Unity Decision Guidance Management System: Analytics Engine and Reusable Model Repository

Mohamad Omar Nachawati[1], Alexander Brodsky[1] and Juan Luo[2]
[1]*Department of Computer Science, George Mason University, 4400 University Drive, Fairfax, VA 22030, U.S.A.*
[2]*Information Technology Unit, George Mason University, 4400 University Drive, Fairfax, VA 22030, U.S.A.*

Keywords: Advanced Analytics, Decision Guidance Management Systems, Decision Support Systems, Decision Management Systems, Knowledge Management, Modeling, Simulation, Optimization, Machine Learning.

Abstract: Enterprises across all industries increasingly depend on decision guidance systems to facilitate decision-making across all lines of business. Despite significant technological advances, current paradigms for developing decision guidance systems lead to a tight-integration of the analytic models, algorithms and underlying tools that comprise these systems, which inhibits both reusability and interoperability. To address these limitations, this paper focuses on the development of the Unity analytics engine, which enables the construction of decision guidance systems from a repository of reusable analytic models that are expressed in JSONiq. Unity extends JSONiq with support for algebraic modeling using a symbolic computation-based technique and compiles reusable analytic models into lower-level, tool-specific representations for analysis. In this paper, we also propose a conceptual architecture for a Decision Guidance Management System, based on Unity, to support the rapid development of decision guidance systems. Finally, we conduct a preliminary experimental study on the overhead introduced by automatically translating reusable analytic models into tool-specific representations for analysis. Initial results indicate that the execution times of optimization models that are automatically generated by Unity from reusable analytic models are within a small constant factor of that of corresponding, manually-crafted optimization models.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Organizations increasingly use automated decision support (ADS) systems to streamline and improve decision-making across a variety of different domains, including manufacturing, supply-chain management, health-care, government operations and marketing (Meleancă, 2013; Taylor, 2015). This paper is concerned with the rapid development of *decision guidance systems*, a class of decision support systems (DSS) geared toward actionable recommendations. These systems are often deployed in a service-oriented architecture that encapsulate executable decision-making logic that can be invoked by different decision-making clients, such as business process management systems, as well as human analysts through integrated and model-driven development environments (Taylor, 2011).

However, despite significant technological advances, current approaches for developing decision guidance systems to automate decision-making lead to a tight-integration between analytic models, algorithms and underlying tools that comprise these systems. Such difficulties can be attributed to diverse and low-level abstractions provided by current paradigms, which preclude the reuse of analytic models across different analytical tasks. Thus the same underlying reality must often be modeled multiple times using different mathematical abstractions, instead of being modeled just once, uniformly (Brodsky et al., 2016a; Brodsky and Luo, 2015). Also, the modeling proficiency required by these languages is typically not within the realm of expertise of many business users, such as executives, analysts and application developers. Consequently, 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 non-reusable, non-extensible, and locked-in to proprietary tool vendors (Brodsky et al., 2016a; Brodsky and Luo, 2015).

To address this problem, earlier work (Brodsky et al., 2016a; Brodsky and Luo, 2015) proposed the Decision Guidance Analytics Language (DGAL) as a solution for developing reusable analytic models in

Nachawati, M., Brodsky, A. and Luo, J.
Unity Decision Guidance Management System: Analytics Engine and Reusable Model Repository.
DOI: 10.5220/0006338703120323
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 1*, pages 312-323
ISBN: 978-989-758-247-9
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

JSONiq, a JSON-based query language itself an extension to XQuery. However, this work was conceptual in nature and did not focus on the problems of compilation and execution of reusable analytic models. We further discuss related work in Section 2.

Addressing these issues is exactly the focus of this paper. Specifically, the contributions of this paper are as follows. First, we developed an analytics engine, called Unity, to support the development of decision guidance systems from a repository of analytic models, which can be reused for different analytical tasks. Unity's uniqueness lies in its core decision guidance algorithms, including optimization and learning, which do not require lower-level level models (e.g., in AMPL for optimization problems), but rather automatically generate lower-level, task- and tool-specific models from reusable analytic models, which are task- and tool-independent. We developed an algorithm to perform deterministic optimization against such models, based on a reduction to standard optimization problem formulation in AMPL and OPL, which can be solved using a variety existing optimization solvers, such as CPLEX and MINOS. To support the reduction, we developed algorithms based on symbolic computation, which output mathematical constraints encoded as a JSON object. These symbolic computation algorithms are also suitable to implement other decision guidance functionality, including machine learning, through reduction to lower-level models.

Second, we propose a conceptual architecture for a NoSQL-based Decision Guidance Management System (DGMS) that is built around Unity to support the seamless integration and interoperability of decision guidance applications, analytic models, algorithms and underlying tools. The uniqueness of this architecture is that it centered around a knowledge base of analytic models, which can be reused for various analytical tasks such as prediction, optimization and statistical learning without the need to manually create lower-level task- and tool-specific models. Finally, we conduct an initial experimental study on the overhead of compiled analytic models. Our evaluation in this paper is limited to the execution time overhead of optimization models generated automatically by Unity. Initial results indicate that the execution times of optimization models that are automatically generated from reusable analytic models are within a small constant factor of that of corresponding, manually-crafted optimization models.

The rest of this paper is organized as follows. In the next section, we briefly discuss some relevant background and related work. In the following section, we propose an architecture for a NoSQL-based DGMS based around Unity, and describe each of its major components. Then we provide an overview of reusable analytic modeling, and show how to develop a simple decision guidance system with Unity. Next, we move on to describe the implementation of the Unity analytics engine, to include the symbolic computation approach, the intermediate representation, and an algorithm for implementing the DGAL operator for deterministic optimization against reusable analytic models. Finally, we present the experimental study and then conclude the paper with some brief remarks on future work.

2 BACKGROUND AND RELATED WORK

In this section, we further discuss related work. Despite its mixed reception and slow adoption due to, among other reasons, unfavorable market conditions and lack of integration and maturity, automated decision system technology is now widely used across many industries (Davenport and Harris, 2005; Patterson et al., 2005). These systems often depend on analytic models to provide actionable recommendations upon which decisions are made. In this paper, we use the term decision guidance system to refer to an advanced class of decision support systems that are designed to provide actionable recommendations using a variety of different analytic models, algorithms and data (Brodsky and Wang, 2008). While decision guidance systems are not restricted to automated decision-making, they often serve to support them. Also, as a clarification, the term decision guidance is distinct from *decisional guidance*, which is known in the literature as the degree to which a DSS influences user decisions (Silver, 1991; Parikh et al., 2001).

Decision guidance systems are built on top of a variety of lower-level tools that provide the full gamut of business analytic capabilities, ranging from descriptive to diagnostic to predictive to prescriptive analytics. There have been several attempts to classify DSSs (Alter, 1980; Arnott, 1998; Hackathorn and Keen, 1981; Haettenschwiler, 2001), including one such classification, by Power, that classifies these systems into five different categories per underlying technology, namely data-driven, model-driven, knowledge-driven, document-driven and communications-driven (Power, 2001). On the other hand, state-of-the-art decision guidance systems often combine multiple approaches into one integrated system to solve complex analytical problems (Brodsky et al., 2016b; Brodsky and Luo, 2015; Luo et al., 2012).

Brodsky and Wang introduced a new type of platform that they referred to as a Decision Guidance Management System (DGMS), which was designed to simplify the development of decision guidance systems 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 (Brodsky and Wang, 2008). While this work laid the foundation for additional research, it did not address the technical challenges surrounding the development of a functional system. Specifically, it did not develop any underlying algorithms to support the decision guidance capabilities of the proposed system, such as simulation, optimization and learning. The proposed architecture was also limited to the relational model, and lacked support for developing analytic models on top of NoSQL data stores, which support more flexible, semi-structured data formats, such as XML and JSON. Furthermore, due to the inherent limitations of SQL, to re-purpose the language for decision guidance modeling and analysis, a few non-standard syntactic extensions were developed, which collectively was called DG-SQL. Introducing new language dialects, however, can break the interoperability of existing development tools, reduce the reusability of existing code and inhibit wide-spread adoption (Lammel and Verhoef, 2001; Shneiderman, 1975).

More recently, progress was made by Brodsky et al. with the proposal of the Decision Guidance Analytics Language (DGAL), which was designed as an alternative to DG-SQL for developing decision guidance systems over NoSQL data stores (Brodsky et al., 2016a; Brodsky and Luo, 2015). Instead of SQL, DGAL is based on JSONiq, which is a more expressive, NoSQL query language. JSONiq was designed specifically for querying JSON documents and NoSQL data stores, and itself is based on the XQuery language (Florescu and Fourny, 2013), which provides highly-expressive querying capabilities centered around the FLWOR construct (Chamberlin et al., 2003). Although this work focused on proposing the DGAL language, it did not propose an architecture for a NoSQL-based DGMS developed around DGAL, nor did it address how to compile and execute analytic models, both of which we cover in this paper.

Rather than extending the syntax of an existing language, as what was done in DG-SQL, DGAL is, by design, syntactically equivalent to JSONiq. Unlike purely library-based approaches for developing analytic models such as the Concert API, the DGAL language is designed to support algebraic modeling, similar to SymPy and JuMP, which allow modelers to specify equations directly using the native expression operators of the host language. While SymPy is a full-blown computer algebra system for Python (Joyner et al., 2012), and JuMP is a domain specific modeling language for optimization in Julia (Lubin and Dunning, 2015), DGAL is a lighter-weight, algebraic modeling language designed to specifically to support the development of reusable analytic models. To support decision guidance, DGAL introduces a small library of core decision guidance operators for different analytical tasks, such as optimization and machine learning, which are exposed as regular JSONiq functions (Brodsky et al., 2016a; Brodsky and Luo, 2015). While these operators appear 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 decision guidance operators have semantics that extend that of the JSONiq language. Although the semantics of these operators are intuitive, the underlying algorithms needed to implement them are significantly more complex.

3 NOSQL-BASED DECISION GUIDANCE MANAGEMENT SYSTEM ARCHITECTURE

In this section, we present a conceptual system architecture for decision guidance systems using the proposed Decision Guidance Management System (DGMS). The architecture we describe is illustrated in Figure 1. The DGMS middleware, represented by the empty black rectangle, is composed of three-layers, namely the application management layer, the decision guidance analytics management layer, and the tool management layer. Within this middleware, the Unity analytics engine, represented by the solid black rectangle, is situated between the client layer and external tool layer, and transparently connects different clients to the lower-level, external tools that support decision guidance. Keeping with the goals that motivated the earlier DGMS proposal (Brodsky and Wang, 2008), our proposed architecture is designed to provide seamless support for data acquisition, learning, prediction, and optimization. However, unlike the former, which uses DG-SQL for analytic modeling, we replace the role of DG-SQL with DGAL. Specifically, in our proposed architecture, DGAL serves as both a language for developing reusable analytic models as well as for executing analytical queries against those models.

The proposed architecture supports several different user roles for interacting with the system, to in-

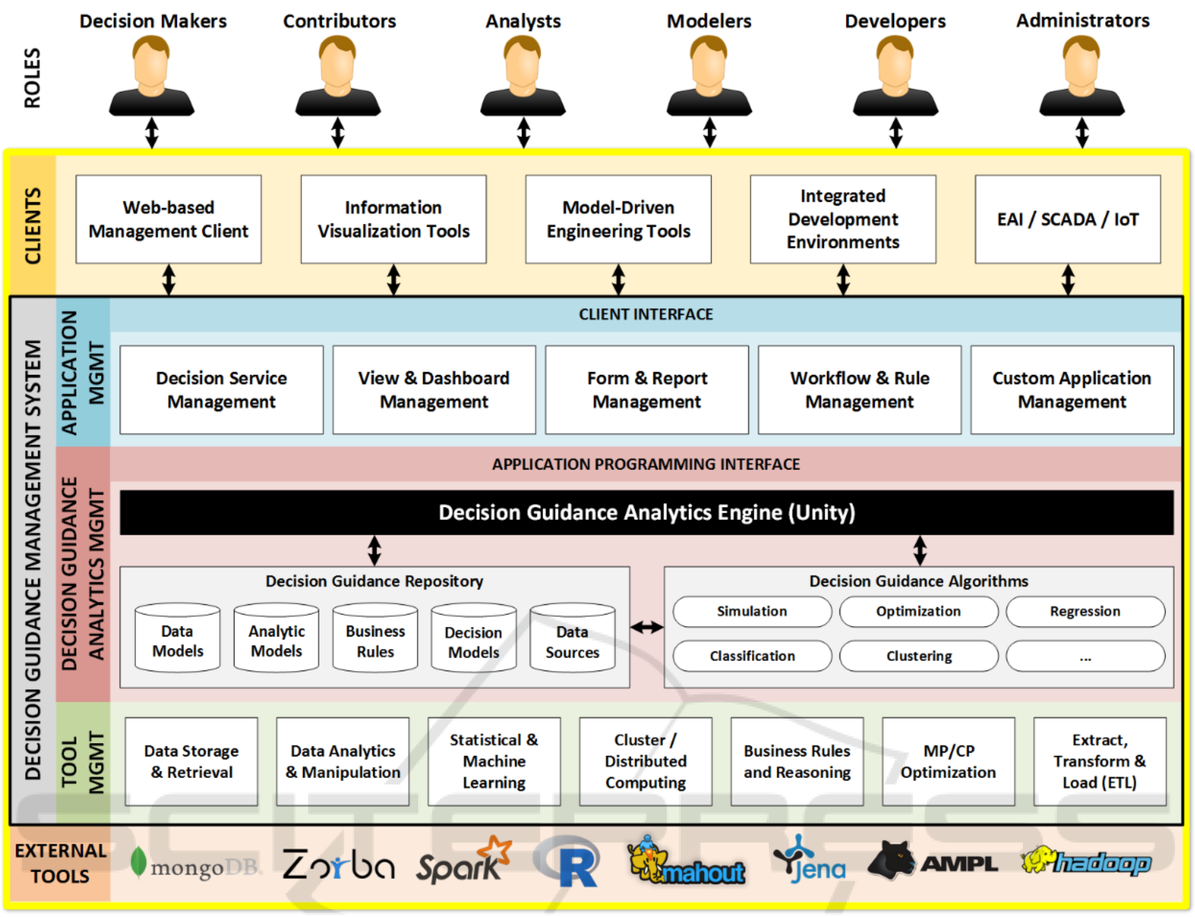
ROLES
Decision Makers
Contributors
Analysts
Modelers
Developers
Administrators
CLIENTS
Web-based Management Client
Information Visualization Tools
Model-Driven Engineering Tools
Integrated Development Environments
EAI / SCADA / IoT
DECISION GUIDANCE MANAGEMENT SYSTEM
APPLICATION MGMT
CLIENT INTERFACE
Decision Service Management
View & Dashboard Management
Form & Report Management
Workflow & Rule Management
Custom Application Management
DECISION GUIDANCE ANALYTICS MGMT
APPLICATION PROGRAMMING INTERFACE
Decision Guidance Analytics Engine (Unity)
Decision Guidance Repository
Data Models
Analytic Models
Business Rules
Decision Models
Data Sources
Decision Guidance Algorithms
Simulation
Optimization
Regression
Classification
Clustering
...
TOOL MGMT
Data Storage & Retrieval
Data Analytics & Manipulation
Statistical & Machine Learning
Cluster / Distributed Computing
Business Rules and Reasoning
MP/CP Optimization
Extract, Transform & Load (ETL)
EXTERNAL TOOLS
mongoDB
Zorba
Spark
R
mahout
Jena
AMPL
hadoop

Figure 1: Proposed Architecture for a NoSQL-based Decision Guidance Management System.

clude decision makers, contributors, analysts, modelers, developers, and administrators. The roles are not mutually exclusive and therefore a single user may serve different roles, depending on their specific requirements. Decision-makers are the end-users of decision guidance, and can be either human operators or a decision-guided applications or services, such as business processes, workflows and intelligent agents. Contributors are users that run ETL processes and provide data-entry into the decision guidance repository. These users can also manage data models and data sources. Analysts are technical users that manage the life-cycle of analytic models in the repository, and can design new views, dashboards, forms and reports. Analysts can also design workflows and business rules for automating repetitive decisions based on decision guidance provided by the system. Modelers are technical users, typically with a data science background, that devise mathematical representations of metrics and constraints for new analytic models to be published in the repository. If such a user does not have the technical expertise to develop analytic models from scratch, they can easily mash up new models by composing and specializing existing ones from the repository. Developers are technical users with experience in software development that build custom applications that provide extended functionality to meet domain-specific requirements. Finally, administrators are users that have administrative access to the system, to handle system configuration, user management and security.

The client layer includes diverse, domain-specific applications that support the development and management of decision guidance systems. The architecture would provide an API for connecting to a variety of third-party clients, including information visualization tools, model-driven engineering tools, integrated development environments and external systems, such as SCADA or IoT devices.

A primary objective of the proposed architecture is to facilitate the development of decision guidance systems by allowing users to work at different levels of abstraction per their skills and expertise. For example, modelers with expertise in operations research can use the full power of mathematical constraints to create reusable analytic models directly in JSONiq. Business analysts without an operations research background could then graphically compose and specialize these expertly crafted models to address specific problems using a model-driven engineering tool. They could also develop analytical views and dashboards using a data visualization tool.

The application management layer provides several services to support the rapid development of decision guidance systems. Decision service management supports the development and operation of decision services based on analytic models contained in the repository. View and dashboard management provide tools to create analytical views and templates for the rapid development of interactive dashboards. Analytical views are like regular database views, except that they are based on one or more analytic models and DGAL operators. Form and report management supports the development and use of forms and notebooks for data collection, reporting and publishing. Workflow and rule management supports the development and execution of workflows and business rules for the rapid development of simple automated decision systems. Finally, custom application management would provide an API for building domain-specific, decision-guided applications.

The analytics management layer hides the complexity of dealing with different external tools that provide the essential analytical capabilities of a decision guidance system. This layer is composed of the Unity analytics engine, a decision guidance repository, and a variety of different decision guidance algorithms. The decision guidance repository provides uniform access to the different artifacts that together constitute the business knowledge used to support decision guidance. The different types of artifacts stored in the decision guidance repository include, but are not limited to data models, analytic models, business rules, DMN-based decision models, data sources, and ontologies. The Unity analytics engine serves as a bridge between the analytic models in the repository, decision guidance algorithms and the lower-level tools used to implement them.

The proposed architecture is designed to support three kinds of analytic models, namely white-box, black-box and gray-box models. With white-box analytic models, the source code of the model is stored in the decision guidance repository, and the execution of such models is performed locally by the analytics engine. While white-box analytic models can help decision-makers 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 analytic models the source code is not provided, and instead what is stored in the decision guidance repository is a description of a web service that provides remote and opaque execution of the model. While this method supports proprietary models, it does not provide a way for the client to reuse the models for different analytical 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 that the execution occurs remotely, however gray-box models return its results in symbolic form. While this exposes part of the model's logic, it allows clients to easily reuse remote models for different types of analytical task.

Finally, the tool management layer serves to provide seamless access to the external tools that are needed to implement decision guidance algorithms as well as to provide additional capabilities. The types of external tools that this layer would support includes packages for data storage and retrieval, data analytics and manipulation, statistical and machine learning, MP/CP optimization and business process and rule execution and reasoning. Ideally, this layer would also support big data analytics and deep learning frameworks such as Apache Spark's Mllib (Meng et al., 2016), Theano (Bergstra et al., 2010), Google Tensorflow (Abadi et al., 2016).

4 REUSABLE ANALYTIC MODELING

As mentioned before, current decision guidance system development paradigms lead to a tight-integration of the analytic models, algorithms and underlying tools that make up these systems, which often inhibit integration and interoperability. This often necessitates the development of specialized models for each analytic task, such as machine learning and optimization, which makes it difficult to reuse and extend existing models via declarative composition and specialization constructs. In this section, we show how Unity can be used to build a rudimentary decision guidance system to support intelligent order management based on a single reusable analytic model expressed directly in JSONiq. In the next section, we describe the implementation of Unity, which uses a symbolic computation-based approach to enable algebraic modeling in JSONiq.

In our intelligent order management scenario, we track suppliers that each supply zero or more items, as well 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 simple JSON object that will serve as the input to our analytic model. An example of this input object is shown below:

```
let $input := { "suppliers": [{
  "sid": "supplier1",
  "supply": [
    { "upc": "47520-81452",
      "ppu": 10.99,
      "qty": 500 }, ... ]
  },{
  "sid": "supplier2",
  "supply": [
    { "upc": "47520-81452",
      "ppu": 11.99,
      "qty": 1500 }, ... ]
  }, ... ],
"customers": [{
  "cid": "customer1",
  "demand": [
    { "upc": "47520-81452", "qty": 1475 }, ... ]
  },{
  "cid": "customer2",
  "demand": [
    { "upc": "32400-24785", "qty": 874 }, ... ]
  }, ... ],
"orders": [{
  "sid": "supplier1", "cid": "customer1",
  "items": [
    { "upc": "47520-81452", "qty": 500 }, ... ]
  }, ... ]
}
```

Based on the data model that one can derive from the above example, we can now define the metrics and constraints for our analytic model, which we will then proceed to implement. While a single analytic model can support multiple metrics, for the purposes of our discussion we will limit our model to a single metric: the computation of the total cost of all ordered items. Assuming the variable `$input` holds the input for our analytic model, the total cost metric can be expressed in JSONiq as follows:

```
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 will now define supply and demand constraints for our order management analytic model. First, we have a supply constraint on orders, which requires 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 particular item across all orders to that supplier. This constraint ensures that an existing order can be fulfilled per current supplier inventory levels. Second, we have a demand constraint on orders, which requires that for each customer, the quantity of each item requested is equal to the sum of the order quantities of that particular item across all orders from that customer. This constraint ensures that we only order items that are specifically requested by customers. We can express both of 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 JSONiq implementation of our analytic model by wrapping the cost metric as well as the supply and demand constraints inside a function containing a single parameter corresponding to our model's input data and whose return value is the computed metrics and constraints:

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

With our reusable analytic model implemented in JSONiq, we can now use different DGAL operators to perform many analytic tasks, such as simulation, optimization and machine learning, without having to redevelop different versions of our model for each analytic task that we want to perform. The work of compiling our reusable analytic model into task- and tool-specific models for analysis is handled seamlessly by the Unity analytics engine. For example, we can simulate our model on some order input using the deterministic simulation operator in DGAL, which is implicitly exposed as a regular JSONiq function invocation:

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

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

In this case, the output object that is returned contains the computed cost metric as a JSON number, and a value of either true or false for the constraints property, depending on if the supply and demand constraints were satisfied for the given input.

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 variables in the place of numeric values for each qty property. This indicates to the Unity analytics engine that we want to solve for the values of those properties. A decision variable is represented as a simple object that contains one of the following properties corresponding to its type: integer?, decimal?, or logical?. The corresponding value indicates the decision variable's identifier, which if set to null will be automatically replaced with a UUID. Two different decision variables that contain the same identifier refer to the same decision variable in the underlying optimization problem. An example of an input object with a decision variable annotation 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 input would, however, result in unexpected behavior. This is because the function that implements the analytic model is expecting a numerically-typed value for the qty property, but it will find a decision variable object instead. Rather, we need to use the dgal:argmin operator to have Unity find specific values for the qty decision variables that minimize the cost objective:

```
let $instantiatedInput := dgal:argmin({
  varInput: $annotatedInput,
  analytics: "scm:OrderAnalyticsModel",
  objective: "cost"
})
```

To maintain complete syntactic equivalence with JSONiq, the DGAL operators provided by Unity are exposed as regular JSONiq functions. As shown above, the DGAL operator for optimization is dgal:argmin, which serves as a tool-independent wrapper over different underlying optimization algorithms, all of which are seamlessly integrated by Unity. For complex problems that require the nesting of multiple optimization operators, Unity also provides seamless solver interoperability.

The dgal:argmin function takes a single object as input, which contains at least three properties, specifically: (1) varInput: the decision variable annotated input, (2) analytics: the analytic model as a qualified name string or function pointer, and (3) objective: the JSONiq path expression string to select the metric property that will serve as the objective from the computed output of the analytic model. If a solution to the resulting optimization problem is feasible, the argmin operator returns an instantiation of the annotated object contained in the varInput property, where all decision variable annotations are replaced with corresponding solution values that together minimize the objective. Finally, to compute the minimized value of the objective metric, one simply invokes the analytic model on the instantiated input object returned from dgal:argmin:

```
return
  scm:OrderAnalyticsModel($instantiatedInput)
```

5 IMPLEMENTATION OF UNITY

In this section, we describe the design and implementation of the Unity analytics engine. Unity integrates several external tools into a seamless decision guidance platform, such as Zorba, GitLab CE[1], AMPL and OPL. Zorba is a NoSQL query engine that supports both XQuery and JSONiq (Bamford et al., 2009). AMPL is a mathematical programming language that was originally developed by Robert Fourer, David Gay, and Brian Kernighan (Fourer et al., 1990). OPL is an newer mathematical programming language that was developed by Pascal Van Hentenryck (Hentenryck, 2002) and is currently maintained by IBM. The engine also includes a compiler for translating reusable analytic models into lower-level, tool-specific models for analysis. A symbolic computation-based approach is used to support algebraic modeling without having to modify the JSONiq query processor (Zorba) by first lowering the analytic model into a tool-independent intermediate representation. We discuss the details of this representation in Section 5.2.

Unity was developed using a combination of Java, C++, JSONiq and XSLT and currently supports simulation, optimization and machine learning against reusable analytic models. Unity is tightly integrated

[1]https://about.gitlab.com/

with GitLab CE, which is used for storage, retrieval and management of decision guidance repository artifacts, such as analytic models, views and datasets. For this purpose, we implemented a custom Zorba URI resolver, which also serves as a hook where the Unity engine transparently transforms JSONiq modules to support algebraic modeling via symbolic computation. To simplify the development of reusable analytic models, we developed a prototype IDE based on Eclipse, as well as an Atom[2] macro for executing DGAL queries remotely over a RESTful API from within the IDE.

5.1 Symbolic Computation and Analysis

While JSONiq query processors support complex data queries and even simple analytical operations they do not directly support the advanced analytics operators that DGAL provides, such as for optimization and machine learn. Executing DGAL queries that depend on such operators require the use of specialized algorithms, 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. As explained before, while syntactically DGAL is backwards compatible with JSONiq, the execution of decision guidance algorithms extends the semantics of JSONiq. Because of this difference, decision guidance algorithms cannot be directly executed on a standard JSONiq query processor. 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, however, JSONiq does not currently support operator overloading.

The process to implement the optimization operator, `dgal:argmin`, consists of 6 steps, as shown in Figure 2. The process begins with the analytic model resolution step, wherein the fully qualified analytic model name is resolved against the content repository (GitLab), to retrieve its JSONiq source code. Next, in

[2]https://atom.io/

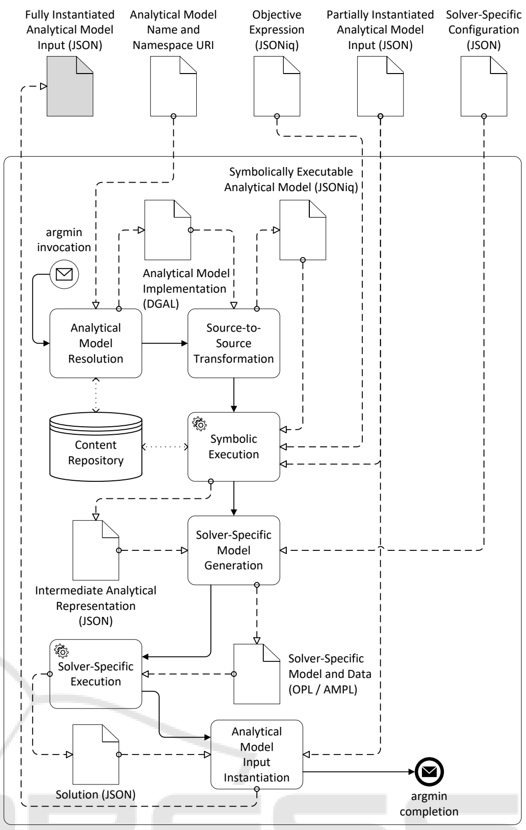
Fully Instantiated Analytical Model Input (JSON)
Analytical Model Name and Namespace URI
Objective Expression (JSONiq)
Partially Instantiated Analytical Model Input (JSON)
Solver-Specific Configuration (JSON)
Symbolically Executable Analytical Model (JSONiq)
argmin invocation
Analytical Model Implementation (DGAL)
Analytical Model Resolution
Source-to-Source Transformation
Content Repository
Symbolic Execution
Intermediate Analytical Representation (JSON)
Solver-Specific Model Generation
Solver-Specific Execution
Solver-Specific Model and Data (OPL / AMPL)
Analytical Model Input Instantiation
Solution (JSON)
argmin completion

Figure 2: Deterministic Optimization Process.

the source-to-source transformation step, the JSONiq source code of the module is transformed into a symbolically executable JSONiq module. Then in the symbolic computation step, the transformed module is executed as a regular JSONiq module using Zorba. This generates output in the intermediate representation. Next in the solver-specific model generation step, the intermediate representation output is used to generate a solver-specific model along with associated data. Currently, the prototype can generate optimization models in either OPL or AMPL. The 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 analytic model input instantiation step, the solution obtained from the solver is merged with the annotated decision variable input to return a fully instantiated input, where all decision variables are replaced with the resultant values in the solution.

5.2 Intermediate Representation

The analytics engine uses a common intermediate language for representing symbolically computed analytic models in a way that is independent of both the front-end modeling language and the tool-specific, back-end language. For this purpose, we use a JSON-based language that is largely compatible with PFA. The Portable Format for Analytics (PFA) is a JSON-based interchange format for deploying analytic models to production environments (Pivarski et al., 2016). Using a common intermediate language allows us to easily extend Unity to support both new modeling language front-ends as well new tool-specific back-ends, such as MPS and NL.

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 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 indicates its type and the property value is its identifier. Unlike in some optimization modeling languages, decision variables in the intermediate representation are not explicitly defined, rather they are implicitly defined when they are used. 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.

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

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

While the above expression is valid, Unity automatically reduces expressions that do not depend on any decision variables or learning parameters. For this expression, the value can be reduced to simply `true`.

5.3 Source-to-Source Transformation

To support algebraic modeling in JSONiq via symbolic computation, Unity performs a source-to-source transformation to redefine the execution semantics of expression operators. The main idea behind this approach is that for each operator in the analytic model, a function call is substituted in its place that when called returns its result as an expression tree in the intermediate representation. Unity attempts to simplify this expression tree by performing constant-folding in cases where the computation does not involve a decision variable or learning parameter.

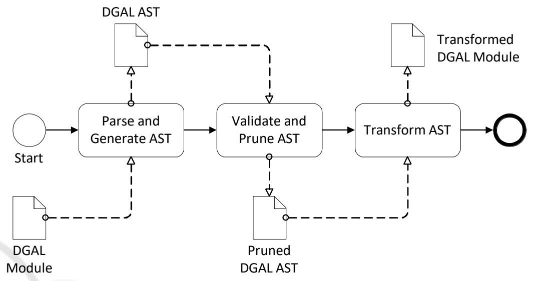

Figure 3: Source-to-Source Transformation Process.

As shown in Figure 3, to perform the source-to-source transformation Unity first parses the source code of the JSONiq+DGAL module to build an abstract syntax tree that is serialized in XML. We used the Rex[3] parser generator to generate a JSONiq parser in Java from the EBNF grammar[4] that is provided in the JSONiq language specification. We then use an XSLT transformation on the resulting XML tree to replace each JSONiq expression operator with a corresponding function that returns its result in the intermediate representation.

To illustrate how the transformation works, consider the supplyConstraint expression from the order analytics example from Section 4. After performing the transformation, the sub-expression `$supplyItem.qty ge fn:sum(...)` is replaced with calls to the corresponding dg:ge and dg:sum functions to enable symbolic computation, 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 symbolic computation functions, like `dg:ge` and `dg:sum`, are implemented completely in

[3]http://www.bottlecaps.de/rex/
[4]http://www.jsoniq.org/grammars/jq++.ebnf

JSONiq. For example, 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
};
```

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

6 PERFORMANCE EVALUATION

In this section, we report on a preliminary experimental study that we conducted to investigate the overhead introduced by automatically translating reusable analytic models into task- and tool-specific models for analysis. The question that we seek to answer is whether our reusable analytic modeling approach is inherently too computationally inefficient to be used for real-world, decision guidance systems. While the amount of acceptable overhead is highly user dependent, we hypothesize that for the case of deterministic optimization, the execution time overhead of automatically generated models is within a small constant factor of that of manually-crafted ones. While our current evaluation is limited to a single compilation target and solver, namely the OPL and CPLEX respectively, we are working to develop a more comprehensive evaluation.

To test our hypothesis, we took the procurement optimization model developed by (Brodsky et al., 2012) and manually translated it into a DGAL reusable analytic model. We also translated the original AMPL model into OPL to serve as the control model for our experiment. We then developed a script to automatically generate isomorphic pairs of randomized input data to feed into our DGAL and OPL models. For our DGAL test model, the script generated input data in the JSON format, while for our OPL control model the script generated input data in the standard *.DAT file format that the OPL compiler accepts.

Using this input data, we conducted a total of 205 trials, where for each trial we measured the wall-clock

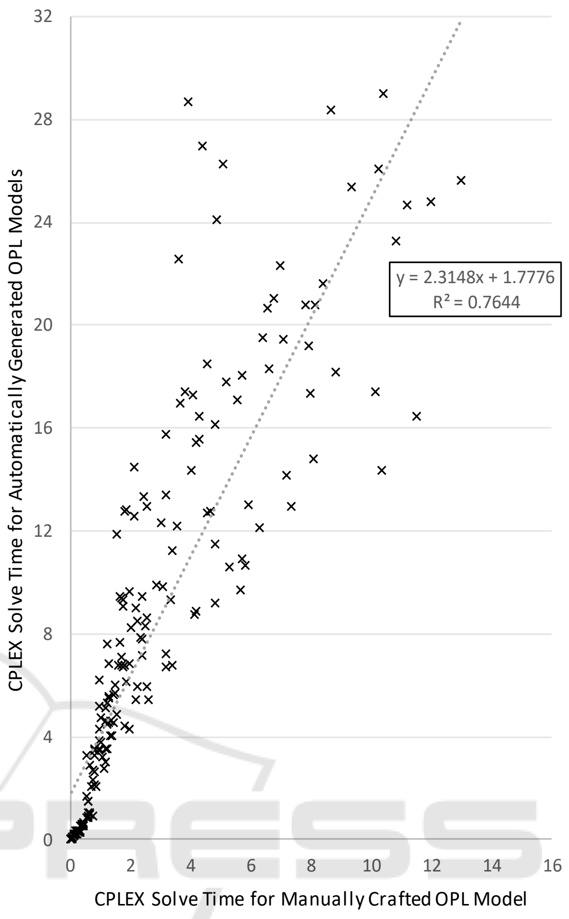

Figure 4: Execution Time Overhead of Automatically Generated OPL Models from DGAL (in seconds).

time that the CPLEX solver took to solve the DGAL-based test optimization problem after being automatically translated into OPL by Unity. For comparison, we also measured the CPLEX execution time for solving the control OPL optimization problem. Because we are only interested in measuring the time it took CPLEX to solve the test and control problems, we excluded from our measurements time spent on other peripheral tasks, such as compiling DGAL models into OPL and loading data into memory. Across all trials, the number of decision variables in the resulting optimization problems ranged from 72 to 16,800.

The trials were conducted on a laptop machine equipped with an Intel Core i5-4210U processor and 16GB of RAM. We used the Java API exposed by CPLEX to automate the execution and measurement of each trial. Wall-clock time was measured by subtracting the difference between the return values of calls to the Java time function, currentTimeMillis(), which was placed immediately before and after the invocation of the CPLEX solve method. To reduce measurement error, we closed extraneous applications and services and ran all problems sequentially.

The results of this experiment are presented as a scatter chart in Figure 4, where the horizontal axis represents the wall-clock time, in seconds, that the CPLEX solver took to solve the OPL control optimization problem, and the vertical axis shows the wall-clock time, in seconds, that the CPLEX solver took to solve the DGAL-based test optimization problem. A linear trend line through the time measurement points gives a slope of 2.3148, which indicates that the execution time of overhead for compiled DGAL models is about 2.3 times that of manually crafted OPL optimization models. Regarding the value of the r-squared statistic, 0.7644, some error is to be expected due to the behavior of underlying algorithms used for MP-based optimization, such as branch and bound, which are often sensitive to how the problem is formulated.

While the purpose of our present research was to develop an analytics engine for reusable models, the current overhead introduced leaves significant room for improvement. There are many techniques that could be used to decrease the overhead of our compiled optimization models. Many solvers provide options to fine-tune the optimization process, such as preprocessing, which needs to be investigated in the future. Also, utilizing a combination of domain-specific decomposition and preprocessing techniques, such as the one proposed by Egge et al. to generate efficient, tool-specific models for certain classes of problems could be fruitful (Egge et al., 2013). However, with regards to our preliminary results, we view the current execution time overhead as a standard trade-off between user productivity and computational efficiency. The current performance of Unity could be acceptable in cases where computational efficiency can be sacrificed to avoid the costly redevelopment of specialized analytic models to support different analytical tasks. In fact, Unity has successfully been used to support the development of reusable analytic models for manufacturing processes (Brodsky et al., 2016a).

7 CONCLUSION AND FUTURE WORK

In this paper, we introduced the Unity analytics engine to support the development of decision guidance systems from a repository of reusable analytic models. We proposed a conceptual architecture for a NoSQL-based Decision Guidance Management System (DGMS) that is built around Unity to support the seamless integration and interoperability of decision guidance applications, analytic models, algorithms and underlying tools. We also demonstrated the use of our analytics engine by constructing a simple decision guidance system for intelligent order management. Finally, we investigated the overhead of our reusable analytic modeling approach by conducting an preliminary experimental study. Initial results indicate that the execution times of optimization models that are automatically generated by Unity from reusable analytic models are within a small constant factor of that of corresponding, manually-crafted optimization models.

Our work opens new research questions that we are currently working on addressing. Particularly, as support for more algorithms against analytic models is developed, the problem of choosing the most appropriate algorithm and settings for a particular problem emerges. Further research is needed to investigate how a meta-optimization solver can be developed and integrated with Unity, along the lines of work on DrAmpl (Fourer and Orban, 2010). The objective here is to automatically determine the set of feasible algorithms for a particular problem, as well appropriate values for algorithm-specific parameters, which is essential for many heuristic or partial-search algorithms. Additionally, we are investigating ways to generalize the work of Egge et al. on decomposition and preprocessing to drastically reduce complexity on a larger class of analytic models (Egge et al., 2013).

ACKNOWLEDGEMENTS

This work has been partially supported by the National Institute of Standards and Technology (NIST), award number 70NANB16H171.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*.

Alter, S. (1980). *Decision support systems: current practice and continuing challenges*, volume 157. Addison-Wesley Reading, MA.

Arnott, D. R. (1998). A framework for understanding decision support systems evolution. In *9th Australasian Conference on Information Systems, Sydney, Australia: University of New South Wales*.

Bamford, R., Borkar, V., Brantner, M., Fischer, P. M., Florescu, D., Graf, D., Kossmann, D., Kraska, T., Muresan, D., Nasoi, S., et al. (2009). Xquery reloaded. *Proceedings of the VLDB Endowment*, 2(2):1342–1353.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7.

Brodsky, A., Egge, N. E., and Wang, X. S. (2012). Supporting agile organizations with a decision guidance query language. *Journal of Management Information Systems*, 28(4):39–68.

Brodsky, A., Krishnamoorthy, M., Bernstein, W. Z., and Nachawati, M. O. (2016a). A system and architecture for reusable abstractions of manufacturing processes. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 2004–2013. IEEE.

Brodsky, A. and Luo, J. (2015). Decision guidance analytics language (dgal)-toward reusable knowledge base centric modeling. In *ICEIS (1)*, pages 67–78.

Brodsky, A., Luo, J., and Nachawati, M. O. (2016b). Toward decision guidance management systems: Analytical language and knowledge base. *Department of Computer Science, George Mason University*, 4400:22030–4444.

Brodsky, A. and Wang, X. S. (2008). 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*, pages 71–71. IEEE.

Chamberlin, D., Florescu, D., Robie, J., Simeon, J., and Stefanescu, M. (2003). Xquery: A query language for xml. In *SIGMOD Conference*, volume 682.

Davenport, T. H. and Harris, J. G. (2005). Automated decision making comes of age. *MIT Sloan Management Review*, 46(4):83.

Egge, N., Brodsky, A., and Griva, I. (2013). An efficient preprocessing algorithm to speed-up multistage production decision optimization problems. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, pages 1124–1133. IEEE.

Florescu, D. and Fourny, G. (2013). Jsoniq: The history of a query language. *IEEE internet computing*, 17(5):86–90.

Fourer, R., Gay, D. M., and Kernighan, B. W. (1990). A modeling language for mathematical programming. *Management Science*, 36(5):519–554.

Fourer, R. and Orban, D. (2010). Drampl: a meta solver for optimization problem analysis. *Computational Management Science*, 7(4):437–463.

Hackathorn, R. D. and Keen, P. G. (1981). Organizational strategies for personal computing in decision support systems. *MIS quarterly*, pages 21–27.

Haettenschwiler, P. (2001). Neues anwenderfreundliches konzept der entscheidungsunterstützung. *Gutes entscheiden in wirtschaft, politik und gesellschaft*, pages 189–208.

Hentenryck, P. V. (2002). Constraint and integer programming in opl. *INFORMS Journal on Computing*, 14(4):345–372.

Joyner, D., Čertík, O., Meurer, A., and Granger, B. E. (2012). Open source computer algebra systems: Sympy. *ACM Communications in Computer Algebra*, 45(3/4):225–234.

Lammel, R. and Verhoef, C. (2001). Cracking the 500-language problem. *IEEE software*, 18(6):78–88.

Lubin, M. and Dunning, I. (2015). Computing in operations research using julia. *INFORMS Journal on Computing*, 27(2):238–248.

Luo, J., Brodsky, A., and Li, Y. (2012). An em-based ensemble learning algorithm on piecewise surface regression problem. *International Journal of Applied Mathematics and Statistics*, 28(4):59–74.

Meleancă, R. (2013). Will decision management systems revolutionize marketing? *Procedia-Social and Behavioral Sciences*, 92:523–528.

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7.

Parikh, M., Fazlollahi, B., and Verma, S. (2001). The effectiveness of decisional guidance: an empirical evaluation. *Decision Sciences*, 32(2):303–332.

Patterson, A., Bonissone, P., and Pavese, M. (2005). Six sigma applied throughout the lifecycle of an automated decision system. *Quality and Reliability Engineering International*, 21(3):275–292.

Pivarski, J., Bennett, C., and Grossman, R. L. (2016). Deploying analytics with the portable format for analytics (pfa). In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 579–588. ACM.

Power, D. J. (2001). Supporting decision-makers: An expanded framework. *Proceedings of Informing Science and IT Education*, pages 1901–1915.

Shneiderman, B. (1975). Experimental testing in programming languages, stylistic considerations and design techniques. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 653–656. ACM.

Silver, M. S. (1991). Decisional guidance for computer-based decision support. *MIS Quarterly*, pages 105–122.

Taylor, J. (2011). *Decision management systems: a practical guide to using business rules and predictive analytics*. Pearson Education.

Taylor, J. (2015). Analytics capability landscape.