
```

(a)

```
Process SandCutProcessParameterless {
        // Base Processes
        Process sand1 = {
                float throughputControl = [1.4,0.6];
                float mValues[cost] = 31.8;
                        ⋮
                };
        Process sand2 = {
                float throughputControl = [1.6,1.4];
                float mValues[cost] = 51.9;
                        ⋮
        };
        Process cut1 = {
                float throughputControl = [1.1,1.9];
                float mValues[cost] = 44.3;
                        ⋮
        };
        Process cut2 = {
                float throughputControl = [0.8,1.2];
                float mValues[cost] = 35.5;
                        ⋮
        };

        // Aggregators
        IQA a1 = {
                int totalTPAlloc = [2,5,1];
                        ⋮
        };
        IA a2 = {
                int invQty = [1,2,2];
                int totalQty = 2;
                        ⋮
        };
        OQA a3 = {
                int totalTPAlloc = [0,4,3];
                        ⋮
                };
        //Metircs
        float mValues[cost] = 163.5;
};
```

(b)

Figure 8: Optimize query query: (a) specialize query on the sand and cut composite process to update process and aggregator metrics with *troughputControl* as decision variable, (b) partial output of the min query on the *SandCutProcessParameterless* component

5 tMQL Syntax and Semantics

This section provides an informal description of the syntax and semantics of the tMQL language. This description builds upon the description of the syntax and semantics provided in the previous section. The formal mathematical outline of the syntax and semantics are provided in appendices A and B, respectively.

An overview of the syntax model of the tMQL language is shown in Fig. 9. This syntax model is made up of four parts. The first part is the tMQL KB. This is the knowledge base that holds the data objects of the allowed type. New entries of the KB are generated using the insert command and each entry of the KB can be reused in queries to populate more entries of the KB.

The second part is concerned with the data types allowed in tMQL. The tMQL language allows for three subsets of types. First, there are OPL types that may be used to define simple number or string type values or for a more complex set or array type of values. Second, there are the tMQL component types that may be the itemFlow, IA, IQA, OQA, or the Process types. As mentioned before, they are used to map physical entities of the manufacturing floor and the queries can only be applied to objects of these component types. Finally, there is a constraint type that are expressions that can be reduced to either true or false.

The third part consists of the values accepted by the data types. The objects accepted by the number and the string types are trivial. When the data type is a set, it means their value is a set of tMQL component types and when the data type is an array, it means that the array is indexed on some tMQL component. The tMQL component type values contain metrics and constraints to enable the capturing of the manufacturing floor data and bounds. They are the built-in component types described in subsection 4.2. Some metrics are modifiable (via the *specialize* query) while some metrics are not. In Fig. 9, modifiable metrics start with a + while the unmodifiable metrics start with a -. Component constraints cannot be modified at this time. Each modifiable metric or OPL value may be of one of the following forms: (a) parameter (. . .) that can specialized later, (b) dvar (?) that can either be specialized later or left as a dvar in an

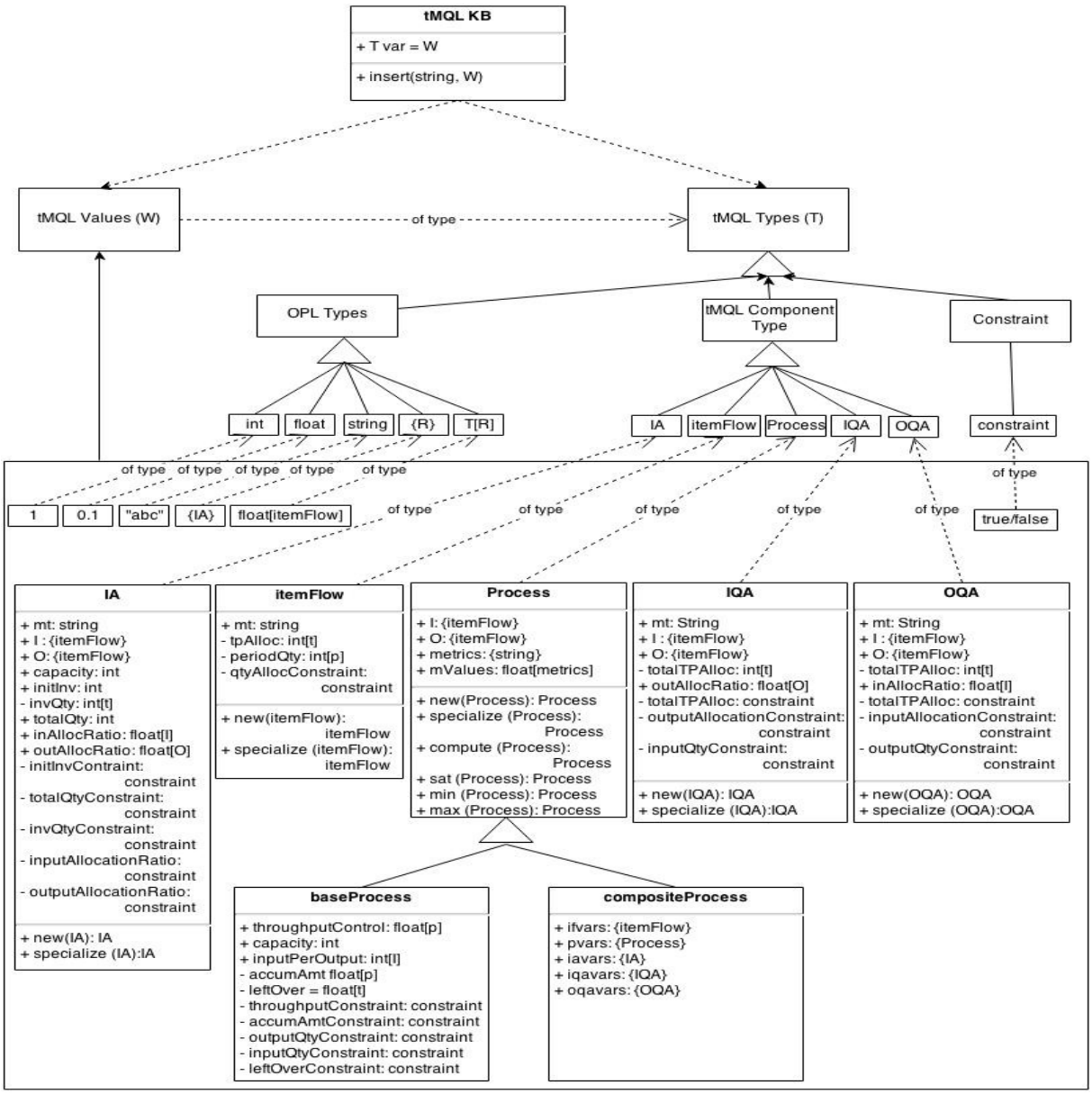

Figure 9: tMQL Syntax Model

optimize query, (c) expression using the metrics of the component or KB that belong to the same declared type, and (d) constants of the appropriate type. Each component value may be in three states at any times. These states are: (a) parameterless, when all OPL types in the component are initialized by a constant, expression or parameter and all component types in the component are also parameterless, (b) dvarless, when all OPL types in the component are initialized by a constant, expression, dvar or parameter and all component types in the component are also dvarless, and (c) grounded, when all OPL types in the component have values that are constant and all component types in the component are also grounded.

Finally, the syntax model consists of the queries. The *new* and *specialize* queries can be used on all tQML component type values to either create a copy or update some metrics of the components. The *compute* and *optimize* (sat/min/max) queries can only be used on components of Process type. Thus, either a machine's metrics can be computed or optimized or the metrics of the machine composed of other processes, aggregators, and flows may be computed or optimized. A *compute* query can be applied on a dvarless component while the optimize queries can be applied on a parameterless component.

The semantics of the *new* query is to make a copy of the component value for one of entries in the KB. The *specialize* query updates the modifiable metrics of the components and returns the updated component. The

semantics of the *compute* and *optimize* queries is to find values for the metrics that satisfy the constraints. Therefore, we first describe the mathematical definitions of the constraints in the components. In Fig. 9, unique names were provided to the constraints within the components. Each of these constraints are described below.

The itemFlow constraints include the qtyAllocConstraint, which is the number of items that the flow can carry at each time period (*periodQty*) is the upper bounded by the amount of items that the aggregators can accept at each time period ($tpAlloc$) shown in equation 1.

$$\underset{p}{\forall}\, periodQty[p] \leq tpAlloc[p-1] \tag{1}$$

The Process components have five constraints: (1) throughputConstraint: The speed of the machine is constrained by the maximum speed that the machine can be operated at as shown in equation 2, (2) outputQtyConstraint: The number of items produced by the process for a time period is the minimum of the amount of items that the aggregators can accept (output allocation), and the sum of any left over items from the previous time periods and the number of items produced during the current time period (throughputControl) as shown in equation 3, (3) leftOverConstraint: The number of items left over at a time period is the sum of any left over items from the previous time periods and the number of items produced during the current time period (throughputControl) but not the number of items that left the process via its output flows as shown in equation 4, (4) accumAmtConstraint: The number of items accumulated in the process at each time period is the sum of any left over items from the previous time periods and the number of items produced during the current time period (throughputControl) as shown in equation 5, and (5) inputQtyConstraint: The number of items the process requires at its input is the number of outputs that need to produced times the number of particular input required per output produced (inputPerOutput) as shown in equation 6.

$$\underset{p}{\forall}\, throughputControl[p] \leq capacity \tag{2}$$

$$\underset{p}{\forall}\, O.periodQty[p] == min(O.alloc[p-1], \lfloor leftOver[p-1] + throughputControl[p]) \rfloor \tag{3}$$

$$\underset{p}{\forall}\, leftOver[p] = (leftOver[p-1] + throughputControl[p]) - O.periodQty[p] \tag{4}$$

$$\underset{p}{\forall}\, accumAmt[p] = leftOver[p-1] + throughputControl[p] \tag{5}$$

$$\underset{p}{\forall}\underset{i\in I}{\forall}\, i.periodQty[p] == O.periodQty[p] \times inputPerOutput[i] \tag{6}$$

The IA components have five constraints: (1) initInvConstraint: Initially, the number of items in the IA (invQty) is set externally via the initInv metric as shown in equation 7, (2) totalQtyConstraint: The total number of items is upper bounded by the physical capacity of the IA as shown in equation 8, (3) invQtyConstraint: At each time point, the number of items in the IA is the sum of the items remaining from the previous time intervals and the items that came in via the input flows but not the items that were dispensed via its output flows as shown in equation 9, (4) inputAllocationConstraint: At each time point, the number of items that the IA accepts from an input flow is the product of the ratio allocated to that input and the space remaining in the IA as shown in equation 10, and (5) outputAllocationConstraint: At each time point, the number of items that the IA dispenses via an output flow is the product of the ratio allocated to that output and the number of items in the IA as shown in equation 11.

$$invQty[0] == initInv \tag{7}$$

$$totalQty \leq capacity \tag{8}$$

$$\underset{t}{\forall}\, invQty[t] == invQty[t-1] + \sum_{i\in I} i.periodQty[t] - \sum_{o\in O} o.periodQty[t] \tag{9}$$

$$\underset{t}{\forall}\underset{i\in I}{\forall}\, i.tpAlloc[t] == inAllocRatio[i] \times (totalQty - invQty[t]) \tag{10}$$

$$\underset{t}{\forall}\underset{o\in O}{\forall}\, o.tpAlloc[t] == outAllocRatio[i] \times (invQty[t]) \tag{11}$$

The IQA components have three constraints: (1) totalTPAllocConstraint: At each time point, the total allocation of the inputs is the sum of allocations on all the inputs as shown in equation 12, (2) outputAllocationConstraint: At each time point, the output allocation on each output is the product of the ratio allocated to that output and the total allocation of the inputs as shown in equation 5, and (3) inputQtyConstraint: At each time period, the number of outputs coming out of the IQA should be the same as the number of inputs coming into the IQA as shown in equation 14.

$$\underset{t}{\forall}\, totalTPAlloc[t] == \sum_{i\in I} i.tpAlloc[t] \tag{12}$$

$$\underset{t}{\forall}\underset{o\in O}{\forall}\, o.tpAlloc[t] == outAllocRatio[o] \times totalTPAlloc[t] \tag{13}$$

$$\underset{p}{\forall}\left(\sum_{i\in I} o.periodQty[p] == \sum_{o\in O} o.periodQty[p]\right) \tag{14}$$

The OQA component also has three constraints. The totalTPAllocConstraint is the same as the one described

in IQA. In addition, OQA also has constraints on the input allocation and the output quantity. These constraints are symmetrical to those described for IQA and hence have been left out here.

The semantics of the compute query is that given a dvarless component, all its expressions and the expressions in any internal components are evaluated to constants and all the constraints are computed to be true or false. The semantics of the *sat* query is that given a parameterless component, the possible values for the decision variables in the component is found. If any of these values result in making the constraints in this component and all of its internal components true, then the first such satisfied component is returned. If no such component can be found for all possible values then the original component object is returned with an unbounded or infeasible status. Similarly for min or max queries, all possible values for the decision variables are found and the values that makes all constraints true and minimizes or maximizes a numeric metric is returned. If no such value can be found for the decision variables, then the original component object with an infeasible or unbounded status is returned.

6 Implementation of deterministic tMQL

This section describes the implementation of the deterministic tMQL model. We describe the implementation of the tMQL components to Optimization Programming Language (OPL) by IBM [12]. OPL is an optimization software package that solves integer programming and very large linear programming problems. The optimization results are then provided as decision guidance to decision makers. Each tMQL component is modeled in OPL as a generic module. Here, these generic modules are described with the help of code-snippets from OPL. Then, we describe the method of using these modules to run an optimization query by showing how the data file is manually generated for any tMQL model.

First the global time points and the period metrics are generalized. Here it is assumed that all time points start from 0 and periods start from 1. The *periodLength* metric is used to describe the length of each period. Given the number of periods (*noPeriods*), the last time point (*lastTP*) is the same as the number of periods. Except that the time points range from 0 to *lastTP* and the periods range from 1 to *noPeriods* as shown by the following OPL code snippet.

```
int periodLength = ...;
int noPeriods = ...;
int lastTP = noPeriods;
range timeRange = 0..lastTP;
range periodRange = 1..noPeriods;
```

In order to generalize the components, the Process, itemFlow, IA, IQA and OQA are stored as an array of their respective ids. The metrics of each module are indexed on their respective ids. If the metric is also indexed on other units such as time points or periods, then the metric is stored as a 2D array, first indexed on the component ids and then indexed on the units. Further, the metrics of the components whose values are defined as parameters (. . .) in Fig. 5 and Fig. 6 have the values of . . . in the implementation. This is to indicate that OPL will search for the values of these metrics in the data file. The metrics of the components whose values are defined as decision variables (?) in Fig. 5 and Fig. 6 are defined as *dvar* variables in the implementation. These variables are decision variables in the optimization problem and will be determined by OPL while satisfying the constraints and/or optimizing some numeric value. An example of this strategy is shown via the code snippet of the itemFlow metric definition in OPL.

```
{string} itemFlowIds = ...;
string itemFlow_mt[itemFlowIds] = ...;
dvar int+ itemFlow_tpAlloc[itemFlowIds][timeRange];
dvar int+ itemFlow_periodQty[itemFlowIds][periodRange];
```

Here the itemFlow component is described as ids. The match type (*mt*) variable is indexed on these ids and it is shown as a parameter. The *tpAlloc* and *periodQty* variables are shown as decision variables that will be calculated by OPL in the optimization problem.

The constraints for each component is shown using their mathematical definitions shown in section 5. As an example, the *qtyAllocConstraint* expression for the itemFlow component is show below.

```
forall(id in itemFlowIds){
    forall(p in 1..noPeriods){
        itemFlow_periodQty[id][p] <=
         itemFlow_tpAlloc[id][p-1];
    }
}
```

For all indexes in the itemFlow component ids and for all periods, this constraint expression bounds the *periodQty* value by the respective *tpAlloc* value. Since the *periodQty* value is indexed on the periods and *tpAlloc* value is indexed on the time points, the *periodQty* value at period p is upper bounded by the *tpAlloc* value at the previous time point, $p-1$. The other constraints are expressed similarly in OPL.

In order to manually translate a system layout to an OPL data file that can be run with the OPL model, the following steps are taken.

1. Convert the system layout to the graphical language by generating a composite process graph for the manufacturing floor like the one shown in Fig. 3 for the sand and cut example.

2. Divide the composite process graph into one or more composite process modules

3. For each composite process module, identify the tMQL components: aggregators (IA, IQA, OQA), the atomic processes (baseProcess) and then the flows that connect them.

4. Add each itemFlow identified as a unique itemFlow id to the data file.

5. Provide the inputs and outputs for each of the other components (IA, IQA, OQA and baseProcess) as the same ids added as itemFlow ids thus connecting them manually with each other. Add these components to the data file.

6. Provide the other interface constants for IA, IQA,OQA and baseProcess. This could be done in tMQL by using the *specialize* query.

7. Perform steps 3-6 for all the composite process modules identified in #2.

8. Connect the different composite process modules by using the same itemFlow id as inputs/outputs to these modules

9. Run the satisfy or min/max query on the composed model in OPL.

7 Conclusion and future work

This paper described an analytical modeling and query language for composing different functions on a manufacturing floor. In order to perform this composition, we provide the following tMQL components: itemFlow, Process, IA, IQA, and OQA. In addition, the composite process can encapsulate one or more of the other components such that the itemFlow acts as an interface among them. Each component comes with a rich set of metrics and constraints such that there is a flexibility to represent the data and bound it appropriately. The representation components are simple assignments stored into a tMQL KB. This allows for reusing already created components by simply using the left var of its assignment in the tMQL KB or by creating a copy of these assigned components using the *new* query. In order to analyze the composed data we present queries that can create a new component, specialize the assignments of a component, compute values for a component, find decision variables that satisfy constraints and/or minimize or maximize numeric values. This allows for the composition data to be queried in variety of ways to ask what-if questions, metric optimization or metric computation queries. Once a component has been queried, the language also allows for the result object to be inserted as a new assignment into the tMQL KB using the insert command. In this paper we also provide the formal syntax and semantics of the model composition and querying language (Appendices A and B).

We demonstrate the workings of the tMQL by using an example bookshelf manufacturing floor. We show how to convert the manufacturing floor functions into tMQL Components. Then, we demonstrate how the different components can be connected using the left vars of type itemFlow from the tMQL KB. Then, one can run the compute, sat or min/max query on these composed models so as to obtain meaningful results for these queries. We demonstrate this by manually converting the bookshelf composite model into a deterministic OPL model and the associated data file (tMQL KB) and then running optimization tasks to ask metric calibration and what-if analysis queries. This paper shows the results of running these queries (Appendix C).

We are currently investigating: (a) a simulation algorithm to accompany the current optimization queries so that the composed model can be run as a simulation, (b) heuristic techniques to design and run tMQL on stochastic manufacturing floor functions, (c) a library of the components discussed here for different manufacturing scenarios, and (d) sustainability metrics functions that can be incorporated into the tMQL.

Acknowledgements

This work is partially supported by NIST grant No. 70NANB12H277.

Appendices

A tMQL syntax

A.1 tMQL Types

Variables in tMQL KB can be of the following types $\boldsymbol{T}$

1. tMQL Component types
   (a) $itemFlow$: item flow type that connects two other tMQL Component type.
   (b) IA : Inventory aggregator type
   (c) IQA : Input quantity aggregator type
   (d) OQA : Output quantity aggregator type
   (e) $Process$: Process type for the base components and composite process type
2. OPL Types: We adopt the OPL atomic types, set, and array types, including:
   (a) int : Integer numbers
   (b) $float$: Floating point numbers
   (c) $string$: Strings of characters
   (d) $\{R\}$: A set of type R where R is a tMQL Component type

(e) $T[R]$: An array indexed on components of type R where R is a tMQL Component type with elements of type T in $\boldsymbol{T}$

3. Constraint type: An expression evaluates this type to either true or false.

A.2 tMQL Values

The types in $\boldsymbol{T}$ have the following values:

1. A value of tMQL Component type is a symbolic expression of the form $\{t_i\ lv_1 = e_1, \ldots, t_n\ lv_n = e_n\}$ where:
   - $t_i (1 \leq i \leq n)$ is one of the types in $\boldsymbol{T}$ and represents the type of expression e_i
   - $lv_i (1 \leq i \leq n)$ is a unique left var name within the component
   - $e_i (1 \leq i \leq n)$ is an expression and can be one of the following
     - a constant of type t_i
     - a var v_k from tMQL KB defined previously $(1 \leq k \leq j-1)$ of type t_i
     - a parameter denoted as ...
     - a decision variable or dvar denoted as ?, if t_i is a numeric type: int or float

   (a) A value of type *itemFlow* is tMQL Component value of the form:

   {
   string mt = ... or a string constant,
   int tpAlloc[t] = ?,
   int periodQty[p] = ?,
   constraint qtyAllocConstraint[p] = expr: returns true or false
   }

   - A component of type itemFlow is a component where none of the expressions $e_i (1 \leq i \leq n)$ are vars in tMQL KB
   - We say that itemFlow component is
     i. *parameter-less* if none of the expressions $e_i (1 \leq i \leq n)$ are "..."
     ii. *dvar-less* if none of the expressions $e_i (1 \leq i \leq n)$ are "..." or "?"
     iii. *grounded* if all expressions $e_i (1 \leq i \leq n)$ are constants

   (b) A value of type IA is tMQL Component value of the form:

   {
   string mt = ... or string constant,
   {itemFlow} I = ... or a set of itemFlow components,
   {itemFlow} O = ... or a set of itemFlow components,
   int capacity = ... or integer constant,
   int initInv = ... or integer constant,
   int totalQty = ... or ? or integer constant,
   float inAllocRatio[I] = ... or integer constant indexed on itemFlows in I,
   float outAllocRatio[O] = ... or integer constant indexed on itemFlows in O,
   int invQty[t] = ?,
   constraint initInvConstraint = expr: returns true or false,
   constraint totalQtyConstraint = expr: returns true or false,
   constraint invQtyConstraint = expr: returns true or false,
   constraint inputAllocationConstraint = expr: returns true or false,
   constraint outputAllocationConstraint = expr: returns true or false,
   }

   We say that IA component is
   i. *parameter-less* if none of the expressions $e_i (1 \leq i \leq n)$ are "..." and all itemFlows (on the sets assigned to I and O) used in the component is also parameter-less
   ii. *dvar-less* if none of the expressions $e_i (1 \leq i \leq n)$ are "..." or "?" and all itemFlows (on the sets assigned to I and O) used in the component is also dvar-less
   iii. *grounded* if all expressions $e_i (1 \leq i \leq n)$ are constants and all itemFlows (on the sets assigned to I and O) used in the component is also grounded

   (c) A value of type IQA is tMQL Component value of the form:

   {
   string mt = ... or string constant,
   {itemFlow} I = ... or a set of itemFlow components,
   {itemFlow} O = ... or a set of itemFlow components,
   float outAllocRatio[O] = ... or integer constant indexed on itemFlows in O,
   int totalTPAlloc[t] = ?,
   constraint totalTPAllocConstraint = expr: returns true or false,
   constraint outputAllocationConstraint = expr: returns true or false,
   constraint inputQtyConstraint = expr: returns true or false,
   }

We say that IQA component is

i. *parameter-less* if none of the expressions $e_i(1 \leq i \leq n)$ are "..." and all itemFlows (on the sets assigned to I and O) used in the component is also parameter-less
ii. *dvar-less* if none of the expressions $e_i(1 \leq i \leq n)$ are "..." or "?" and all itemFlows (on the sets assigned to I and O) used in the component is also dvar-less
iii. *grounded* if all expressions $e_i(1 \leq i \leq n)$ are constants and all itemFlows (on the sets assigned to I and O) used in the component is also grounded

(d) A value of type *OQA* is tMQL Component value of the form:

{
string mt = ... or string constant,
{itemFlow} I = ... or a set of itemFlow components,
{itemFlow} O = ... or a set of itemFlow components,
float inAllocRatio[O] = ... or integer constant indexed on itemFlows in O,
int totalTPAlloc[t] = ?,
constraint totalTPAllocConstraint = expr: returns true or false,
constraint inputAllocationConstraint = expr: returns true or false,
constraint outputQtyConstraint = expr: returns true or false,
}

We say that OQA component is

i. *parameter-less* if none of the expressions $e_i(1 \leq i \leq n)$ are "..." and all itemFlows (on the sets assigned to I and O) used in the component is also parameter-less
ii. *dvar-less* if none of the expressions $e_i(1 \leq i \leq n)$ are "..." or "?" and all itemFlows (on the sets assigned to I and O) used in the component is also dvar-less
iii. *grounded* if all expressions $e_i(1 \leq i \leq n)$ are constants and all itemFlows (on the sets assigned to I and O) used in the component is also grounded

(e) A value of type *Process* is tMQL Component value that contains the following types and variables: {itemFlow} I, {itemFlow} O, {string} metrics and float mValues[metrics]. There may be two types of values of type *Process*: *baseProcess* and *compositeProcess*.

i. A baseProcess is a tMQL Component *Process* value of the form:

{
{itemFlow} I = ... or a set of itemFlow components,
{itemFlow} O = ... or a set of itemFlow components,
float throughputControl[p] = ? or ... or array of floats indexed on noPeriods,
float capacity = ... or floating point constant,
int inputPerOutput[I] ... or integer constant indexed on itemFlows in I,
float accumAmt[p] = ?,
float leftOver[t] = ?,
{string} metrics = ... or set of strings,
float mValues[metrics] = ... or array or floats indexed on the set of metrics,
constraint throughputConstraint = expr: returns true or false,
constraint accumAmtConstraint[p] = expr: returns true or false,
constraint outputQtyConstraint[p] = expr: returns true or false,
constraint inputQtyConstraint[p] = expr: returns true or false,
constraint leftOverConstraint[t] = expr: returns true or false,
}

We say that baseProcess component is

A. *parameter-less* if none of the expressions $e_i(1 \leq i \leq n)$ are "..." and all itemFlows (on the sets assigned to I and O) used in the component is also parameter-less
B. *dvar-less* if none of the expressions $e_i(1 \leq i \leq n)$ are "..." or "?" and all itemFlows (on the sets assigned to I and O) used in the component is also dvar-less
C. *grounded* if all expressions $e_i(1 \leq i \leq n)$ are constants and all itemFlows (on the sets assigned to I and O) used in the component is also grounded

ii. A compositeProcess is a tMQL Component *Process* type value of the form:

{
{itemFlow} I = ... or a set of itemFlow components,
{itemFlow} O = ... or a set of itemFlow components,
{itemFlow} ifvars = ... or set of itemFlow components
{Process} pvars = ... or set of Process components

{IA} iavars = ... or set of IA components
{IQA} iqavars = ... or set of IQA components
{OQA} oqavars = ... or set of OQA components
{string} metrics = ... or set of strings, float mValues[metrics] = ... or array or floats indexed on the set of metrics,
}

We say that compositeProcess component is

A. *parameter-less* if none of the expressions $e_i(1 \leq i \leq n)$ are "..." and all itemFlows (on the sets assigned to I, O, ifvars), Processes (on the sets assigned in pvars), IAs (on the sets assigned in iavars), IQAs (on the sets assigned in iqavars), IQAs (on the sets assigned in iqavars) and OQAs (on the sets assigned in oqavars) used in the component are also parameter-less

B. *dvar-less* if none of the expressions $e_i(1 \leq i \leq n)$ are "..." or "?" and all itemFlows (on the sets assigned to I, O, ifvars), Processes (on the sets assigned in pvars), IAs (on the sets assigned in iavars), IQAs (on the sets assigned in iqavars), IQAs (on the sets assigned in iqavars) and OQAs (on the sets assigned in oqavars) used in the component are also dvar-less

C. *grounded* if all expressions $e_i(1 \leq i \leq n)$ are constants and and all itemFlows (on the sets assigned to I, O, ifvars), Processes (on the sets assigned in pvars), IAs (on the sets assigned in iavars), IQAs (on the sets assigned in iqavars), IQAs (on the sets assigned in iqavars) and OQAs (on the sets assigned in oqavars) used in the component are also grounded

(f) Finally the value for the tMQL Component type can be a parameter denoted as ...

2. OPL type values
   (a) A value of type *int* is a positive integer or negative integer or ... or ?
   (b) A value of type $float$ is a positive decimal number or negative decimal number or ... or ?
   (c) A value of type *string* is a sequence of characters or ...
   (d) A value of type $\{R\}$ is a set of components of type R where R is a tMQL Component type or ...
   (e) A value of type $T[R]$ is an array of elements of type T in $\boldsymbol{T}$ indexed by an index set of elements of type R or ...
3. Constraint type values: A value of type *constraint* is an expression that evaluates to true or false

A.3 tMQL KB

tMQL KB is a sequence of assignments of the form:
$T_1\ v_1 := w_1$
$T_2\ v_2 := w_2$
$\vdots$
$T_n\ v_n := w_n$, where:

- $T_i(1 \leq i \leq n)$ is one of the tMQL types in $\boldsymbol{T}$.
- $v_i(1 \leq i \leq n)$ is a unique var names
- $w_i(1 \leq i \leq n)$ is a tMQL value of type T_i

One can add new query assignments into the tMQL KB. An insert is used to put new variables into the tMQL KB
:
insert newVar $:= q_i$, where:

- newVar is a new tMQL var name
- the type of newVar is that of q_i
- q_i is a tMQL Query

The insert operator computes the query q_i resulting in a tMQL value , w of type T (determined implicitly from q_i) and adds "T newVar = w" as the last entry in the KB.

A.4 tMQL Query

A tMQL Query (or just query) is one of the following:

1. Create a copy of the tMQL component.
   new v_j, where
   - v_j is a tMQL Component in tMQL KB

   Returns in a copy of the tMQL Component v_j
2. Specialize a tMQL component
   specialize $v_j\{t_1\ lv_1 := e_1; \ldots; t_k\ lv_k := e_k;\}$, where:
   - v_j is a tMQL Component in tMQL KB,
   - (t_i, lv_i) $(1 \leq i \leq k)$ is the type/var pair used in v_j,
   - e_i $(1 \leq i \leq k)$ is:
     (a) if lv_i type in v_j is a tMQL Component type T (Process, itemFlow, IA, IQA, OQA), then e_i is a query that returns type T
     (b) if type of lv_i in v_j is {T} (set of T) where T is a tMQL Component type, then e_i is of the form $\{e_{i1}, \ldots, e_{ik}\}$ where $e_{i1}, \ldots, e_{ik}$ are tMQL Query of type T

(c) if lv_i type in v_j is numeric (int or float) then e_i is either a numeric constant or ... or ?

(d) for lv_i of any other type (not tMQL Component type nor numeric), e_i can be a constant of that type or ...

- Every e_i ($1 \leq i \leq n$) must be more specialized than the previous e_i in v_j where "more specialized" denoted as $\succ$, is a partial order defined as follows:
  $constant \succ expression \succ \cdots \succ ?$
- The query type is that of v_j

Note that *specialize* $v_j\{\}$ is equivalent to *new* v_j

3. Compute query on a tMQL component
   compute v_j, where:
   - v_j is a dvar-less tMQL Component in tMQL KB,
   - Returns in a grounded tMQL Component of the type of v_j where all the expressions are computed into constants
4. Satisfy constraints query on a tMQL component
   sat v_j, where:
   - v_j is a parameter-less tMQL Component in tMQL KB,
   - Returns in a grounded tMQL Component of the type of v_j where all the constraints are true and expressions are computed into constants or an INFEASIBLE object where some of the constraints are false (see precise semantics definition)
5. Optimize a numeric variable on a tMQL Component
   min/max v_j.*numVar*, where:
   - v_j is a parameter-less tMQL Component in tMQL KB
   - *numVar* is one of the variables in v_j that is to be minimized or maximized
   - Returns in a grounded tMQL Component of the type of v_j where all the constraints are true for a minimum or maximum numVar value and expressions are computed into constants or an INFEASIBLE object where some of the constraints are false (see precise semantics definition)

B tMQL semantics

The semantics is a function:

$$QK \rightarrow U \times Status$$

where:

- QK is a set of all valid query-KB pairs (q, KB). Here "valid" means that one of the queries from tMQL Query is called on a tMQL Component type variable present in tMQL KB.
- U is a set of all tMQL Component values.
- Status is a set of relevant status. Status informs the state of the returned value U. Some status include *success*, $infeasible$, *unbounded*, *optimal* and $feasible$.

In the following definitions, we assume that a KB is of the form $T_1\ v_1 := w_1 \ldots T_n\ v_n := w_n$

1. $Sem(new\ v) = (w, success)$ if $v = v_i$ in KB, where w is a tMQL value constructed by making a copy of the value w_i. A *success* status is also returned.
2. $Sem(specialize\ v\{t_1\ lv_1 = e_1, \ldots, t_k\ lv_k = e_k\}) = (w', success)$, if $v = v_i$ in KB, where w' is a tMQL value constructed from w_i by replacing the RHS of $lv_1, \ldots, lv_k$ with $e1, \ldots, e_k$, that must be more specialized. A *success* status is also returned.
3. $Sem(compute\ v\{t_1\ lv_1 = e_1, \ldots, t_k\ lv_k = e_k\}) = (w', success)$, if $v = v_i$ in KB, where w' is a tMQL value constructed from w_i such that:
   (a) if w_i is a dvar-less component value like the one of type itemFlow that does not contain any other vars from tMQL KB then, w' is a grounded component value where each expression in w_i is replaced by a constant of the appropriate type by computing the expression
   (b) if w_i is a dvar-less component like the one of type IA, IQA, OQA or Process that contain other vars from tMQL KB then, w' is a grounded component value where each expression is replaced by a constant of the appropriate type by computing the expression and each component value within w_i is also computed to obtain a grounded component value. The components within w_i are computed in the following order of tMQL Component types:
      i. itemFlow
      ii. IA
      iii. IQA
      iv. OQA
      v. Process

   A *success* status is also returned.
4. $Sem(sat\ v\{t_1\ lv_1 = e_1, \ldots, t_k\ lv_k = e_k\})$ is a tMQL value constructed as follows:
   (a) If $v = v_i$ in KB and all constraints in w' are satisfied, then
   $Sem(sat\ v\{t_1\ lv_1 = e_1, \ldots, t_k\ lv_k = e_k\}) = (w', feasible)$

(b) Otherwise, $Sem(sat\ v\{t_1\ lv_1 = e_1, \ldots, t_k\ lv_k = e_k\}) = (w, infeasible\ or\ unbounded)$

This query takes the following steps:

(a) Find the possible instances for the dvar variables in the parameter-less component value w_i. For each such instances do the following

i. Run *specialize* on the component value w_i to replace the dvar variables with the instance

ii. Run *compute* on the specialized component value to obtain a grounded component value. This resulting value follows the semantics of the *compute* query.

iii. If all the constraints in the computed component value are satisfied then this component value is returned as w' with a $feasible$ status

(b) If some the constraints in the computed component remain unsatisfied, then the original component value (w) is returned with with the *unbounded* or $infeasible$ status.

5. $Sem(min/max\ v\{t_1\ lv_1 = e_1 \ldots, t_k\ lv_k = e_k\}.numVar)$ is a tMQL value constructed as follows:

(a) If $v = v_i$ in KB and all constraints in w' are satisfied for a min/max numVar, then $Sem(min/max\ v\{t_1\ lv_1 = e_1 \ldots, t_k\ lv_k = e_k\}.numVar) = (w', optimal)$

(b) Otherwise, $Sem(min/max\ v\{t_1\ lv_1 = e_1 \ldots, t_k\ lv_k = e_k\}.numVar) = (w, infeasible\ or\ unbounded)$

This query will take the following steps:

(a) Find the possible instances for the dvar variables in the parameterless component w_i. For each such instances do the following:

i. Run *specialize* on the component w_i to replace the dvar variables with the instance

ii. Run *compute* on the specialized component value to obtain a grounded component value. This resulting value follows the semantics of the *compute* query.

(b) In all such computed component values where all constraints are satisfied, the value with minimum or maximum numVar value is returned as w' with a *optimal* status

(c) If some the constraints in all the computed component values remain unsatisfied, then the original component value (w) is returned with with the *unbounded* or $infeasible$ status.

C Application example

This section presents the workings of tMQL via the bookshelf example application introduced in section 2. Here we model the bookshelf example and ask appropriate queries against the model. This section demonstrates how the data model for a complex manufacturing floor can be built one module at a time and queries can be asked against it. Table 1 shows some parts of the data model that define the connections of the components for the bookshelf component diagram of Fig. 1. These include the match types, input flows and output flows of the IA, IQA and OQA. For the itemFlow, these include the match type and for the baseProcess, these include the input flows and the output flows. The itemFlows are not explicitly listed but they have the same match types of the aggregator they originate or end in.

The remainder of this section show how to query the model. This can be done in two ways: computation and optimization.

Optimization allows us to perform metric optimization and constraint satisfaction queries. Using these queries one can perform what-if analysis and metric calibration. As described in section 6, we use OPL to describe the model and optimize the decision variables. The convention of <*matchType*>*From*<*componentName*> is used to for itemFlow ids. Due to lack of space, the connections for all components and match types for aggregators of only some of the components in the bookshelf example are shown in Table 1. The connections and match types for the remaining components are similar. The other parts of the data model are described below:

- The global metrics used were: noPeriods = 10, periodLength = 2
- IQA a1 has an equal outAllocRatio among its outputs
- For all baseProcesses, accumAmt, leftOver and throughputControl are decision variables (dvar)
- Base processes Packaging, Assembly1 and Assembly2 have inputPerOutput of one finished left plank, one finished right plank, three shelves and one accessory package. The accessory gathering process packages 12 screws, 10 nails and four rollers. Other processes require one input per every output produced.
- Each baseProcess has its own capacity and it is provided in Table 2
- Each baseProcess also has a process cost that is computed as a function of the dvar throughputControl. $\{cost : (5 \leq x(p) \leq 10 \rightarrow 5 \times x(p)) \wedge (10 < x(p) \leq 20 \rightarrow 5.5 \times x(p)) \wedge (20 < x(p) \leq 30 \rightarrow 6 \times x(p)) \wedge (x(p) > 30 \rightarrow 7.5 \times x(p))\}$ where x is the throughputControl for period p.

Table 1: Component connections data for the bookshelf example

Type	Component Name	Connection Data
IQA	a1	mt = plywood I = {plywoodToA1} O = {plywood1FromA1, plywood2FromA1}
Process	Sanding1	I = {plywood1FromA1} O = {sandedwoodFromSanding1}
IA	a2	mt = sandedwood I = {sandedwoodFromSanding1, sandedwoodFromSanding2} O = {sandedwood1FromA2, sandedwood2FromA2, sandedwood3FromA2}
Process	Cutting1	I = {sandedwood1FromA2} O = {cutwoodFromCutting1}
Process	Paint & Dry1	I = {cutwoodFromA3} O = {pdwoodFromPaintDry1}
Process	Finishing1	I = {pdwoodFromA5} O = {finishedwoodFromFinishing1}
Process	Packaging	I = {packagedAccessory1FromAB, finishedwood1FromA8, finishedwood1FromA9, finishedwood1FromAA} O = {packagedwoodFromPackaging}
Process	Accessory Gathering	I = {screws, nails, roller} O = {packagedAccessory}
IA	aB	mt = accessoryPackaged I = {packagedAccessory} O = {packagedAccessory1FromAB, packagedAccessory2FromAB, packagedAccessory3FromAB}
Process	Assembly1	I = {packagedAccessory2FromAB, finishedwood2FromA8, finishedwood2FromA9, finishedwood2FromAA} O = {assembledBookShelf1}
OQA	aC	mt = assembledBookShelf I = {assembledBookShelf1, assembledBookShelf2} O = {assembledBookShelf}

- Each baseProcess also has a carbon emission value that is computed as a function of the dvar throughputControl. $\{CO_2 : (5 \leq x(p) \leq 10 \rightarrow 2 \times x(p)) \wedge (10 < x(p) \leq 20 \rightarrow 3 \times x(p)) \wedge (20 < x(p) \leq 30 \rightarrow 4 \times x(p)) \wedge (x(p) > 30 \rightarrow 5 \times x(p))\}$ where x is the throughputControl for period p. Carbon emissions are measured in kg.
- Each IA has its own capacity provided in Table 2. The initInv of each IA is set to 0.
- For each IA, totalQty and invQty are dvar.
- The inAllocRatio and outAllocRatio for each IA is set to distribute the inputs and outputs equally among the flows entering and exiting the IA.
- OQA aC has an equal inAllocRatio among its inputs.

The model was run on OPL and the following optimization and what-if analysis queries were analyzed. Each query below show the total quantity (totalQty) of items that the IA needs to hold. Also, each query shows the optimized values of the state (cost or carbon emissions) for each process. Finally, the total optimized value obtained for each query is:

1. **Minimize the process cost such that at least four bookshelves are assembled and one bookshelf packaged**
   - TotalQty: $a2 = 29, a3 = 42, a4 = 40, a5 = 42, a6 = 40, a7 = 40, a8 = 40, a9 = 40, aA = 40, aB = 40$
   - Cost: $Sanding1 = 35, Sanding2 = 37.5, Cutting1 = 15, Cutting2 = 40, Cutting3 = 15, Paint\&Dry1 = 15, Paint\&Dry2 = 40, Paint\&Dry3 = 15, Finishing1 = 15, Finishing2 = 40, Finishing3 = 15, AccessoryGathering = 15, Packaging = 2.5, Assembly1 = 7.5, Assembly2 = 2.5$

Table 2: Capacity values for the baseProcesses and IAs in the bookshelf example

Type	Component Name	Capacity	Type	Component Name	Capacity
Process	Sanding1	5	Process	Sanding2	5
Process	Cutting1	15	Process	Cutting2	12
Process	Cutting3	11	Process	Paint & Dry1	13
Process	Paint & Dry2	12	Process	Paint & Dry3	9
Process	Finishing1	8	Process	Finishing2	10
Process	Finishing3	9	Process	Accessory Gathering	4
Process	Packaging	3	Process	Assembly1	4
Process	Assembly2	2	IA	a2	45
IA	a3	42	IA	a4	40
IA	a5	40	IA	a6	42
IA	a7	40	IA	a8	40
IA	a9	40	IA	aA	40
IA	a9	40			

- TotalCost = 310

2. **Minimize carbon emissions such that at least four book shelves are assembled and one bookshelf packaged**

   - TotalQty: $a2 = 28, a3 = 42, a4 = 40, a5 = 42, a6 = 40, a7 = 40, a8 = 40, a9 = 40, aA = 40, aB = 40$
   - Carbon Emissions: $Sanding1 = 14, Sanding2 = 15, Cutting1 = 6, Cutting2 = 16, Cutting3 = 6, Paint\&Dry1 = 6, Paint\&Dry2 = 16, Paint\&Dry3 = 6, Finishing1 = 6, Finishing2 = 16, Finishing3 = 6, AccessoryGathering = 6, Packaging = 1, Assembly1 = 1, Assembly2 = 3$
   - TotalCarbonEmisssions = 124

3. **Minimize the process cost such that the carbon emission is at most 100 kg and at least four bookshelves are assembled and one bookshelf packaged**

   - TotalQty: $a2 = 45, a3 = 42, a4 = 40, a5 = 42, a6 = 40, a7 = 40, a8 = 40, a9 = 40, aA = 40, aB = 40$
   - Cost: $Sanding1 = 40, Sanding2 = 17.5, Cutting1 = 12.5, Cutting2 = 37.5, Cutting3 = 5, Paint\&Dry1 = 12.5, Paint\&Dry2 = 37.5, Paint\&Dry3 = 5, Finishing1 = 12.5, Finishing2 = 37.5, Finishing3 = 5, AccessoryGathering = 15, Packaging = 2.5, Assembly1 = 5, Assembly2 = 5$
   - TotalCost = 250

4. **Optimize (maximize) the number of bookshelves that can be produced such that the total cost is at most $500 and carbon emissions is at most 180 kg**

   - TotalQty: $a2 = 44, a3 = 42, a4 = 40, a5 = 42, a6 = 40, a7 = 40, a8 = 40, a9 = 40, aA = 40, aB = 40$
   - Carbon Emissions: $Sanding1 = 22, Sanding2 = 22, Cutting1 = 8, Cutting2 = 24, Cutting3 = 9, Paint\&Dry1 = 8, Paint\&Dry2 = 24, Paint\&Dry3 = 8, Finishing1 = 8, Finishing2 = 24, Finishing3 = 8, AccessoryGathering = 8, Packaging = 4, Assembly1 = 2, Assembly2 = 1$
   - TotalCarbonEmisssions = 180
   - Cost: $Sanding1 = 55, Sanding2 = 55, Cutting1 = 20, Cutting2 = 60, Cutting3 = 22.5, Paint\&Dry1 = 20, Paint\&Dry2 = 60, Paint\&Dry3 = 20, Finishing1 = 20, Finishing2 = 60, Finishing3 = 20, AccessoryGathering = 20, Packaging = 10, Assembly1 = 5, Assembly2 = 1.5$
   - TotalCost = 450
   - BookShelf Packaged: 4
   - BookShelf Assembled: 3

These queries shows how a model for the bookshelf composite model can be used to ask what-if analysis queries by using optimization and metric calibration queries. The model gives the flexibility to use different process metrics as decision variables while allowing the constraints to bound the physical specifications of the manufacturing floor. The same model composition and connection data can be used to ask different queries against the model. Metric calibration can be

performed by repeatedly performing the constraint satisfaction queries against the data. When in some cases, we cannot provide exact data to quantify a metric, it is possible to provide them as a function of another metric. This is shown in the model and query above via the cost metric, which is a function of the throughputControl. It is also possible to add the solutions to these queries into the component repository and use them in the composition of other more complex composite models.

References

[1] M. Fu, F. Glover, and J. April, "Simulation optimization: a review, new developments, and applications," in *Simulation Conference, 2005 Proceedings of the Winter*, pp. 13 – 95, Dec 2005.

[2] M. C. Fu, C.-H. Chen, and L. Shi, "Some topics for simulation optimization," in *Proceedings of the 40th Conference on Winter Simulation*, pp. 27–38, Winter Simulation Conference, 2008.

[3] M. Goossens, F. Mittelbach, and A. Samarin, *Advanced Modeling and Optimization of Manufacturing Processes*. Springer Series in Advanced Manufacturing, Springer London, 2011.

[4] S. Melouk, N. Freeman, M. Miller, and M. Dunning, "Simulation optimization-based decision support tool for steel manufacturing," *International Journal of Production Economics*, vol. 141, no. 1, pp. 269 – 276, 2013.

[5] P. Raska and Z. Ulrych, "Simulation optimization in manufacturing systems," *23rd International DAAAM Symposium*, vol. 23, no. 1, pp. 221 – 224, 2012.

[6] J. B. Dabney and T. L. Harman, *Mastering SIMULINK 4*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 2001.

[7] P. Fritzson, *Principles of object-oriented modeling and simulation with Modelica 2.1*. Piscataway, NJ, USA: Wiley-IEEE Press, 2004.

[8] J. Åkesson, M. Gäfvert, and T. Tummescheit, "Jmodelica-an open source platform for optimization of modelica models," in *Proceedings of MATHMOD 2009-6th Vienna International Conference on Mathematical Modelling*, 2009.

[9] D. Brück, H. Elmqvist, S. E. Mattsson, and H. Olsson, "Dymola for multi-engineering modeling and simulation," in *Proceedings of Modelica*, 2002.

[10] J. Hřebíček and M. Řezáč, "Modelling with maple and maplesim," in *Proceedings of the 22nd European Conference on Modelling nad Simulation ECMS*, pp. 60–66, 2008.

[11] R. Fourer, D. Gay, and B. Kernighan, *AMPL: a modeling language for mathematical programming*. Cengage Learning, 2002.

[12] P. Van Hentenryck, *The OPL optimization programming language*. Cambridge, MA, USA: MIT Press, 1999.

[13] R. F. Boisvert, S. E. Howe, and D. K. Kahaner, "Gams: A framework for the management of scientific software," *ACM Transactions on Mathematical Software)*, vol. 11, no. 4, pp. 313–355, 1985.

[14] J. Bisschop and R. Entriken, *AIMMS: The modeling system*. Paragon Decision Technology, 1993.

[15] A. Brodsky, G. Shao, and F. Riddick, "Process analytics formalism for decision guidance in sustainable manufacturing," *Journal of Intelligent Manufacturing*, pp. 1–20, 2014.

[16] A. Brodsky, N. E. Egge, and X. S. Wang, "Supporting agile organizations with a decision guidance query language," *Journal of Management Information Systems*, vol. 28, no. 4, pp. 39–68, 2012.

[17] A. Brodsky, M. M. Bhot, M. Chandrashekar, N. E. Egge, and X. S. Wang, "A decisions query language (DQL): High-level abstraction for mathematical programming over databases," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, (New York, NY, USA), pp. 1059–1062, ACM, 2009.

[18] A. Brodsky, "Constraint databases: Promising technology or just intellectual exercise?," *Constraints J.*, vol. 2, no. 1, pp. 35–44, 1997.

[19] A. Brodsky and V. E. Segal, "The C3 constraint object-oriented database system: An overview," in *Second International Workshop on Constraint Database Systems, Constraint Databases and Their Applications*, (London, UK, UK), pp. 134–159, Springer-Verlag, 1997.

[20] A. Brodsky and H. Nash, "CoJava: Optimization modeling by nondeterministic simulation," in *Principles and Practice of Constraint Programming-CP 2006*, pp. 91–106, Springer, 2006.

[21] A. Brodsky and H. Nash, "CoJava: A unified language for simulation and optimization," in *Principles and Practice of Constraint Programming - CP 2005* (P. van Beek, ed.), vol. 3709 of *Lecture Notes in Computer Science*, pp. 877–877, Springer Berlin Heidelberg, 2005.