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

Unity: A NoSQL-based Platform for Building Decision Guidance Systems from Reusable Analytics Models

Mohamad Omar Nachawati
mnachawa@gmu.edu

Alexander Brodsky
brodsky@gmu.edu

Juan Luo
jluo2@gmu.edu

Technical Report GMU-CS-TR-2016-4

Abstract

Enterprises across all industries increasingly depend on decision guidance systems (DGSs) to facilitate decision-making across all lines of business. Despite significant technological advances, current DGS development paradigms lead to a tight-integration of the analytics models, methods and underlying tools that comprise these systems, often inhibiting extensibility, reusability and interoperability. To address these limitations, this paper focuses on the development of the first NoSQL decision guidance management system (NoSQL-DGMS), called Unity, which enables decision-makers to build DGSs from a repository of analytics models that can be automatically reused for different analytics methods, such as simulation, optimization and machine learning. In this paper, we provide the Unity NoSQL-DGMS reference architecture, and develop the first implementation, which is centered around a modular analytics engine that symbolically executes and automatically reduces analytics models, expressed in JSONiq, into lower-level, tool-specific representations. We conduct a preliminary experimental study on the overhead of OPL optimization models automatically generated from JSONiq using Unity, compared with manually-crafted OPL models. Preliminary results indicate that the execution time of OPL models that are automatically reduced from JSONiq is within a small constant factor of corresponding, manually-crafted OPL models.

1 Introduction

Enterprises across all industries increasingly depend on decision guidance systems (DGS) to facilitate decision-making across all lines of business. DGSs are an advanced class of decision support systems (DSS) that are designed to provide actionable recommendations using a variety of different analytics models, algorithms and data. These systems are often built on top of a variety of different lower level tools that provide decision-makers with the full gamut of business analytics, from descriptive to diagnostic to predictive to prescriptive analytics [1]. While DSSs are traditionally classified into five different categories according to underlying technology, namely data-driven, model-driven, knowledge-driven, document-driven and communications-driven [2], state-of-the-art DGSs often combine multiple approaches into one integrated system to solve complex analytical problems [1].

However, despite significant technological advances, current DGS development paradigms lead to a tight-integration of the analytics models, tasks and underlying tools that comprise these systems, often inhibiting extensibility, reusability and interoperability. As stated in [1], the development of DGSs are typically one-off, hard-wired to specific problems, and usually require significant interdisciplinary expertise to build. This is similar to how database systems were developed long before the invention of the first DBMS. Consequently, DGSs end up being highly complex, costly, and non-extensible, and non-reusable. These deficiencies originate mainly from the diversity of the required analytics tools and algorithms, which are each designed for a specific analytics task, such as data manipulation, predictive what-if analysis, decision optimization, statistical learning and data mining. These tools each may require the use of a different mathematical abstraction and language to construct analytics models, which inhibits the interoperability such models across different analytics tools and tasks [1].

Overcoming the aforementioned difficulties faced when building DGSs is an important research problem [1]. These difficulties can be attributed to a diverse, low level abstractions provided by current paradigms, which preclude the reuse of analytics models across different analytics tasks. Thus the same underlying reality must often be modeled multiple times using different mathematical abstractions to support different tasks, instead of being modeled just once, uniformly [1]. Also, the modeling proficiency required by these languages is typically not within the realm of expertise of many of the users of a DGS, including the stakeholders, business analysts and application developers. Consequently, DGS development projects often require a team with diverse interdisciplinary expertise, are prone to budget overruns and

unexpected delays, and often result in software that is highly-proprietary, non-reusable, non-extensible, and locked-in to specific tool vendors [1].

To date, there has been extensive research focusing on the application of DGSs to solve complex problems in a variety of different domains [3]–[5]. However, far less effort has been expended on surmounting the above-mentioned obstacles encountered in the development of DGSs in general, and thus several research gaps still exist. An initial step forward was made in [6] with the introduction of a new type of platform that they refer to as a decision guidance management system (DGMS), which was designed to simplify the development of DGSs by seamlessly integrating support for data acquisition, learning, prediction, and optimization on top of the data query and manipulation capabilities typically provided by a DBMS [6]. While the work in [6] laid the foundation for additional research [1], [7]–[10], it did not address the technical challenges surrounding the development of a DGMS. In particular, the work in [6] did not provide any underlying algorithms to support the decision guidance capabilities provided by the proposed DGMS, such as simulation, optimization and learning. The proposed architecture was also limited to the relational model, and lacked support for developing analytics models on top of NoSQL data stores for semi-structured data in more flexible formats, such as XML or JSON. Furthermore, due to the inherent limitations of SQL, to repurpose the language for decision guidance modeling and analysis, a number of non-standard syntactic extensions were developed, collectively called DG-SQL. Introducing new language dialects, however, can break the interoperability of existing development tools and inhibit wide-spread adoption [11] and also affect the reusability of existing code [12].

Further progress was made in [1], [10] with the proposal of the Decision Guidance Analytics Language (DGAL) designed as an alternative to DG-SQL for developing DGSs over NoSQL data stores. Instead of SQL, DGAL is based on the more expressive JSON Query Language (JSONiq), which itself is based on XQuery. JSONiq is a popular query language for JSON document-oriented NoSQL data stores, and provides highly-expressive query capabilities centered around the original FLWOR construct of XQuery [13]. Rather than extending the syntax of an existing language, as what was done in DG-SQL, DGAL is, by design, syntactically equivalent to JSONiq. To support decision guidance, a number of analytics services, such as for optimization and learning, are proposed in [1]. While these services are exposed as regular functions in JSONiq, they require a non-standard interpretation of the language to implement. Thus, while DGAL is syntactically equivalent to JSONiq, the analytics services it provides have semantics that extend that of the JSONiq language. However, while [1] focused on proposing DGAL as a language for developing DGSs, it did not provide a reference architecture or implementation of a DGMS developed around DGAL, nor did it address the problem of compiling DGAL analytics models into lower-level, tool-specific representations.

Lifting the aforementioned limitations is exactly the focus of this paper. Specifically, the contributions of this paper are as follows. First, we propose a reference architecture for DGSs that is based on Unity NoSQL-DGMS operating as a middleware to connect higher-level decision guidance applications and their clients to the lower-level tools needed for supporting different analytics tasks. The uniqueness of this architecture is that it centered around a knowledge base of DGAL analytics models, which can be reused for various analytics tasks such as prediction, optimization and statistical learning without the need to manually create lower level task-specific models.

Second, we develop Unity NoSQL-DGMS, the first system of its kind, which is designed to enable decision-makers to build DGSs from a repository of DGAL analytics models that can be automatically reused for different analytics tasks. Unity's uniqueness lies in its core analytics services, including optimization and learning, which do not require lower-level level models (e.g., in AMPL for optimization problems), but rather automatically generate the lower-level task- and tool-specific models from the higher level task- and tool-independent analytics models in the AKB.

Third, as part of Unity NoSQL-DGMS, we develop algorithms for its execution engine based on a symbolic computation to reduce analytics models in the knowledge base into a JSON-based intermediate representation. This intermediate representation is then translated into tool-specific representations. The execution engine is used to provide analytics services for deterministic optimization against DGAL analytics models using mathematical programming (MP). To implement these services, we develop a code generator from the intermediate representation that targets both AMPL and OPL to support a wide range of different optimization solvers.

Finally, we conduct a preliminary experimental study on the overhead of automatically generated task- and tool-specific models. Our evaluation is currently limited to just the execution time overhead of OPL optimization models that are automatically generated from DGAL analytics models using the deterministic optimization service developed in this paper. Initial results indicate that the execution time of automatically derived OPL optimization models is within a constant factor of the execution time of corresponding, manually-crafted OPL optimization models.

The rest of this paper is organized as follows. In the next section, we present the Unity NoSQL-DGMS reference architecture, and describe each of its major layers and components. In the following section, we provide an overview of the DGAL language, and show how to use develop a DGS using DGAL and Unity. Then we move on to describe the prototype implementation of Unity, to include the DGAL execution engine, intermediate representation, and implementation of the argmin and argmax services for performing deterministic optimization against DGAL analytics models. Finally, we describe the preliminary experimental study and then conclude the paper with some brief remarks on future work.

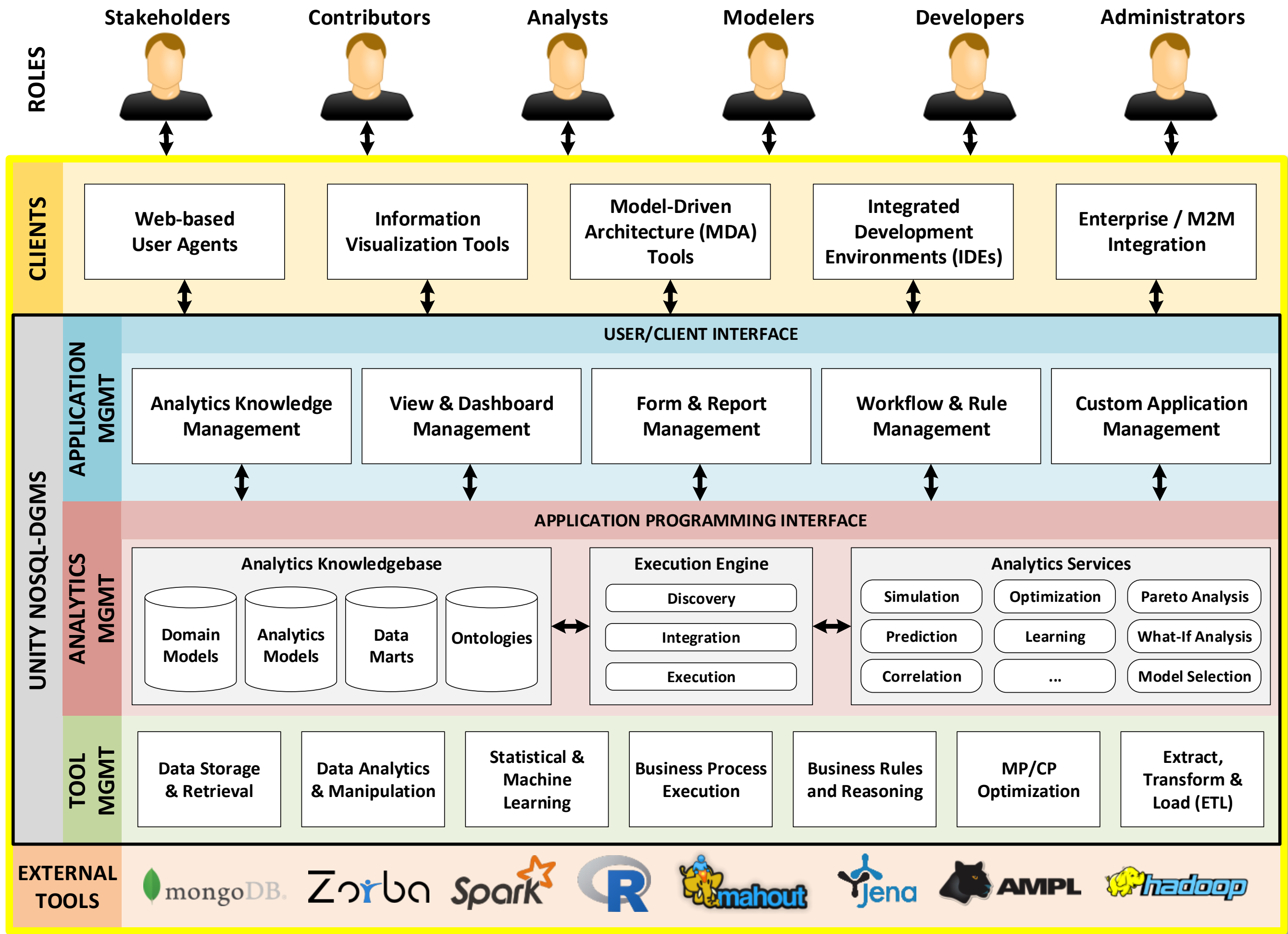

Figure 1. Unity NoSQL-DGMS Reference Architecture

2 Unity NoSQL-DGMS Reference Architecture

In this section, we describe the Unity NoSQL-DGMS reference architecture, which is depicted in figure 1. The boundary of Unity is enclosed in a black rectangle in the diagram. Unity serves as a middleware between the client layer and external tool layer of a typical DGS, and is internally comprised of three-layers, namely the application management layer, the analytics management layer, and the tool management layer. As a middleware, Unity simplifies connecting these different clients and the users they support to external tools. Similar to the original DGMS vision presented in [6], Unity is designed to provide seamless support for data acquisition, learning, prediction, and optimization. However, unlike former, which uses DG-SQL for analytics modeling and analysis, Unity replaces the role of DG-SQL with DGAL. Specifically, in the Unity, DGAL serves as both a language for defining reusable analytics models, and for executing analytics services against those models. As a middleware, Unity provides DGAL as a unified abstraction to hide the complexities of dealing with a diverse range of lower-level tools that are needed to implement a DGS.

Unity supports six primary user roles typical of a DGS, namely stakeholders, contributors, analysts, modelers, developers, and administrators. The roles are not mutually exclusive, and therefore a single user may serve different roles. Stakeholders consist of decision-makers at all levels of an organization, from supervisors to the board of directors. A stakeholder is the end-user of the actionable, decision-guided recommendations provided by a particular DGS. Contributors provide data-entry and manage the domain models, data marts, and ontologies in the analytics knowledgebase. Analysts are advanced business users that manage the analytics models of the knowledgebase, and additionally is able to design new views, dashboards, forms and reports. Analysts can also design workflows and business rules for automating repetitive decisions based on the actionable recommendations provided by a particular DGS. Modelers are technical users with an operations research background that develop the metrics and constraints equations for new analytics models. While an analyst may not have the technical expertise to develop new analytics models from scratch, they can mash up new models by composing and specializing existing models in the knowledge base. Developers are technical users with experience in software

development that build custom applications that provide extended functionality to meet domain-specific requirements.

We envision that Unity would support client layer of diverse, domain-specific clients that used in conjunction with DGSs to support the different types of user that interact with a DGS. Unity provides out-of-the-box support for web-based user agents for managing DGSs developed using Unity. Unity also provides a REST-based API for integrated third-party clients ranging from information visualization tools, model-driven architecture (MDA) tools, integrated development environments and for integrating with other information systems. Another objective Unity is to facilitate the development of DGSs by allowing users to work at different levels of abstractions according to their skills and expertise. These abstractions support the seamless integration of different analytical models, tools and data. For example, modelers and developers can use the full power of mathematical constraints in DGAL to create reusable analytics models, which are interoperable and composable and extensible, while analysts and non-technical users are able to compose these expertly crafted analytics models to precisely model problems they need to analyze.

The application management layer provides a number of services to support the rapid development of DGS. Analytics knowledge management supports the creation, querying, and modification of the different artifacts housed in the analytics knowledgebase. View and dashboard management provides tools for creating analytics views and templates for the rapid development of interactive dashboards. Analytics views are similar to regular database views, except that they are based on one or more analytics models and services. Form and report management supports the development and use of forms for data collection and reporting. Workflow and rule management supports the development and execution of rules and workflows for building automated decision systems (ADS). Finally, custom application management provides tools for building domain-specific DGS-based applications.

The analytics management layer hides the complexity of dealing with the different external tools that provide the essential analytics and other capabilities of a DGS. This layer is comprised of the analytics knowledge base, the execution engine and a variety of different analytics services. At the core of the architecture, the knowledge base provides uniform access to the different analytics artifacts that together constitute the domain-specific knowledge used for decision guidance. The different types of analytics artifacts include, but are not limited to analytics models, domain models and instances, ontologies.

The Unity reference architecture is designed to support three kinds of analytics models, namely white-box, black-box and grey-box models. With white-box analytics models, the DGAL source code of the analytics model is stored in the knowledge base, and the execution of such models is performed locally by the execution engine. While white-box analytics models can help decision-maker better understand the logic behind the computation of metrics and constraints, they would not be suitable for models containing proprietary knowledge. On the other hand, with black-box analytics models the DGAL source code is not provided, and instead are stored in the knowledge base as web service descriptions, were the execution of such models would occur remotely. While this can support proprietary models, it does not provide a way for the client to reuse the models for different analytics tasks. It also requires users to send possibly sensitive data to third parties for processing. Finally, gray-box models are like black-box models in the execution occurs remotely, however the gray-box models are capable of returning its results in symbolic form. While this exposes part of the logic at a low level (akin to assembly), it allows clients to easily reuse remote models for different types of analytics.

The execution engine serves as a bridge between analytics models in the knowledge base, analytics services, such as simulation and optimization, and the lower-level tool used to implement them. The execution engine is built on top of external tools, such as Zorba for JSONiq query processing and AMPL for MP-based optimization, which are managed in the tool management layer. The execution engine includes a compiler for translating DGAL analytics model into lower-level, tool specific models. Rather than directly translating DGAL analytics models into different tool-specific models, such as AMPL, the approach we use to implement the analytics model compiler involves a symbolic execution using a standard JSONiq query processor to first lower the analytics model in DGAL into a simpler, JSON-based intermediate representation. We discuss the details of the intermediate representation in the following section. While JSONiq query processors support complex data queries and even simple analytical operations they do not directly support the advanced analytics services that DGAL provides, such as optimization and learn. Executing DGAL queries that depend on such services require the use of specialized algorithms to implement, which are often readily available as third-party tools. By utilizing a simpler intermediate representation, support for new third-party tools can be developed without having to re-implement the entire DGAL language.

Finally, the tool management layer manages the external tools that are needed to implement the various analytics services, as well as to provide other capabilities of the NoSQL-DGMS. It provides a uniform access to the different tools, ranging from data storage and retrieval, data analytics and manipulation, statistical and machine learning, MP/CP optimization and business process and rule execution and reasoning.

3 Reusable Analytics Modeling with Unity and DGAL

In this section we provide a brief overview of DGAL and show how one can use DGAL in conjunction with Unity to develop a reusable analytics model for order analytics to support the development of intelligent supply chain management systems. In the interest of space, we will limit our discussion to analytics modeling, simulation and deterministic optimization. For more information on DGAL and other analytics services, such as statistical learning and stochastic optimization, we refer the reader to our work in [1].

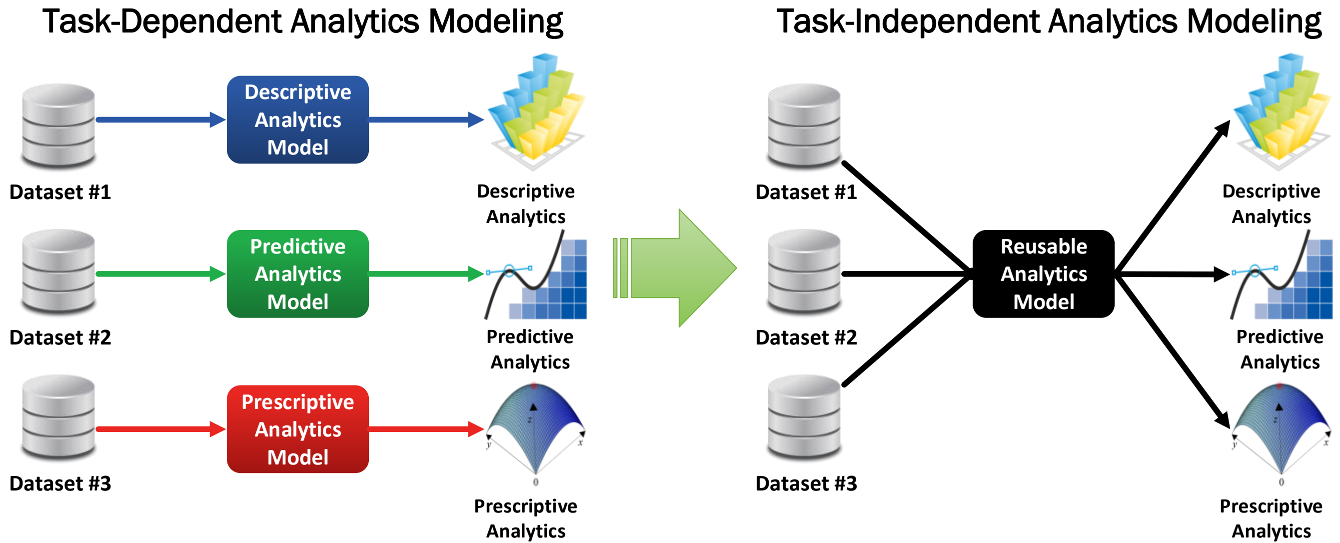

Figure 2. Paradigm Shift from Task-Dependent to Task-Independent Analytics Modeling

As mentioned earlier, current DGS development paradigms lead to a tight-integration of the analytics models, tasks and underlying tools that comprise these systems, often inhibiting extensibility, reusability and interoperability. As illustrated on left in figure 2, the task-specific analytics modeling paradigm requires a specialized analytics model for each analytics task, such as those for descriptive, predictive or prescriptive analytics. This inhibits composition, specialization, generalization and reuse of analytics models. Overcoming these limitations by way of a paradigm shift from non-reusable, task-dependent modeling to task-independent modeling was the motivation for the development of DGAL [1]. As shown on the right in figure 2, DGAL supports a task-independent approach to modeling analytics knowledge where a single model can be used for multiple analytics tasks, such as simulation, optimization and learning.

DGAL is based on the JSON Query Language (JSONiq), which itself is based on XQuery. JSONiq is a popular query language for JSON document-oriented NoSQL data stores, and provides highly-expressive query capabilities centered around the original FLWOR construct of XQuery [13]. Rather than extending the syntax of JSONiq, DGAL is by design, syntactically equivalent to JSONiq. To support decision guidance, a number of analytics services, such as for optimization and learning, are proposed in [1]. While these services are exposed as regular functions in JSONiq, they require a non-standard interpretation of the language to implement. Thus, while DGAL is syntactically equivalent to JSONiq, the analytics services it provides have semantics that extend that of the JSONiq language.

In DGAL, an analytics model is implemented as a regular function in JSONiq. Such function must accept its input as a single JSON object, and must return a JSON object that contains a top-level `constraint` property evaluating to true if the constraints of the model are satisfied for a particular input and false otherwise. It can also contain any number of metrics, which are numerically or logically-typed properties that are computed and derived from the input to the function. The input to the function can have certain numerically or logically-typed properties replaced with decision variables or learning parameters, which can then be solved for by invoking one or more of the analytics services provided by Unity. Restrictions on the properties of the input that can be replaced with decision variables or learning parameters depend on the analytics service invoked. In the case of deterministic optimization, for example, decision variables are restricted to only those properties that contribute to the computation of either the objective metric or constraint. For more details on DGAL analytics models we direct the reader to [1].

```
"suppliers": [{
  "sid": "supplier1",
  "supply": [
    { "upc": "47520-81452", "ppu": 10.99, "qty": 500 },
    { "upc": "32400-24785", "ppu": 19.99, "qty": 400 }]
  },{
  "sid": "supplier2",
  "supply": [
    { "upc": "47520-81452", "ppu": 11.99, "qty": 1500 },
    { "upc": "32400-24785", "ppu": 18.99, "qty": 1295 },
    { "upc": "14752-47748", "ppu": 29.99, "qty": 2500 }]
  }],
"customers": [{
  "cid": "customer1",
  "demand": [
    { "upc": "47520-81452", "qty": 1475 },
    { "upc": "14752-47748", "qty": 475 }]
  },{
  "cid": "customer2",
  "demand": [
    { "upc": "32400-24785", "qty": 874 },
    { "upc": "47520-81452", "qty": 254 },
    { "upc": "14752-47748", "qty": 987 }]
  }],
"orders": [{
  "sid": "supplier1",
  "cid": "customer1",
  "items": [ { "upc": "47520-81452", "qty": 500 } ]
  }, ...
```

In our ordering system scenario, we track a group of suppliers that each supply zero or more items, as well as a group of customers that each have a demand for zero or more items. We also maintain a list of orders that represent the flow of items from suppliers to customers. We can represent this information using a JSON object that will serve as the input to our DGAL analytics model, an example of which is shown above. Based on the data model that can be derived from the above example, we can now define the metrics and constraints for our analytics model that we will then proceed to implement. While a single analytics model can support multiple metrics, for the purposes of our discussion we will limit our model to only compute the total cost of all orders, which can be computed using the following JSONiq expression (assuming that the variable $input holds the input to our analytics model):

```
let $orders := $input.orders[]
let $items := $input.items[]
let $suppliers := $input.suppliers[]

let $cost :=
  for $order in $orders, $item in $items
  return
      fn:sum(
        $suppliers[$$.sid eq
$order.sid].supply[][$$.upc eq $item.upc].ppu *
$item.qty
      )
```

We now need to define the constraints of our analytics model. In our order analytics model, we have two basic constraints. The first constraint is a supply constraint on orders that stipulates that for each supplier, the quantity of each item in stock is greater than or equal to the sum of the order quantities of that item across all orders to that supplier. The second constraint is a demand constraint on orders that stipulates that for each customer, the quantity of each item requested is equal to the sum of the order quantities of that item across all orders from that customer. We can express these constraints in JSONiq as follows:

```
let $suppliers := $input.suppliers[]
let $customers := $input.customers[]
let $orders := $input.orders[]

let $supplyConstraint :=
 for $supplier in $suppliers, $item in
$supplier.supply[]
 return $item.qty ge fn:sum($orders[$$.sid eq
$supplier.sid].items[][$$.upc eq
$item.upc].qty)

let $demandConstraint :=
 for $customer in $customer, $item in
$customer.demand[]
 return $item.qty eq fn:sum($orders[$$.cid eq
$customer.cid].items[][$$.upc eq
$item.upc].qty)
```

We finish the implementation of our DGAL analytics model for order analytics by wrapping the JSONiq expressions for computing metrics and constraints inside a JSONiq function:

```
declare function
scm:OrderAnalyticsModel($input)
{
  let $cost := ...
  let $supplyConstraint := ...
  let $demandConstraint := ...
  let $constraints :=
    $supplyConstraint and $demandConstraint
  return {
    cost: $cost
    constraints: $constraints
  }
};
```

Using our reusable analytics model for order analytics, implemented in DGAL, we can perform a variety of different tasks, such as simulation, optimization and learning, without having to redevelop new task-specific models for individual task. The work of reducing DGAL analytics models into tool-specific models for execution and analysis is handled seamlessly by Unity. Simulation in DGAL involves the computation of the metrics and constraints for an input object. We can compute the metrics and constraints by simply invoking the `scm:OrderAnalyticsModel` function on that input object:

```
scm:OrderAnalyticsModel($input)
```

In this case, the output JSON object that is returned from the invocation of that function contains only numerically or logically-typed values for metric properties, and a value of either true or false for the constraints property, depending if the constraints were satisfied for the given input object.

What if we wanted to find the optimal item order quantities, `qty`, for each supplier such that the total cost is minimized? To do this, we can annotate our original input object with decision variable objects in place of numeric values for each `qty` property to indicate that we want Unity to solve for the values of those properties. A decision variable object is a JSON object that contains one of the following properties corresponding to its type: `integer?`, `decimal?`, or `logical?`. The corresponding property value indicates the decision variable identifier, which if set to `null` will replaced with a UUID. Two different decision variable objects that contain identical identifiers refer the same decision variable in the underlying optimization problem. The decision variable annotated input is shown below:

```
"orders": [{
    "sid": "supplier1",
    "cid": "customer1",
    "items": [ { "upc": "47520-81452", "qty": {
"integer?": null } } ]
}]
```

Invoking the `scm:OrderAnalyticsModel` function directly on the decision variable annotated input would, however, result in undefined behavior. This is because the function that implements the analytics model is expecting a numerically-typed value for the `qty` property, but we provide a decision variable object instead. Rather, we can invoke an analytics service provided by Unity for deterministic optimization against the analytics model and annotated input to find specific values for `qty` that minimizes the `cost` metric as follows:

```
let $ instantiatedInput := dgal:argmin({
  varInput: $annotatedInput,
  analytics:
"Q{http://example.org/scm}OrderAnalyticsModel",
  objective: "cost"
})
```

In maintaining complete syntactic equivalence with JSONiq, all analytics services provided by Unity are exposed as a regular JSONiq functions by DGAL. For deterministic optimization, DGAL provides the `dgal:argmin` function which simply serves as a wrapper around the analytics service for deterministic optimization that is provided by Unity. The `dgal:argmin` function takes single JSON object as input that contains at least three properties, specifically: (1) varInput: the decision variable annotated input as a JSON object, (2) analytics: the analytics model as a qualified name string, and (3) objective: the JSONiq path expression string to select the property of the objective metric that is computed by analytics model. If a solution to the resulting optimization problem is feasible, the `dgal:argmin` function returns an instantiation of the annotated object contained in the varInput property, where all decision variable annotations are instantiated with values that together minimize the objective metric. To compute the minimized value of the objective metric, one simply invokes the analytics model on the instantiated input object returned from `dgal:argmin`:

```
return
Q{http://example.org/scm}OrderAnalyticsModel($i
nstantiatedInput)
```

4 System Implementation

In this section we describe a prototype implementation of Unity that follows the reference architecture presented earlier. The Unity prototype was developed using a combination of Java, C++ and JSONiq and currently supports for computation and deterministic optimization of analytics models implemented in DGAL. We use Java Content Repository (JCR), as implemented in Apache Jackrabbit, to store, retrieve and manage the analytics knowledge base artifacts, such as analytical models, views and data. To simplify the development of DGAL analytics models, we developed an initial DGAL IDE based on Eclipse, as shown in figure 3, as well as a package for Atom containing a macro for executing DGAL queries from within the IDE.

The execution engine was developed on top of the Zorba query processor, which was adapted to operate against the JCR content repository, rather than from the local file system. For this purpose, we developed a custom URI resolver for Zorba in C++. For the application layer, we use Jetty as an embedded HTTP server and Java servlet container to host a custom developed Web application that provides support for executing DGAL queries and manipulating and managing

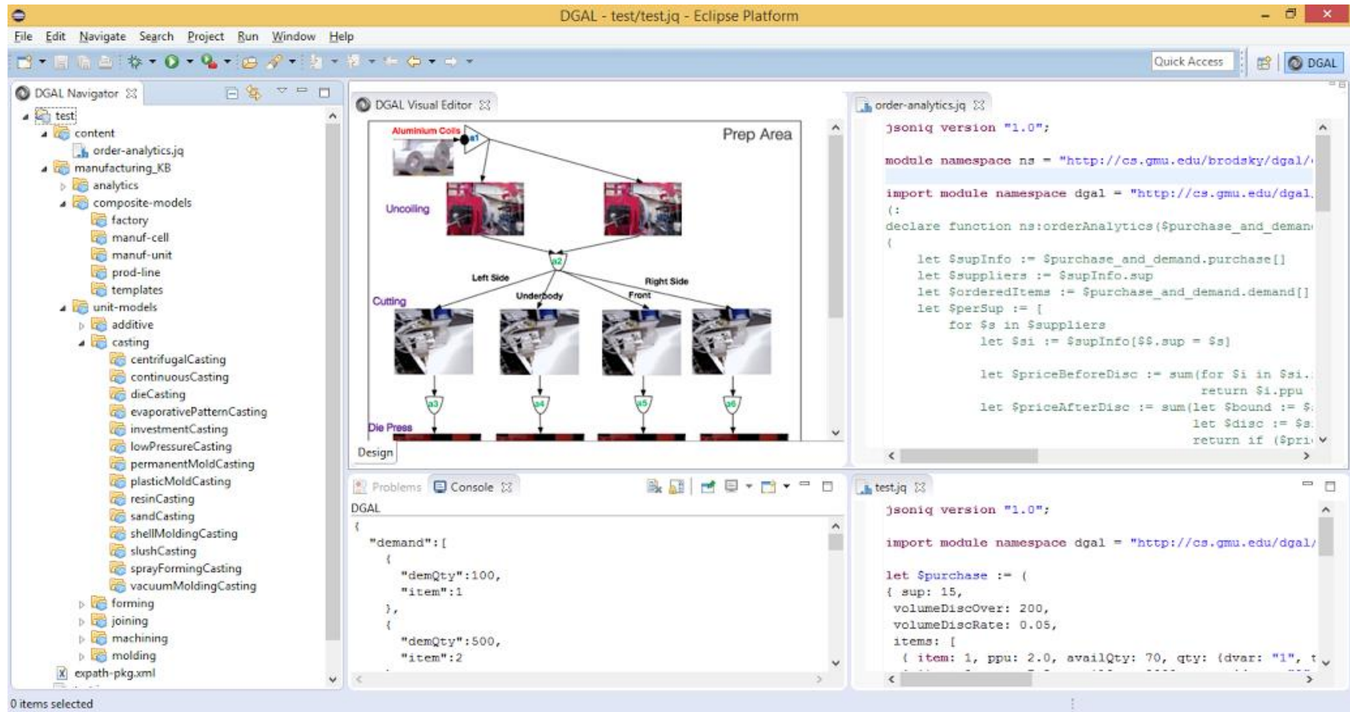

Figure 3. Eclipsed-based IDE for DGAL Analytics Model Development

analytics models and data in the repository. For the client-tier, the Unity DGMS prototype provides remote access through a number of different protocols, including HTTP/HTML, REST/JSON and WebDAV.

The process provided by Unity that implements the deterministic optimization function, `dgal:argmin`, consists of 6 steps, as shown in figure 4. The process begins with the analytics model resolution step, where in the fully qualified analytics model name is resolved against the content repository to retrieve its implementation as a DGAL source module. Next, in the source-to-source transformation step, the DGAL source module is transformed into a symbolically executable JSONiq library module. Then in the symbolic execution step, the transformed module is executed as a regular JSONiq module by the Zorba engine, which generates its output in an intermediate representation. Next in the solver-specific model generation step, the output from the previous step is used to generate a solver-specific model with associated data. Currently the prototype can generate optimization models in either OPL or AMPL. The newly generated model is then dispatched, in the solver specific execution step, to the solver specified in the configuration object, such as CPLEX, MINOS or SNOPT. Finally, in the analytics model input instantiation step the solution obtained from the solver is merged with the decision variable annotated input to return an instantiated input, where all decision variables are replaced with the resultant values in the solution.

The prototype implementation is built around an intermediate representation for analytics models. The intermediate representation is a JSON-based language that captures a partially translated analytics model in a way that is independent of both the source modeling language and the target, tool-specific language. While we currently support DGAL for analytical modeling, other languages like DGQL [7] could also be supported with the development of an analytics compiler to transform models in other languages into this intermediate representation.

In the intermediate representation, mathematical expressions whose values depend on decision variables or learning parameters are encoded as symbolic expression objects, which are JSON objects that capture the abstract syntax tree of the expression to be computed, rather than its computed result. Decision variables are represented in the intermediate representation as simple JSON objects that capture the variable's name, type and optionally its estimated value, which is often crucial for non-linear optimization tasks. The property name of a decision variable object indicates the type of the decision variable and the property value is a string that holds the identifier of the decision variable. Unlike in some optimization modeling languages, such as AMPL and OPL, decision variables in the intermediate representation are not explicitly defined, rather they are implicitly defined as part of their usage. For this reason, care must be taken to ensure that if multiple decision variable symbols with the same identifier are used within a single intermediate representation model, the decision variable types must all be consistent. Just like decision variables, learning parameters are represented in the intermediate representation as simple JSON objects that capture the parameter's name, type and optionally its estimated value.

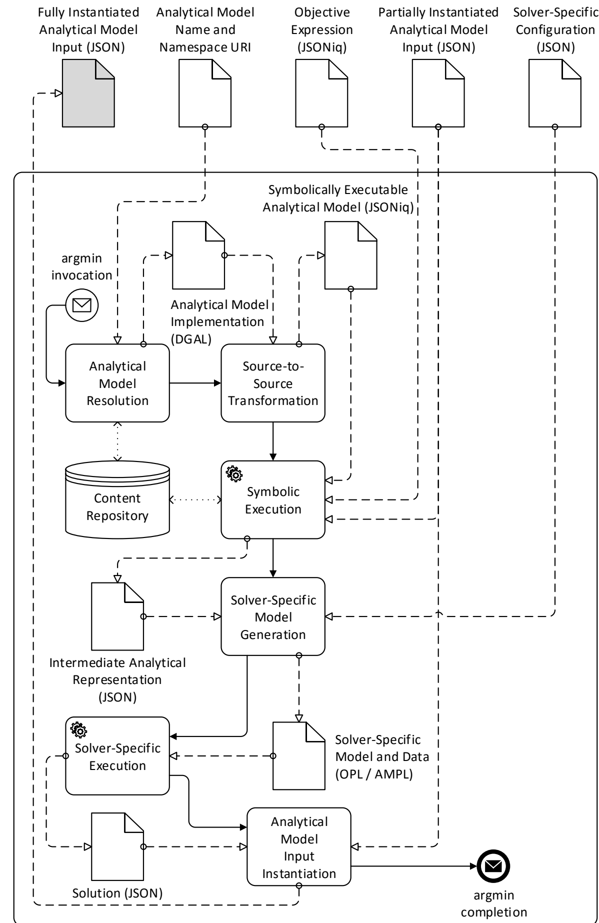

Figure 4. Deterministic Optimization Process (`dgal:argmin`)

Expressions are encoded as single-property JSON objects where the property name indicates the expression operator and the corresponding value is a JSON array containing the values of the operands. The intermediate analytical representation supports many kinds of expression operators, including arithmetical, logical, conditional, quantified expression operators. While user-defined functions are currently not supported, Unity DGMS provides a number of built-in functions, such as aggregation and piecewise functions. For instance, the JSONiq expression `100 + 250 eq 350` can be encoded in the intermediate representation as follows:

```
{
    "==": [
        { "+": [ 100, 250 ] },
        350
    ]
}
```

While this expression is valid in the intermediate representation, the analytics engine automatically reduces

expressions that do not depend on any decision variables or learning parameter, and thus can be computed at the time of symbolic execution. For this expression, the value can be reduced to simply `true`.

As explained before, while syntactically DGAL is backwards compatible with JSONiq, the execution of analytics services extends the semantics of JSONiq. Because of this difference, analytics services cannot be directly executed on a standard JSONiq query processor, such as Zorba. One way to support the alternative execution semantics of DGAL is to re-develop a new JSONiq query processor that natively supports DGAL. However, as the objective of Unity is to promote interoperability and reuse, we opted for a different approach. If JSONiq supported operator overloading, like in C++, another approach would be to overload the expression operators supported by DGAL to re-define their execution semantics. For descriptive analytic tasks that are supported directly in JSONiq, the execution semantics would remain unchanged. For predictive or prescriptive analytic tasks, however, the execution of these overloaded operators would generate results in the intermediate analytical representation Unfortunately, JSONiq does not currently support operator overloading.

To support advanced analytics using a stock JSONiq query processor, the analytic model compiler first performs a source-to-source transformation to redefine the execution semantics of expression operators by direct modification of the source code. The main idea behind this approach is that for each expression operator in the analytical model, a function call is substituted in its place that when called returns the result in the intermediate analytical representation. In cases where the computation of an expression does not involve a decision variable or learning parameter, the intermediate analytical representation would be identical that of a standard interpretation of JSONiq.

As shown in figure 5, to perform the source-to-source transformation we first parse the JSONiq source text and build an abstract syntax tree. For this purpose, we used the REx parser generator [14] to generate a JSONiq parser in Java from the EBNF grammar that is provided in the JSONiq language specification. We then use an XSLT transformation on the resulting XML syntax tree to replace each JSONiq expression operator with a corresponding function that returns its result in the intermediate analytical representation.

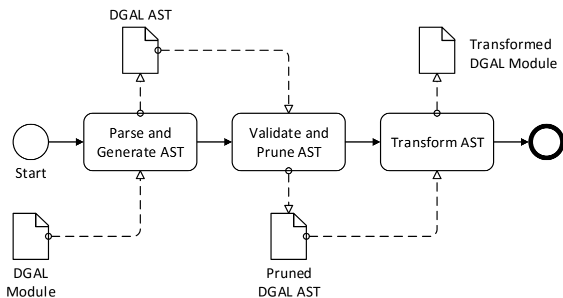

Figure 5. DGAL Source-to-Source Transformation Process

For example, consider the supplyConstraint expression from supply chain analytics example presented in section 2. After performing the source-to-source transformation, the sub-expression $supplyItem.qty ge fn:sum(...) is replaced with a calls to the corresponding dg:ge and dg:sum functions, as shown below:

```
let $supplyConstraint :=
for $supplier in $input.suppliers[],
    $supplyItem in $supplier.supply[]
return dg:ge($supplyItem.qty,
         dg:sum($input.orders[][dg:eq($$.sid,

$supplier.sid)].items[][dg:eq($$.upc,
$supplyItem.upc)].qty))
```

All such expression functions, like dg:ge and dg:sum, are implemented completely in JSONiq. The complete JSONiq definition of the dg:eq function is provided below:

```
declare function dg:eq($operand1, $operand2)
{
    if ($operand1 instance of object
     or $operand2 instance of object) then
      { "==": [$operand1, $operand2] }
    else
      $operand1 eq $operand2
};
```

It takes two parameters, $operand1 and $operand2, which correspond to the left and right operands of the binary equality operator in JSONiq. A quick simplification is done in the event that neither operand depends on a decision variable or learning parameter, whereby the fully computed result is returned, otherwise an intermediate analytical expression tree object is returned.

5 Experimental Evaluation

While the Unity NoSQL-DGMS prototype that we described in the previous section demonstrates one approach for implementing deterministic optimization against analytics models expressed in DGAL, a vital question is whether our approach is too computationally inefficient to be used for real-world, DGSs. In this section, we report on a preliminary experimental study that we conducted to investigate the overhead of OPL optimization models automatically generated from DGAL using Unity, compared with manually-crafted OPL models that are formally-equivalent. Our hypothesis is that for OPL models that are automatically generated from DGAL, the solve time is within a small constant factor of the time it takes to solve manually crafted OPL models that are formally equivalent. To test our hypothesis, we manually translated an AMPL optimization model for procurement analytics, which we borrow from [8], into two formally equivalent models, one in DGAL and the other in OPL, which we refer to as the experimental input and control input respectively.

A program was written to randomly generate pairs of instance data to use for our DGAL and OPL models, which

we refer to as the experimental input and control input respectively. Care was taken to ensure that each input pair was formally equivalent, or in other words the same randomly generated value was used for each corresponding property in the input. We conducted a total of 205 trials, where for each trial we measured the wall-clock time that the CPLEX solver took to solve the experimental optimization problem and the control optimization problem. We only measured the difference in how long it takes CPLEX to solve the experimental optimization problem versus the control problem, so we do not include the time spent on other associated processes in our measurements, such as the time spent compiling DGAL models and loading data into CPLEX.

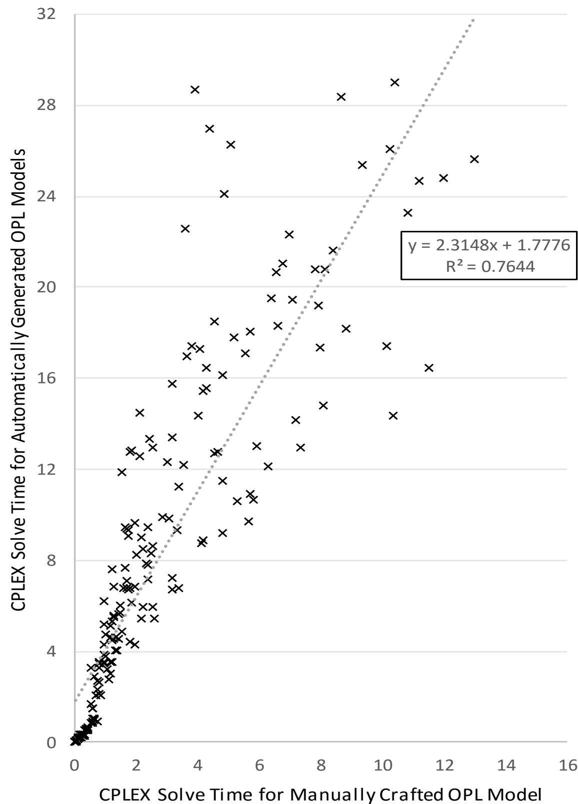

Figure 6. Execution Time Overhead of Automatically Generated OPL Models from DGAL (in seconds)

The Java API provided by CPLEX was used to automate the execution of the experiment's 205 trials. The wall-clock time was measured by subtracting the difference between the return values of calls to System.currentTimeMillis() placed immediately after and before the invocation of the CPLEX solve() method. Within a trial, the experimental optimization problem consisted of the OPL model and input that was automatically generated through compilation of the experimental DGAL model on the experimental input. The control optimization problem consisted of the control OPL model and the control input. The number of decision variables in the resulting optimization problems across all trials range from 72 to 16,800. The tests were conducted on a single machine equipped with an Intel® Core™ i5-4210U processor and 16GB of RAM. To reduce measurement errors due to interference, we closed all other extraneous applications and services, and each test was run sequentially.

The results of this experiment are presented as a scatter chart in figure 6, where the horizontal axis represents the wall-clock time that the CPLEX solver took, in seconds, to solve the control optimization problem, and the vertical axis shows the wall-clock time that the CPLEX solver took, in seconds, to solve the experimental optimization problem. A linear trend line through the time measurement points gives a slope of 2.3148, which indicates that compiled DGAL models are on average about 231% slower than manually crafted OPL. On closer inspection, a value of 0.7644 for the coefficient of determination, indicates that almost 24% of the variance in time measurements is not explained by the independent variable of our experiment, which was the optimization model used, either the OPL model automatically generated from DGAL or the manually crafted one. Some variance is to be expected because the behavior of underlying algorithms used for MP-based optimization, such as branch and bound, are often sensitive to how the problem is formulated.

While our objective is not to compete with commercial solvers in terms of faster execution times, but rather to develop a NoSQL-DGMS based around DGAL to facilitate the development of DGSs, there are a number of techniques that could be employed to decrease the overhead of automatically generated models. The CPLEX solver provides a number of options to fine-tune the optimization process, like pre-solve, which need to be investigated in the future. Also, utilizing a combination of domain-specific pre-processing techniques, such as the one proposed in [4], to generate efficient, tool-specific models for certain classes of problems. However, with regards to our preliminary evaluation, we view the current optimization time overhead as a standard tradeoff between user productivity and computational efficiency. Thus, Unity NoSQL-DGMS would still be practical in cases where computational efficiency can be sacrificed to avoid the costly re-development of specialized analytics models to support different tasks.

6 Conclusion and Future Work

This paper reported on the development of the first NoSQL decision guidance management system (NoSQL-DGMS) – Unity, which enables decision-makers to build DGSs from a repository of analytics models that can be automatically reused for different analytics methods, such as simulation, optimization and machine learning. We provided the Unity NoSQL-DGMS reference architecture, and developed its first implementation, which is centered around a modular analytics engine that symbolically executes and automatically reduces analytics models, expressed in DGAL, into lower-level, tool-specific representations. We also demonstrated the use of Unity and DGAL in building a simple DGS for order analytics. Finally, we conducted a preliminary experimental study on the overhead of OPL optimization models automatically generated from DGAL using Unity, compared with manually-crafted OPL models

that are formally-equivalent. Preliminary results indicate that the execution time of OPL models that are automatically reduced from DGAL is within a small constant factor of corresponding, manually-crafted OPL models.

The possibility of building DGSs from reusable analytics models by way of a NoSQL-DGMS represents a paradigm shift from the current approaches, which are task-dependent and lead to a tight-integration of the analytics models, methods and underlying tools. Our work opens up new research questions that we plan to work on in the future. Particularly, as support for additional analytics tools are developed, the problem of choosing the most appropriate tool for a specific task, such as optimization, emerges. Further research is needed to investigate how a tool recommender system can be developed to determine the set of candidate tools feasible for a particular analytics model and task, and then to rank the set of candidate tools according to some user-defined objective, such as predicted execution time or accuracy. Second, more work is necessary to develop algorithms to automatically reduce the complexity of analytics models using techniques such as domain-specific heuristics, pre-processing and relaxation. This is essential because in DGAL analytics models can use the full expressive power of JSONiq, which can lead to the development of analytics models are too complex to be solved efficiently using MP-based techniques alone.

References

[1] A. Brodsky, J. Luo, and M. O. Nachawati, “Toward Decision Guidance Management Systems: Analytical Language and Knowledge Base,” Department of Computer Science, George Mason University, 4400 University Drive MSN 4A5, Fairfax, VA 22030-4444 USA, GMU-CS-TR-2016-1, 2016.

[2] D. J. Power, “Supporting decision-makers: An expanded framework,” in *e-Proceedings Informing Science Conference, Krakow, Poland*, 2001, pp. 431–436.

[3] H. Altaleb and A. Brodsky, “A Primary Market for Optimizing Power Peak-Load Demand Limits,” *Int. J. Decis. Support Syst. Technol. IJDSST*, vol. 5, no. 2, pp. 21–32, 2013.

[4] N. Egge, A. Brodsky, and I. Griva, “Distributed Manufacturing Networks: Optimization via Preprocessing in Decision Guidance Query Language,” *Int. J. Decis. Support Syst. Technol.*, vol. 4, no. 3, pp. 25–42, 33 2012.

[5] C.-K. Ngan and A. Brodsky, “DGLS System: Decision Guidance for Optimal Load Shedding in Electric Power Microgrids,” in *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, 2013, p. 1.

[6] A. Brodsky and 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*, 2008, pp. 71–71.

[7] A. Brodsky, N. Egge, and X. S. Wang, “Reusing relational queries for intuitive decision optimization,” in *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, 2011, pp. 1–9.

[8] A. Brodsky, N. E. Egge, and X. S. Wang, “Supporting agile organizations with a decision guidance query language,” *J. Manag. Inf. Syst.*, vol. 28, no. 4, pp. 39–68, 2012.

[9] A. Brodsky, S. G. Halder, and J. Luo, “DG-Query, XQuery, mathematical programming,” in *16th International Conference on Enterprise Information Systems (ICEIS 2014)*, 2014.

[10] A. Brodsky and J. Luo, “Decision Guidance Analytics Language (DGAL) - Toward Reusable Knowledge Base Centric Modeling:,” 2015, pp. 67–78.

[11] R. Lammel and C. Verhoef, “Cracking the 500-language problem,” *IEEE Softw.*, vol. 18, no. 6, pp. 78–88, 2001.

[12] B. Shneiderman, “Experimental testing in programming languages, stylistic considerations and design techniques,” in *Proceedings of the May 19-22, 1975, national computer conference and exposition*, 1975, pp. 653–656.

[13] J. Robie, G. Fourny, M. Brantner, D. Florescu, T. Westmann, and M. Zaharioudakis, “Jsoniq the complete reference,” 2015.

[14] G. Rademacher, *REx Parser Generator*. .

Appendix 1 – Experimental DGAL Analytics Model

```
jsoniq version "1.0";
module namespace ns = "http://cs.gmu.edu/dgal/procurementAnalytics.jq";
declare function ns:procurementAnalytics($procurementData)
{
  let $Vendors := $procurementData.Vendors[]
  let $Items := $procurementData.Items[]
  let $Locations := $procurementData.Locations[]
  let $Stock := $procurementData.Stock[]
  let $Order := $procurementData.Order[]
  let $total_cost := sum(for $v in $Vendors, $i in $Items, $l in $Locations
                         return $Stock[$$.Vendor eq $v].Items[][$$.Item eq $i].PricePerUnit *
                                $Order[$$.Location eq $l].Items[][$$.Item eq $i].ItemOrder[][$$.Vendor
eq $v].Orders)
  let $available_vs_purchased := every $v in $Vendors, $i in $Items satisfies
      $Stock[$$.Vendor eq $v].Items[][$$.Item eq $i].Available >=
      sum(for $l in $Locations
          return $Order[$$.Location eq $l].Items[][$$.Item eq $i].ItemOrder[][$$.Vendor eq $v].Orders)
  let $requested_vs_delivered := every $l in $Locations, $i in $Items satisfies
      $Order[$$.Location eq $l].Items[][$$.Item eq $i].Requested <=
      sum(for $v in $Vendors
          return $Order[$$.Location eq $l].Items[][$$.Item eq $i].ItemOrder[][$$.Vendor eq $v].Orders)
  let $order_placed := every $l in $Locations, $i in $Items, $v in $Vendors  satisfies
    if ($Order[$$.Location eq $l].Items[][$$.Item eq $i].ItemOrder[][$$.Vendor eq $v].OrderPlaced eq 0)
then
        $Order[$$.Location eq $l].Items[][$$.Item eq $i].ItemOrder[][$$.Vendor eq $v].Orders eq 0
    else
        $Order[$$.Location eq $l].Items[][$$.Item eq $i].ItemOrder[][$$.Vendor eq $v].Orders >= 1
  let $items_purchased := every $v in $Vendors, $l in $Locations satisfies
    $Order[$$.Location eq $l].LocationOrder[][$$.Vendor eq $v].ItemsPurchased eq
    sum(for $i in $Items return $Order[$$.Location eq $l].Items[][$$.Item eq $i].ItemOrder[][$$.Vendor
eq $v].OrderPlaced)
  let $vendor_shipped := every $l in $Locations, $v in $Vendors satisfies
    if ($Order[$$.Location eq $l].LocationOrder[][$$.Vendor eq $v].VendorShipped eq 0) then
        $Order[$$.Location eq $l].LocationOrder[][$$.Vendor eq $v].ItemsPurchased eq 0
    else
        $Order[$$.Location eq $l].LocationOrder[][$$.Vendor eq $v].ItemsPurchased >= 1
  let $vendor_restriction := every $l in $Locations satisfies
    sum(for $v in $Vendors return $Order[$$.Location eq $l].LocationOrder[][$$.Vendor eq
$v].VendorShipped) <= 20
  let $constraints := (((($available_vs_purchased and $requested_vs_delivered) and $order_placed) and
                     $items_purchased) and $vendor_shipped) and $vendor_restriction)
  return {
      procurementData: $procurementData,
      procurementCost: $total_cost,
      available_vs_purchased: $available_vs_purchased,
      requested_vs_delivered: $requested_vs_delivered,
      order_placed: $order_placed,
      items_purchased: $items_purchased,
      vendor_shipped: $vendor_shipped,
      vendor_restriction: $vendor_restriction,
      constraints: $constraints
  }
};
```