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

Toward Reusable Models: System Development for Optimization Analytics Language (OAL)

Abdullah Alrazgan
aalrazga@gmu.edu

Alexander Brodsky
brodsky@gmu.edu

Technical Report GMU-CS-TR-2014-4

Abstract

Decision optimization has been broadly used in many areas including economics, finance, manufacturing, logistics, and engineering. However, optimization modeling used for decision optimization presents two major challenges. First, it requires expertise in operation research and mathematical programming, which most users, including software developers, engineers, business analysts and end-users, typically do not have. Second, optimization models are usually highly customized, not modular, and not extensible. Sustainable Process Analytics Formalism (SPAF) has been recently proposed at National Institute of Standards and Technology (NIST) to address these challenges, by transitioning from the redevelopment of optimization solutions from scratch towards maintaining an extensible library of analytical models, against which declarative optimization queries can be asked. However, the development of practical algorithms and a system for SPAF presents a number of technical challenges, which have not been addressed. In this paper we (1) propose an object-oriented extension of SPAF, named Optimization Analytics Language (OAL); (2) present OAL system development based on methods to reduce OAL models and queries into a formal mathematical programming formulation; (3) showcase OAL through a simple case study in the domain of manufacturing; and (4) conduct a preliminary experimental study to assess the overhead introduced by OAL.

1 Introduction

Making complex decisions is prevalent in various domains including economics, finance, manufacturing, logistics, and engineering. For example, decisions are made to allocate production loads of a product to different machines on the manufacturing floor in order to meet demand at minimum cost, to source raw materials from suppliers on a restricted budget, or to schedule airline crews to begin and end their shifts in the same city to minimize cost. Making such decisions may involve analyzing many complex alternatives that are beyond human capacity to do manually. Furthermore, this often would result in outcomes that are far from optimal.

To support decision making, enterprises turned to Decision Support Systems (DSS) [1] and, more recently, to Decision Guidance Systems (DGS) [2] to help with analyzing complex problems. DSS are information systems that support decision-making by providing useful information, visualization, and trends but may not necessarily suggest actionable recommendations. DGS, on the other hand, are a class of DSS that does provide actionable recommendations. This often involves analyzing streams of data from a variety of sources, building, learning, and using models for prediction and what-if analyses, and performing deterministic or stochastic optimization.

DGS require formal optimization models to be constructed and solved using Mathematical Programming (MP) or Constraint Programming (CP) which have been extensively studied; however, modeling an optimization problem or a system presents two major challenges. First, many potential users, such as process engineers and business analysts, do not have expertise in MP/CP modeling and optimization. Second, optimization models are typically highly customized, not modular, and not extensible.

Moreover, building systems for optimization-based analytics is typically a sequential, non-reusable process, which may involve a wide span of expertise in operation research (OR) and information technology. This process is depicted in Figure 1 (Conventional Approach). It starts with an OR expert modeling it. Second, it would need to implement data collection, manipulation, and integration by IT professionals using appropriate data management tools. IT experts may also develop end-user tools that a business analyst can use. Third, business analysts

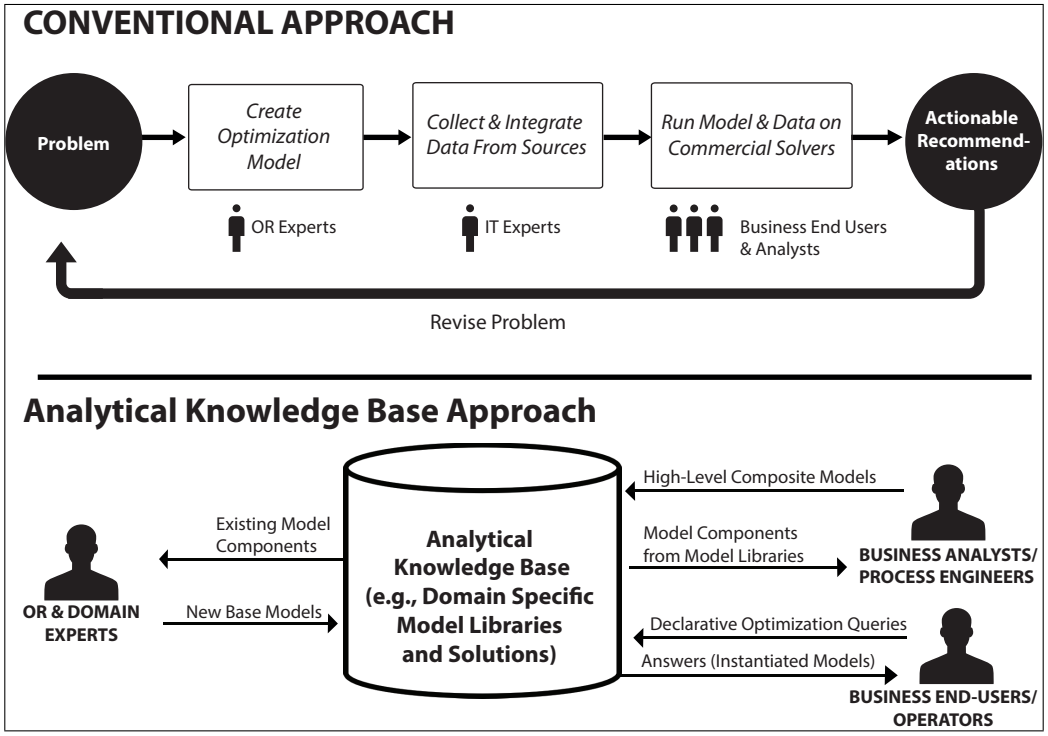

Figure 1: Paradigm Shift in Development of Decision Guidance Solutions

would run the constructed problem model on collected data using a commercial optimization solver to obtain actionable recommendations. Finally, the process may have to be restarted to define a modified problem since the models are highly customized. Because of this rather complex process, an intelligent and easier modeling approach for process engineers and business analysts is needed to make the development of DGS much more agile and flexible.

We address this need, in this paper, by advocating a paradigm shift from the sequential, non-reusable approach to a more modular approach centered around the Analytical Knowledge Base (Figure 1). The Analytical Knowledge Base (AKB) stores models that may not necessarily be optimization models but rather models that describe an underlying physical reality such as machines, devices, or processes. In the new paradigm, OR experts do not need to model every decision problem faced by business users. Instead, OR experts would create a domain-specific component library and store it in the AKB. For example, in the domain of manufacturing, they would create models for processes such as die casting, turning, gas metal arc welding, etc. Using these component models, a business analyst, who may be a process engineer but not an OR expert, would construct a model (possibly using a Graphical User Interface (GUI)) of a manufacturing process in terms of flow of resources and work pieces among the machines on a manufacturing floor. This business analyst can create a library of various composite models, such as those for different types of processes. Finally, a business end-user, who may be a production operator in the manufacturing domain, can ask a variety of declarative queries against a model in the AKB. For example, against a manufacturing process model, the operator can ask: 1) given the settings of the machines (e.g., on/off and speed) and the load distribution among the machines, what is the total power consumption needed to fulfill the demand; or, 2) a decision query, such as which machines should be turned on and off, and how to setup machines' control settings so that the production demand will be satisfied at a minimal cost subject to a limitation on carbon emissions. The new paradigm allows for modular, extensible construction of models mostly by business analysts (except for domain-specific libraries). Furthermore, it provides the ability to easily reuse the models for the construction of new composite models and then answering declarative queries (including what-if analysis and optimization) posed by business end-users against the composite models.

Sustainable Process Analytics Formalism (SPAF) [3] was proposed at the National Institute of Standards and Technology (NIST) to target reusability of optimization models, which is a first step towards the new paradigm. SPAF is comprised of two layers: the base generic *Analytics Language* called AL and an SPAF library with built-in models for processes, flows, sustainability metrics and process composition. However, the SPAF paper did not address the problem of building an SPAF system.

In this paper, we focus on the system development

of the Optimization Analytics Language (OAL), which is a generalized and improved form of SPAF/AL. More specifically, the contributions of this paper are as follows. First, we define the syntax and semantics of OAL by generalizing the AL syntax with clean object-oriented features of modularity, encapsulation, and inheritance. We demonstrate OAL's features through the use of a manufacturing production example. Second, we develop the OAL system based on the compilation of an OAL class library and a declarative query into a formal optimization model expressed in Optimization Programming Language (OPL). This machine-generated OPL model is solved using a commercial optimization solver, namely IBM CPLEX. Finally, we conduct a preliminary experimental study to compare the performance of an OPL machine-generated model by an OAL compiler versus a manually constructed OPL model for the same problem. This very preliminary study shows only a relatively marginal run-time overhead.

The remainder of this paper is organized as follows: In Section 2, we give a brief overall survey of decision optimization and discuss how OAL can bridge the gap. In Section 3, we give an overview of OAL and showcase it via a manufacturing example. In Section 4, we present our reduction algorithm and system implementation architecture of OAL. In Section 5, we give results of a preliminary experimental analysis of OAL vs. manually generated models. We conclude in Section 6, and briefly discuss directions for future work.

2 Related Work

Engineers and business analysts may use a variety of different tools to support decision-making. These tools can be broadly classified into four categories: 1) End-User Domain-Specific; 2) Optimization Modeling; 3) Simulation Modeling; and 4) Black-box Integration.

Domain-Specific tools are designed for specific, limited tasks and typically provide a GUI that is easy to use by end-users. The implementation may use optimization tools and integrate them with other systems such as Enterprise Resource Planning (ERP). Examples include optimizing price-revenue, transportation, sourcing, and production plans, among others (e.g., [4]; [5]). However, this approach is not extensible, which may lead to "silo" optimization and not achieve the global optimum.

Optimization Modeling tools typically use MP or CP algorithms. 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 [6], OPL [7], GAMS [8], or AIMMS [9]. However, as mentioned in the introduction, 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.

Simulation Modeling tools allow engineers and business analysts 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 GUI. Tools like SIMULINK [10] and Modelica-based ones [11] like JModelica [12], Dymola [13], and MapleSim [14] allow users to model complex systems in mechanical, hydraulic, thermal, control, and electrical power. Modelica comes with over 1000 generic model components that can all be reused. However, optimization using simulation modeling tools amounts to a heuristically-guided trial and error approach where simulation serves as a black box. Simulation-based optimization is significantly inferior to MP/CP optimization in terms of optimality or quality of results and computational complexity because it does not utilize the mathematical structure of the underlying problem the way MP/CP approaches do.

Lastly, Black-Box Integration tools are designed for the intersection of simulation tools and domain-specific tools to automate a computational process that requires the use of unified tools or packages. They help integrate an input to one system as an output from another using scripting or direct integration nodes as with many manufacturing applications such as Computer Aided Design (CAD). Engineers and business analysts can use these tools to help arrive at decisions that may use optimization techniques, data-mining, and probabilistic simulation. Among these tools are ModeFrontier [15] and OptiY [16]; while they provide an easy-to-use GUI for the business analyst, they present a similar issue as simulation tools in that they do not provide a way to arrive at a global optimal.

Based on all the decision and analysis tools discussed, an engineer or business analyst may not be able to use specialized OR tools without the help of an OR expert. They may use simulation or black-box integration tools, but arriving at a global optimization through the use of MP and CP methods is missing. Moreover, the current research lacks of a modeling language that is modular, extensible, and can use constructs an engineer or a business analyst would understand. There have been various attempts to tackle this issue; among them are CoJava [17, 18], CoReJava [19], and OptimJ [20], all of which extend Java with optimization constructs; this allows easier modeling for Java developers. Decision Guidance Query Language (DGQL) [21, 22] which, in turn, builds on the work [23, 24, 25], allows seamless integration of optimization on the data manipulation

language SQL. However, while CoJava is a fully object-oriented language, it requires the skills of a software developer experienced in using Java. DGQL, on the other hand, would be relatively easy to use by business analysts but does not support the modular AKB-centric modeling approach. OAL would aim to bridge that gap.

3 Optimization Analytics Language (OAL)

The development of OAL as a modeling tool is to achieve reusability and modularity that a business analyst or an engineer can use and understand. Reusability is achieved through the introduction of an analytical knowledge base that stores basic component models that can be extended from and form a specific process that can be re-saved. Modularity enables models to be instantiated with different data and integrated with other models by increasing coherence and reducing coupling of model components together. The implementation of each model is self-contained; that is, it describes a physical reality such as machines, devices, or processes.

To model decision optimization in OAL, an engineer does not need to formulate the problem as a mathematical model. Alternatively, he may use process-flow notations to describe the model. OAL allows the user to define a generic class or specific types found in SPAF, such as flow, flow aggregator, and process. A class is a general term to describe any specification (i.e., a set of properties or constraints) without any restrictions. A flow is a special type of class that describes the input and output of a specification. A process or sub-process is essentially a class that must have the inputs and outputs in addition to properties or constraints. A flow aggregator aggregates the flows of an output from multiple processes as a single input to another process. Finally, once all the models are defined, the engineer can indicate an optimization query, i.e., an objective function, expressed as a statement or a declarative query to view the data. A formal syntax and semantics has been presented in SPAF [3].

We adopted an equational syntax of OPL for implementation purposes and added semantics of object-oriented features and specific process-flow notations, that would help an engineer understand OAL. OPL is an algebraic modeling language that follows a certain mathematical structure. This structure usually defines the data and decision variables, the objective function, and the constraints. It is a strongly typed language that has various operators including arithmetic, relational, and logical. It also includes data types such as integers, strings, with the addition of piecewise and stepwise functions. OPL also supports built-in certain data structures such as range, arrays, sets, and tuples. OPL can read data defined internally or from an external file, make connections to databases, and retrieve data from spreadsheets. Since OPL is an algebraic modeling language, the "how" of solving the formulated optimization problem is left to an external mathematical solver. In IBM ILOG CPLEX Optimization Studio, OPL can be used with two optimization solvers: CPLEX for MILP or CP solver for constraint programming. The solver can search for a solution using complex mathematical techniques to derive at an optimal solution. OAL supports all the features discussed with the addition of the introduction of component-based models for reusability, changeability, and extensibility. The models can be generic classes, or specific types such as process, flow, and flow aggregator.

3.1 OAL by Example

To explain OAL, we will introduce it through a resource allocation example in a production plant that was proposed at NIST. Depicted in Figure 2, we assume that we have three raw external inputs to the overall production plant that consists of seven processes. A die casting process uses a metal powder as its input, represented as `dieCastingInput`, and produces a raw metal as its output. Once the die casting process produces an output, it is fed into any of the three turning machines. Each machine has a minimum and maximum amount of throughput as well as an associated cost to have the machine running based on the energy used. The die casting output can flow into any of the turning machines. After the turning machines produce a part, it flows into the gas metal arc welding process. The injection molding process uses one of the external inputs of raw plastic, as represented in `injectionMoldingInput`, and produces a plastic piece. The final process is threaded fastening, which takes an external input of a bolt (`threadedFastInput`), the plastic piece from the injection molding, and the metal from the gas metal arc welding to arrive at a finished product represented as `output`. We can query this OAL model to compute data, perform what-if analysis, or solve a decision optimization query. Examples include: What is the maximum throughput of turning machine 2?; how much would it cost to run two machines as opposed to three?; or which optimal allocation will meet demand at minimum energy cost?

To explain how this process is modeled using OAL, assume for now that we already have a library of predefined models that can be organized based on topics and includes the known properties and equations. For example, under a manufacturing library, we have models that consist of die casting, turning machines, welding, injection molding, or threaded fastening as similarly shown in Figure 3. Each model corresponds to a process definition that is independent of or extended from the others. Furthermore, the built-in library has connectors of flows and flow aggregators, each of which have definitions written in OAL. Flows correspond to the arrows depicted in Figure 2 whereas flow aggregators correspond to the black triangles. Given this library, the process

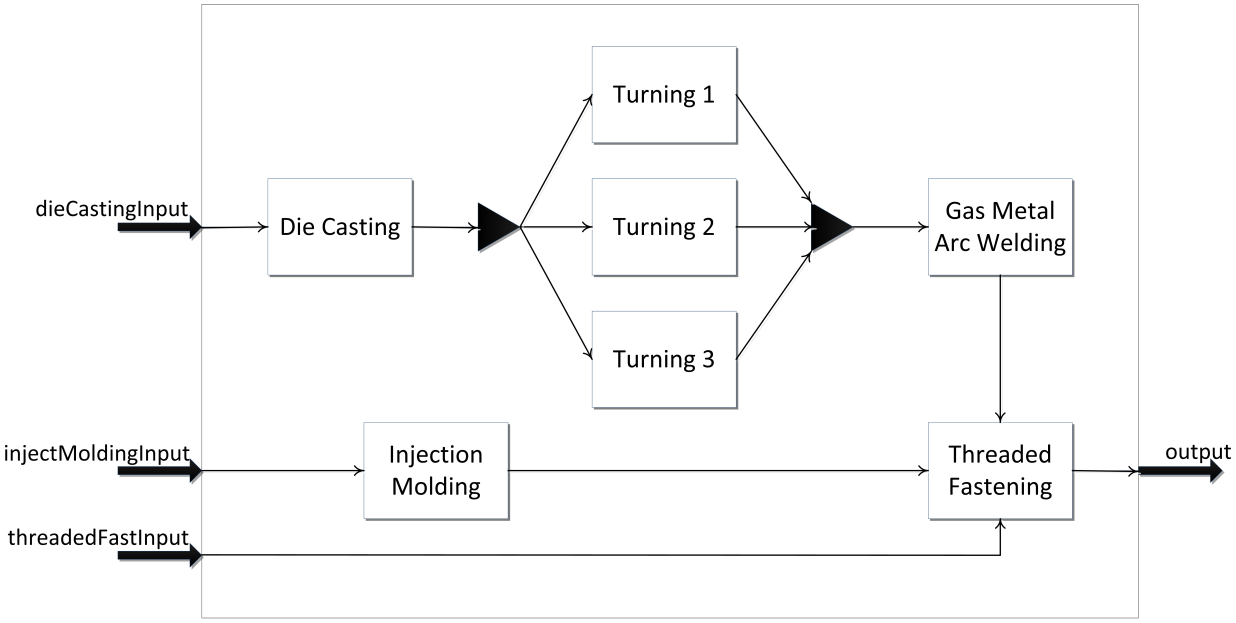

Figure 2: Modeling Approach of OAL

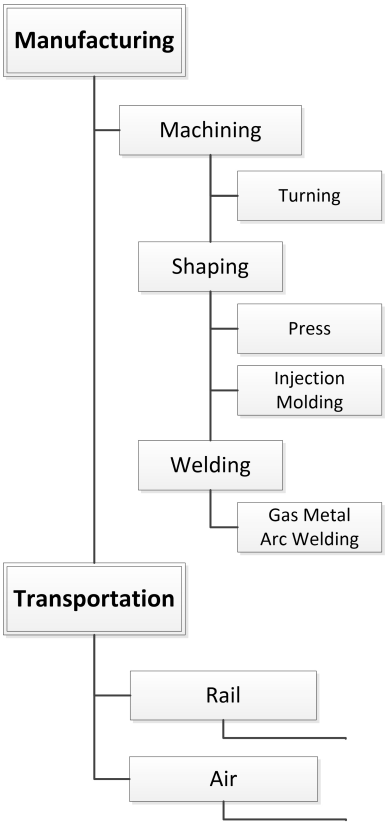

Figure 3: OAL Built-In Libraries

to construct Figure 2 can be easily done through a GUI that would allow a drag-and-drop of processes, flows, and flow aggregators. This type of modeling allows an engineer to easily construct different scenarios within a manufacturing floor without having an in-depth understanding of OR such as the type of constraints, the solver method, the different parameters of each process, or the general mathematical form to construct an optimization query.

Listing 1 mimics Figure 2. While it might be verbose, the general idea is that this process would be automatically generated based on a diagram developed by a user using a GUI. It defines the sub-processes and flow inputs and outputs of the overall process as well as flow aggregators. Finally, it can include metrics that an engineer or other user would be interested in solving.

To drill down on defining a sub-process, Listing 2 references a sample of OAL code that models die casting. Each sub-process must define its input flows and output flows. The process can contain three types of statements: 1) a variable definition, 2) constraint, and 3) a model call or statement. A variable definition can be a declared statement, a constant statement, an expression statement, or an instantiated statement (i.e., three dots). All variable definitions follow similar syntax to OPL and must be defined with a basic type such as string, integers, float, range, range floats, boolean, or any user-defined type through a tuple definition. A declared statement is one that is not equated with a constant or expression but a question mark that signifies that the system will use as part of the decision process of the overall model. A constant or expression statement is basically instantiated with a constant or an expression that is valid. Expression statements include stepwise or piecewise functions such

```
1  class ManufacturingFloor {
2
3  process dieCastingProc = new DieCasting{};
4  process turning1Proc = new Turning1{};
5  process turning2Proc = new Turning2{};
6  process turning3Proc = new Turning3{};
7  process gasMetalArcWeldingProc =
8          new GasMetalArcWelding{};
9  process injectionMoldingProc = new InjectionMolding{};
10 process threadedFasteningProc = new ThrededFastening{};
11
12 {process} subProcesses = {dieCastingProc, turning1Proc,
13         turning2Proc, turning3Proc, gasMetalArcWeldingProc,
14         injectionMoldingProc, threadedFasteningProc};
15
16 // Setup main flows
17 flow dieCastingInput = dieCastingProc.input;
18 flow injectMoldingInput = injectionMoldingProc.input;
19 flow threadedFastInput =
20         threadedFasteningProc.inThreadedBolt;
21 flow output = threadedFasteningProc.output;
22
23 // Setup aggregators
24 flowAggr dieCastingOutAggr = new flowAggr {
25         inputFlows = {dieCastingProc.input};
26         outputFlows = {turning1Proc.input,
27                 turning2Proc.input, turning3Proc.input}
28         };
29
30 flowAggr turningOutAggr = new flowAggr {
31         inputFlows = {turning1Proc.output,
32                 turning2Proc.output, turning3Proc.output};
33         outputFlows = {gasMetalArcWeldingProc.input}
34         };
35
36 float energyTotal = sum(i in subProcesses) i.energyPerHour;
37 }
```

Listing 1: Manufacturing Floor Process

```
1  class DieCasting {
2  {flow} inputFlows = new flow{};
3  flow outputFlow = new flow{};
4
5  float minimumThru = 0.0;
6  float initialEnergy = 50.0;  //kWh
7  float maximumThru = 150;
8
9  range sRange = 1..4;
10 range bRange = 1..3;
11 float slope[sRange] = [0.2, 0.25, 0.30, 0.5];
12 float break[bRange] = [10,60,100];
13
14 pwlFunction energyFunction = piecewise{slope[1] -> break[1];
15         slope[2] -> break[2]; slope[3] -> break[3]; slope[4]}
16         (minimumThru, initialEnergy);
17
18 float thru = outputFlow.unitPerHour;
19 minimumThru <= thru <= maximumThru;
20 float energyPerHour = energyFunction(thru);
21
22 float inputPerOutput[inputFlows] = [5.7];
23
24 forall(i in inputFlows)
25         i.unitPerHour ==
26         outputFlow.unitPerHour * inputPerOutput[i];
27 }
```

Listing 2: Diecasting Process Definition

as the one defined in line 14. The piecewise function is defined into 3 breakpoints and 4 slopes. In Listing 2, the slope begins with a positive slope of 0.2 until the first breakpoint of 10. After the first 10 values, it slopes up to 0.25 until the second breakpoint of 60, and so on. The piecewise starts with an offset of the points `minimumThru` and `initialEnergy`. The last type of variable definition is an instantiated statement, which is defined by using three dots. The three dots indicates that data will be defined at a later point when one does a model call, as we will see later on. Furthermore, the second type of statement is constraints and it can be defined either through the use of "for all" statements or "if" statements or a simple constraint such as the one in line 19. This constraint limits the minimum throughput and maximum throughput. Lastly, a model call begins with the keyword "new" and the name of the model, as in the case of lines 2 and 3. On the other hand, a model statement is typically instantiated with dots statement; it signifies that it will be associated with another model at a later point when one does a model call. One interesting thing to note is that one can refer to other model variables as shown in Listing 2 on line 17 or 24. The syntax uses the dot operator followed by a variable name defined in that specific model.

In Listing 3, it shows the flow model definition with the name `class flow`. Flows represent the connectors between processes. In our example, we are interested in how many units per hour to produce. Flow aggregators consolidate the arrows and may add a constraint as shown in the listing. The constraint in the example is a balancing constraint.

Since most of the processes in this example have the same type of properties and constraints for simplification purposes, OAL allows us to define a model that we can extend or inherit. Listing 4 includes the updated model definition. One can see most of the variable definitions are instantiated with dots. The generic model has multiple inputs and outputs and the parameters are not given. To use this model, one must instantiate a model definition with `extends` as shown in the updated die casting process model — Listing 5 shows how. It is defined as a process instead of a class because we must define an input and output whereas a class can be a generic type. One can define new variables or constraints as shown with `flow input`. This parameter can be passed as value to `baseThruMachine`. Note here that all unde-

```
class flow {
float unitPerHour = ?;
}

class flowAggr {
{flow} inputFlows = ...;
{flow} outputFlows = ...;
sum(i in inputFlows) i.unitPerHour ==
                    sum(j in outputFlows) j.unitPerHour;
}
```

Listing 3: Flow and Flow Aggregator Definition

```
class baseThruMachine {
{flow} inputFlows = ...;
flow outputFlow = ...;

float minimumThru = ...;
float initialEnergy = ...;
float maximumThru = ...;

range sRange = 1..4;
range bRange = 1..3;
float slope[sRange] = ...;
float break[bRange] = ...;
pwlFunction energyFunction = piecewise{slope[1] -> break[1];
        slope[2] -> break[2]; slope[3] -> break[3]; slope[4]}
        (minimumThru, initialEnergy);

float thru = outputFlow.unitPerHour;
minimumThru <= thru <= maximumThru;
float energyPerHour = energyFunction(thru);

float inputPerOutput[inputFlows] = ...;

forall(i in inputFlows)
        i.unitPerHour ==
        outputFlow.unitPerHour * inputPerOutput[i];
}
```

Listing 4: A Base Model That Can Be Extended

```
process DieCasting extends baseThruMachine {
        flow input = new flow{};
        flow output = new flow{};

        inputFlows = {input};
        outputFlow = output;
        minimumThru = 0.0;
        initialEnergy = 50.0;
        maximumThru = 150;
        slope[] = [0.2, 0.25, 0.30, 0.5];
        break[] = [10,60,100];
        inputPerOutput[] = [5.7]
}
```

Listing 5: An Updated Die Casting Process Definition

```
class ManufacturingFloor {
        process dieCastingProc = DieCasting{
                input = flow{ unitPerHour = 116.85};
                output = flow{ unitPerHour = 20.5};
                inputFlows = {input}; outputFlow = output;
                initialEnergy = 50.0;
                maximumThru = 150;
                slope[] = [0.2, 0.25, 0.30, 0.5];
                break[] = [10,60,100];
                thru = 20.5;
                };
        // Same for other processes ...

        // Main flows as is
        flow dieCastingInput = dieCastingProc.input;
        flow injectMoldingInput = injectionMoldingProc.input;
        // ...

        // Aggregators statements as is

        float energyTotal = 371.4;
}
```

Listing 6: Manufacturing Floor Results

fined values of `class baseThruMachine` must be instantiated with data. For example, minimumThru that was already defined in `class baseThruMachine` now has a parameter of 0.0 of the die casting process. The other processes of the example are defined in a similar fashion and follow the same model call with different data inputs.

So far the constructed models do not constitute an optimization or simulation model. They merely capture analytical knowledge about the data or constraints. OAL allows us to do various non-trivial computations about the data or optimization queries. In our manufacturing example, we try to minimize the total energy using a minimize statement. One can also define a maximize statement or a `stat` command to compute a query. This allows users to query the data or find an optimal configuration. Users can also perform what-if analyses to explore various scenarios of the problem. The query or objective statement is always the last statement following all model definitions. Listing 6 shows the results that can be obtained once a minimize statement is executed. The statement is

```
minimize ManufacturingFloor.energyTotal
```

The metrics we are interested in are `unitPerHour` for each process and the total energy. As shown in the listing, the values are passed as part of the process and the statement of `energyTotal` is replaced with an optimal parameter; and hence saved in the analytical knowledge base for future reference.

```
class Metrics {
        {string} names = ...;
        {string} descreteNames = ...;
        float v[names] = ?;
        int dv[descreteNames] = ?;

        forall(n in descreteNames) v[n] == dv[n];
}
```

Listing 7: Base Metric Class

```
class CompositeProcess {
{process} subProcesses = ...;
{flowAggr} flowAggregators = ...;

{string} allMetricNames =
        union(p in subProcesses) p.metrics.names;
{string} allDescreteNames =
        union(p in subProcesses) p.metrics.descreteNames;

float v[n in allMetricNames] = sum(p in subProcesses,
  smn in p.metrics.names : smn == n) p.metrics.v[n];
int dv[n in allDescreteNames] = sum(p in subProcesses,
  smn in p.metrics.descreteNames : smn == n) p.metrics.dv[n];

Metrics m = new Metrics { v = v; dv = dv; };
}
```

Listing 8: Base Composite Process Class

3.2 OAL Library

OAL comes with a built-in library of classes for reuse. This library of classes is built manually and is augmented with more definitions and models as they are being built. In our example, most of the definitions can be considered part of a library for later use with other models. Moreover, there are basic library items that make our example easier to define; among them are metrics and composite processes.

Metrics is a basic class that states what we are interested in minimizing or maximizing. We can compute for competing metrics at each run to enable what-if analysis. Specific metrics can be cost, profit, energy consumption, CO_2 emissions, etc. In our example, we only had one metric, which was energy. Listing 7 lists the basic metric class that can be extended from.

Composite process is the building block that a main process can use to build the overall model. As was shown in Listing 1, manufacturing floor model follows a specific format by defining sub-processes, flows, and flow aggregators, followed by a metric to be computed. Listing 8 shows the basic composite process that the manufacturing process model can extend from. `CompositeProcess` requires that you define the subprocesses and flow aggregators which `ManufacturingFloor` does. The general idea is that composite process would allow easier construction of models using a GUI. Assuming one is in place, the user would only have to construct the GUI skeleton structure similar to Figure 2 and the code would be automatically generated as shown in Listing 9. The overall model is listed in Appendix A.

```
class ManufacturingFloor extends CompositeProcess {
process dieCastingProc = new DieCasting{};
process turning1Proc = new Turning1{};
process turning2Proc = new Turning2{};
process turning3Proc = new Turning3{};
process gasMetalArcWeldingProc = new GasMetalArcWelding{};
process injectionMoldingProc = new InjectionMolding{};
process threadedFasteningProc = new ThrededFastening{};

subProcesses = {dieCastingProc, turning1Proc, turning2Proc,
        turning3Proc, gasMetalArcWeldingProc, injectionMoldingProc,
        threadedFasteningProc};

// Setup main flows
flow dieCastingInput = dieCastingProc.input;
flow injectMoldingInput = injectionMoldingProc.input;
flow threadedFastInput = threadedFasteningProc.inThreadedBolt;
flow output = threadedFasteningProc.output;

// Setup aggregators
flowAggr dieCastingOutAggr = new flowAggr {
        inputFlows = {dieCastingProc.input};
        outputFlows = {turning1Proc.input, turning2Proc.input,
                        turning3Proc.input}
        };

flowAggr turningOutAggr = new flowAggr {
        inputFlows = {turning1Proc.output, turning2Proc.output,
                        turning3Proc.output};
        outputFlows = {gasMetalArcWeldingProc.input}
        };
}
```

Listing 9: Manufacturing Floor Based of CompositeProcess

4 OAL System Development

The process of developing a system for OAL involves a series of steps. Figure 4 gives a general overview of the components that are involved. It is separated into three main phases: Syntax Analysis, Preprocess Analysis and Transformation, and Code Generation. A final phase is to compile and run the translated files using IBM CPLEX solver to perform the query that is presented in the OAL code. Note that we can target different solvers based on the type of problem. This is usually determined by the number of variables, the type of variables (real, binary, integers), the constraints, and whether the objective function is linear, quadratic, or polynomial. OAL would need to do the analysis to determine which solver is the best to use, but this is outside the scope of this paper.

First, the *syntax analysis* phase involves the use of a

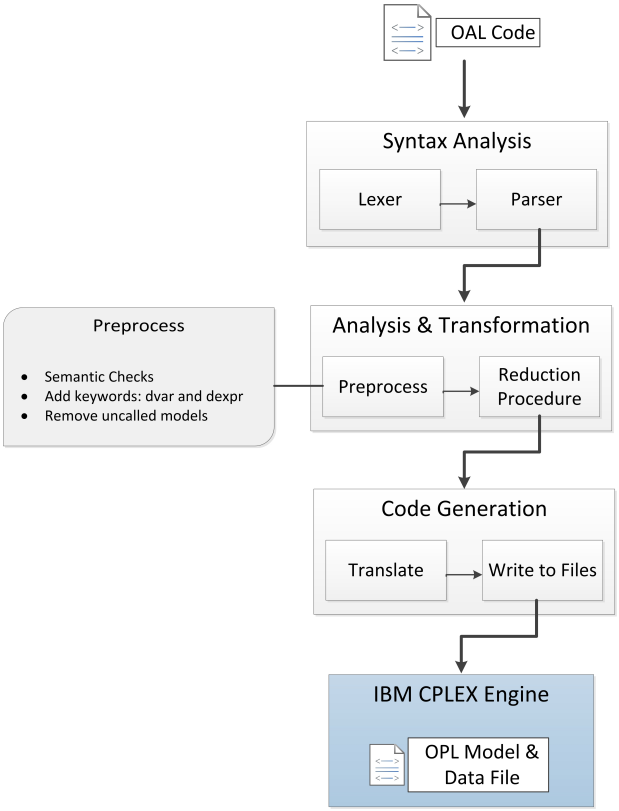

Figure 4: OAL System Structure with Preprocess Procedure

lexer and parser. The lexer, or lexical analysis, converts a sequence of characters into tokens based on the lexical grammar, i.e., rules that define the syntax of the language. Once no errors are reported, it moves to the second step which is the parser. The parser checks if OAL tokens form an allowable expression or statement according to the OAL context-free-grammar. To describe a context-free-grammar, we make use of a language similar to Extended Backus-Naur Form (EBNF). This EBNF would then be translated to a source code of the compiler that provides an abstract syntax tree (AST). The AST is a syntactic structure of the OAL code that will be used for the next phase. If a syntax error is found in the OAL code, it would alert there is a syntactical error.

Secondly, the *pre-process analysis* and *transformation* phase consists of two components: a pre-process and reduction procedure. The reduction procedure involves various transformations to the AST that will be discussed in detail in the next subsection, but first a pre-process procedure must be done to do three checks. First, it does a number of tree traversals of the AST to do semantic checks. Semantic checks include checking for repeated variable names, type checking (the left and right hand of an assignment statement must have compatible types), the "if" conditionals must evaluate to a binary value, and if the instantiated statements (i.e., three dots) are accompanied with model calls that contain data. Second, it adds keywords of decision variables and decision expressions, i.e., `dvar` and `dexpr` to the AST. A decision variable is a variable definition of a declare statement; this means it is assigned to a question mark. A decision expression, on the other hand, is any variable definition statement that includes the decision variable. This is needed to conform to the OPL syntax. Finally, the preprocessing procedure removes any uncalled models to make the model compact.

The last phase is *code generation*, which receives two ASTs from the reduction procedure. The first AST contains the model structure and the other contains the data. The translate component would transform the model structure to follow the structure of all model data and decision variables, the objective function or query statement, then followed by the constraints found in the models. It also checks that all the variable definitions are defined in the data structure of the AST. It would then generate two files, one is a model file and the other is a data file, all of which conform to OPL syntax and semantics. Then, we follow the algorithm presented in Algorithm 1 and obtain a result using a CPLEX solver. It is then presented to the user and replaces the instantiation of variables with the updated solution to be saved in a library.

Algorithm 1 OAL Query Computation

input: A OAL query $M_1, \ldots, M_n; \phi$
output: Query Result
1: Perform Reduction Procedure in Figure 5 to produce model file $(s_1, \ldots, s_n, O, C)$ and data
2: **if** ϕ = `sat` **then**
3: Solve using a MP solver without a valid objective
4: **else**
5: Solve using MP `minimize` or `maximize`
6: **end if**
7: Replace the variable instantiation that are dvars and dexprs with updated solution

4.1 OAL Reduction Procedure to Mathematical Programming

The main component of the OAL system is the reduction component. It introduces various transformations applied to the AST. Since OAL mostly uses OPL syntax, translating most expressions is the same. However, introducing new semantics such as class, process, and flows, brings a few challenges to consider. Assuming the OAL model passes syntactic (i.e., grammar rules) and semantic checks, translating to an OPL model follows the steps defined in Figure 5.

The process is spilt into static and dynamic parts. Static refers to handling the model part while dynamic deals with the data. Each model will embed itself in a string of IDs. This string holds all IDs that this model

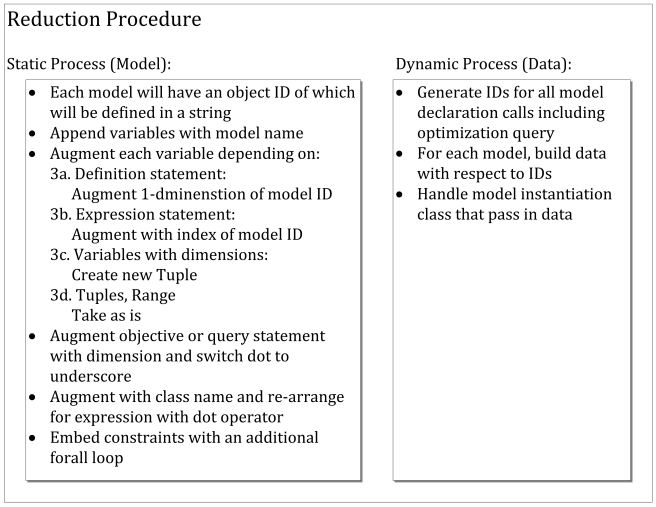

Figure 5: OAL Reduction Steps of Static and Dynamic Process

has called and will be used as a handle for processing. This is analogous to object ID that most languages have behind the scenes. Beginning with each model, all local basic variables (i.e., integers, strings, floats, user-defined) along with the newly created ID will be appended with a model name. This is to prevent variables from having the same name that may appear in other models. This process also includes the model statement of inputs and outputs but will be treated as strings. The compiler then augments each variable with a one-dimension of model IDs. Below is an example of how a simple model is translated to OPL. This is the flow item that was defined in the previous section.

```
class flow {
float unitPerHour = ?;
}
```

OAL

```
{string} flow_IDS = ...;
dvar float flow_unitPerHour[flow_IDS];
```

Translated OPL

In the case of a variable definition that is either a constant statement or an expression statement, we append a special dimension with an index that conforms to OPL syntax. This index traverses through the model ID and we add that dimension of the index to each variable found in the expression statement. In case it is a constant statement (i.e., one with explicit value), we just add the dimension of index in the model ID. Below is how to deal with the case from our previous example.

For variables that have dimensions (i.e., array type), the translating process involves a number of steps. We first have to expand it by utilizing a tuple definition, a data structure that combines multiple basic variables together and can be used to define types. In this case, the tuple will consist of two variables: one is the model ID and the second is the type of dimension it had. Once

```
float energyPerHour =
  energyFunction(thru);
```

OAL

```
dexpr float bETM_energyPerHour[i in bETM_IDS] =
  bETM_energyFunction[i](bETM_thru[i]);
```

Translated OPL

we have a tuple defined, we have to create tuple index so that it may be used for our next step. The final step would be to augment the variable with the dimension of the newly defined tuple index. This will have a further effect when we come into constraints that involve variables that had a dimension. To illustrate, recall that in the base model it had an `inputPerOutput` variable with a dimension of `inputFlows`, and recall that `inputFlows` will be treated as a string since it is a model statement but not a model call. Below is how that OAL statement is translated to OPL.

```
float inputPerOutput[inputFlows] = ...;
```

OAL

```
tuple bETM_inputPerOutput_t {
  string ID;
  string bETM_inputFlows;
};

{bETM_inputPerOutput_t} bETM_index
  = { <id, i> | id in bETM_IDS, i in bETM_inputFlows[id]};

float bETM_inputPerOutput[bETM_index] = ...;
```

Translated OPL

Lastly in terms of different data structures, we have set, range, and tuple definitions. Sets follow the same logic of transformation as basic variable definitions, regardless of whether they are sorted or not. For ranges, whether they are integer or floats, the translation is not needed but is actually taken as defined in OAL. It is a basic data structure that defines the lower and higher bounds to be used as an array indexer. Lastly, tuple definitions are not being transformed and are taken as is because they are defining a data type. Only if they are used as the data type of a variable definition will the compiler apply the variable definition rules.

The static part is now left with objective query, expression using variables that appear in other models through the dot operator, and constraints. The objective statement or query includes the class name followed by a variable that appeared in that class. Since the reduction procedure append all variables with the model name, the objective statement or query involves a simple step

of switching the dot operator with an underscore. It appends with a dimension that includes an actual ID. Generating the ID is part of the dynamic process but the objective query, along with the transformation, is shown here for completeness.

```
1 minimize ManufacturingF.energyTotal;
```

OAL

```
1 minimize ManufacturingF_energyTotal["I−33"];
```

Translated OPL

Secondly, expression using the dot operator involves a series of steps to translate to OPL. This expression consist of two parts other than the dot operator; a local variable followed by the variable that appears in another model. The local variable is the object handle of the model and refers to a model ID. To translate to OPL, the reduction procedure must find what is the class of the object handle. Once that is known, it is a simple step of re-arranging the variable and augmenting it with the class name. It also must be augmented with a dimension that includes the object handle. Below is a an example that showcases this type of transformation to OPL. Recall that `outputFlow` is a type of the class `flow` under class `baseEnergyThruMachine`.

```
1 float thru = outputFlow.unitPerHour;
```

OAL

```
1 float bETM_thru[i in bETM_IDS]
2         = flow_unitPerHour[bETM_outputFlow[i]];
```

Translated OPL

As a last step in the static process, constraints are statements that do not begin with a data type but are comparison statements either in arithmetic, relational, and logical to be evaluated as boolean expression. It can include flow control using "if-else" statements or `forall` statements. For each constraint found in each OAL model, the compiler embeds a `forall` statement traversing through the model IDs. The variables that appear in a constraint statement are augmented with a one-dimension of model IDs. If the variable already has a dimension, it would be augmented with a dimension of tuple that consists of a model ID and the variable that was used. To illustrate the transformation process to OPL, these are two constraints (See Translate 1) that were defined in the manufacturing production example. Since the constraints appeared in the `baseEnergyThruMachine` model, they would be augmented with the IDs from that model. This would complete the static process.

The dynamic process would start by having a duplicate AST of the static one. The dynamic process would do the following three steps for each new model call. First, it begins the process of dynamically generating IDs for each new model call including the objective query. It does this by traversing the tree to reach each model declaration. Once a model call is found, it generates an ID and adds it to the list of model IDs. Second, any data that is provided inside the model would associated with that ID; this includes the basic data types as well as sets and user-generated tuple types. Lastly, it would check if the model call has passed over data. If data was passed over, like in Listing 5, the data takes precedence and is substituted with any data that the model may have. Once this process is done, the dynamic process traverses the static AST and removes any data declarations with an instantiated expression (i.e., three dots). It also adds the ID, if necessary, to the objective query in the static AST. While doing this process, the dynamic AST makes sure it adheres to the correct syntax and semantics that OPL data file has. This concludes the reduction procedure and the process then moves on to code generation stage. While this reduction procedure includes many cases to consider and may seem complex, we found that the compiler transforms the OAL code to OPL with reasonable run-time overhead.

```
1 minimumThru <= thru <= maximumThru;
```

OAL

```
1 forall(id in bETM_IDS)
2   bETM_minThru[id] <=
3   bETM_thru[id] <= bETM_maxThru[id];
```

Translated OPL

```
1 forall(i in inputFlows)
2   i.unitPerHour ==
3   outputFlow.unitPerHour *
4   inputPerOutput[i];
```

OAL

```
1 forall(id in bETM_IDS)
2   forall(i in bETM_inputFlow[id])
3     flow_unitPerHour[i] ==
4     flow_unitPerHour[bETM_outputFlow[id]] *
5     bETM_inputPerOutput[<id, i>];
```

Translated OPL

Translate 1: Two Constraint Transformation from OAL to OPL

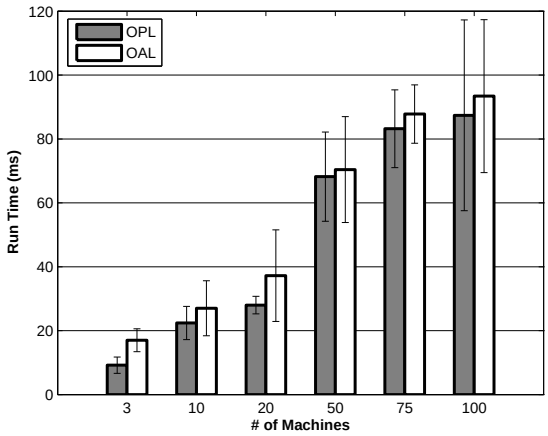

Figure 6: OAL vs. OPL Run Time from 3 to 100 Machines

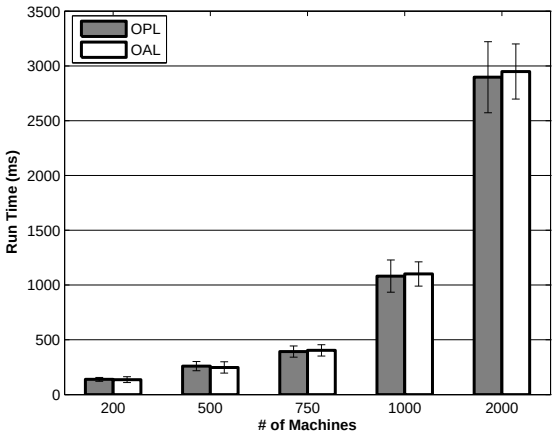

Figure 7: OAL vs. OPL Run Time from 200 to 2000 Machines

5 OAL Experimental Performance

OAL has the potential to achieve an intuitive, high-level abstraction for modeling complex systems to arrive at optimal decisions by engineers or business analysts. The running time of arriving at a solution is as equally important as modeling a problem. To model a mathematical optimization problem, it is usually sensitive to how it is represented; modeling it poorly may result in longer time to find an optimal solution. While OAL is heavily based on OPL, OAL introduces new variables and tuples as discussed in our reduction procedure which may have an effect on the running time. To test the overhead that may be caused by OAL, we did an initial experimental study by comparing the manufacturing process model that was written in OAL and a manually constructed OPL code of the same model, as shown in Listing 13 under Appendix C.

The OAL and OPL model were both ran using IBM ILOG CPLEX Optimization Studio 12.4 using Java Concert API. The specification of the system that was used to conduct the experiment was a workstation running Windows 8.1 on a 2.6GHz Core i5 "Ivy Bridge" processor with 16GB of RAM. We parametrize the problem based on the number of machines and measure the running time in finding a solution for the manually constructed OPL and the OPL by an OAL compiler. The running time only includes the CPU time to find an optimal solution and does not include loading from a disk or compiler time. We randomized the parameters and ran the models ten times each time as we increase the number of machines. We only accounted for finding feasible solutions and ignored results that were infeasible as they were found in under 10 milliseconds. The result is shown in Figure 6 and 7.

As can been seen from the figure, OPL was faster on average in arriving at an optimal solution when the number of machines is less than 200. As we increase that number we see the relation is almost on bar. For machines less than 200, we see that OAL is slower by a factor of 16% but all solutions were under a second. This could be related to introducing tuples as opposed to multi-dimensional arrays. Although the amount of code generated by OAL was greater than manually constructed OPL, they both had the same amount of running time as the number of machines increases to more than 200. This could be attributed to the compiler optimization by CPLEX solver to the resulting OAL code.

While this experiment is limited, it gave a glimpse of OAL overhead and how closely it is related to OPL models. It will be interesting to test on different classes of problems than a classic resource allocation problem such as scheduling and very large scale combinatorial problems. Future work will investigate further with different models to compare against and give a better overall picture than this preliminary study.

In addition, these results do not include the compile

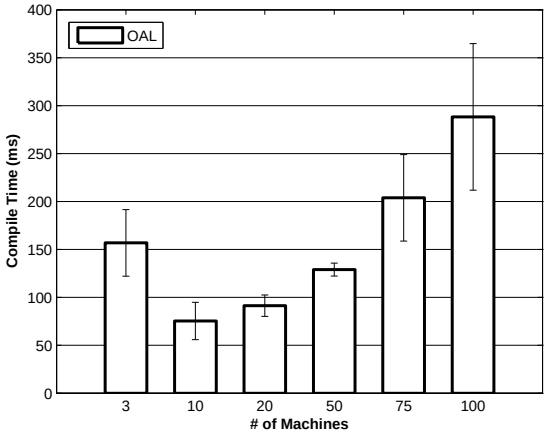

Figure 8: OAL Compile Time for 3 to 100 Machines

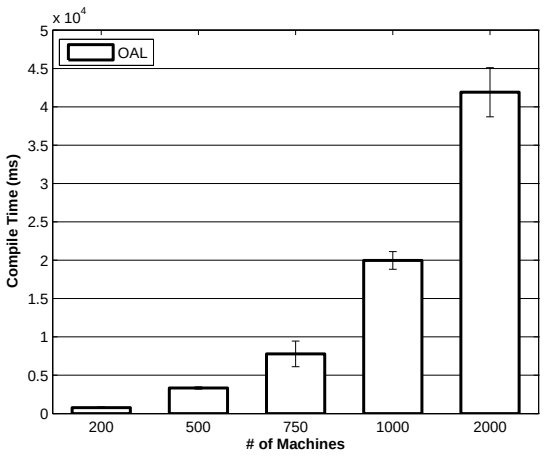

Figure 9: OAL Compile Time for 200 to 2000 Machines

time and this is heavily dependent upon the details of the implementation. Figure 8 shows how fast the compiler for up to 100 machines but Figure 9 shows a longer compilation time as the number of machines increase. The current compiler needs to be optimized for handling large amounts of data and that will be the focus for future implementation.

6 Conclusion & Future Work

In this paper, we have developed an Optimization Analytics Language that targets reusable models. We refined the syntax and semantics and presented a manufacturing production example using OAL. The language's main goal is to provide easier semantics for engineers and business analysts to easily model mathematical optimization. We focused on the system development of the language and presented techniques to reduce OAL models and queries into formal mathematical programming problems. Finally, we conducted an initial experimental study to demonstrate that the language's run time when compared with a manually constructed mathematical model.

We consider this language to be the back-end for what we believe could be part of our future work. Developing an easy-to-use application as a front layer would be highly beneficial for an engineer to easily construct decision optimization using a drag-and-drop to model an overall process. It would be interesting to showcase a real world problem that an engineer can easily build using this graphical user interface with OAL as the system layer.

OAL currently relies on using a mixed integer linear programming solver to run the optimization. More specifically, it uses CPLEX as its current solver. The system has the potential to introduce behind the scenes techniques to determine which solver is best for solving the problem at hand. Moreover, it could introduce algorithms that can help optimize and speed-up processing time.

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied. Certain commercial software systems are identified in this paper to facilitate understanding. Such identification does not imply that these software systems are necessarily the best available for the purpose.

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," *Decision support systems*, vol. 33, no. 2, pp. 111–126, 2002.

[2] A. Brodsky and X. Wang, "Decision-guidance management systems (dgms): Seamless integration of data acquisition, learning, prediction and optimization," in *Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, pp. 71–81, 2008.

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

[4] T. Sandholm, D. Levine, M. Concordia, P. Martyn, R. Hughes, J. Jacobs, and D. Begg, "Changing the game in strategic sourcing at procter & gamble: Expressive competition enabled by optimization," *Interfaces*, vol. 36, no. 1, pp. 55–68, 2006.

[5] M. Levy, D. Grewal, P. K. Kopalle, and J. D. Hess, "Emerging trends in retail pricing practice: implications for research," *Journal of Retailing*, vol. 80, no. 3, pp. xiii–xxi, 2004.

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

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

[8] 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.

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

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

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

[12] 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.

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

[14] 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.

[15] ModeFrontier, "3.1.1," *Documentation provided by ESTECO*, 2011.

[16] T. Pham, "Optiy software and documentation version 2.3," *OptiY, www.optiy.de*, 2007.

[17] A. Brodsky and H. Nash, "Cojava: a unified language for simulation and optimization," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 194–195, ACM, 2005.

[18] 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.

[19] A. Brodsky, J. Luo, and H. Nash, "Corejava: learning functions expressed as object-oriented programs," in *Machine Learning and Applications, 2008. ICMLA'08. Seventh International Conference on*, pp. 368–375, IEEE, 2008.

[20] P. Viry, "Object-Oriented Modeling with OptimJ," Feb. 2008.

[21] 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*, pp. 1059–1062, ACM, 2009.

[22] 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.

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

[24] A. Brodsky, L. Kerschberg, and S. Varas, "Resource management in agent-based distributed environments," in *Cooperative Information Agents III*, pp. 61–85, Springer, 1999.

[25] A. Brodsky and V. Segal, "The c 3 constraint object-oriented database system: An overview," in *Constraint Databases and Applications (CDB'97 and CP'96 Constraints and Databases)*, pp. 134–159, Springer Berlin/Heidelberg LNCS, 1996.

Appendix A

```
class Metrics {
        {string} names = ...;
        {string} descreteNames = ...;
        float v[names] = ?;
        int dv[descreteNames] = ?;

        forall(n in descreteNames) v[n] == dv[n];
}

class CompositeProcess {
        {process} subProcesses = ...;
        {flowAggr} flowAggregators = ...;

        {string} allMetricNames = union(p in subProcesses) p.metrics.names;
        {string} allDescreteNames = union(p in subProcesses) p.metrics.descreteNames;

        float v[n in allMetricNames] = sum(p in subProcesses, smn in p.metrics.names : smn == n) p.metrics.v[n];
        int dv[n in allDescreteNames] = sum(p in subProcesses, smn in p.metrics.descreteNames : smn == n) p.metrics.dv[n];

        Metrics m = new Metrics { v = v; dv = dv; };
}

class flow {
        float unitPerHour = ?;
}

class flowAggr {
        {flow} inputFlows = ...;
        {flow} outputFlows = ...;
        sum(i in inputFlows) i.unitPerHour == sum(j in outputFlows) j.unitPerHour;
}

class baseThruMachine {
        {flow} inputFlows = ...;
        flow outputFlow = ...;

        float minimumThru = ...;
        float initialEnergy = ...;
        float maximumThru = ...;

        range sRange = 1..4;
        range bRange = 1..3;
        float slope[sRange] = ...;
        float break[bRange] = ...;
        pwlFunction energyFunction = piecewise{slope[1] -> break[1]; slope[2] -> break[2];
                                        slope[3] -> break[3]; slope[4]}  (minimumThru, initialEnergy);

        float thru = outputFlow.unitPerHour;
        minimumThru <= thru <= maximumThru;
        float energyPerHour = energyFunction(thru);
        float inputPerOutput[inputFlows] = ...;

        forall(i in inputFlows)
                i.unitPerHour == outputFlow.unitPerHour * inputPerOutput[i];

        Metrics metrics = new mscMetrics{Names = {"energyPerHour"}; DescreteNames ={}};
        metrics.v["energyPerHour"] == energyPerHour;
}
```

Listing 10: Generic Library

```
process DieCasting extends baseThruMachine {
        flow input = new flow{};
        flow output = new flow{};

        inputFlows = {input};
        outputFlow = output;
        minimumThru = 0.0;
        initialEnergy = 50.0;
        maximumThru = 150;
        slope[] = [0.2, 0.25, 0.30, 0.5];
        break[] = [10,60,100];
        inputPerOutput[] = [5.7]
}

process Turning1 extends baseThruMachine {
        flow input = new flow{};
        flow output = new flow{};

        inputFlows = {input};
        outputFlow = output;
        minimumThru = 0.0;
        initialEnergy = 50.0;
        maximumThru = 150;
        slope[] = [0.2, 0.25, 0.30, 0.5];
        break[] = [10,60,100];
        inputPerOutput[] = [1]
}


process Turning2 extends baseThruMachine {
        flow input = new flow{};
        flow output = new flow{};

        inputFlows = {input};
        outputFlow = output;
        minimumThru = 0.0;
        initialEnergy = 50.0;
        maximumThru = 3;
        slope[] = [0.15, 0.2, 0.25, 0.5];
        break[] = [10,60,100];
        inputPerOutput[] = [1]
}


process Turning3 extends baseThruMachine {
        flow input = new flow{};
        flow output = new flow{};

        inputFlows = {input};
        outputFlow = output;
        minimumThru = 0.0;
        initialEnergy = 50.0;
        maximumThru = 10;
        slope[] = [0.1, 0.11, 0.12, 0.5];
        break[] = [10,60,100];
        inputPerOutput[] = [1]
}

process GasMetalArcWelding extends baseThruMachine {
        flow input = new flow{};
        flow output = new flow{};
```

```
62
63          inputFlows = {input};
64          outputFlow = output;
65          minimumThru = 0.0;
66          initialEnergy = 50.0;
67          maximumThru = 150;
68          slope[] = [0.2, 0.25, 0.30, 0.5];
69          break[] = [10,60,100];
70          inputPerOutput[] = [1]
71  }
72
73  process InjectionMolding extends baseThruMachine {
74          flow input = new flow{};
75          flow output = new flow{};
76
77          inputFlows = {input};
78          outputFlow = output;
79          minimumThru = 0.0;
80          initialEnergy = 50.0;
81          maximumThru = 150;
82          slope[] = [0.2, 0.25, 0.30, 0.5];
83          break[] = [10,60,100];
84          inputPerOutput[] = [1]
85  }
86
87  process ThrededFastening extends baseThruMachine {
88          flow inPlasticPart = new flow{};
89          flow inWeldPart = new flow{};
90          flow inThreadedBolt = new flow{};
91          flow output = new flow{};
92
93          inputFlows = {inPlasticPart, inWeldPart, inThreadedBolt};
94          outputFlow = output;
95          minimumThru = 0.0;
96          initialEnergy = 50.0;
97          maximumThru = 150;
98          slope[] = [0.2, 0.25, 0.30, 0.5];
99          break[] = [11,60,100];
100         inputPerOutput[] = [1, 1, 1]
101 }
102
103 class ManufacturingFloor {
104
105         process dieCastingProc = new DieCasting{};
106         process turning1Proc = new Turning1{};
107         process turning2Proc = new Turning2{};
108         process turning3Proc = new Turning3{};
109         process gasMetalArcWeldingProc = new GasMetalArcWelding{};
110         process injectionMoldingProc = new InjectionMolding{};
111         process threadedFasteningProc = new ThrededFastening{};
112
113         {process} subProcesses = {dieCastingProc, turning1Proc, turning2Proc,
114                                   turning3Proc, gasMetalArcWeldingProc, injectionMoldingProc,
115                                   threadedFasteningProc};
116
117         // Setup main flows
118         flow dieCastingInput = dieCastingProc.input;
119         flow injectMoldingInput = injectionMoldingProc.input;
120         flow threadedFastInput = threadedFasteningProc.inThreadedBolt;
121         flow output = threadedFasteningProc.output;
122
123         // Setup aggregators
```

```
124         flowAggr dieCastingOutAggr = new flowAggr {
125                         inputFlows = {dieCastingProc.input};
126                         outputFlows = {turning1Proc.input, turning2Proc.input, turning3Proc.input}
127                         };
128
129         flowAggr turningOutAggr = new flowAggr {
130                         inputFlows = {turning1Proc.output, turning2Proc.output, turning3Proc.output};
131                         outputFlows = {gasMetalArcWeldingProc.input}
132                         };
133 }
```

Listing 11: Domain Specific Library

Appendix B[1]

```
1  {string} flow_IDS = ...;
2  dvar float flow_unitPerHour[flow_IDS];
3
4
5  {string} flowAggr_IDS = ...;
6  {string} flowAggr_inputFlows[flowAggr_IDS] = ...;
7  {string} flowAggr_outputFlows[flowAggr_IDS] = ...;
8
9  // Changed from baseEnergyThruMachine to bETM
10 {string} bETM_IDS = ...;
11 {string} bETM_inputFlows[bETM_IDS] = ...;
12 string bETM_outputFlow[bETM_IDS] = ...;
13 float bETM_minimumThru[bETM_IDS] = ...;
14 float bETM_initialEnergy[bETM_IDS] = ...;
15 float bETM_maximumThru[bETM_IDS] = ...;
16 range bETM_sRange = 1..4;
17 range bETM_bRange = 1..3;
18
19 tuple bETM_slope_tuple {
20   string ID;
21   int bETM_sRange;
22 };
23 {bETM_slope_tuple} bETM_slope_index = {<id, i> | id in bETM_IDS, i in bETM_sRange};
24 float bETM_slope[bETM_slope_index] = ...;
25
26 tuple bETM_break_tuple {
27   string ID;
28   int bETM_bRange;
29 };
30 {bETM_break_tuple} bETM_break_index = {<id, i> | id in bETM_IDS, i in bETM_bRange};
31 float bETM_break[bETM_break_index] = ...;
32
33 pwlFunction bETM_energyFunction[i in bETM_IDS] =
34                     piecewise{bETM_slope[<i, 1>] -> bETM_break[<i, 1>];
35                               bETM_slope[<i, 2>] -> bETM_break[<i, 2>];
36                               bETM_slope[<i, 3>] -> bETM_break[<i, 3>];
37                               bETM_slope[<i, 4>]}
38                              (bETM_minimumThru[i], bETM_initialEnergy[i]);
39
40
41 dexpr float bETM_thru[i in bETM_IDS] = flow_unitPerHour[bETM_outputFlow[i]];
42 dexpr float bETM_energyPerHour[i in bETM_IDS] = bETM_energyFunction[i](bETM_thru[i]);
43
```

[1] Based on original example with no CompositeProcess or Metrics model

```
44
45 tuple bETM_inputPerOutput_tuple {
46   string ID;
47   string bETM_inputFlows;
48 };
49 {bETM_inputPerOutput_tuple} bETM_inputPerOutput_index =
50                             {<id, i> | id in bETM_IDS, i in bETM_inputFlows[id]};
51 float bETM_inputPerOutput[bETM_inputPerOutput_index] = ...;
52
53
54 {string} DieCasting_IDS = ...;
55 string DieCasting_input[DieCasting_IDS] = ...;
56 string DieCasting_output[DieCasting_IDS] = ...;
57
58 // Same for other processes ...
59
60 // Changed from ManufacturingFloor to MF
61 {string} MF_IDS = ...;
62 string MF_dieCastingProc[MF_IDS] = ...;
63 string MF_turning1Proc[MF_IDS] = ...;
64 string MF_turning2Proc[MF_IDS] = ...;
65 string MF_turning3Proc[MF_IDS] = ...;
66 string MF_turning3Proc[MF_IDS] = ...;
67 string MF_gasMetalArcWeldingProc[MF_IDS] = ...;
68 string MF_injectionMoldingProc[MF_IDS] = ...;
69 string MF_threadedFasteningProc[MF_IDS] = ...;
70 {string} MF_subProcesses[MF_IDS] = ...;
71
72 string MF_dieCastingInput[i in MF_IDS] = DieCasting_input[MF_dieCastingProc[i]];
73 string MF_injectMoldingInput[i in MF_IDS] = InjectionMolding_input[MF_injectionMoldingProc[i]];
74 string MF_threadedFastInput[i in MF_IDS] = ThrededFastening_inThreadedBolt[MF_threadedFasteningProc[i]];
75 string MF_output[i in MF_IDS] = ThrededFastening_output[MF_threadedFasteningProc[i]];
76
77 string MF_dieCastingOutAggr[MF_IDS] = ...;
78 string MF_turningOutAggr[MF_IDS] = ...;
79
80 dexpr float MF_energyTotal[i in MF_IDS] = sum(s in MF_subProcesses[i]) bETM_energyPerHour[s];
81
82 minimize MF_energyTotal["P-01"];
83
84 subject to {
85   forall(id in flowAggr_IDS)
86         sum(i in flowAggr_inputFlows[id])
87                 flow_unitPerHour[i] ==
88         sum(j in flowAggr_outputFlows[id])
89                 flow_unitPerHour[j];
90 };
91
92 subject to {
93   forall(id in bETM_IDS)
94         bETM_minimumThru[id] <= bETM_thru[id] <= bETM_maximumThru[id];
95 };
96
97 subject to {
98   forall(id in bETM_IDS)
99     forall(i in bETM_inputFlows[id])
100       flow_unitPerHour[i] == flow_unitPerHour[bETM_outputFlow[id]] * bETM_inputPerOutput[<id, i>];
101 };
```

Listing 12: Sample of Machine Generated Manufacturing Production Model in OPL

Appendix C

```
using CPLEX;

{string} machines = ...;

float minThru[machines] = ...;
float initEnergy[machines] = ...;
float maxThru[machines] = ...;

float s[machines][1..4] = ...;
float b[machines][1..3] = ...;

pwlFunction energyFunction[m in machines] = piecewise{s[m][1] -> b[m][1];
                    s[m][2] -> b[m][2]; s[m][3] -> b[m][3]; s[m][4]} (minThru[m], initEnergy[m]);

dvar float thru[machines];
dexpr float energyPerHour[m in machines] = energyFunction[m](thru[m]);
dexpr float energyTotal = sum(m in machines) energyPerHour[m];

{string} inputs[machines] = ...;
{string} outputs[machines] = ...;

{string} items = union(i in machines) inputs[i] union union(m in machines) outputs[m];
dvar float unitPerHour[items];
float inputPerOutput[machines] = ...;

minimize energyTotal;

subject to {
  forall(m in machines) {
    minThru[m] <= thru[m] <= maxThru[m];
  }

  forall(m in machines)
        forall(o in outputs[m]) {
                thru[m] == unitPerHour[o];
   }

  forall(m in machines)
    forall(i in inputs[m])
        forall(o in outputs[m]) {
          unitPerHour[i] == unitPerHour[o] * inputPerOutput[m];
        }
}
```

Listing 13: Manually Constructed Manufacturing Production Model in OPL