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 Decision Guidance Management Systems: Analytical Language and Knowledge Base

Alexander Brodsky brodsky@gmu.edu
Juan Luo jluo2@gmu.edu
M. Omar Nachawati mnachawa@masonlive.gmu.edu

Technical Report GMU-CS-TR-2016-1

Abstract

Decision guidance systems are a class of decision support systems that are geared toward producing actionable recommendations, typically based on formal analytical models and techniques. This paper proposes the Decision Guidance Analytics Language (DGAL) for easy iterative development of decision guidance systems. DGAL allows the creation of modular, reusable and composable models that are stored in the analytical knowledge base independently of the tasks and tools that use them. Based on these unified models, DGAL supports declarative queries of (1) data manipulation and computation, (2) what-if prediction analysis, (3) deterministic and stochastic decision optimization, and (4) machine learning, all through formal reduction to specialized models and tools, and in the presence of uncertainty.

1. Introduction

Decision Support Systems (DSS) are widely used to support organizational and personal decision-making in diverse areas such as engineering systems, finance, business, economics and public policy. They are becoming increasingly critical with the information overload from the Internet. While the scope of DSS is broad, we view Decision Guidance Systems (DGS) as a class of DSS geared to elicit knowledge from domain experts and provide actionable recommendations to human decision-makers, with the goal of arriving at the best possible course of action. To this end, DGS may need to:

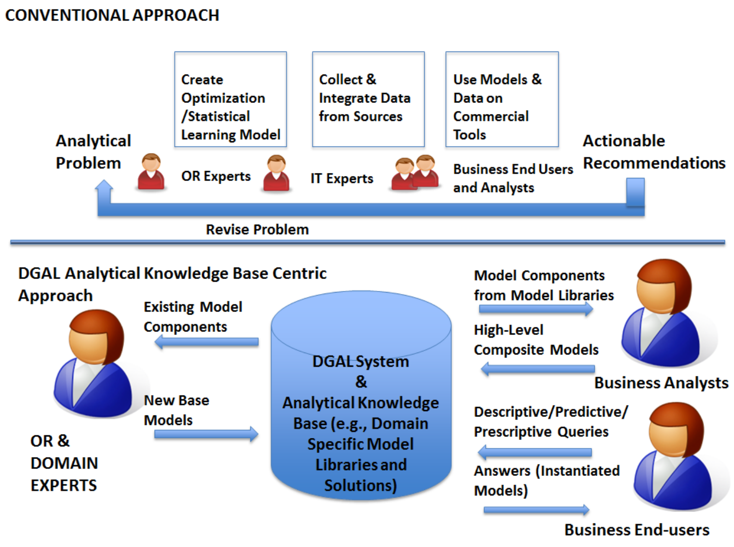
- Use and mine large amounts of data collected from multiple sources;
- Elicit knowledge about the underlying model structure from domain experts;
- Learn deterministic or stochastic models from historical data;
- Elicit metrics, performance indicators and decision objectives from decision-makers;
- Perform analytical tasks, including what-if prediction analysis and (deterministic or stochastic) optimization under diverse constraints, e.g., originating from business or engineering considerations and laws of nature;
- Present and explain actionable recommendations to decision-makers; and
- Solicit decision-makers feedback for iterative improvement.

Consider an example of DGS for procurement and sourcing, e.g., for a manufacturing facility. Given databases and sources on possible suppliers, a procurement officer needs to monitor, make and execute procurement decisions, e.g., which items in what quantities and from which suppliers should be procured, as to satisfy business constraints (production schedule, risk mitigation, inventory capacity etc.) and minimize the total cost. For example, the procurement officer may want to monitor the status of orders, inventories, schedules, and DGS will need to generate different aggregated views of this information, continuously, over time. These activities resemble those of database management systems, dealing with data manipulation and transformation of data (especially temporal sequences) from multiple sources. When given the current inventory and orders status, as well as uncertainty in future pricing and supply, the procurement officer may want to estimate the level of inventories and financial status over the next month and identify risks. These decisions involve the techniques of stochastic simulation and statistical learning for regression, classification and estimation [1]. Prediction and estimation of uncertain outcomes, in turn, may involve regression analysis of functions [2] such as for cost, time, risk, or building classifiers for different categories of outcomes (e.g., normal operation, schedule delay, and financial default). The procurement officer may also ask for a recommendation on procurement, e.g., which items in what quantities from which suppliers and at what time should be purchased, as to satisfy business rules and constraints and minimize the total procurement cost. These decision-makings involves optimization and sensitivity analysis [3], which would typically correspond to a deterministic or stochastic optimization problem, possibly using multiple criteria and under various business assumptions. The goal of the Decision Guidance Analytics Language (DGAL) and Analytical Knowledge Base (AKB), which we propose in this paper, is to support easy development and reuse of models of the descriptive, predictive and prescriptive analytical tasks in decision guidance systems.

Until now, the practice of building DG applications has resembled the practice of developing database applications decades ago before the invention of the relational DBMS. It typically follow the “linear” style described in the conventional approach of Figure 1. DG applications are typically one-off and hard-wired to specific problems; require significant interdisciplinary expertise to build; are highly complex and costly; and are not extensible, modifiable, or reusable.

We believe that these deficiencies originate mainly from the diversity of the required computational tools and algorithms, each designed for a different task (such as data manipulation, predictive what-if analysis, decision optimization, statistical learning and data mining). The computational tools require the use of diverse mathematical abstractions/languages to construct input models (e.g., languages such as OPL, AMPL and GAMS for modeling mathematical programming and constraint programming problems).

Figure 1. Conventional Approach vs. DGAL Approach


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

This introduces two major issues. First, the same underlying reality must often be modelled multiple times using different mathematical abstractions for different tasks/tools, instead of being modeled only once, uniformly. Second, the modeling expertise required by these abstractions/languages is typically not within the realm of DG users – neither domain-specific users (e.g., business professionals) nor DB application and software developers (who may be used to SQL-like languages and OO programming languages such as Java). This, in turn, leads to long-duration, expensive and non-reusable development of DG applications, which must involve a team with diverse interdisciplinary expertise.

We further discuss related work and its limitations in Section 2.

Overcoming the outlined limitations of decision guidance modeling is the focus of this paper. More specifically, the contributions of this paper are as follows. First, we introduce the concept of and the design principles for the Decision Guidance Analytics Language (DGAL) and the Analytical Knowledge-Base (AKB). The key idea is a paradigm shift from the non-reusable task-centric modeling approach to reusable-AKB-centric approach (see lower part of Figure 1). In the latter approach, modular reusable and composable models, which we call Analytical Models (AMs), are created and stored in the KB independently of the tasks and the tools that may use these models. Second, we provide formal definition of syntax & semantics of DGAL. Third, we discuss a case study to show how we apply a DGAL-based methodology to create (graphical) domain specific languages for anaytical end-users. We describe a high-level implementation architecture for DGAL.

The paper is organized as follows. In Section 2 we discuss the related work and also its limitations. Section 3 presents the conceptual architecture and desired properties of DGAL. Section 4 overviews JSON and JSONiq for descriptive analytics, with its computational DGAL extensions. Section 5 describes the construction and of Analytical Knowledge and explains how computation, optimization and learning queries can be posed against AMs and their semantics. Section 6 describes modeling of uncertainty in AMs and the corresponding declarative query operators. Section 7 describes the DGAL Knowledge Base. Section 8 describes a supply chain case study represented by DGAL knowledge base. Section 9 concludes, provides more details on related work, and outlines future research questions.

2. Related Work

To understand the current state of the art and its limitations, consider the six classes of tools/languages relevant to modelling analytical tasks in decision guidance systems: (1) closed domain-specific end-user oriented tools, e.g., strategic sourcing optimization modules within procurement applications [4]; (2) data manipulation languages, such as SQL, XQuery [5] and JSONiq [6]; (3) simulation modelling languages, such as Modelica [7]; (4) simulation languages, such as Jmodelica and Simulink [8][9]; (5) optimization modelling languages, such as AMPL [10], GAMS

[11], and OPL [12] for Mathematical Programming (MP) and Constraint Programming (CP); and (6) statistical learning languages/interfaces, such as PMML [13]. Domain specific tools may be easy to use for a particular well-defined task, but are not extensible to reflect the diversity of emerging descriptive, predictive and prescriptive analytical tasks. Nor do they support ality, i.e., the ability to compose their (white-box) models to achieve global (system-wide) optimal predictions and/or prescriptions (e.g., actionable recommendations), rather than local (silo) optimal predictions and prescriptions. Data manipulation languages, obviously, do not support predictive what-if analysis, optimization or statistical learning.

Simulation languages and tools have the advantage of their modelling expressivity, flexibility, and OO modularity, which support reusability and interoperability of (black-box) simulation models. However, performing optimization using simulation models/tools is based on (heuristically-guided) trial and error. Because of that, simulation-based optimization is significantly inferior, in terms of optimality of results and computational complexity, to MP and CP tools/algorithms, for problems expressible in supported analytical forms (e.g., MILP) [14]. Also, while sufficiently expressive, simulation languages were not designed for declarative and easy data manipulation provided by data manipulation languages such as SQL, XQuery and JSONiq.

Because optimization modelling languages such as AMPL [10], GAMS [11] or OPL [12] are used with MP and CP solvers, which use a range of sophisticated algorithms that leverage the mathematical structure of optimization problems, they significantly outperform simulation-based optimization, in terms of optimality and running time. However, optimization modelling languages are not modular, extensible, reusable or support ality; nor do they support low-level granularity of simulation models (which is expressed through OO programs.) Statistical learning languages/tools have similar limitations and advantages, because most are based on optimization. Like the simulation tools, optimization and statistical learning tools were not designed for easy data manipulation, compared to data manipulation languages such as SQL, XQuery, and JSONiq.

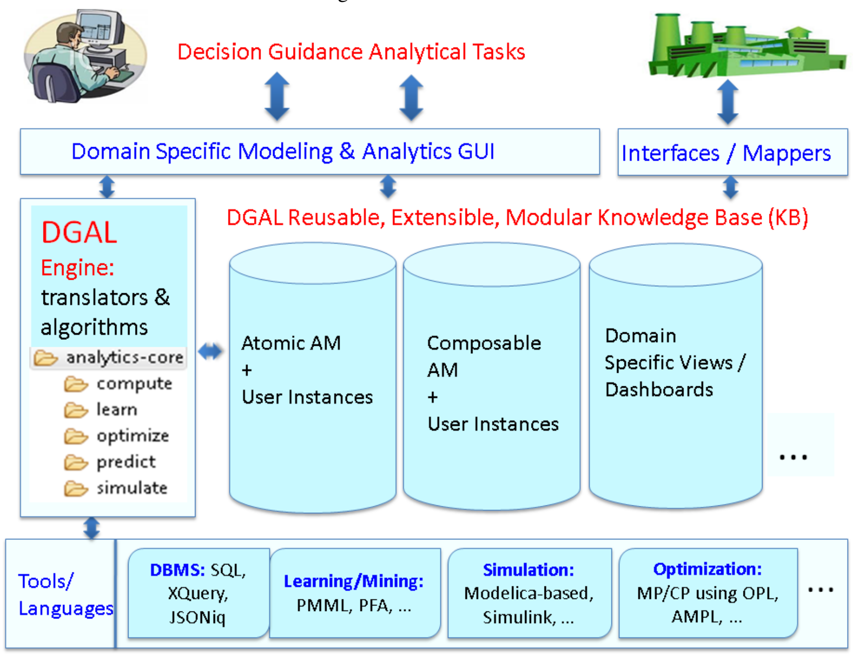
The Modelica simulation modeling language was designed to reuse knowledge. It allows a detailed level of abstraction, including Object-Oriented code and differential equations [7]. However, Modelica by itself is not a language for performing optimization, learning, or prediction. But there are tools such as JModelica for simulation, and Optimica for simulation-based optimization [8]. However, because of the low level of abstraction allowed in Modelica, general Modelica models cannot be automatically reduced to MP/CP models and solved by MP/CP solvers.

3. Conceptual Architecture & Desired Properties

The following are the design principles we identified for the proposed DGAL language/system:

- Reusable-KB-centric approach: DGAL must support the paradigm shift from
- Non-reusable task-centric modelling approach to reusable-KB-centric modelling approach.
- Task-independent representation of analytical knowledge: DGAL AMs must represent analytical knowledge (including data and its structure, parameters, control/decision variables, constraints and uncertainty) uniformly, regardless of the tasks (e.g., computation, prediction, optimization or learning) that may be using it.
- Unified language for analytical knowledge manipulation: DGAL must uniformly support (1) data manipulation (with the ease of data manipulation languages like SQL and JSONiq), (2) deterministic and stochastic computation/prediction, (3) decision optimization based on MP/CP, (4) statistical/machine learning, and (5) construction of AMs.
- Flexible construction of analytical knowledge: DGAL must support modular AM , generalization, specialization and reuse.
- Algebra over analytical KB: DGAL operators must form an algebra over the set of well-defined AMs, that is, operators applied to AMs (in the AKB) must return an AM. Thus, the resulting AMs can be put back into the AKB, and then used by other operators. Note, this is analogous to data manipulation languages (such as SQL, XQuery and JSONiq), which are algebras over the corresponding data model (relational, XML or JSON).
- Declarative high-level language: DGAL analytical knowledge manipulation operators (compute, optimize, learn) must be declarative and simple for end users.

Figure 2. DGAL Framework



Decision Guidance Analytical Tasks
Domain Specific Modeling & Analytics GUI
Interfaces / Mappers
DGAL Reusable, Extensible, Modular Knowledge Base (KB)
DGAL Engine: translators & algorithms
analytics-core
compute
learn
optimize
predict
simulate
Atomic AM + User Instances
Composable AM + User Instances
Domain Specific Views / Dashboards
...
Tools/ Languages
DBMS: SQL, XQuery, JSONiq
Learning/Mining: PMML, PFA, ...
Simulation: Modelica-based, Simulink, ...
Optimization: MP/CP using OPL, AMPL, ...
...

- Compact language core: It is desirable for DGAL to have a compact core, and allow additional functionality through built-in and user-developed libraries (in the knowledge-base).
- Ease of use by modellers: DGAL should be easy to use by mathematical modellers and software/DB developers.
- Ease of use by end users: DGAL should enable built-in KB libraries of AMs to raise the level of abstraction (obscure mathematical details, etc.), which would make it easy to use by end users (such as business analysts and managers)

The high-level DGAL framework and functionality is depicted in Figure 2. Central to the framework is the Analytical Knowledge Base, which is a collection of Analytical Models (AMs), possibly organized in different Viewpoint Libraries. AM is the base component of analytical knowledge. At the high-level, AM describes metrics and constraints of a system as a function of parameters and controls. More specifically, each AM composes of:

- Input JSON object schema that encodes parameters and contril variables of a system (e.g. machine, process, supply chain)
- Output JSON object schema that encodes metrics of interest (e.g. cost, time, energy, QoS) and the boolean value constraints, which is true if all system's feasibility constraints (e.g. laws of physics, engineering limitations, safety rules) are satisfied.
- JSONiq function that computes the output (with metrics and constraints) from an input (with parameters and controls). The function may use stochastic primitives drawing a value from a probability distribution.
- Constraints, using JSONiq syntax for Boolean expressions, including for universal quantification.
- Uncertainty, by adding distribution functions to expressions (in functions and constraints), which implicitly define random variables. All DGAL operators are applied to AM and return an AM, and so DGAL constitutes an algebra (like data manipulation languages SQL, XQuery and JSONiq). The DGAL operators are of four key types (Upper part of Figure 2).

- Construct: this class of operators allows to construct an AM from scratch, from another AM by specialization/-generalization, or by composing an AM from previously defined AMs.

DGAL Enginine provides the following core analytics functions to an AM:

- Compute: this class of operators instantiate an AM by performing computation of functions (may involve uncertainty quantification).
- Simulate: this class of operators instantiate an AM by performing computation of functions (may involve uncertainty quantification) multiple times and each time it returns a diffent output.
- Predict: this class of operators instantiate an AM by assigning values to variables and performing computations, given values of its parameters.
- Learn: this class of operators instantiate an AM by funding values of its parameters, as to minimize an estimation error against a learning set.
- Optimize: this class of operators instantiate an AM by finding values of decision variables that optimize an objective, and then compute it with the optimal values.

The (deterministic) optimization and learning operators are performed by creating a formal mathematical or constraint programming model and solving it using an appropriate solvers (Lower part of Figure 2). The AM, AKB and DGAL operators are designed according to the designed principles outlined in this section.

4. Overview of JSON and JSONiq

JavaScript Object Notation (JSON) is rapidly becoming a data model for descriptive (big data) analytics. For that reason, we would like to use JSONiq, a query language over JSON as a foundation of DGAL. Both JSON and JSONiq are reviewed in this section. We borrow here the description of JSONiq from [15][6].

JSON data model is a "lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate" [15]. JSON is an open standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML [6]. Similar to the fact that Xquery is a query and processing language designed for XML data model, JSON data model also have a specially designed powerful query language, JSONiq.

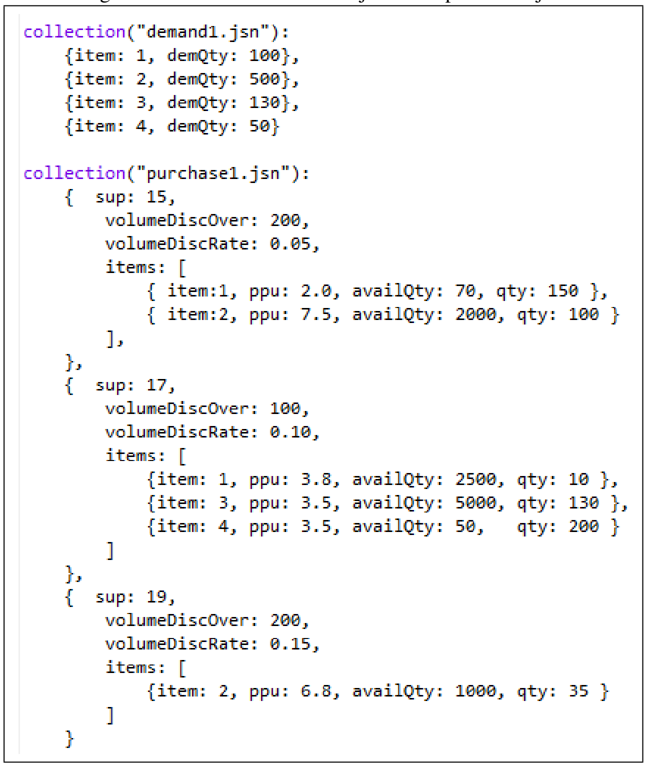
JSONiq is very similar to XQuery, adapted to JSON, including "the structure and semantics of the FLWR (FOR-LET-WHERE-RETURN) construct, the functional aspect of the language, the semantics of comparisons in the face of data heterogeneity, and the declarative, snapshot-based updates" [15]. However, Xquery is more complex and has more language constructs than JSONiq due to the fact that XML data model is more complex than that of JSON. For example, the element type of XML can be mixed with attributes, elements, or text. The order of children elements in XML is significant even with the same contents. The namespaces, QNames, and XML schema can be complicated to describe specific types of XML documents [6].

The FLWR construct is an iteration structure for both JSONiq/Xquery. It is different from regular control structure of programming languages by considering the simplicity of JSON data model. FLWR makes the JSONiq a powerful, clean, and straightforward data processing language. In addition to querying collections of data in JSON format, JSONiq can extract, transform, clean, select, enrich, or join hierarchical or heterogeneous data sets [6].

The main technical characteristics of JSONiq (and XQuery) are as follows:

- "It is a set-oriented language. While most programming languages are designed to manipulate one object at a time, JSONiq is designed to process sets (actually, sequences) of data objects" [6].
- It falls into the category of the functional programming paradigm. Different from the procedural programming paradigm such as Object-Oriented programming languages, expression is the basic unit of the JSONiq programs. Every language construct is an expression and expressions are composed from one or more previous expressions [6].

Figure 3. Collections “demand1.jsn” and “purchase1.jsn”



```
collection("demand1.jsn"):
    {item: 1, demQty: 100},
    {item: 2, demQty: 500},
    {item: 3, demQty: 130},
    {item: 4, demQty: 50}

collection("purchase1.jsn"):
    {  sup: 15,
        volumeDiscOver: 200,
        volumeDiscRate: 0.05,
        items: [
            { item:1, ppu: 2.0, availQty: 70, qty: 150 },
            { item:2, ppu: 7.5, availQty: 2000, qty: 100 }
        ],
    },
    {  sup: 17,
        volumeDiscOver: 100,
        volumeDiscRate: 0.10,
        items: [
            {item: 1, ppu: 3.8, availQty: 2500, qty: 10 },
            {item: 3, ppu: 3.5, availQty: 5000, qty: 130 },
            {item: 4, ppu: 3.5, availQty: 50,   qty: 200 }
        ]
    },
    {  sup: 19,
        volumeDiscOver: 200,
        volumeDiscRate: 0.15,
        items: [
            {item: 2, ppu: 6.8, availQty: 1000, qty: 35 }
        ]
    }
```

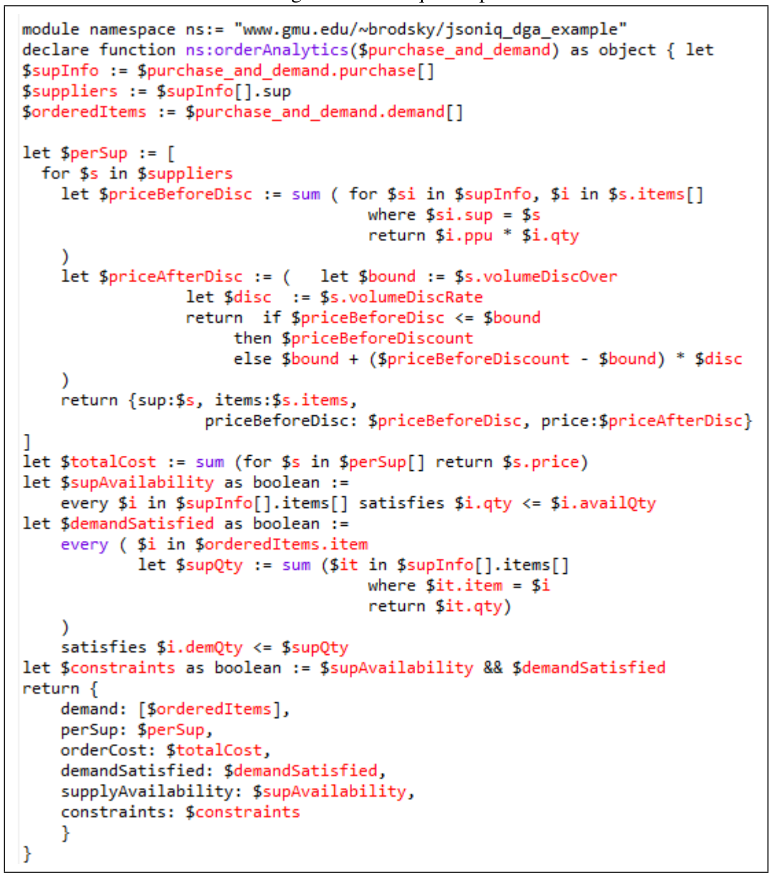
- It is a declarative language. It describes what computation will be performed, instead of how the computation (process) will be done. It does not consider the implementation details such as specific data structures, algorithms, memory allocations, and indexing in databases. The component of the declarative language usually has clear corresponding relationship to mathematical logic [6].
- It is designed to process hierarchical or heterogeneous data which is sometimes semi-structured. The JSON data type does not need to follow any specific pattern but can be heterogeneous. It can be nested structures in multiple levels. Consequently it is hard to define a schema which can describe the JSON data well. Sometimes the schema may only be able to describe the data partially [6].

To exemplify JSON and JSONiq, first consider JSON collections “demand1.jsn” and “purchase1.jsn” in Figure 3. JSON collections are sequences of objects, identified by a name which is a string, e.g. “demand1.jsn.” In the example, the collection “demand1.jsn” is a sequence that contains four objects, {item: 1, demQty: 100}, {item: 2, demQty: 500}, {item: 3, demQty: 130}, and {item: 4, demQty: 50}. Objects are unordered sets of key/value pairs, separated by comma. A key is a string and a value can be any JSON building block. Each key/value pair is separated by a semicolon. Four items are defined in this collection, with the quantity of demand specified.

Similarly the collection “purchase1.jsn” contains a sequence of three suppliers objects. For each supplier object, four sets of key/value pairs indicate supplier identifier, the overall supplier cost before and after volume discount, and the list of items being purchased from this supplier. Given “items” as the key of the key/value pair, a sequence of items is assigned as the value and of the type JSON Array. Array represents an ordered list of items (in any category) and can nest. For example, for the first supplier with key/value pair of “sup: 15”, two different items are purchased as {item:1, ppu: 2.0, availQty: 70, qty: 150} and {item:2, ppu: 7.5, availQty: 2000, qty: 100}. Each item is represented as a JSON object, with four key/value pairs. Each item has a unique identifier (e.g. 1), price per unit, available quantity from the specific supplier, and purchased quantity.

JSONiq is a language that makes computations of structures from the input collections easy. In the JSONiq example of Figure 4, a JSONiq function orderAnalytics is defined with variable `$purchase_and_demand` as argument

Figure 4. JSONiq Example



```
module namespace ns:= "www.gmu.edu/~brodsky/jsoniq_dga_example"
declare function ns:orderAnalytics($purchase_and_demand) as object { let
$supInfo := $purchase_and_demand.purchase[]
$suppliers := $supInfo[].sup
$orderedItems := $purchase_and_demand.demand[]

let $perSup := [
  for $s in $suppliers
    let $priceBeforeDisc := sum ( for $si in $supInfo, $i in $s.items[]
                                    where $si.sup = $s
                                    return $i.ppu * $i.qty
    )
    let $priceAfterDisc := (   let $bound := $s.volumeDiscOver
                  let $disc  := $s.volumeDiscRate
                  return  if $priceBeforeDisc <= $bound
                       then $priceBeforeDiscount
                       else $bound + ($priceBeforeDiscount - $bound) * $disc
    )
    return {sup:$s, items:$s.items,
                 priceBeforeDisc: $priceBeforeDisc, price:$priceAfterDisc}
]
let $totalCost := sum (for $s in $perSup[] return $s.price)
let $supAvailability as boolean :=
    every $i in $supInfo[].items[] satisfies $i.qty <= $i.availQty
let $demandSatisfied as boolean :=
    every ( $i in $orderedItems.item
            let $supQty := sum ($it in $supInfo[].items[]
                                    where $it.item = $i
                                    return $it.qty)
    )
    satisfies $i.demQty <= $supQty
let $constraints as boolean := $supAvailability && $demandSatisfied
return {
    demand: [$orderedItems],
    perSup: $perSup,
    orderCost: $totalCost,
    demandSatisfied: $demandSatisfied,
    supplyAvailability: $supAvailability,
    constraints: $constraints
    }
}
```

and object as return type. The argument variable `$purchase_and_demand` is of the composite structure of both collections "purchase1.jsn" and "demand1.jsn" as follows:

collection("purchaseAndDemand1.jsn"):

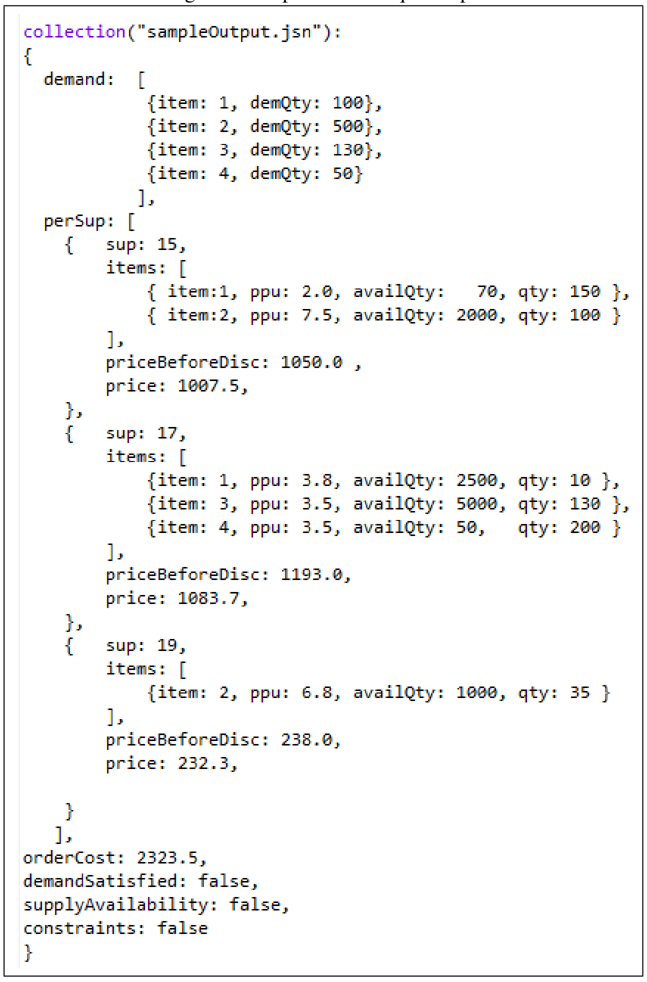
purchase: [collection("purchase1.jsn")], demand: [collection("demand1.jsn")]

The function implements a small supply chain with items, demand, and supplies. In the function body, the variable $supInfo is assigned as an array which contains the sequence of objects in the collection "purchase1.jsn." The variable $suppliers is assigned as an array of all supplier identifiers such as 15 or 17. The variable $orderedItems is assigned as an array of all the items and demand quantities in the collection "demand1.jsn."

The variable $perSup is defined to calculate for each supplier, the total cost charged for all items supplied by this supplier. The for clauses is used to iterate each supplier of $suppliers. For all items supplied by that specific supplier, within the inner "for" clause of the second let statement, the variable $priceBeforeDisc represents the cost to purchase items, which are calculated from price per unit and item quantity. The variable $priceAfterDisc is defined to adjust the item cost by considering the volume discount rate. The cost after discount is calculated based on a volume discount formula. If the overall cost is more than the volumeDiscOver, it will be calculated at a discount rate volumeDiscRate. The variable $totalCost sums up the total cost of all suppliers. The variable $supAvailability is defined as a Boolean variable to enforce the business rule that the item quantity purchased must be less than or equal to the available quantity to each specific supplier. The variable $demandSatisfied is defined as another Boolean variable to enforce the business rule that the total market demand for each item cannot beyond the total item supply from all suppliers. The Boolean variable $constraint is the logical '&&' of both $supAvailability and $demandSatisfied.

The object returned by this function consists of six key-value pairs representing market demand, detailed information for each supplier (identifier, supplied items, price before volume discount, and price after volume discount),

Figure 5. Output of JSONiq Example



```
collection("sampleOutput.jsn"):
{
  demand:  [
            {item: 1, demQty: 100},
            {item: 2, demQty: 500},
            {item: 3, demQty: 130},
            {item: 4, demQty: 50}
           ],
  perSup: [
    {   sup: 15,
        items: [
            { item:1, ppu: 2.0, availQty:   70, qty: 150 },
            { item:2, ppu: 7.5, availQty: 2000, qty: 100 }
        ],
        priceBeforeDisc: 1050.0 ,
        price: 1007.5,
    },
    {   sup: 17,
        items: [
            {item: 1, ppu: 3.8, availQty: 2500, qty: 10 },
            {item: 3, ppu: 3.5, availQty: 5000, qty: 130 },
            {item: 4, ppu: 3.5, availQty: 50,   qty: 200 }
        ],
        priceBeforeDisc: 1193.0,
        price: 1083.7,
    },
    {   sup: 19,
        items: [
            {item: 2, ppu: 6.8, availQty: 1000, qty: 35 }
        ],
        priceBeforeDisc: 238.0,
        price: 232.3,

    }
   ],
orderCost: 2323.5,
demandSatisfied: false,
supplyAvailability: false,
constraints: false
}
```

total cost of all purchases, if market demand has been satisfied, if the supply availability has been satisfied, and if both availability have been satisfied. The output of this JSONiq example is displayed in Figure 5, given the argument as composite collections in Figure 3.

The FLWOR expression is probably the most powerful JSONiq construct, which corresponds to the SQL Select-From-Where clause, but the JSONiq construct is more general and flexible. The FOR clause allows iteration over a sequence.

5. DGAL by Example: Deterministic Knowledge Manipulation

5.1. Computation and Data Manipulation

Performing computation/data manipulation with Analytical Model is just an invocation of a JSONiq function, like in the example in Figure 6. First, we need to import the module in which the function *orderAnalytics* is defined. Then we assign values to the argument of the function. The function is then called to perform the computation. In the example, it returns an object representing demand, cost of each supplier, total cost, and a true/false Boolean value for demand being satisfied, the supply availability being satisfied, and both constraints being satisfied.

Figure 6. Computation

```
module namespace my = "http://cs.gmu.edu/~brodsky/sandbox"
import module namespace sp = "www.gmu.edu/~brodsky/jsoniq_dga_example"
// computation:
let $purchase1 := collection("purchase1.jsn")
let $demand1 := collection("demand1.jsn")
return sp:orderAnalytics({purchase: $purchase1, demand: $demand1})
```

Figure 7. varPurchase1.jsn

```
collection("varPurchase1.jsn"):
// indexable object
{  sup: 15,
    volumeDiscOver: 200,
    volumeDiscRate: 0.05,
    items: [
        { item:1, ppu: 2.0, availQty: 70,   qty: "int ?" },
        { item:2, ppu: 7.5, availQty: 2000, qty: "int ?" }
    ],
 },
 {  sup: 17,
    volumeDiscOver: 100,
    volumeDiscRate: 0.10,
    items: [
        {item: 1, ppu: 3.8, availQty: 2500, qty: "int ?" },
        {item: 3, ppu: 3.5, availQty: 5000, qty: "int ?" },
        {item: 4, ppu: 3.5, availQty: 50,   qty: "int ?" }
    ]
 },
 {  sup: 19,
    volumeDiscOver: 200,
    volumeDiscRate: 0.15,
    items: [
        {item: 2, ppu: 6.8, availQty: 1000, qty: "int ?" }
    ]
 }
```

5.2. *Optimization*

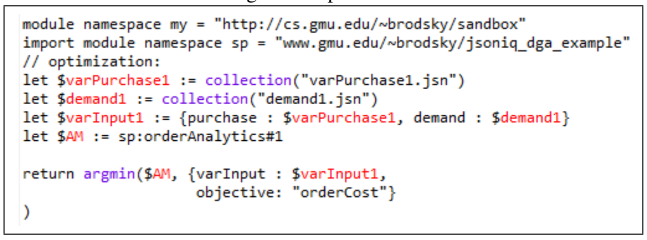
If the quantities in the collection "purchase1.jsn" used as argument to JSONiq query in Figure 4 were not known, the procurement officer may want to find them as to make sure that the constraints of both Boolean expressions "$demandSatisfied" and "$supAvailability" are satisfied (i.e., computed to be true), and the total cost is minimized. Intuitively, the DGAL optimization operators are designed to do this kind of "reverse" computation.

To perform optimization in DGAL, we first need to annotate the input object to the analytical model function (*orderAnalytics* in the example) to indicate which values (*qty*s in the example) are not known but we would like the system to find (i.e., decision variables). Figure 7 gives an example of such annotation in the collection "varPurchase1.jsn." This collection is identical to the collection "purchase1.jsn" in Figure 3, with the exception that the key qty does not have a numerical value, but has instead a special annotation "*int ?*" to indicate that *qty* will now be a decision variable. The annotated collection must have the following structural limitation. Informally, the structure of the output object should not depend on the values of the annotated decision variables; only numerical values in the output object may depend on the decision variables.

With the annotated input collection, performing optimization is very simple: it is performed by invoking the DGAL function *argmin* (or *argmax*) as exemplified in Figure 8.

The DGAL function argmin in Figure 8 is invoked with the following input object. The first parameter $AM is assigned as a type of function reference in JSONiq and points to the Analytical Model used, i.e., sp:orderAnalytics in the example. The second parameter is an array which has two key-value pairs. The first key *varInput* has the value which is the annotated input to *orderAnalytics*, i.e., when we use the annotated collection "*varPurchase1.jsn*" from Figure 7 instead of the original collection "*purchase1.jsn*" from Figure 3. The second key *objective* has the value that indicates the key of the numeric value in the output object of *orderAnalytics* *orderCost* in the example - that we

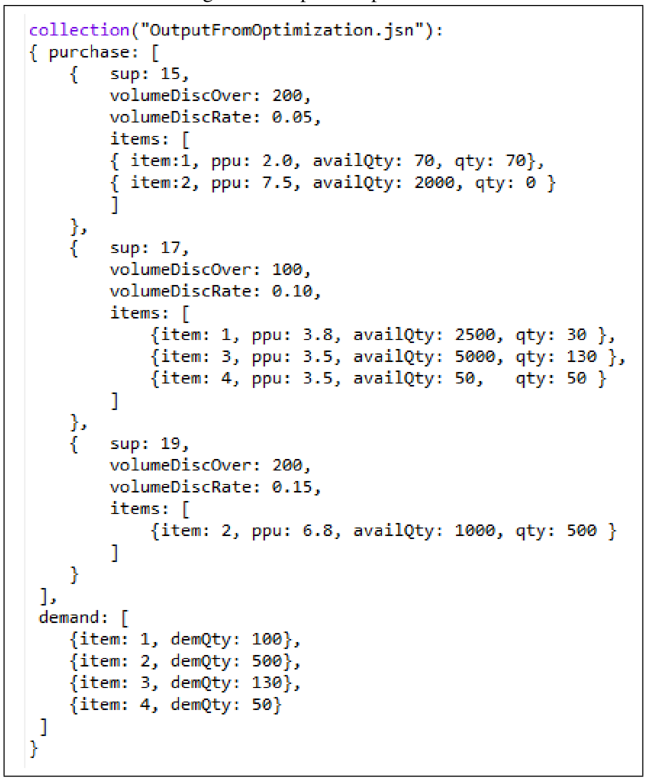
Figure 8. Optimization



```
module namespace my = "http://cs.gmu.edu/~brodsky/sandbox"
import module namespace sp = "www.gmu.edu/~brodsky/jsoniq_dga_example"
// optimization:
let $varPurchase1 := collection("varPurchase1.jsn")
let $demand1 := collection("demand1.jsn")
let $varInput1 := {purchase : $varPurchase1, demand : $demand1}
let $AM := sp:orderAnalytics#1

return argmin($AM, {varInput : $varInput1,
                     objective: "orderCost"}
)
```

Figure 9. Output of Optimization



```
collection("OutputFromOptimization.jsn"):
{ purchase: [
    {   sup: 15,
        volumeDiscOver: 200,
        volumeDiscRate: 0.05,
        items: [
        { item:1, ppu: 2.0, availQty: 70, qty: 70},
        { item:2, ppu: 7.5, availQty: 2000, qty: 0 }
        ]
    },
    {   sup: 17,
        volumeDiscOver: 100,
        volumeDiscRate: 0.10,
        items: [
            {item: 1, ppu: 3.8, availQty: 2500, qty: 30 },
            {item: 3, ppu: 3.5, availQty: 5000, qty: 130 },
            {item: 4, ppu: 3.5, availQty: 50,   qty: 50 }
        ]
    },
    {   sup: 19,
        volumeDiscOver: 200,
        volumeDiscRate: 0.15,
        items: [
            {item: 2, ppu: 6.8, availQty: 1000, qty: 500 }
        ]
    }
 ],
 demand: [
    {item: 1, demQty: 100},
    {item: 2, demQty: 500},
    {item: 3, demQty: 130},
    {item: 4, demQty: 50}
 ]
}
```

would like to use as the objective to be minimized.

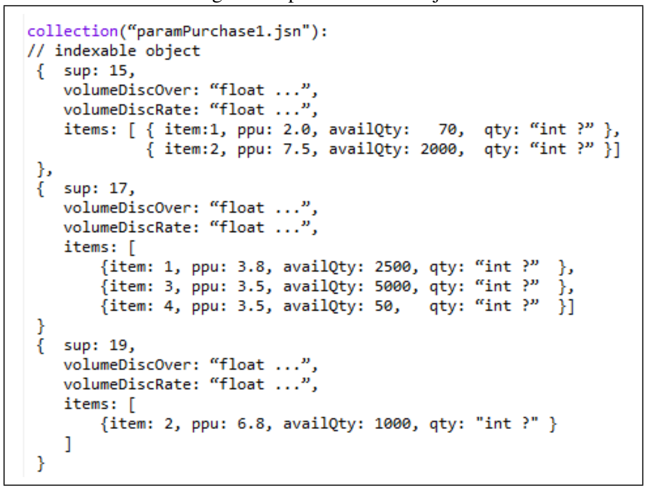
The output of the *argmin* function is an object identical to the annotated input object, with the exception that all annotations "*int ?*" (which denote decision variables) are now replaced with actual numerical values that (1) satisfy the *constraints*, i.e., result in the value true computed for the key *constraints* in the output object; and (2) minimize the numerical value computed for the key *orderCost*, which was designated as the objective in the invocation of the *argmin* function. The collection "*OutputFromOptimization.jsn*" in Figure 9 exemplifies the optimization output.

5.3. Learning

If the volume discount parameters, *volumeDiscOver* and *volumeDiscRate*, were not known in the collection "purchase1.jsn", how can we learn them (via regression analysis) from historical data? The DGAL function *learn* is provided for this purpose.

To use the learn function in DGAL, we first need to annotate the input object to the Analytical Model (orderAnalytics in the example) to indicate which parameters we would like to learn. In the example these parameters are *volumeDiscOver* and *volumeDiscRate* for each supplier in collection "varPurchase.jsn." Figure 10 gives an example

Figure 10. paramPurchase1.jsn



```
collection("paramPurchase1.jsn"):
// indexable object
 {  sup: 15,
    volumeDiscOver: "float ...",
    volumeDiscRate: "float ...",
    items: [ { item:1, ppu: 2.0, availQty:   70,  qty: "int ?" },
             { item:2, ppu: 7.5, availQty: 2000,  qty: "int ?" }]
 },
 {  sup: 17,
    volumeDiscOver: "float ...",
    volumeDiscRate: "float ...",
    items: [
        {item: 1, ppu: 3.8, availQty: 2500, qty: "int ?"  },
        {item: 3, ppu: 3.5, availQty: 5000, qty: "int ?"  },
        {item: 4, ppu: 3.5, availQty: 50,    qty: "int ?"  }]
 }
 {  sup: 19,
    volumeDiscOver: "float ...",
    volumeDiscRate: "float ...",
    items: [
        {item: 2, ppu: 6.8, availQty: 1000, qty: "int ?" }
    ]
 }
```

of such annotation in the collection "paramPurchase1.jsn." This collection is identical to the collection "varPurchase1.jsn" in Figure 7, with the exception that the keys *volumeDiscOver* and *volumeDiscRate* do not have a numerical value, but have instead a special annotation "float ..." to indicate that they will now be parameters to be learned using regression.

We also need to create a learning set collection, which captures a set of (historical) input-output pairs of a function we are trying to regress. The collection "learningSet1.jsn" in Figure 11 exemplifies the learning set. It contains a sequence of JSON objects, each having keys input and output. The value for the key output in every object gives a historical value for orderCost. The value for the key input in every object gives a "partial" input object for the Analytical Model orderAnalytics. It is "partial" because we only need information on the quantities for each item order from each supplier, but do not need any other information. With the "paramPurchase1.jsn" collection and the learning set object, performing regression learning is done by invocation of DGAL function learn, as shown in Figure 12.

The function *learn* is invoked with an object with key-value pairs for *paramInput*; analytics to indicate the Analytical Model to be learned (*orderAnalytics* in the example); *outValue*, to indicate which value the function computes (*orderCost* in the example), and the previously constructed *learningSet*.

The output of function *learn* is exemplified in Figure 13. The structure of output is exactly as the input collection "*paramPurchaseAndDemand1*" with the exception that parametric annotations "float ..." replaced by values. However, the decision variables annotations "int ?" are still left as they were. Note that both parameters and decision variables can be either of type int or float. The values for parameters are constructed to minimize the summation of squares of learning errors for each input-output pair in the learning set.

6. DGAL by Example: Uncertainty Management (Simulation, Prediction, and Stochastic Optimization)

Consider Figure 14, which exemplifies how uncertainty is represented in AMs.

In the example, the *stochOrderAnalytics* function is identical to the (deterministic) *orderAnalytics* function in Figure 4 with the exception that, in the *$ppu* computation, a random value drawn from the normal or Gaussian distribution is used (see a box in Figure 14). The Gaussian distribution is constructed with the mean of zero and standard deviation of 0.05. When this is done, the variable *$ppu* on the left of the assignment statement represents a random variable. Furthermore, all expressions that are dependent on it in the computation, directly or indirectly, also represent random variables.

In the context of uncertainty, we can talk about simulation, prediction and stochastic optimization (see example in Figure 15). Simulation is done exactly as computation, namely by invoking the Analytical Model *stochOrderAnalytics*. Because a random value is drawn as the noise added to the unit price of products (see Figure 14), every invocation

Figure 11. Learning Set

```
collection("learningSet1.jsn"):
{ input: { purchase: [
        { sup: 15,
          items: [ {item: 1, qty: 150 }, { item: 2, qty: 100 }],
        },
        { sup: 17,
          items: [ {item: 1, qty: 10 }, {item: 3, qty: 130}, {item: 4, qty: 200}]
        },
        { sup: 19,
          items: [ {item: 2, qty: 35 }]
        }]},
  output: 2329.4
},
{ input: { purchase: [
        { sup: 15,
          items: [ { item: 1, qty: 250 }, { item: 2, qty: 55 }, {item: 3, qty: 50}],
        },
        { sup: 17,
          items: [ {item: 1, qty: 25 }, {item: 4, qty: 130 }]
        },
        { sup: 19,
          items: [ {item: 2, qty: 35 }, {item: 3, qty: 80}]
        }]},
  output: 2390.3
},
     ....    ....
{ input: { purchase: [
        { sup: 15,
          items: [ { item: 1, qty: 90 }, { item: 2, qty: 210 }],
        },
        { sup: 17,
          items: [ {item: 1, qty: 110 }, {item: 3,qty: 28 }, {item: 4, qty: 185 }]
        },
        { sup: 19,
          items: [ {item: 2, qty: 76 }]
        }]},
  output: 3210.6
}
```

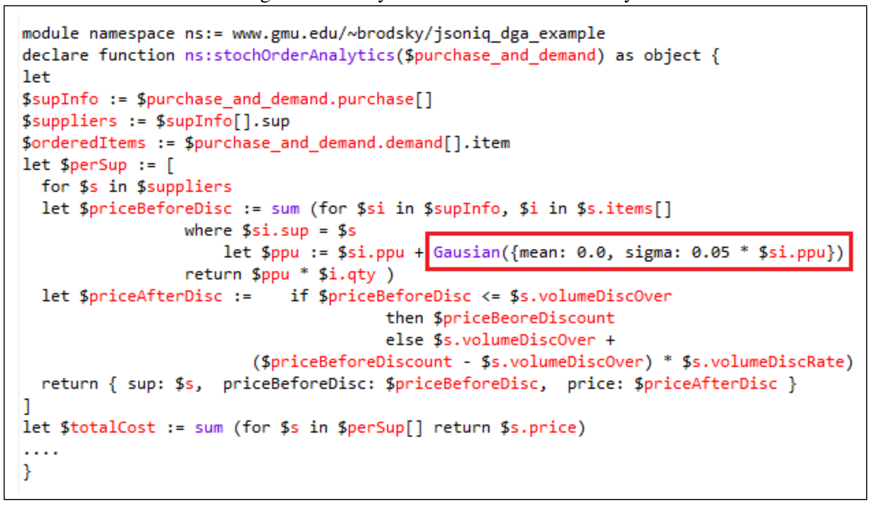
Figure 12. Learning

```
module namespace my = "http://cs.gmu.edu/~brodsky/sandbox"
import module namespace sp = "www.gmu.edu/~brodsky/jsoniq_dga_example"
// learning
let $paramPurchaseAndDemand1:={paramPurchase:collection("paramPurchase1.jsn"),
                                             demand:collection("demand1.jsn")}
let $learningSet1:=collection("learningSet1.jsn")
return learn({paramInput: $paramPurchaseAndDemand1,
            analytics: "sp:orderAnalytics",
            outValue: "orderCost"
            learningSet: [$learningSet1]
})
```

Figure 13. Output from Learning

```
collection("OutputFromLearning.jsn"):
// indexable object
 {  sup: 15,
    volumeDiscOver: 200,
    volumeDiscRate: 0.05,
    items: [ { item:1, ppu: 2.0, availQty:   70,  qty: "int ?" },
             { item:2, ppu: 7.5, availQty: 2000,  qty: "int ?" }]
 },
 {  sup: 17,
    volumeDiscOver: 100,
    volumeDiscRate: 0.10,
    items: [
         {item: 1, ppu: 3.8, availQty: 2500, qty: "int ?"  },
         {item: 3, ppu: 3.5, availQty: 5000, qty: "int ?"  },
         {item: 4, ppu: 3.5, availQty: 50,   qty: "int ?"  }]
 }
 {  sup: 19,
    volumeDiscOver: 200,
    volumeDiscRate: 0.15,
    items: [
         {item: 2, ppu: 6.8, availQty: 1000, qty: "int ?" }
    ]
 }
```

Figure 14. Analytical Model with Uncertainty



```
module namespace ns:= www.gmu.edu/~brodsky/jsoniq_dga_example
declare function ns:stochOrderAnalytics($purchase_and_demand) as object {
let
$supInfo := $purchase_and_demand.purchase[]
$suppliers := $supInfo[].sup
$orderedItems := $purchase_and_demand.demand[].item
let $perSup := [
  for $s in $suppliers
  let $priceBeforeDisc := sum (for $si in $supInfo, $i in $s.items[]
                 where $si.sup = $s
                     let $ppu := $si.ppu + Gausian({mean: 0.0, sigma: 0.05 * $si.ppu})
                 return $ppu * $i.qty )
  let $priceAfterDisc :=    if $priceBeforeDisc <= $s.volumeDiscOver
                                      then $priceBeoreDiscount
                                      else $s.volumeDiscOver +
                        ($priceBeforeDiscount - $s.volumeDiscOver) * $s.volumeDiscRate)
  return { sup: $s,  priceBeforeDisc: $priceBeforeDisc,  price: $priceAfterDisc }
]
let $totalCost := sum (for $s in $perSup[] return $s.price)
....
}
```

of stochOrderAnalytics may result in a different answer, which is a result of stochastic simulation.

We can also perform prediction, e.g., using the Monte-Carlo method, to estimate the expectation of the random variables ($ppu and all expressions that depend on it in Figure 14). This is done using the DGAL function predict, which is invoked with the input object that has key-value pairs *input, analytics, sigmaUpperBound, confidence and timeUpperBound* (see *prediction* part in Figure 15). The value for input is the same as in the computation (i.e., collection *purchaseAndDemand1*); *analytics* indicates the used analytical function (*stochOrderAnalytics* in the example). The estimates of the random variables are done so that the standard deviation would not exceed the *sigmaUpperBound* with the indicated statistical *confidence*, unless computation time exceeds the indicated *timeUpperBound*, which is an optional parameter. The form of the output is the same as in the deterministic computation (the example in Figure 5), with annotation exception that instead of the (deterministically) computed numeric values, e.g. ppu with the values of 2.0, 7.5, 3.8, 3.5, or 6.8 per supplier and per item, the output object contains JSON objects that capture estimations of the expectation and standard deviation of random variables ppu.

Finally, a one-stage stochastic optimization is performed by invoking the DGAL function *argmin* (or *argmax*). The input object is the same as in the deterministic case, with the exception that it is extended with key-value pairs for the required *constraintSatProb* to indicate with what minimal probability the constraints must be satisfied and with what statistical *confidence*. Optionally, a maximum computation *budget* can be indicated.

The output of the *argmin* function has the same form as in the deterministic case. Semantically, *argmin* in this case is interpreted as a (one-stage) stochastic optimization to minimize the expectation of the indicated objective (*orderCost* in the example), which is now interpreted as a random variable.

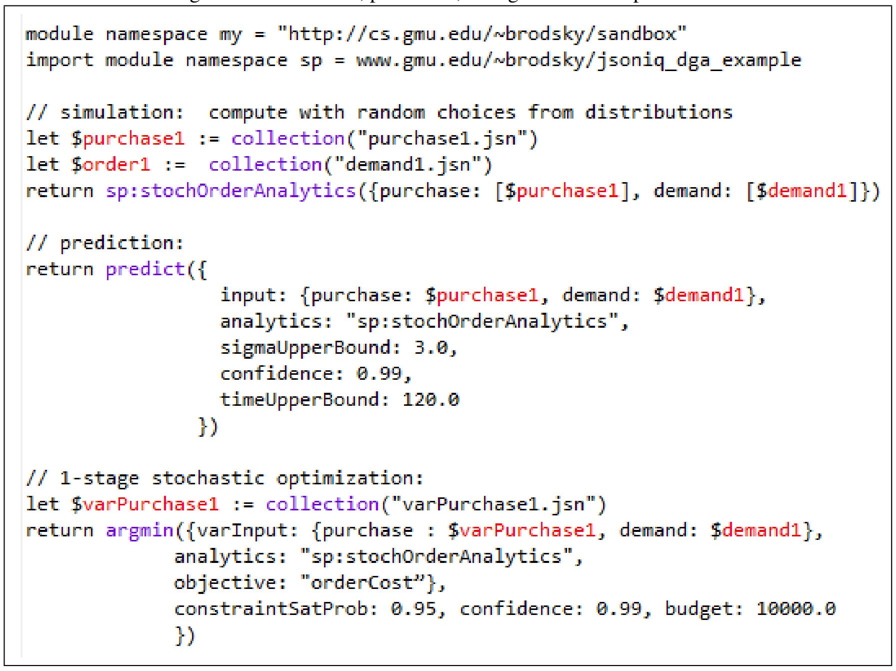
7. DGAL Knowledge Base

We describe the components of the DGAL knowledge base in Figure 16.

The analytical knowledge base component provides a facility for the storage and retrieval of analytical knowledge artifacts. An analytical knowledge artifact is an abstract concept that refers to any data element that can be identified by an URI. Data semantics are determined by the artifact type. The DGAL engine currently supports different types of analytical knowledge artifacts such as JSON, JSONiq, JSON Schema, DGAL, and Package.

A repository in the JSON/JSONiq tier is a set of information resources that can be uniquely referenced by a Uniformed Resource Indicator (URI). Examples of such resources include JSON documents, JSONiq modules and JSON schemas. Within the DGAL tier, a DGAL Knowledge Base is a specialization of a JSONiq repository that provides decision-guidance capabilities, such learning and optimization. An Analytica Model is a specialization of a JSONiq module that include analytical model functions and schemas for input and output. Analytical model functions

Figure 15. Simulation, prediction, 1-stage stochastic optimization



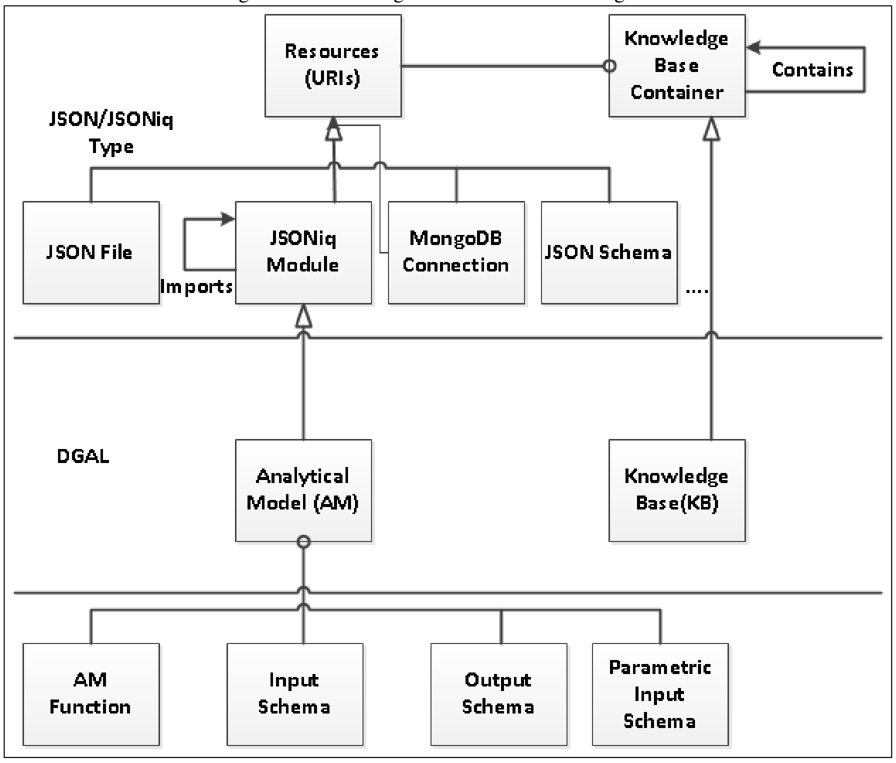
```
module namespace my = "http://cs.gmu.edu/~brodsky/sandbox"
import module namespace sp = www.gmu.edu/~brodsky/jsoniq_dga_example

// simulation:  compute with random choices from distributions
let $purchase1 := collection("purchase1.jsn")
let $order1 :=  collection("demand1.jsn")
return sp:stochOrderAnalytics({purchase: [$purchase1], demand: [$demand1]})

// prediction:
return predict({
                 input: {purchase: $purchase1, demand: $demand1},
                 analytics: "sp:stochOrderAnalytics",
                 sigmaUpperBound: 3.0,
                 confidence: 0.99,
                 timeUpperBound: 120.0
               })

// 1-stage stochastic optimization:
let $varPurchase1 := collection("varPurchase1.jsn")
return argmin({varInput: {purchase : $varPurchase1, demand: $demand1},
             analytics: "sp:stochOrderAnalytics",
             objective: "orderCost"},
             constraintSatProb: 0.95, confidence: 0.99, budget: 10000.0
             })
```

Figure 16. UML diagram for DGAL Knowledge Base



are essentially JSONiq functions that can be interpreted using alternative, decision-guidance semantics described in previous sections.

Clearly, AM functions, like any JSONiq function, perform data manipulation. For example, to compute information about order analytics from the purchase and demand data, one can simply invoke the JSONiq function *orderAnalytics*, described in Figure 4. However, we will also use AM functions for optimization, statistical learning, simulation and prediction with the same ease as data manipulation.

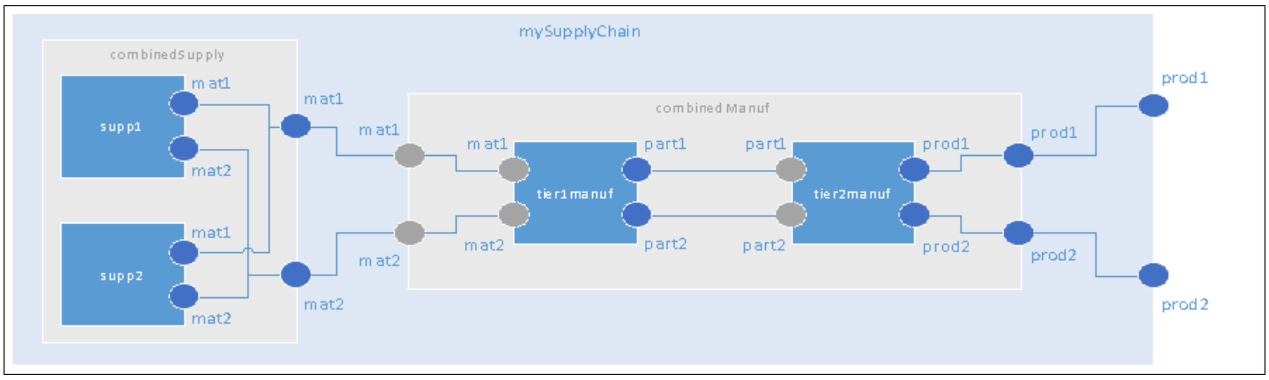
8. A Composable Supply Chain Case Study

In this section, we demonstrate the ability of composing analytical models in DGAL with a case study regarding supply chain optimization. Consider the simple supply chain model shown in Figure 17 that describes the structure and interactions of the various suppliers and manufacturing involved in the production of two supposed products: prod1 and prod2. At the highest level, the entire supply chain is represented by the mySupplyChain composite component. This component provides two outputs, namely prod1 and prod2, and requires no input. As a composite type component, it is itself composed of two sub-components, which in this case are combinedSupply and combinedManuf.

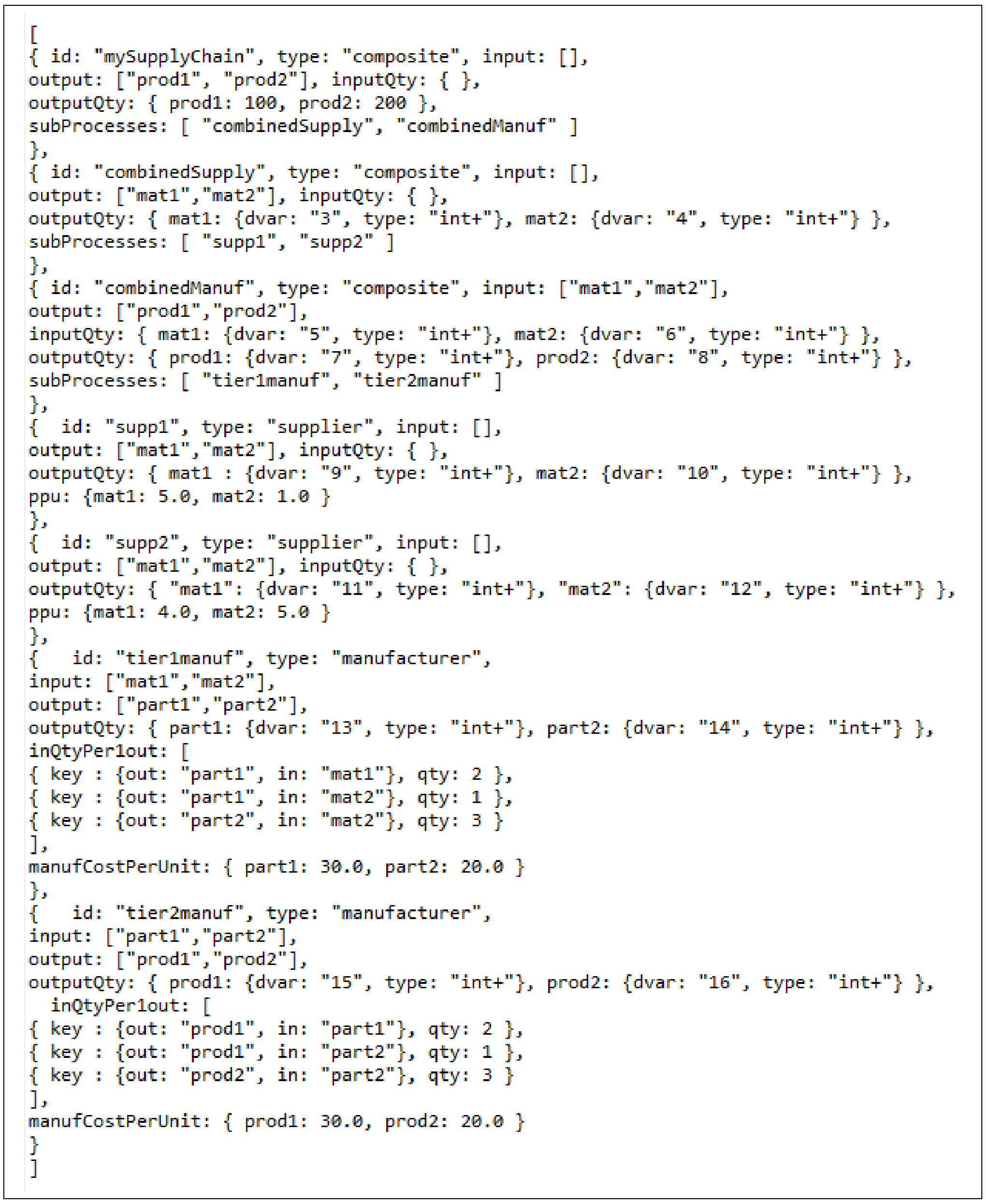
The combinedSupply component is a composite type component with two outputs that correspond to the raw materials used during the manufacturing process. The combinedSupply component is in turn composed of two atomic components, supp1 and supp2, each representing an individual supplier of raw material. This provides a layer of abstraction over the individual suppliers and allows us to reference both suppliers using the single, composite component.

The combinedManuf component is also a composite type component and has two inputs, mat1 and mat2, which are connected to the two outputs of the combinedSupply component. It has two outputs, prod1 and prod2, which directly correspond to the two outputs of the mySupplyChain component. The combinedManuf component is in turn composed of two atomic components, namely tier1manuf and tier2manuf, which are arranged in sequence, as depicted in Figure 17.

Figure 17. A Composable Supply Chain Model



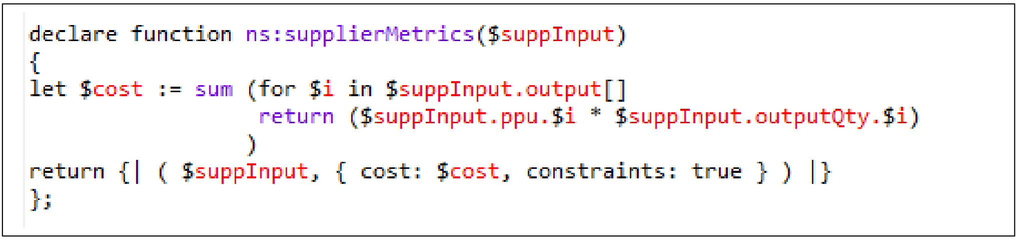
Based on the description above, we can develop a formal supply chain model in DGAL to be used for decision guidance analytics, like optimization. The first step is to formalize the structure of the components and connections in our supply chain model. This can be represented in JSON, which provides a compact syntax for expressing nested data relationships. A schema can also be developed to enforce a standard hierarchy or other structural constraints using JSON Schema or JSound. Our structural model consists of a JSON-array of objects that represent each individual component of our supply chain model. Each represented component contains the following object properties: a unique identifier, the type of component (either composite, supplier, or manufacturer), the input types and input quantities, the output types and output quantities. Composite type components have an additional property that specify any sub-processes that they compose. Supplier type components contain a property that indicate the price-per output unit, while manufacturer type components contain property that specifies the quantities of each input type needed to produce one unit of output. The full JSON representation for the structure of the supply chain example is provided below:



```
[
{ id: "mySupplyChain", type: "composite", input: [],
output: ["prod1", "prod2"], inputQty: { },
outputQty: { prod1: 100, prod2: 200 },
subProcesses: [ "combinedSupply", "combinedManuf" ]
},
{ id: "combinedSupply", type: "composite", input: [],
output: ["mat1","mat2"], inputQty: { },
outputQty: { mat1: {dvar: "3", type: "int+"}, mat2: {dvar: "4", type: "int+"} },
subProcesses: [ "supp1", "supp2" ]
},
{ id: "combinedManuf", type: "composite", input: ["mat1","mat2"],
output: ["prod1","prod2"],
inputQty: { mat1: {dvar: "5", type: "int+"}, mat2: {dvar: "6", type: "int+"} },
outputQty: { prod1: {dvar: "7", type: "int+"}, prod2: {dvar: "8", type: "int+"} },
subProcesses: [ "tier1manuf", "tier2manuf" ]
},
{  id: "supp1", type: "supplier", input: [],
output: ["mat1","mat2"], inputQty: { },
outputQty: { mat1 : {dvar: "9", type: "int+"}, mat2: {dvar: "10", type: "int+"} },
ppu: {mat1: 5.0, mat2: 1.0 }
},
{  id: "supp2", type: "supplier", input: [],
output: ["mat1","mat2"], inputQty: { },
outputQty: { "mat1": {dvar: "11", type: "int+"}, "mat2": {dvar: "12", type: "int+"} },
ppu: {mat1: 4.0, mat2: 5.0 }
},
{   id: "tier1manuf", type: "manufacturer",
input: ["mat1","mat2"],
output: ["part1","part2"],
outputQty: { part1: {dvar: "13", type: "int+"}, part2: {dvar: "14", type: "int+"} },
inQtyPer1out: [
{ key : {out: "part1", in: "mat1"}, qty: 2 },
{ key : {out: "part1", in: "mat2"}, qty: 1 },
{ key : {out: "part2", in: "mat2"}, qty: 3 }
],
manufCostPerUnit: { part1: 30.0, part2: 20.0 }
},
{   id: "tier2manuf", type: "manufacturer",
input: ["part1","part2"],
output: ["prod1","prod2"],
outputQty: { prod1: {dvar: "15", type: "int+"}, prod2: {dvar: "16", type: "int+"} },
  inQtyPer1out: [
{ key : {out: "prod1", in: "part1"}, qty: 2 },
{ key : {out: "prod1", in: "part2"}, qty: 1 },
{ key : {out: "prod2", in: "part2"}, qty: 3 }
],
manufCostPerUnit: { prod1: 30.0, prod2: 20.0 }
}
]
```

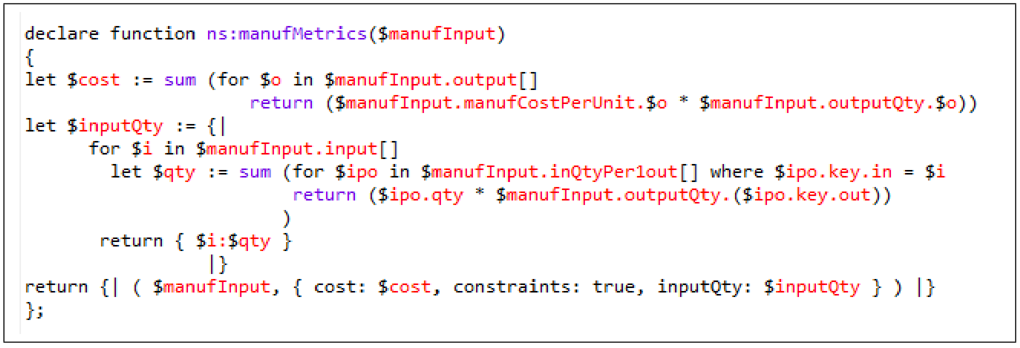
Once the structure of the components and connections have been specified, the next step is to specify the interaction semantics for each type of component in our model. For each component type, a DGAL analytical function, corresponding to a JSONiq function, needs to be provided that specifies how components are composed along with any constraints, and computes analytical metrics, such as cost.

For supplier components, the following DGAL analytical function computes a cost metric based on the sum of the cost of each supplied item:



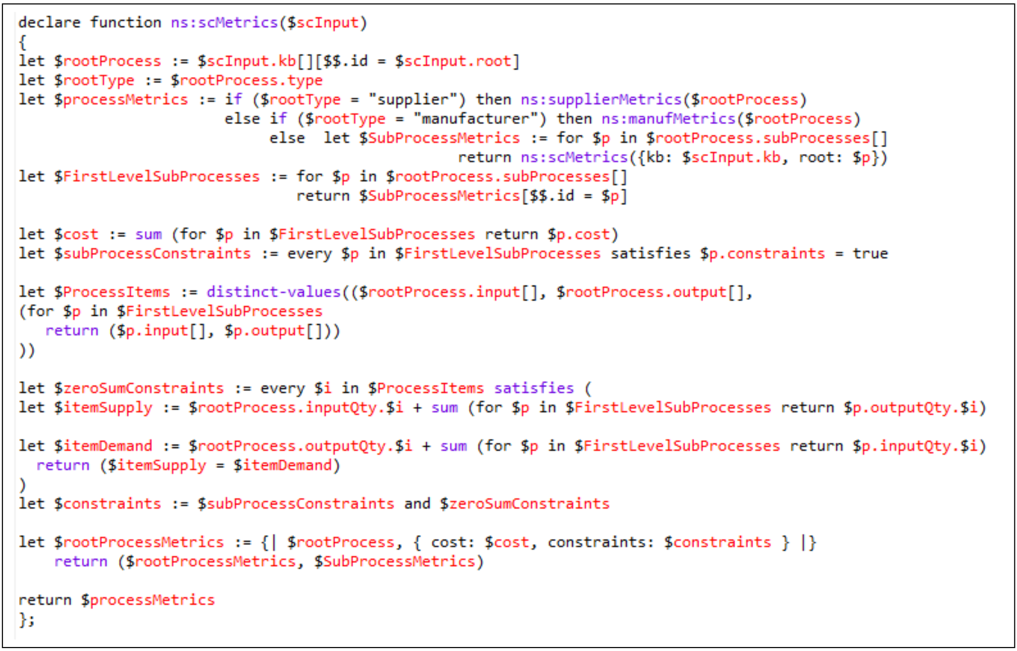
```
declare function ns:supplierMetrics($suppInput)
{
let $cost := sum (for $i in $suppInput.output[]
                  return ($suppInput.ppu.$i * $suppInput.outputQty.$i)
                 )
return {| ( $suppInput, { cost: $cost, constraints: true } ) |}
};
```

For manufacturer components, the following DGAL analytical function computes a cost metric based on the sum of the cost of each output product:



```
declare function ns:manufMetrics($manufInput)
{
let $cost := sum (for $o in $manufInput.output[]
                  return ($manufInput.manufCostPerUnit.$o * $manufInput.outputQty.$o))
let $inputQty := {|
      for $i in $manufInput.input[]
        let $qty := sum (for $ipo in $manufInput.inQtyPer1out[] where $ipo.key.in = $i
                         return ($ipo.qty * $manufInput.outputQty.($ipo.key.out))
                        )
       return { $i:$qty }
                   |}
return {| ( $manufInput, { cost: $cost, constraints: true, inputQty: $inputQty } ) |}
};
```

Next, a DGAL analytical function is provided for composite components. The function specifies how sub- components are composed, ensuring that metrics are properly aggregated and additional constraints are enforced:



```
declare function ns:scMetrics($scInput)
{
let $rootProcess := $scInput.kb[][$$.id = $scInput.root]
let $rootType := $rootProcess.type
let $processMetrics := if ($rootType = "supplier") then ns:supplierMetrics($rootProcess)
                       else if ($rootType = "manufacturer") then ns:manufMetrics($rootProcess)
                            else  let $SubProcessMetrics := for $p in $rootProcess.subProcesses[]
                                                           return ns:scMetrics({kb: $scInput.kb, root: $p})
let $FirstLevelSubProcesses := for $p in $rootProcess.subProcesses[]
                               return $SubProcessMetrics[$$.id = $p]

let $cost := sum (for $p in $FirstLevelSubProcesses return $p.cost)
let $subProcessConstraints := every $p in $FirstLevelSubProcesses satisfies $p.constraints = true

let $ProcessItems := distinct-values(($rootProcess.input[], $rootProcess.output[],
(for $p in $FirstLevelSubProcesses
   return ($p.input[], $p.output[]))
))

let $zeroSumConstraints := every $i in $ProcessItems satisfies (
let $itemSupply := $rootProcess.inputQty.$i + sum (for $p in $FirstLevelSubProcesses return $p.outputQty.$i)

let $itemDemand := $rootProcess.outputQty.$i + sum (for $p in $FirstLevelSubProcesses return $p.inputQty.$i)
  return ($itemSupply = $itemDemand)
)
let $constraints := $subProcessConstraints and $zeroSumConstraints

let $rootProcessMetrics := {| $rootProcess, { cost: $cost, constraints: $constraints } |}
    return ($rootProcessMetrics, $SubProcessMetrics)

return $processMetrics
};
```

Finally to put things all together, the DGAL argmin function, specified in previous sections, can be used to optimize our supply chain model via our composite analytical functions. In this case we pass to the argmin function our supply chain JSON structure, the name and namespace of our analytical composition function, scMetrics, and the objective that we want to minimize, cost. The result of calling argmin will be a fully instantiated supply chain model, where all the decision variables have been solved.

```
return [ns:scMetrics(dgal:argmin({ varInput: {kb: $exampleVarCompositeInput, root: "mySupplyChain"},
                                   ns: "http://cs.gmu.edu/dgal/supplyChain.jq",
                                   analytics: "scMetrics", objective: "cost"
}))]
```

9. Conclusions and Future Work

In this paper we proposed the Decision Guidance Analytics Language (DGAL) for easy iterative development of decision guidance systems. The work on DGAL leverages our prior work on decision guidance and optimization languages. In particular, the unification of computation and equational syntax comes from CoJava [16], SC-CoJava [17] and DGQL [18], CoReJava [19][20] on adding regression to DGAL, and DG-Query [21] which are designed to seamlessly add deterministic optimization and machine learning to Java, SQL and XQuery code, respectively, via automatic reduction to MP, CP or specialized algorithms. Also, DGAL fits into the framework of, but is significantly more general than, Decision Guidance Management Systems, proposed in [22]. Finally, the concept of centralized AKB is borrowed from our work on the Process Analytics Formalism [23][24], which was limited to MP/CP optimization only.

Many research questions remain open. They include (1) specific reduction algorithms from DGAL queries and AMs to specialized formal models of optimization, statistical learning, and uncertainty quantification; it may be promising to borrow from the work on constraint databases to support symbolic constraint (2) development specialized algorithms for stochastic optimization in DGAL that can leverage deterministic approximation encoded in DGAL Analytical Models; (3) development of specialized algorithms that can utilize pre-processing of stored (and therefore, static) AMs, to speed up optimization, generalizing the results in [25]; and, (4) developing graphical user interfaces for domain specific languages based on DGAL.

References

[1] G. Shmueli, O. R. Koppius, Predictive analytics in information systems research, Mis Quarterly 35 (3) (2011) 553–572.

[2] D. C. Montgomery, E. A. Peck, G. G. Vining, Introduction to linear regression analysis, Vol. 821, John Wiley & Sons, 2012.

[3] P. J. Haas, P. P. Maglio, P. G. Selinger, W. C. Tan, Data is dead... without what-if models., PVLDB 4 (12) (2011) 1486–1489.

[4] S. B. Katz, Y. Labrou, M. Kanthanathan, K. M. Rudin, Method for managing a workflow process that assists users in procurement, sourcing, and decision-support for strategic sourcing, uS Patent 7,870,012 (Jan. 11 2011).

[5] M. Rys, D. Chamberlin, D. Florescu, Xml and relational database management systems: the inside story, in: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, ACM, 2005, pp. 945–947.

[6] G. Fourny, JSONiq The SQL of NoSQL, CreateSpace Independent Publishing Platform, 2013.

[7] P. Fritzson, V. Engelson, Modelicaa unified object-oriented language for system modeling and simulation, in: ECOOP98Object-Oriented Programming, Springer, 1998, pp. 67–90.

[8] J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, H. Tummescheit, Modeling and optimization with optimica and jmodelica. orglanguages and tools for solving large-scale dynamic optimization problems, Computers & Chemical Engineering 34 (11) (2010) 1737–1749.

[9] Y. Zhang, C. Li, A principal-agent approach to incentive mechanisms in supply chains, in: Service Operations and Logistics, and Informatics, 2006. SOLI'06. IEEE International Conference on, IEEE, 2006, pp. 358–363.

[10] R. Fourer, D. M. Gay, B. W. Kernighan, AMPL: A mathematical programming language, AT&T Bell Laboratories Murray Hill, NJ 07974, 1987.

[11] R. E. Rosenthal, Gams–a user's guide.

[12] P. Van Hentenryck, L. Michel, L. Perron, J.-C. Régin, Constraint programming in opl, in: Principles and Practice of Declarative Programming, Springer, 1999, pp. 98–116.

[13] A. Guazzelli, M. Zeller, W.-C. Lin, G. Williams, Pmml: An open standard for sharing models, The R Journal 1 (1) (2009) 60–65.

[14] V. Jain, I. E. Grossmann, Algorithms for hybrid milp/cp models for a class of optimization problems, INFORMS Journal on computing 13 (4) (2001) 258–276.

[15] JavaScript Object Notation, World Wide Web, `http://json.org/`.

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

[17] A. Brodsky, M. Al-Nory, H. Nash, Sc-cojava: A service composition language to unify simulation and optimization of supply chains, in: Modeling for Decision Support in Network-Based Services, Springer, 2012, pp. 118–142.

[18] A. Brodsky, S. C. Mana, M. Awad, N. Egge, A decision-guided advisor to maximize roi in local generation & utility contracts, in: Innovative Smart Grid Technologies (ISGT), 2011 IEEE PES, IEEE, 2011, pp. 1–7.

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

[20] J. Luo, A. Brodsky, Piecewise regression learning in corejava framework, International Journal of Machine Learning and Computing 1 (2) (2011) 163–169.

[21] A. Brodsky, S. G. Halder, J. Luo, Dg-query: An xquery-based decision guidance query language, in: ICEIS 2014 - 16th International Conference on Enterprise Information Systems, SCITEPRESS – Science and Technology Publications, 2014, pp. pp–152.

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

[23] A. Brodsky, G. Shao, F. Riddick, Process analytics formalism for decision guidance in sustainable manufacturing, Journal of Intelligent Manufacturing (2013) 1–20.

[24] A. Alrazgan, A. Brodsky, Toward reusable models: System development for optimization analytics language (oal).

[25] N. Egge, A. Brodsky, I. Griva, An efficient preprocessing algorithm to speed-up multistage production decision optimization problems, in: System Sciences (HICSS), 2013 46th Hawaii International Conference on, IEEE, 2013, pp. 1124–1133.