NISTIR 7961

Processes Analytics Formalism for Decision Guidance in Sustainable Manufacturing

Alexander Brodsky
Guodong Shao
Frank Riddick

http://dx.doi.org/10.6028/NIST.IR.7961

National Institute of Standards and Technology
U.S. Department of Commerce

AUTHENTICATED
U.S. GOVERNMENT
INFORMATION
GPO

NISTIR 7961

Processes Analytics Formalism for Decision Guidance in Sustainable Manufacturing

Alexander Brodsky
George Mason University

Guodong Shao
Frank Riddick
System Integration Division
Engineering Laboratory

http://dx.doi.org/10.6028/NIST.IR.7961

November 2013



U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Under Secretary of Commerce for Standards and Technology and Director

TABLE OF CONTENTS

1. INTRODUCTION 1
2. The needs for SUSTAINABLE process analytics formalism 1
3. The concept of sustainable process analytics formalism 5
   3.1 Context of SPAF 5
   3.2 SPAF concept through a manufacturing example 6
   3.3 Structures of SPAF 7
   3.4 Potential users of the SPAF 12
4. Illustrative example using SPAF 13
   A. SPAF model syntax 26
   B. SPAF formal semantics 28
   C. SPAF Query Computation 30

ABSTRACT
This paper introduces NIST's Sustainable Process Analytics Formalism (SPAF) to facilitate the use of simulation and optimization technologies for decision support in sustainable manufacturing. SPAF allows formal modeling of modular, extensible, and reusable process components and enables sustainability performance prediction, what-if analysis, and decision optimization based on mathematical programming. SPAF models describe (1) process structure and resource flow, (2) process data, (3) control variables, and (4) computation of sustainability metrics, constraints, and objectives. This paper presents the SPAF syntax and formal semantics, provides a sound and complete algorithm to translate SPAF models into formal mathematical programming models, and illustrates the use of SPAF through a manufacturing process example.

1. INTRODUCTION

To be successful in today's complex, rapidly changing, and highly competitive world, manufacturers must begin using sustainable practices throughout their manufacturing operations. The United States Department of Commerce (DOC) identifies Sustainable Manufacturing (SM) as one of its high-priority performance goals, defining SM as the "creation of manufactured products that use processes that minimize negative environmental impacts, conserve energy and natural resources, are safe for employees, communities, and consumers, and are economically sound" (DOC 2010).

Increasingly, some large companies are making efforts to make their operations and manufacturing processes more sustainable (Fujitsu 2011, GM 2010, Rockwell Automation 2010). However, most of these projects are customized and conducted on a piecemeal basis. The solutions are normally not easily reusable and not easily extensible. The effect of many complex interactions is often overlooked. Furthermore, most of the small and medium-sized enterprises (SMEs) lack the capability to manage energy and material efficiency in a systematic, quantitative, and optimal manner required to meet their sustainability goals. To address these challenges, a standard formal methodology is needed to model, exchange, and reuse manufacturing process knowledge for effective sustainability performance analysis.

This is the focus of this paper. A Sustainable Process Analytics Formalism (SPAF) has been developed at the National Institute of Standards and Technology (NIST) to facilitate the use of simulation and optimization technologies for decision support in sustainable manufacturing. SPAF allows formal modeling of modular, extensible and reusable process components and enables sustainability performance prediction, what-if analysis and decision optimization based on mathematical programming. More specifically, the contributions of this paper include (1) the concept of SPAF that enables formal representation of sustainable process structure and resource flow, data, control parameters, metrics, and constraints; (2) the syntax and formal semantics of SPAF; (3) a sound and complete algorithm to translate SPAF models into formal mathematical programming models; and (4) a sustainable manufacturing example that illustrates SPAF.

The rest of this paper is organized as follows. Section 2 briefly discusses the needs for the SPAF; section 3 introduces the context and concept of SPAF and potential users; section 4 explains the SPAF using an example; in section 5, a summary is provided and future work are discussed. Finally, the appendix presents the detail SPAF syntax and semantics.

2. THE NEEDS FOR SUSTAINABLE PROCESS ANALYTICS FORMALISM

Formal description and representation of sustainable processes also provides a basis for standardization. Such standardization in turn is the foundation for system integration, process analysis, and decision optimization, all are essential to the improvement of decision-making on factory floors (NIST 2010, Tanzil and Beloff 2006, NRC 1999). Complex sustainability analysis requires formal simulation (e.g., Delmia Quest) or optimization models (e.g., A Modeling Language for Mathematical Programming (AMPL)) (Berglund et al. 2011, AMPL 2011). Modeling and optimization have been identified as a key enabler for improving SM in the future (SMLC 2011), but require significant modeling expertise and a substantial development effort. SPAF eases the modeling process by providing standard description and tools associated with it, so that the availability, use, and effectiveness of modeling and optimization technologies can be increased. Currently, different analysis tools such as simulation, optimization, and database query languages require different data representation and mathematical abstractions for modeling. Thus, even for the same manufacturing process, the knowledge needs to be represented differently multiple times, rather than just once. This makes model development, modification, and extension very difficult. SPAF facilitates the increase of model reuse. In summary, SPAF is designed to help companies, especially for SMEs, overcome the following major challenges: (1) lack of modeling and

operation research expertise and (2) duplication of modeling efforts. To satisfy manufacturers' needs, we have decided to include the following desirable features for SPAF:

Data manipulation and querying: SPAF supports data storage, manipulation, and querying. For example, given a model of a specific milling machine, users may want to query the machine specification data provided by the vendor.

What-if analysis: SPAF supports what-if analysis by computing a range of sustainability metrics as a function of non-controllable parameters and control variables in manufacturing processes, based on the formal representation of manufacturing processes and sustainability metrics. For example, given a particular setup of a milling machine, users may want to compute the energy consumption for that setup.

Decision optimization: SPAF enables the formulation of optimization problems for deriving the best option among all alternatives of the operational setting of machines, production plan, and investment options. For example, given a model of a specific milling machine, users may want to find out a setting of the machine that produces the required part while minimizing the energy consumption.

Unified modeling for different tasks: SPAF allows the sustainable process knowledge being represented once, used many times, for different analyses such as data query, what-if analysis, and decision optimization. Figure 1 shows the comparison between the current modeling approach (left hand side of the Figure) and the unified SPAF modeling approach (right hand side of Figure 1). In current modeling approaches, duplicated modeling efforts are needed even for the same manufacturing problem for different kinds of analysis tasks. For example, simulation, optimization, and Life Cycle Assessment (LCA) of a machining process have totally different abstractions and modeling methods. They are independent of each other. However, the unified SPAF modeling approach enables modeling the machining process using SPAF once, and the same model then can be used for data query, what-if analysis, and decision optimization.

Built-in support for process modeling and sustainability metrics: SPAF provides modeling capability for hierarchical composition of processes and resource flows. Representation of sustainable metrics such as CO_2 emissions, energy and material consumption, and cost are stored in a model library for reuse.

Modular, extensible, and reusable models: SPAF enables modular model design and creation of model libraries. Modular model design provides definitions for structuring process knowledge into discrete, scalable, and reusable modules consisting of isolated, self-contained functional components and linking these components through well-defined interfaces. A model library stores these model components. Model components in a library can be used as building blocks to formulate new SPAF models for different problems. For example, the users should be able to compose a machine shop model from a number of machine and assembly model components in the library.

Ease of use: SPAF makes process analysis modeling more intuitive and straightforward for domain users such as manufacturing engineers or decision makers. Composing a bigger model using existing model components in a library should be an easy process. No extensive programming, mathematics, operation research, and optimization knowledge is required. The task could be simply drag-and-drops if a graphical user interface is developed.

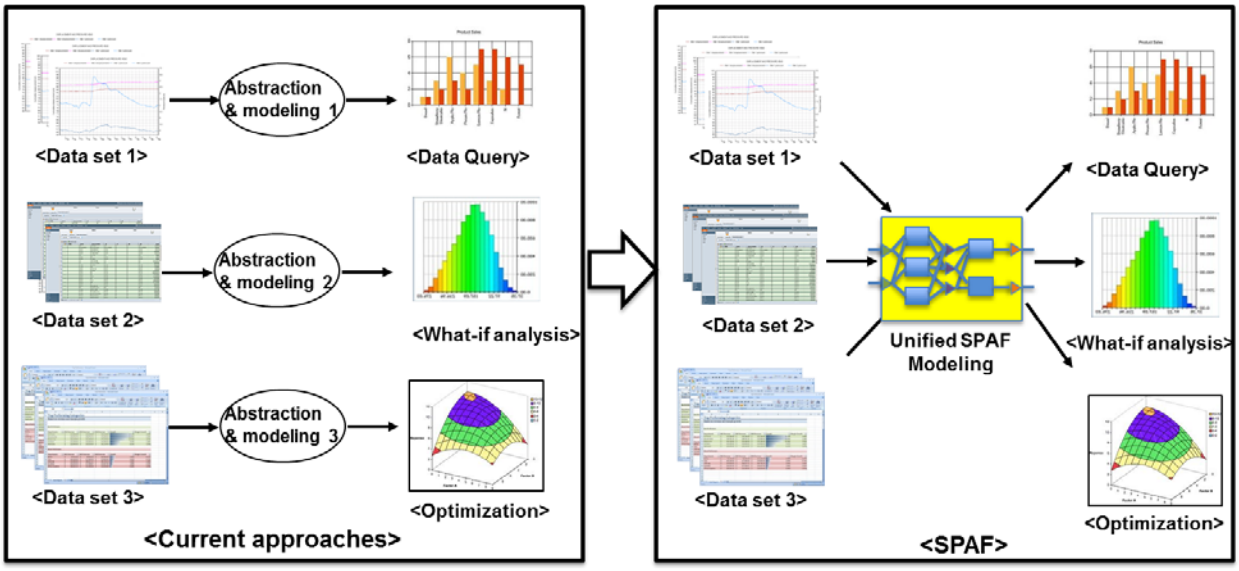


Figure 1 Current modeling approach and the unified SPAF modeling approach

To better understand the SPAF requirements, a variety of modeling languages and formalisms, listed in Table 1, have been analyzed from the perspective of the desirable features and functionalities discussed above. These languages and formalism are selected because of their suitability for at least one of the features discussed above. They include:

- Process description languages such as Process Specification Language (PSL) (ISO 2004), Business Process Model and Notation (BPMN) (OMG, 2010), and Systems Modeling Language (SysML) (OMG, 2012).
- Database query languages such as Structured Query Language (SQL) (ISO 2011) and Extensible Markup Language (XML) Query (XQuery).
- Simulation languages such as SIMAN – a general-purpose SIMulation ANalysis program for modeling combined discrete-continuous systems (Pegden et al. 1995) and Object-Oriented (OO) languages.
- Optimization languages such as AMPL (AMPL, 2011), The General Algebraic Modeling System (GAMS) (GAMS, 2010), and Optimization Programming Language (OPL) (IBM, 2012).
- Non-deterministic optimization semantics for corresponding formalism, which is used for CoJava (Brodsky & Nash, 2005) and Decision Guidance Query Language (DGQL) (Brodsky & Wang, 2008).

Process description languages are designed for process description and modeling with a modular, extensible, and reusable approach and can be easy to use via a graphical user interface. The SysML parametric models support mathematical expression for (e.g., performance constraints) the system being designed and provide a foundation for what-if analysis. However, they do not support direct data manipulation and querying, optimization, and unified modeling of different tasks.

Database query languages are specifically designed for data manipulation and querying. They are relatively easy to use, SQL-like skills are sufficient for problem modeling. However, they only allow some limited what-if analysis and optimization for what can be expressed. There is no unified modeling of different tasks. These languages do not have built-in process and sustainability metrics modeling, they are not easily reusable.

Simulation languages are excellent for what-if analysis. Some simulation tools support process modeling and have user-friendly graphical user interfaces. A few of them even started to support sustainability modeling, e.g., Witness (Waller, 2012). In most cases, simulation languages support modular, extensible, and reusable modeling. However, they are not the appropriate tools for data querying and optimization. Optimization by simulation approach is time-consuming and the results may not be as accurate as those derived by using optimization tools. There is no unified modeling capability for different tasks discussed above. Basic simulation modeling of processes requires object-oriented programming skills that most manufacturing or process engineers do not have.

Optimization languages are designed for optimization modeling. Some optimization languages such as OPL provide basic support for data manipulation and querying. However, they are not designed for what-if analysis and do not provide unified modeling capability. There is no built-in support for process and sustainability modeling. Current optimization modeling languages are not developed for reuse and modular model construction. Mathematical and optimization modeling skills are required to use them.

Optimization semantics for OO programming and database query languages are developed to provide features such as data manipulation and querying, what-if analysis, optimization, and unified modeling of these different tasks. However, there is no built-in support for process and sustainability metrics modeling, even though it potentially can be built on top of CoJava (Brodsky and Nash, 2005), which requires Java programming skills. On the other hand, DGQL (Brodsky and Wang, 2008) is relatively easy to use, just like SQL.

SPAF is designed to allow data querying, what-if analysis, optimization, and unified modeling of these different tasks. SPAF provides built-in support for process and sustainability metrics modeling with a components' library. SPAF also supports modularity, extensibility, reusability, and ease of use especially, with a graphical interface. However, the modeling effort will be similar to OPL for new process model components if there is no model library.

Table 1 A comparison table of SPAF and other languages

Model Languages / Features	Process Description Languages (PSL, BPMN, SysML)	Database Query Languages (SQL, XQuery)	Simulation Languages (SIMAN, OO languages)	Optimization Modeling Languages (e.g., AMPL, GAMS, OPL)	Optimization Semantics for OO and Query Lang's (CoJava, DGQL)	Design goal for SPAF
Data manipulation and querying	Not directly	Yes	Require modeling and programming	AMPL and GAMS are not designed for query processing; OPL has some built-in support	Yes	Yes
What-if analysis	Process structure and flow etc, not analytics	Limited (only what can be expressed as DB queries)	Yes	No	Yes	Yes
Optimization	No	Limited and not efficient	Limited and not efficient	Yes	Yes	Yes
Unified modeling for different tasks	No	No	No	No	Yes	Yes
Built-in support	Can be	Can be	Can be built on	No	Can be	Yes with a

for process modeling and sustainability metrics	extended	extended	top		extended	components library
Modular, extensible, and reusable	Yes	Does not support OO extensibility	Yes	Difficult to reuse models	CoJava - Yes; DGQL – just like SQL	Yes with a components library
Ease of use (by manufacturing and business users)	Can be easy via graphical interface	Relatively easy (SQL skills)	Programming skills to model analytics; Many allow high-level composition functionality	Math/optimization modeling skills	CoJava (programming skills); DGQL (SQL skills)	Easy for composite process, esp. if a graphical interface is added; similar to OPL for atomic process models

3. THE CONCEPT OF SUSTAINABLE PROCESS ANALYTICS FORMALISM

3.1 Context of SPAF

To explain the context of SPAF, a five-stage SM improvement methodology is depicted in Figure 2. This methodology is based on the ideas of the Six Sigma DMAIC (Define, Measure, Analyze, Improve, and Control) methodology (Chieh, 2010). The methodology proceeds through the following stages:

- Stage 1 - High-Level Assessment: Each factory assesses its sustainability level and status, defines high-level sustainability goals, and identifies areas for improvement regarding its organizational sustainability performance (for both its processes and facilities).
- Stage 2 - Problem Identification and Data Collection: To address areas of improvement identified in Stage 1, more specific case scenarios need to be defined. Modeling objectives, constraints, metrics, and control variables related to each case scenario need to be identified. Relevant data, both manufacturing process- and sustainability- related information, need to be measured, collected, and/or estimated. In reality, process- and sustainability- related data are not always available and when they are, they may exist in various forms, and would typically not yet be formalized.
- Stage 3 - Formal Process Modeling and Data Representation: To prepare for formal analysis and optimization modeling, case scenarios defined in Stage 2 need to be formally described, data collected need to be formally represented, and inputs and controls need to be modeled in a way so that the values of decision variables could be instantiated.
- Stage 4 - Decision Guidance through What-if Analysis and Decision Optimization: The formal process modeling and data representation completed in Stage 3 need to be translated into models that can be solved by commercial off-the-shelf (COTS) tools. Different tasks such as data querying, what-if analysis, and decision optimization will be performed for evaluation and analysis purposes. The analyses provide actionable recommendations to decision makers for improvement implementation.
- Stage 5 - Implementation/Execution: Decision makers can implement and execute the actionable recommendations derived from Stage 4 for sustainability improvement. Occasionally, the evaluation in the previous stage may determine that the goals cannot be achieved using the identified alternative and hence this implementation stage may involve abandoning the now determined-to-be flawed improvement plan. In either case, upon completion of Stage 5, users can continue to the next iteration of the continuous improvement cycle.

The SPAF supports this methodology at stages 3 and 4 in Figure 2. Note that the formal process modeling and data representation is done uniformly and only once for both what-if analysis and decision optimization as shown in the right hand side of Figure 1. The SPAF models enable decision makers to ask questions in the form of queries that provide computation and optimization solutions as actionable recommendations. SPAF queries include:

1. Process data queries that resemble typical database queries and can be asked directly against the explicit data.
2. What-if analysis queries to compute certain metrics for different scenarios based on available input information.
3. Decision optimization queries to find the best one (minimum or maximum as required) out of all alternatives that satisfy the constraints by using decision variables.

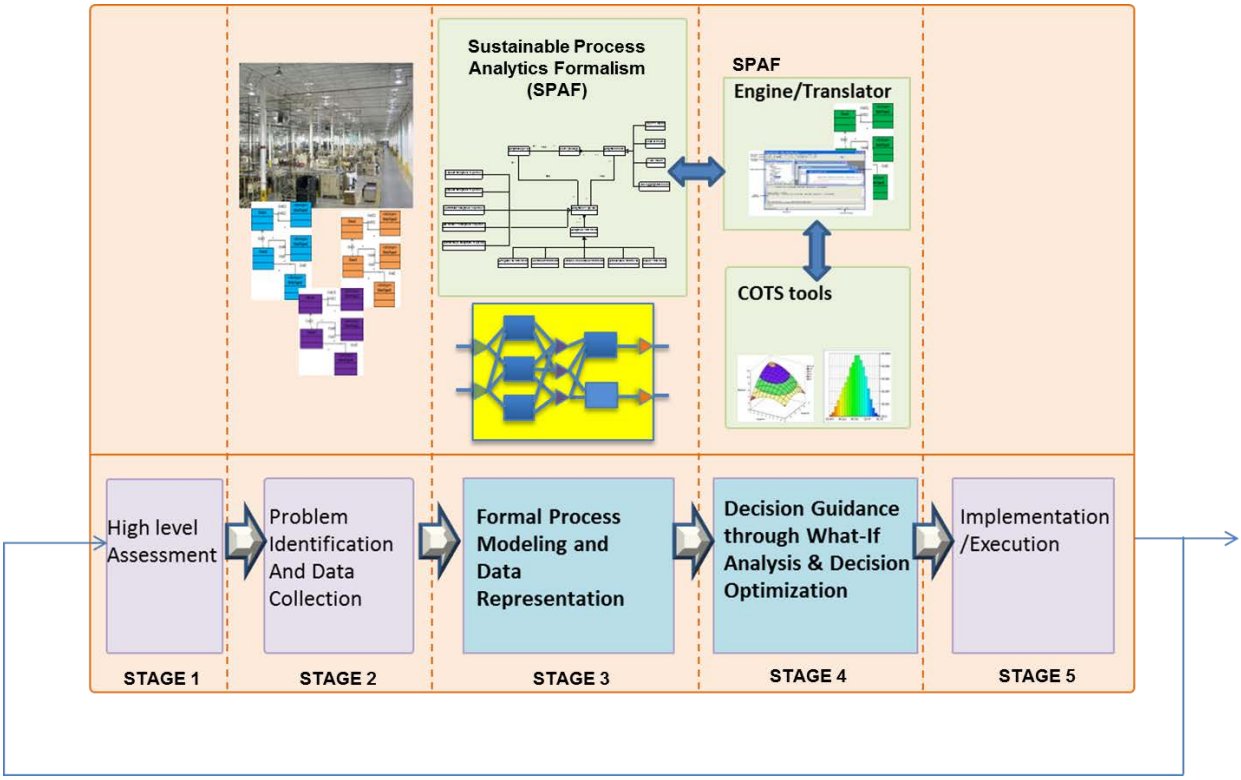


Figure 2 A model-based SM improvement methodology with SPAF

3.2 SPAF concept through a manufacturing example

The detailed SPAF syntax and formal semantics will be presented as an appendix. In this subsection, the concept of SPAF modeling is illustrated using an example of a manufacturing process. Assuming we have decided to analyze and optimize sustainability performance for a manufacturing process and collected data for the study. We need to first describe formally the process using SPAF and then solve the problem. The manufacturing process, depicted in Figure 3, has five sub-processes, three machining processes and two assembly processes. The composite process (large rectangle), the sub-processes (small rectangles), flows (lines), and flow aggregators (triangles) are depicted. Two parts, Part 1 and Part 2, provide input for

the three machining processes. The machining processes produce three intermediate components, Comp 1, Comp 2, and Comp 3. The components produced by the machines, A to C, flow to the assembly processes to be assembled into final products, Product 1 and Product 2. In this example, metrics that can be used to describe the composite process are cost and CO_2 emissions. In this example, three specific kinds of questions decision makers pose may include:

1. Process data questions, e.g., what is the maximum capacity of Machine A? How many of Product 2 needs to be produced over a scheduled week?
2. What-if analysis questions, e.g., what are the total cost, energy consumption, and CO_2 emissions for a scheduled weekly production under a particular production plan?
3. Decision optimization questions, e.g., how should production plans be set for the machines, the assembly stations, and the flow distributions among them so that the scheduled weekly production can be met within the weekly CO_2 cap and at a minimal cost?

To answer these questions, the process structure, flow, sub-process relationships, and associated data need to be clearly understood; and the objective, metrics, constraints, and control variables need to be identified. The models can be expressed with identified data and variables and metrics computation expressions. Optimization models can be formulated with constraints and objectives. The detail SPAF modeling will be discussed in Section 4.

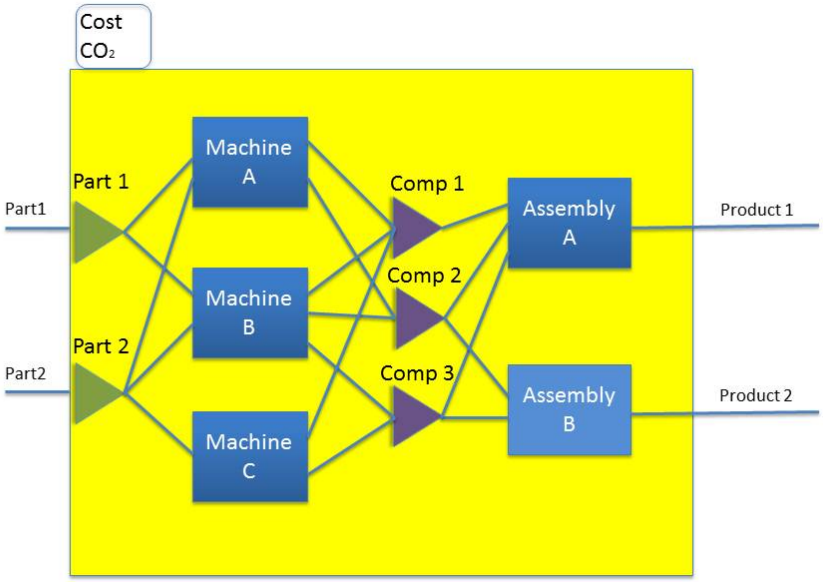


Figure 3 An example: a two-product-manufacturing process for SPAF modeling

3.3 Structures of SPAF

The goals of SPAF development are as follows. On the one hand, SPAF needs to be sufficiently expressive for the SM key performance indicators such as energy and material consumption, emission, and cost in industrial scenarios. On the other hand, the formalism needs to be simple for ease of use, which means that high-level abstraction needs to be used. Relevant industry-accepted languages, standards, and tools should be used. The SPAF models are human and machine readable, ready for database storage, translator development, and exchange among modelers/users/systems.

The SPAF includes two major parts: (1) Generic analytics language – represents generic analytics knowledge and (2) Process description and sustainability metrics model templates - supports sustainable process modeling and sustainability metrics computation.

Figure 4 presents a class diagram for the generic part of the SPAF to describe the structure of the components and their relationships. The components are explained as follows:

- An *analytical statement* may be assignments, constraints, or decision variable declarations.
- An *analytical sequence* is a sequence of analytical statements. SPAF analytical sequences can be an *explicit, implicit, constraints, alternatives,* or *optimization* analytical sequence to satisfy the different modeling needs.
  - *Explicit analytical sequence* provides all the required process structure and data in the form of assignment statements that assign a data value to each variable in the sequence.
  - *Implicit analytical sequence* extends an *explicit analytical sequence* with assignment statements that assign an expression (which computes a value from previously defined variables) to a new variable.
  - *Constraint analytical sequence* extends *implicit analytical sequence* with constraint expressions in terms of previously defined variables.
  - *Alternative analytical sequence* extends a *constraint analytical sequence* with declaration of variables that do not have an assigned value. Intuitively, an alternative sequence defines a set of feasible computations, corresponding to all possible assignment of values into the declared variables that satisfy the constraints.
  - *Optimization analytical sequence* is an *alternatives analytical sequence* with the added optimization directive “minimize” or “maximize.” Intuitively, it states the optimization problem of finding an instantiation of values into declared variables that satisfy all the constraint statements, and minimizes/maximizes the indicated objective.
- An *analytical model* is an (non-optimization) analytical sequence with the added model name and a possible parameter Id.
- A *model package* is a set of *analytical models*. Intuitively, it serves as a reusable repository of analytical knowledge, which could be used for different applications.
- An *analytical query* is a pair of an analytical sequence and a model package that includes all models that are referred to, directly or indirectly, from the process model.

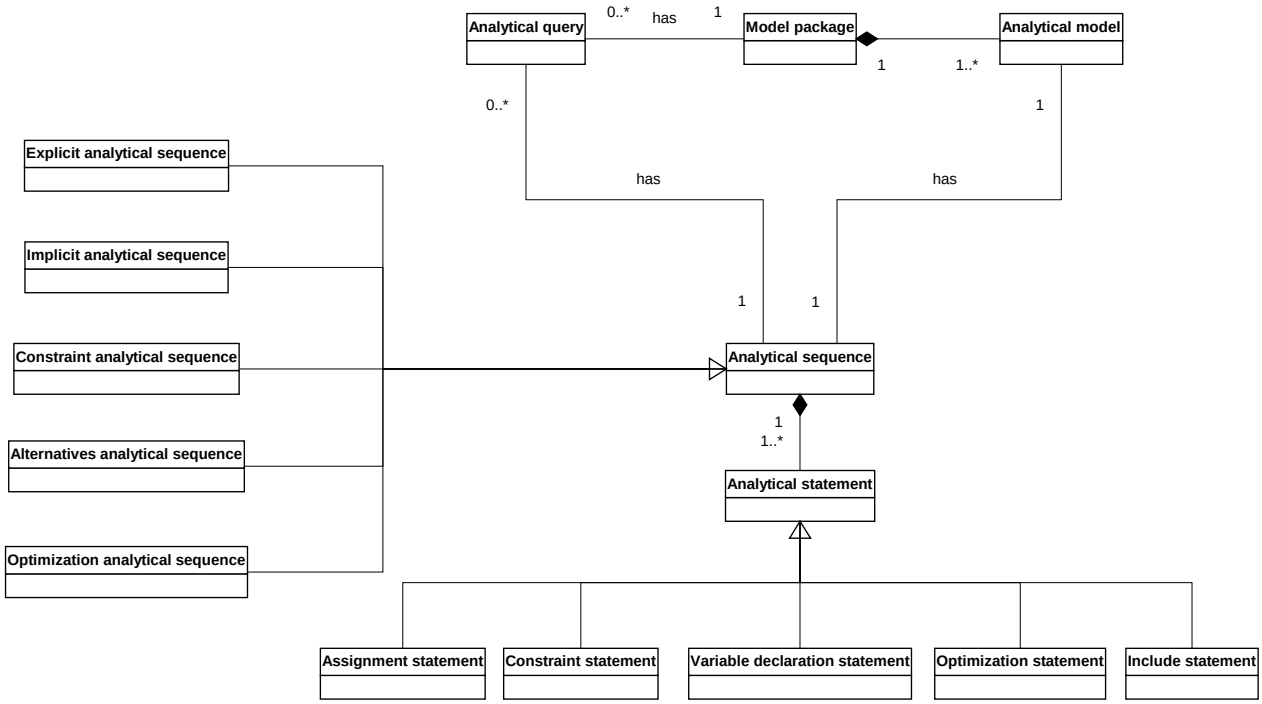


Figure 4 SPAF class diagram – generic analytics language

Figure 5 shows a hierarchical diagram for the process description and sustainability metrics model templates part of the SPAF. For the purpose of formal process representation, four different SPAF model components have been defined. These model components include context, flow, flow aggregator, and process. Context describes data that are globally accessible by all model components; context model includes context ID and associated data attributes. Flow describes entities that physically flow into and out of a process. Flow aggregator aggregates multiple sources of the same type of flows and distributes the outputs as inputs to the other processes. The sum of all inputs of an aggregator must equal the sum of its outputs. A process can be a composite process or a sub-process. The attributes of each component are shown in Figure 6. The process description and sustainability metrics model templates part includes syntax of the SPAF *analytical models* such as *process model, flow model, flow aggregator model,* and *context model.* These analytical models must adhere to a more specialized structure. Figure 5 and Figure 6 are connected through the *analytical model.*

A process model may be a generic process model or a specific process model. A generic process model can be stored in a model library and reused for developing specific process models. Flow and flow aggregator models may be those for discrete flows, continuous flows, or batch flows. Sustainability metric aggregator models are specifically designed for sustainability metric aggregation for environmental indicators such as energy, emission, material, and waste and economic indicators such as investment, revenue, cost, and return on investment (ROI).

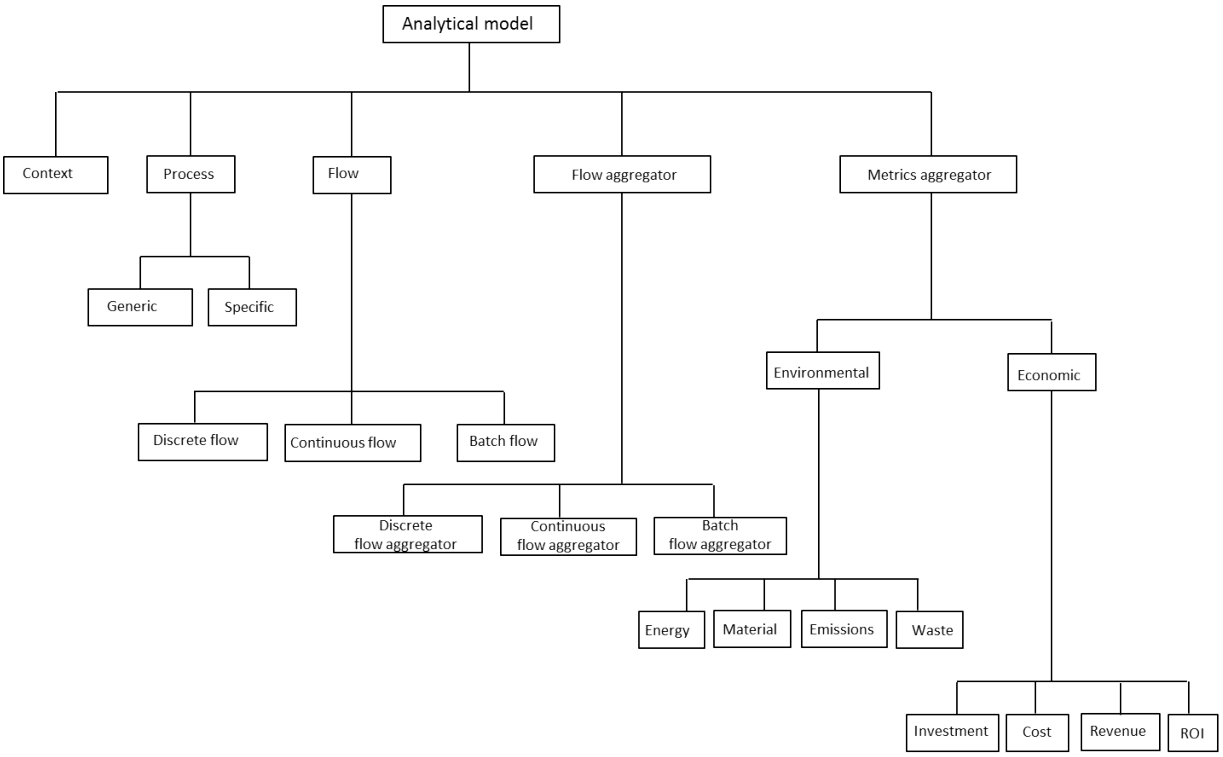


Figure 5 A hierarchical diagram for the process description and sustainability metrics model templates

- context
  - model name
  - context data
- flow
  - id
  - model name
  - context
  - flow data
- flow aggregator
  - id
  - flow name
  - context
  - input flow
  - output flow
- process
  - id
  - model name
  - context
  - input flows
  - output flows
  - sub-process
  - flow aggregators

Figure 6 SPAF process description model components

SPAF libraries collect model components of both generic and specific models. Metrics model components can also be stored in a library. Figure 7 shows examples of generic model components in a SPAF library, e.g., process model components such as “baseSeqTransform” and “baseProcessComposer”, and flow and

flow aggregator models for discrete and continuous flows, and metrics models for environmental and economic indicators. Figure 8 lists examples of specific model components in the library; these models were developed for the two-product-manufacturing example. It includes specific process models for "machine A," "machine B," "machine C," "assembly A," "assembly B," and composite model "twoProdManuf" for the overall process. The model components in the SPAF library provide reusable building blocks and can be used as templates for a family of manufacturing processes; each model or template can be reused with some adjustment for different cases within the family. New models can be added to the SPAF library. Moreover, the existing models within the library can be executed with new data so that different companies that have the same problems could use the models by inputting their data to seek company-specific decision guidance. Using model components in the library, modelers can create SPAF models and queries for their operations more effectively and efficiently.

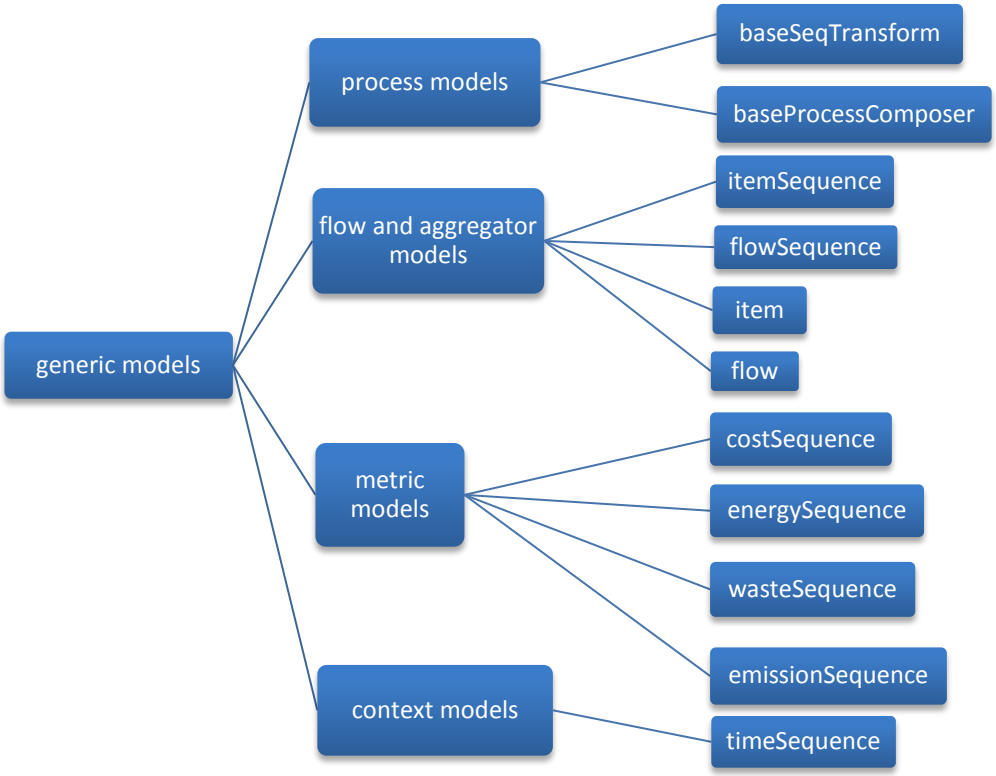


Figure 7 SPAF component library: example of generic models

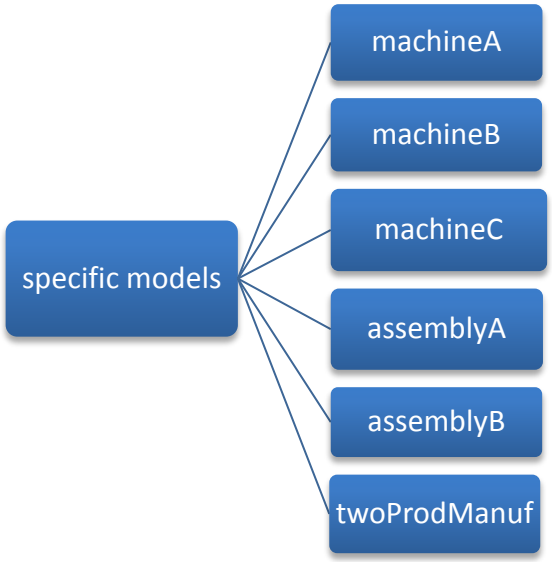


Figure 8 SPAF component library: examples of specific models

3.4 Potential users of the SPAF

Most people within the factory floor do not have the expertise to formulate and solve optimization problems using Mathematical Programming (MP) or Constraint Programming (CP). The SPAF makes process analysis modeling more intuitive and straight forward for domain users such as manufacturing engineers. Figure 9 shows a high-level use-case model for a typical implementation of a decision guidance management system using SPAF. There are three types of potential users, each with different roles in using and/or maintaining the system.

- SPAF *analyst*: The analyst is a primary user of the system. S/he may be an engineer who is in charge of defining case scenarios, setting up analysis model objectives together with decision makers, and identifying sustainability metrics, constraints, and controls. S/he will collect data from the factory floor and represent process structure, flow, and metrics using SPAF.
- *Decision maker*: The decision maker is the end user of the system. S/he identifies the SM objectives/goals and provides model requirements. The decision maker will query the knowledge base, ask what-if analysis questions, and/or make optimization requests with applied constraints and control data for a specific problem. S/he may also compose SPAF models simply using model components in the SPAF library.
- SPAF *knowledge-base administrator*: The knowledge-base administrator serves as a system administrator who is responsible for updating system data, creating reusable knowledge artifacts, helping the analyst develop new applications, maintaining the SPAF model library, modifying/improving the system design, and maintaining/enhancing the system.

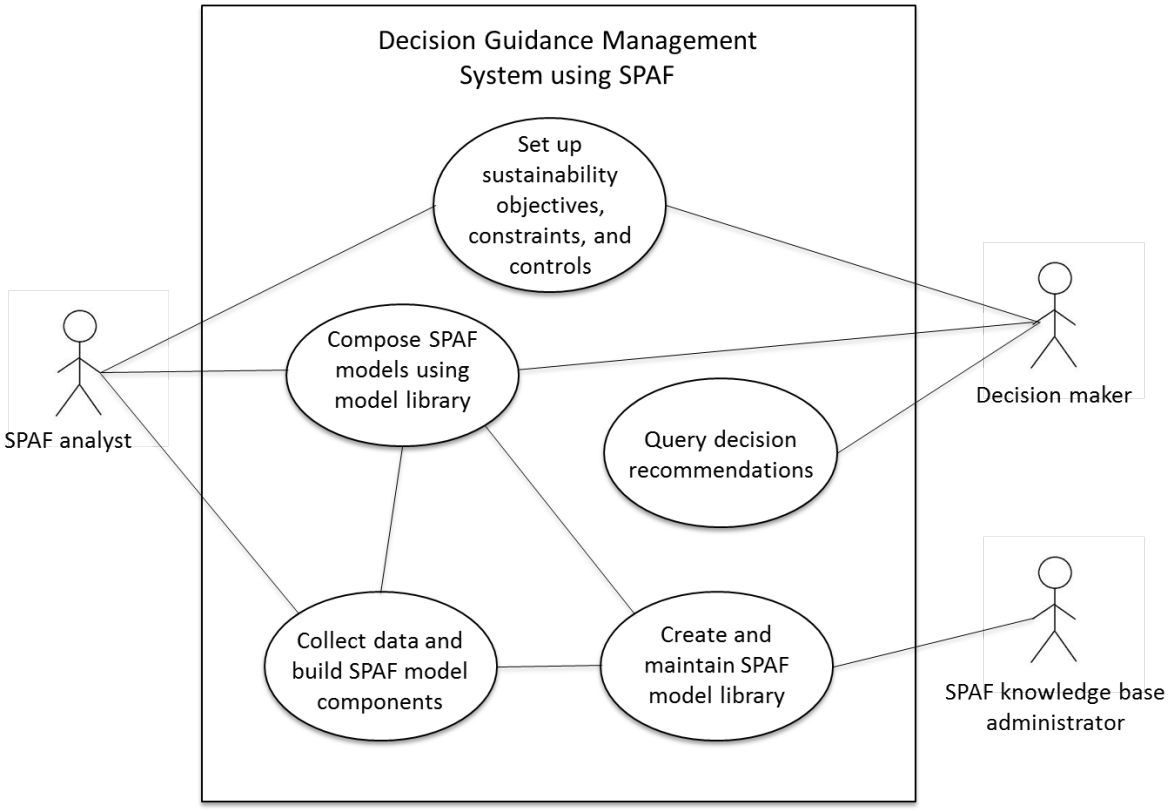


Figure 9 Use case for implementation of decision guidance management system using the SPAF

4. ILLUSTRATIVE EXAMPLE USING SPAF

In this section, the two-product-manufacturing example introduced in Section 3.2 is modeled using SPAF and discussed in details. Figure 10 to Figure 20 show the detailed SPAF process models and possible queries. First, assuming the SPAF model, *twoProductsManuf ()*, is developed and all data are provided, a what-if analysis query requires only four statements (Figure 10). The first two statements include the data models for product demand data indicating quantities for each final product (i.e., Product 1 and Product 2) and production plan data that describes the numbers of components should be produced by each machining or assembly process. The third statement includes the SPAF model for the two-product-manufcturing process and finally a constratint statement that indicates the total amount of CO_2 generated by this manufacturing scenario should be less than or equal to 50 metric tons. Since all data required for the query are available, the query is actually a deterministic computational model that calculates the total cost within the limit of total CO_2 less than 50 metric tons. The query provides answers for both total cost and total CO_2.

```
include data productDemand();
include data productionPlanData();
include process twoProductsManuf ();
twoProductsManuf.totalCO2 ≤ 50;
```

Figure 10 A what-if analysis query for the two-product-manufacturing process

In a case where the production plan data are not provided, an optimal production plan with minimal total cost needs to be determined, given the same CO_2 limitation as a constraint. The same SPAF model, *twoProductsManuf ()*, is used as shown in Figure 11, but the production plan data (as shown in Figure 10) is not provided and an optimizaiton statement is added to "*minimize*" the total cost. There are still only four lines of code; however, since the production plan data are unknown, all data previously provided explicitly in the production plan data model become decision variables that need to be instantiated to satisfy all the constraints. Now it is no longer a deterministic computational model. It actually describes a set of non-deterministic computational paths, each corresponding to an instantiation of set of values for the decision variables. Some of the non-deterministic computation paths are "feasible," i.e., they satisfy all of the constraints (the total CO_2 constraints as well as the internal constraints) while others are not feasible. The semantics of the optimizaion query in Figure 11 is to find a non-deterministic optimizaiton path that leads to the minimal total cost among all feasible computation paths. The query results inlcude not only both total cost and total CO_2 but also the optimal production plan configuation (i.e., the optimal number of components being produced by each machining or assembly process).

```
include data productDemand();
include process twoProductsManuf ();
twoProductsManuf.totalCO2 ≤ 50;
Minimize twoProductsManuf.totalCost;
```

Figure 11 Optimization query for the two-product-manufacturing process

As stated above and shown in Figure 10 and Figure 11, the same SPAF model, *twoProductsManuf ()*, can be used for different kind of queries, such as what-if analysis (in Figure 10) and decision optimization (in figure 11). These query examples demonstrate that SPAF provides a unified modeling capability. The queries against SPAF models are simple and straightforward.

Figure 12 shows the model for the "context" component, its model name is "*timeSequence*." "*timeSequence*" is declared as a set of string "*day*," which is a tuple consisting of three fields "*day*, *month*, and *year*" of type integer. The three dots "…" expresses the missing data that need to be instantiated as a constant, or an expression before the data is used.

```
context timeSequence() {
tuple day {
      int day;
      int month;
      int year;
   };
{day} timeSequence = ...;
}
```

Figure 12 Context model for the two-product-manufacturing example

Figure 13 shows the model for the "flow" component, the model name is "*itemSequence*." The "*Id"*parameter of "*itemSequence*" will be replaced by the value of a parameter in an *include* statement. An *include* statement calls another model. It is similar to a subroutine call. The context model *itemSequence ()* is included using a *include* statement. A one-dimensional array "*Id.qty*" is an integer

array. "*Id.qty*" is indexed by the finite set of tuples defined by the "*timeSequence*" variable from the *context* model. The elements of the array represent quantities of the flow in that day.

```
flow itemSequence (Id) {
string Id.matchName = …;
include context timeSequence();
int  Id.qty[timeSequence];
forall (d in timeSequence)   Id.qty[d] ≥ 0 ;
}
```

Figure 13 Flow model for the two-product-manufacturing example

Figure 14 shows a model for the "flow aggregator" component, its model name is "*itemSeqAggr*." As described earlier, "*Id"* is a parameter whose value will be provided by an *include* statement. In the first statement of this model, the context model *itemSequence ()* is included. Next, a variable "*Id.flowType*" is declared as a string "*itemSequence.*" "*Id.inputFlows*" and "*Id.outputFlows*" are declared as a set of strings and will be instantiated separately. "*Id.flows,*" the union of "*Id.inputFlows*" and "*Id.outputFlows,*" is also a set of strings. For every flow in "*Id.flows,*" its quantity for the day in "*timeSequence*" is an integer. The *forall* statement defines a constraint for each day in "*timeSequence,*" it indicates that the total number of the "inputFlows" for a day must equal the total number of the "outputFlows."

```
flow aggregator  itemSeqAggr (Id) {
include context timeSequence();
string   Id.flowType = "itemSequence";
{string}    Id.inputFlows = ...;
{string} Id.outputFlows = ...;
{string} Id.flows = Id.inputFlows union Id.outputFlows
for (i in Id.flows) int i.qty[timeSequence];
forall (d in timeSequence)
             sum (i in Id.inputFlows) i.qty[d]
          == sum (o in Id.outputFlows) o.qty[d];
}
```

Figure 14 Flow aggregator model for the two-product-manufacturing example

Figure 15 shows a model of a generic atomic process, which is an end process in which there is no sub-process, e.g., Machine A. The model name is "*baseSeqTransform.*" *Id* is provided when it is called. For every output flow, the flow model is being *included* with a parameter of the flow name. Two arrays of

floats are declared for both "*Id.costPerUnit*" and "*Id.CO_2PerUnit*"; their index set is the set of output flows for this atomic process. A two-dimensional array of integer "*Id.inputPerOutput*" represents the number of input flows required for each output flow. For each production day, the cost of the atomic process is computed as the unit cost of each output flow times the number of output flows produced in that day; the CO_2 emission is computed as the unit CO_2 emission from each output flow times the number of output flows produced in that day. A constraint is that the total number of input flows needed in that day must equal the number of output flows produced in the same day times the number of input flows required for each output flow.

Once the generic atomic process model component is developed, it can be saved and reused for generating specific atomic process model components.

```
process baseSeqTransform(Id) {
include context timeSequence();
string Id.name = …;
{string} Id.inputFlows = …;
{string} Id.outputFlows = …;
for (i in Id.outputFlows) include flow itemSequence(i);
float Id.costPerUnit[Id.outputFlows] = ...;
float Id.CO2PerUnit[Id.outputFlows] = ...;
int   Id.inputPerOutput[Id.outputFlows][Id.inputFlows] = …;
float Id.cost[d in timeSequence] =
           sum(r in outputFlows) Id.costPerUnit[r] * r.qty[d];
float Id.CO2[d in timeSequence] =
           sum(r in outputFlows) Id.CO2PerUnit[r] * r.qty[d];
for (i in Id.inputFlows) { i.qty[d in timeSequence] =
    sum (o in Id.outputFlows) Id.inputPerOutput[o][i] * o.qty[d];
include flow itemSequence(i);
   }
}
```

Figure 15 An atomic process model for the two-product-manufacturing example

Figure 16 shows a specific atomic process model for Machine A. Model name is "*machine.*" It starts with the instantiation of the declarations. "*Id.name*" is given as "*machine.*" "*Id.inputFlows*" is a set of two strings, "*part1toMaA*" and "*part2toMaA.*" Input flows of "*part1toMaA*" and "*part2toMaA*" are given names "*part1*" and "*part2*" respectively. "*Id.outputFlows*" is a set of two strings "*comp1fromMaA*" and "*comp2fromMaA.*" Output flows, "*comp1fromMaA*" and "*comp2fromMaA,*" are given names of "*comp1*" and "*comp2*" respectively. Two float type arrays for "*Id.costPerUnit*" and "*Id.CO_2 PerUnit*" are both given in a pair (index, value) of elements as ["*comp1fromMaA*": 35.0, "*comp2fromMaA*": 65.0] and ["*comp1fromMaA*": 0.05, "*comp2fromMaA*": 0.02] respectively. The two-dimensional array

"*Id.inputPerOutput*" is instantiated as a pair of ["*comp1fromMaA*": ["*part1toMaA*": 1,"*part2toMaA*": 1], "*comp2fromMaA*": ["*part1toMaA*": 1,"*part2toMaA*": 3]] The last step is to *include* the generic model "*baseSeqTransform.*" "*machineA*" is the parameter.

Other atomic processes in the two-product-manufacturing example including Machine B, Machine C, Assembly A, and Assembly B are similar to the process model of Machine A.

```
process machineA () {
string Id = "machineA";
{string} Id.inputFlows = {"part1toMaA","part2toMaA"};
string part1toMaA.name = "part1";
string part2toMaA.name = "part2";
{string} Id.outputFlows = {"comp1fromMaA","comp2fromMaA};
string comp1fromMaA.name = "comp1";
string comp2fromMaA.name = "comp2";
float Id.costPerUnit [Id.outputFlows] = ["comp1fromMaA": 35.0,
      "comp2fromMaA": 65.0];
float Id.CO2PerUnit [Id.outputFlows] = ["comp1fromMaA": 0.05,
      "comp2fromMaA": 0.02];
int Id.inputPerOutput [Id.outputFlows][Id.inputFlows] =
   ["comp1fromMaA": ["part1toMaA": 1,"part2toMaA": 1],
   "comp2fromMaA": ["part1toMaA": 1,"part2toMaA": 3]]
include process baseSeqTranform ("machineA");
}
```

Figure 16 Atomic process model for Machine A

Depicted in Figure 17 is a generic process composer model, which includes all flow models and all sub-processes models, and formulates the flow aggregator models automatically instead of being given explicitly. Again, three dots indicate that the input and output flows, and sub-processes need to be instantiated before this generic model is called. For every flow, the model needs to be included and its model name, "*matchName,*" and aggregator name need to be defined before this generic model is called. "*flowsToAggregators,*" a set of strings, are the union of input flows to the composite process and output flows from all sub-processes. "*flowsFromAggregators,*" another set of strings, are the union of all input flows to all sub-processes. All aggregator flow names are in the set of strings that include all "*matchName*" of the flows. For every flow "*matchName,*" if the name of input flow is in the "*flowsToAggregators*" and the name of the output flow is in the "*flowsFromAggregators,*" then include the flow aggregator model with the flow's "*matchName*" as a parameter.

```
process   processComposer(id) {
{string} Id.inputFlows = …;
{string} Id.outputFlows = …;
```

```
{string} Id.subProcesses = …;
{string}  Id.flows = Id.inputFlows union Id.outputFlows;
for (f in Id.flows) {
   string f.model = ...;
   include flow f.model(f);
   string f.matchName = ...;
   string f.aggrModel = ...;
};
for (p in Id.subProcesses) {
   string p.model = ...;
   include process p.model(p);
}
{string} Id.flowsToAggregators =
   Id.inputFlows union union(p in Id.subProcesses) p.outputFlows;
{string} Id.flowsFromAggregators =
   Id.outputFlows union union(p in Id.subProcesses) p.inputFlows;
{string} Id.allFlows =
             Id.flowsToAggregators union Id.flowsFromAggregators;
{string} Id.matchNames =
             distinct({f.matchName | f in Id.allFlows});
for (n in Id.matchNames) {string Id.n.aggrModel =
      first({f.aggrModel |f in Id.allFlows : f.matchName == n});
{string} Id.n.inputFlows = {
          f | f in Id.flowsToAggregators : f.matchName == n};
{string} Id.n.outputFlows = {
         f | f in Id.flowsFromAggregators : f.matchName == n};
include flow aggregator Id.n.aggrModel(Id.n);
};
```

Figure 17 Generic composite process model

Figure 18 shows the metrics aggregator models that compute daily total cost and CO_2. The daily total cost and CO_2 are the sum of cost and CO_2 for all sub-processes.

```
metric aggregator costSequence(Id) {
include context timeSequence ();
```

```
{string} Id.subProcesses = ...;
float Id.cost[t in timeSequence] =
                    sum(p in Id.subProcesses) p.cost[t];
}

metric aggregator CO2Sequence(Id) {
include context timeSequence ();
{string} Id.subProcesses = ...;
float Id.CO2[t in timeSequence] =
                sum(p in Id.subProcesses) p.CO2[t];
}
```

Figure 18 Metric aggregator model

Figure 19 shows the composite process model. Model *Id* is "*twoProductsManuf.*" It includes the context model *itemSequence ().* "*Id.inputFlows*" is given as a set of two strings {"*part1in*", "*part2in*"}. "*Id.outputFlows*" is given as a set of two strings of {"*product1*", "*product2*"}. "*matchNames*" are also given. "*Id.subProcessess*" is instantiated as a set of five strings of {"*machine,*" "*machineB*", "*machine,*" "*assembly,*" "*assemblyB*"}. The generic process model *processComposer* is called to include all atomic sub-processes models defined previously. *Float* type of data for extra facility cost and CO_2 per day ($1 750 and 0.3 metric tons) are provided. The metric aggregator models, *costSequence (Id) and CO2Sequence(Id),* are included. Total cost for each day is the extra facility cost plus daily cost for all sub-processes. Total CO_2 for each day is the extra facility CO_2 plus total sub-processes CO_2.

An alternative modeling method is to explicitly instantiate all flows and flow aggregators, e.g., inputs and outputs of Part1, Part2, Comp1, Comp2, and Comp3 are all specified as sets of strings. Then every flow aggregator is *included* with its name as a parameter.

```
process   twoProductsManuf () {
string Id = "twoProductsManuf";
include context timeSequence();
{string} Id.inputFlows = {"part1in", "part2in"};
{string} Id.outputFlows = {"product1", "product2"};
string part1in.matchName = "part1";
string part2in.matchName = "part2";
string product1.matchName = "product1";
string product2.matchName = "product2";
{string} Id.flows = Id.inputFlows union Id.outputFlows;
for (f in Id.flows) f.model = "itemSequence";
{string} Id.subProcessess = {
```

```
        "machineA", "machineB", "machineC","assemblyA", “assemblyB"};
    for (p in Id.subProcesses) p.model = p;
    include process processComposer(Id);
    float Id.extraCostSequence[t in timeSequence] = 1750.0;
    float Id.extraCO2Sequence[t in timeSequence]= 0.3;

    include   metric aggregator costSequence(Id);
    include   metric aggregator CO2Sequence(Id);
    float Id.totalCost =
        sum(t in timeSequence)(Id.cost[t] + Id.extraCostSequence[t]);
    float Id.totalCO2 =
        sum(t in timeSequence)(Id.cost[t] + Id.extraCO2Sequence[t]);
    }
```

Figure 19 Composite process model for the two-product-manufacturing process

After we explained all the SPAF model components for the example, we need to examine the data required by the queries in Figure 10 and Figure 11. A context data sequence is shown in Figure 20. Its product demand data model is listed in Figure 21, in which the quantities of the two products are given for each production day. For example, *[<4, 9, 2012>: 6]* in the first line means demand for Product 1 on September 4th, 2012 is 6.

```
{day} timeSequence = {
<4, 9, 2012>, <5, 9, 2012>,
<6, 9, 2012>, <7, 9, 2012>, <8, 9, 2012>,
}
```

Figure 20 A context data sequence for the two-product-manufacturing process

```
int product1.qty [timeSequence] = [<4, 9, 2012>: 6, <5, 9, 2012>: 8, <6, 9, 2012>: 5, <7, 9, 2012>: 7, <8, 9, 2012>: 4];
int product2.qty [timeSequence] = [<4, 9, 2012>: 5, <5, 9, 2012>: 6, <6, 9, 2012>: 3, <7, 9, 2012>: 4, <8, 9, 2012>: 5];
```

Figure 21 Product demand data model for product 1 and product 2

A what-if scenario for the example is described as follows: if the process engineer uses a predefined production plan, i.e., all the data such as numbers of part 1 and part 2, numbers of components flows into

and out of Machine A, Machine B, Machine C, and number of components flows into Assembly A, and Assembly B each day are fixed. This means that all the data needed in the SPAF model are explicitly provided and can be used to computer metrics using formulas. The four lines of what-if query (as shown in Figure 10) can be expanded as in Figure 22 while the constraint keeps the same as before.

```
{day} timeSequence = { <5, 11, 2012>, <6, 11, 2012>, <7, 11, 2012>, <8, 11, 2012>, <9, 11, 2012>, }
int product1.qty [timeSequence] = [6, 8, 5, 7, 4];
int product2.qty [timeSequence] = [5, 6, 3, 4, 5];
// data for fixed production plan
int part1.qty [timeSequence] = [ 98, 128, 73, 107, 56];
int part2.qty [timeSequence] = [127, 166, 96, 139, 221];
int part1ToMaA.qty [timeSequence] = [0, 0, 4, 0, 0];
int part2ToMaA.qty [timeSequence] = [0, 0, 0, 0, 0];
int part1ToMaB.qty [timeSequence] = [98, 128, 69, 107, 56];
int part2ToMaB.qty [timeSequence] = [127, 166, 96, 139, 61];
int part2ToMaC.qty [timeSequence] = [0, 0, 0, 0, 160];
int comp1FromMaA.qty [timeSequence] = [0, 0, 0, 0, 0];
int comp2FromMaA.qty [timeSequencetimeSequence] = [0, 0, 4, 0, 0];
int comp1FromMaB.qty [timeSequence] = [30, 40, 25, 35, 20];
int comp2FromMaB.qty [timeSequence] = [23, 30, 14, 25, 17];
int comp3FromMaB.qty [timeSequence] = [22, 28, 16, 22, 2];
int comp1FromMaC.qty [timeSequence] = [0, 0, 0, 0, 0];
int comp3FromMaC.qty [timeSequence] = [0, 0, 0, 0, 16];
int comp1ToAsA.qty [timeSequence] = [30, 40, 25, 35, 20];
int comp2ToAsA.qty [timeSequence] = [18, 24, 15, 21, 12];
int comp3ToAsA.qty [timeSequence] = [12, 16, 10, 14, 8];
int comp2ToAsB.qty [timeSequence] = [5, 6, 3, 4, 5];
int comp3ToAsB.qty [timeSequence] = [10, 12, 6, 8, 10];
include process twoProductsManuf ();
twoProductsManuf.totalCO2 ≤ 50;
```

Figure 22 What-if query for the two-product-manufacturing example

This is a deterministic computational model, however, since there is a constraint statement in the query and there are also other data integrity constraints within the models, the answers have to satisfy all the constraints. The results of the what-if scenario are: the total cost is $30 000 with a total of 35.11 metric tons of CO_2. Note that changes in any input data will result in a different set of solutions.

For the optimization query listed in Figure 11, input data such as weekly production schedule and customers' demand for Product 1 and Product 2 are provided. The sustainability goal is to determine an optimal production plan that minimizes the total cost within a CO_2 bound of 50 metric tons. The optimization model performs multiple non-deterministic computations, each instantiates decision variables (quantities of flows in each configuration) using values that satisfy all the constraints. Among those sets of configurations that satisfy all the constraints, the system will automatically find a configuration (i.e., a production plan) that minimizes the total cost. Figure 23 shows the optimization result screen of an implementation using IBM ILOG CPLEX. The optimal production plan for the scheduled five days is derived. The optimization results show that the minimal total cost is $28 023 with total 36.72 metric tons of CO_2. The results also indicate that due to the higher operation cost of the Machine B, it is not recommended to use Machine B to produce any of the components, i.e., Comp1, Comp2, and Comp3. Note that changes in any of the input data and constraints will also affect the values of decision variables and decision expressions.

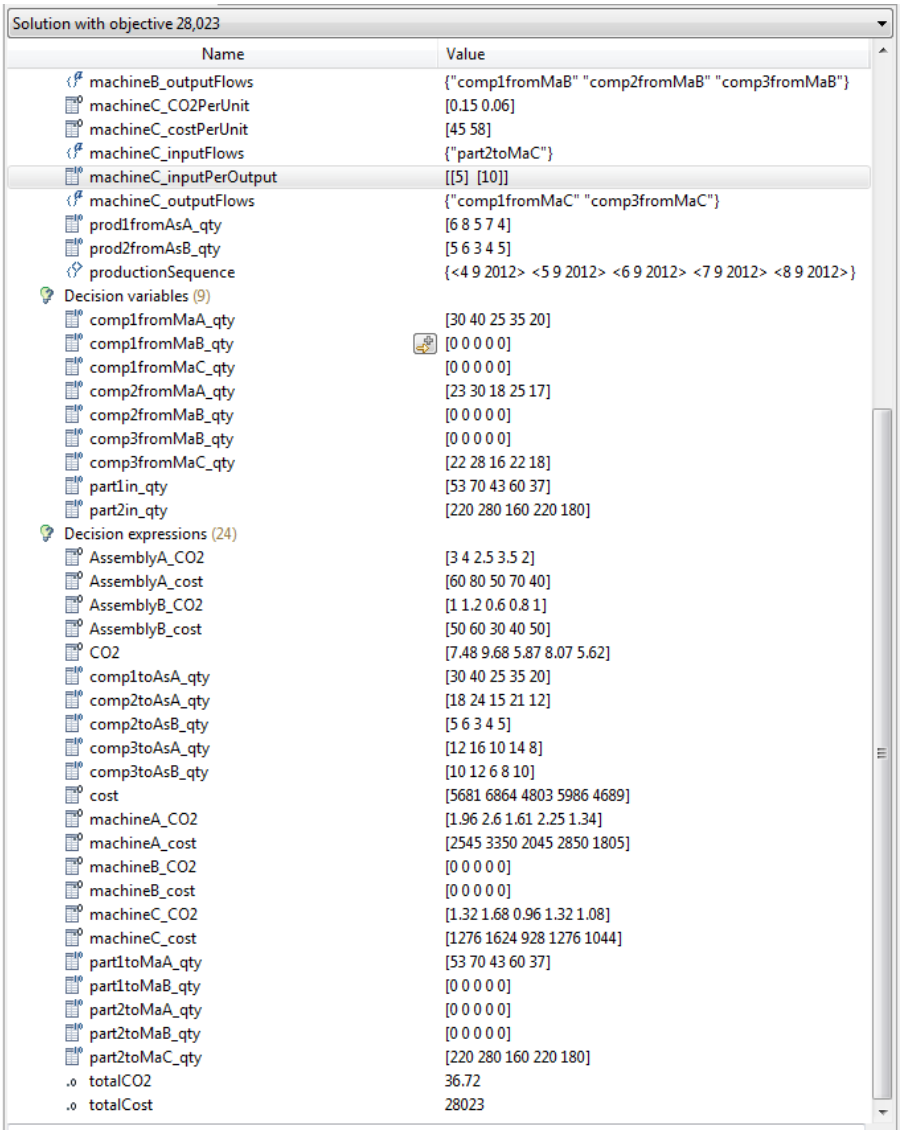


Figure 23 Optimal solution screen of two-product-manufacturing example

5. CONCLUSION AND FUTURE WORK

This paper proposed a NIST-developed Sustainable Process Analytics Formalism that allows manufacturers to: (1) formally represent sustainable process structure, flow, process data, control variables, and process analytical model of sustainability metrics and constraints for quantitative sustainability analysis; and (2) analyze and make decisions on improvement alternatives with modeling and optimization tools. The formalism provides platform-independent process-knowledge description and supports what-if analysis and decision optimization for decision makers. The use of the SPAF formalism is illustrated through a two-product manufacturing process example. The SPAF syntax, formal semantics, and query computation algorithm are presented in the appendix.

The formalism will be deployed to industry through case studies and contributions to standard development efforts. When implemented for real manufacturing applications, the formalism will help manufacturers quantify their sustainability efforts for improvement of energy and material efficiency, lower emissions, and save cost.

Future work includes (1) examining diverse manufacturing processes to identify extra process analytical needs; (2) supporting taxonomies, and metrics from unit manufacturing, assembly processes, and production planning; (3) supporting smart manufacturing by enhancing the SPAF; (4) developing translators that automatically translate SPAF to formal optimization/simulation models, which can then be solved by commercial optimization tools; (5) developing graphical representation of SPAF based on modeling language such as UML, SysML, or BPMN; (6) performing industrial case studies to evaluate and validate the formalism and the capabilities; and (7) standardizing the SPAF.

DISCLAIMER

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

REFERENCES

AMPL. (2011). "A Modeling Language for Mathematical Programming." *Available via <http://www.ampl.com/> [accessed Jan. 2012]*.

Berglund, J. K., Michaloski, J. L., Leong, S. K., Shao, G., Riddick, F. H., Arinez, J., et al. (2011). Energy Efficiency Analysis for a Casting Production System. *Proceedings of the 2011 Winter Simulation Conference*, (pp. 1060-1071).

Brodsky, A., and Nash, H. (2005). CoJava: a unified language for simulation and optimization. *The Conference on Object Oriented Programming Systems Languages and Applications*, (pp. 194 - 195).

Brodsky, A., and Wang, S. X. (2008). Decision-Guidance Management System (DGMS): Seamless Integration of Data Acquisition, Leaning, Prediction, and Optimization. *The 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, (pp. 71–81). Hawaii.

Chieh, C. (2010). Six Sigma Basics: DMAIC Like Normal Problem Solving. *Available via < http://www.isixsigma.com/new-to-six-sigma/dmaic/six-sigma-basics-dmaic-normal-problem-solving/> [accessed August. 15, 2013]*.

CPLEX. (2011). *Available via <http://en.wikipedia.org/wiki/CPLEX> [accessed August. 2013]*.

DOC. (2010). Sustainable Manufacturing Initiative and Public-private Dialogue. *Available via <http://www.trade.gov/competitiveness/sustainablemanufacturing/index.asp> [accessed Jan. 15, 2012]*.Fujitsu. (2011). Fujitsu Offers Energy-Saving Green Infrastructure Solution. Available via <http://www.fujitsu.com/global/news/pr/archives/month/2007/20071210-02.html> [accessed March 2012].

GAMS. (2010). An Introduction to General Algebraic Modeling System (GAMS). *Available via <http://www.gams.com/>* [accessed July 2013].

GM. (2010). Innovation: Environment. *Available via <http://www.gm.com/corporate/responsibility/environment/facilities/index.jsp> [accessed August 2012]*.

Feng, S. C. and Joung, C. B. (2009). An Overview of a Proposed Measurement Infrastructure for Sustainable Manufacturing. Proceedings of the 7th Global Conference on Sustainable Manufacturing.

IBM. (2012). *Introducing IBM ILOG CPLEX Optimization Studio V12.2.* Available via <http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r2/index.jsp?topic=%2Filog.odms.ide.help%2FContent%2FOptimization%2FDocumentation%2FOPL_Studio%2F_pubskel%2Fglobals%2Feclipse_and_xplatform%2Fps_opl307.html>. [accessed August 2013].

ISO 18629-1:2004. (2004). Industrial Automation Systems and Integration – Process Specification Language – Part 1: Overview and Basic Principles. Available via <http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35431> [Access Sept. 2013].

ISO/IEC 9075-1:2011. (2011). Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework). Available via <http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681> [Access Sept. 2013].NIST. (2010). *Metrics, Standards, and Infrastructure for Sustainable Manufacturing workshop.* Gaithersburg, MD: Available via <http://www.mel.nist.gov/msid/conferences/Agenda_SMW.htm > [access August 2013].

NIST SM. (2012). Sustainable Manufacturing Program. *Available via <http://www.nist.gov/el/msid/lifecycle/sustainable_mfg.cfm> [accessed Sept. 2013]*.

National Research Council. 1999. *Industrial Environmental Performance Metrics: Challenges and Opportunities*. Washington, DC: The National Academies Press.

OECD. (2013). Sustainable Manufacturing Toolkit Prototype. *Available via < http://www.oecd.org/innovation/green/toolkit/48661768.pdf [accessed Sept. 2013]*.

OMG. (2010). Business Process Model and Notation (BPMN). *Available via <http://bpmnhandbook.com/01_specs/BPMN_20_spec.pdf> [accessed Sept. 2013]*.

OMG. (2012). OMG Systems Modeling Language. *Available via <http://www.omgsysml.org/> [accessed Sept. 2013]*.

OPL. (2012). http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r2/index.jsp?topic=%2Filog.odms.ide.help%2FContent%2FOptimization%2FDocumentation%2FOPL_Studio%2F_pubskel%2Fglobals%2Feclipse_and_xplatform%2Fps_opl307.html. *[accessed Sept. 2013]*.

Paju, M., Heilala, J., Hentula, M., Heikkila, A., Johansson, B., Leong, S., and Lyons K. (2010). Framework and Indicators for a Sustainable Manufacturing Mapping Methodology. Proceedings of 2010 Winter Simulation Conference.

Pegden, C, Sadowski, R, Shannon, R. (1995). Introduction to Simulation Using SIMAN. McGraw-Hill, Inc. New York, NY. Pineda-Henson, R., Culaba, A. B. (2002). Developing an Expert System for GP Implementation. *Proceedings of the 2nd World Conference on Green Productivity*. *Available via* http://www.apo-tokyo.org/gp/manila_conf02/resource_papers/narrative/henson_experta4.pdf *[accessed August, 2013]*.

Rockwell Automation. (2010). Taking Energy Management to a Higher Level. *Available via < http://www.managingautomation.com/maonline/reseearch/download/view/Taking_Energy_Management_to_a_Higher_Level_27756351>. [accessed August 2013]*.

Shao, G., Kibira, D., Brodsky, A., & Egge, N. (2011). Decision Support for Sustainable Manufacturing using Decision Guidance Query Language. *The International Journal of Sustainable Engineering,Volume 4, Issue 3* , 251-265.

SMLC. 2011. Implementing 21st Century Smart Manufacturing. https://smart-process-manufacturing.ucla.edu/about/news/Smart%20Manufacturing%206_24_11.pdf. *[accessed August 2013]*.

Tamer G. (2011). "Methodological study on technology integration for sustainable manufacturing in the surface finishing industry." *ETD Collection for Wayne State University*. Paper AAI3469974. http://digitalcommons.wayne.edu/dissertations/AAI3469974

Tanzil, T. and Beloff, B. (2006). “Assessing impacts: Overview on sustainability indicators and metrics,” *Environmental Quality Management*, Volume 15, Issue 4, pp. 41-56.

Waller, A. (2012). Witness Simulation Software. *Proceedings of the 2012 Winter Simulation Conference.*

Appendix: Sustainable Process Analytics Formalism Syntax and Formal Semantics

A. SPAF model syntax

SPAF adopts concepts and ideas from other languages and is based on the OPL data model and the basic OPL syntax of arithmetic and query expressions with minor modifications and extensions. The basic OPL data model, modeling concept, data type, and data structure are listed in (IBM, 2012).

Analytical sequence, *Aseq*, is a sequence (s_1, ..., s_n) of *analytical statements*, $s_{i,,}$

$1 \leq i \leq n$, in one of the forms:

1. $T_i \quad x_i = a_i$
2. $T_i \quad x_i = e_i$
3. $T_i \quad x_i$
4. $T_i \quad x_i = \ldots$
5. C_i
6. *include* M_i *(J)* or *include* M_i *( )*
7. *min* x_i , *max* x_i , or *sat*

where:

- The statements *min* x_i , *max* x_i , or *sat* are only allowed as the last statement s_n
  - T_i is a type
  - x_i is a variable name, which may include a prefix identifier, e.g., *Id.x*.
  - a_i is a constant of type T_i
  - e_i is an expression returning type T_i
  - "…" is a keyword in "$T_i \quad x_i \quad = \ldots$" to indicate that x_i is to be instantiated with a constant before using it later in the sequence
  - C_i is a constraint
  - M_i is a unique name of an analytical model
  - *J* in M_i *(J)* is a string identifier

The first four forms are declaration statements, within which the first two forms are assignment statements. C_i is a constraint statement, *include* M_i is an include statement, and *min* x_i, *max* x_i, or *sat* are optimization statements, i.e., minimization, maximization, and satisfiability. If the last statement s_n of the analytical sequence (s_1, ..., s_n) is *min* x_i, *max* x_i, or *sat*, then the (s_1, ..., s_n) is an *optimization* analytical sequence; otherwise, we say that it is a *non-optimization* analytical sequence.

An *analytical model* is an expression of the form

M(Id) { Aseq } or M( ) { Aseq }

where *M* is a unique name of the model, *Id* is an optional parameter, and *Aseq* is a *non-optimization* analytical sequence.

Let *P* be a set of analytical models. We say that *P* is *closed under reference* (or *closed*) with respect to an analytical sequence *A* (or model *M*) if the following holds: If an *A* has a statement of the form *include M'(J)*, then *P* must contain an analytical model *M'*. We say that *P* is *closed under reference* (or *closed*) if for every model *M* in *P*, *P* is closed with respect to *M*.

An *analytical query* is a pair (*A*, *P*), where *A* is an analytical sequence and *P* is a model package *closed* with respect to *A*.

Let (*A*, *P*) be an analytical query. The *flattened* sequence of *A*, denoted *flat (A, P)*, is an analytical sequence that results from *A* by recursively replacing each *include M( )* with the analytical sequence of

the model *M ( )*, and replacing each *include M(J)* statement with the analytical sequence of the model *M(Id)* in *P*, in which every appearance of *Id* is replaced with *J*.

We say that an analytical query (*A, P*) has a *conflict*, if one of the following holds in

flat (A, P) = (S_1,…, Si ,…, S_j,…, S_n):

- S_j is a declaration statement of the form T_i $x_i = a_i$ or T_i $x_i = e_i$ and S_i is any declaration statement
- S_i and S_j are two declaration statements such that $x_i = x_j$ and $T_i \neq T_j$ (i.e., the same variable is declared twice with conflicting types)

Given a *flat (S_1,…, S_i ,…, S_n)* analytical sequence *A* (i.e., without *include* statement), we say that variable x_i is *data-instantiated* if:

- There is a statement S_i of the form T_i $x_i = a_i$, where a_i is a constant or, recursively
- There is a statement S_i of the form T_i $x_i = e_i$, such that all variables y in e_i are instantiated in the prefix sequence (S_1,…,S_{i-1})

We say that a flat analytical sequence *A* is data instantiated if every variable x in a declaration statement is instantiated.

We say that an analytical query (*A, P*) is *well-formed* if:

- It does not have a *conflict*, and
- For every constraint statement C_i and expression e_i in the declaration statement of the form T_i $x_i = e_i$ or *min* x_i, *max* x_i, or *sat* in *flat (A, P)*, the following holds: it only contains variables that have been declared in a declaration statement earlier in the sequence.
- If *A* is a non-optimization sequence, then, *flat (A, P)* must be data instantiated.
- If *A* is an optimization sequence, then for every statement S_i in *flat (A, P) = (s_1, ..., s_n)* of the form T_i x_i = …, x_i must be instantiated in (S_1,…,S_{i-1}) (i.e., earlier in the sequence).

From now on, only *well-formed* analytical queries are considered.

As discussed earlier, a SPAF model is an analytical model *M (Id)* if it is one of the following forms:

- *Process model*
- *Context model*
- *Flow model*
- *Flow aggregator model*
- *Sustainability metric aggregator model*

A SPAF *process model* with identifier *Id*, *denoted PM (Id)*, is an analytical sequence that contains statements of the following forms:

string *Id.processType = type_flow_string,*

{string} *Id.inputFlow = inputFlowExpr,*

{string} *Id.outputFlow = outputFlowExpr,*

{string} *Id.subProcess = subProcessExpr,*

{string} *Id.flowAggregator = flowAggrExpr,*

and

include M (I), for every *I* in *Id.inputFlow, Id.outputFlow, Id.subProcess,* or *Id.flowAggregator,* where:

- *Id* is used as a prefix for all variables on the left hand side of the declaration statements, except for variables that appear on the left hand side of assignments into variables defined in the included

models, i.e., SPAF models *M (Id')*, where *Id'* is in *Id.inputFlow*, *Id.outputFlow*, *Id.subProcess*, and *Id.flowAggregator* (those are "visible" to the process model)

- *type_process_string* is a string
- *inputFlowExpr*, *outputFlowExpr* are analytical expressions of the type {string} (i.e., return a set of strings)
- *subProcessIdsExpr*, *flowAggrIdsExpr* are analytical expressions of type {string}
- *M (I)* denotes a method that returns a SPAF model with identifier *I*

A SPAF *context model CM ( )*, is an analytical model.

A SPAF *flow model* with identifier *Id*, denoted *FM (Id)*, is an analytical model that contains statements of all of the following forms:

string Id.flowType = type_flow_string,

where:

- *Id* is used as a prefix for all variables on the left hand side of the assignment statements
- *type_flow_string* is a string

A SPAF *flow aggregator model* with identifier *Id*, denoted *FAM (Id)*, is an analytical sequence that contains all of the followings forms:

string *Id.flowType = type_flow_string*,

{string} *Id.flows_to_aggr = inputFlowExpr*,

{string} *Id.flows_from_aggr = outputFlowExpr*,

where:

- *Id* is used as a prefix for all variables on the left hand side of the assignment statements
- *type_flow_string* is a string
- *inputFlowExpr*, *outputFlowExpr* are analytical expressions of the type *{string}* (i.e., return a set of strings)

An SPAF process package is a *model package P*. We say that it is *well-formed* if:

- *P* is closed under references
- *P* satisfies the following *scoping* rules:
  - Process model *M (Id)* can use variables prefixed with identifiers form *Id.inputFlow*, *Id.outputFlow*, *Id.flowAggregator*, or itself, i.e., *Id.*
  - A model *M (Id)* in *P* can use variables from the context model in *P*.
  - Flow Aggregator Model *M (Id)* can use variables that are prefixed with identifiers of flow models that are referenced in it, or itself, i.e., *Id*.
- For every process model *M* in *P*, *A(M)* is a *well-formed* analytical sequence

Note that a *well-formed* SPAF process package *P* provides a modular description of a (*flat*) and *well-formed* analytical sequence. Thus, it is naturally extendable and its components are reusable.

B. SPAF formal semantics

We say that an analytical sequence *A* is *explicit* if all of its analytical statements are of the form

$$T_i \quad x_i = a_i$$

where a_i is a constant, i.e., it is an assignment of a constant to a variable. Intuitively, the symbolic expression of an explicit analytical sequence represents the corresponding data. Note that an explicit analytical sequence is flat. Formally, the semantics of an explicit analytical sequence $(s_1, ..., s_n)$, denoted *Sem* $((s_1, ..., s_n))$, is itself, i.e., its symbolic expression.

We say that an analytical sequence A is *implicit* if all of its analytical statements are of the form

$$T_i \quad x_i = e_i$$

Note that this includes the case when the expression e_i is a constant a_i. Formally, the semantics of a well-formed implicit analytical sequence

$$(T_1 \quad x_1 = e_1, \ldots, \ T_n \quad x_n = e_n)$$

is the explicit analytical sequence

$$(T_1 \quad x_1 = a_1, \ldots, \ T_n \quad x_n = a_n)$$

in which each a_i, $1 \le i \le n$, is a constant of type T_i that is computed by expression e_i, when each variable x_j, $1 \le j \le i-1$, is replaced by the constant a_j.

The semantics of $(s_1, ..., s_n)$ is denoted *Sem* $((s_1, ..., s_n))$. Obviously, an *explicit* analytical sequence is a particular case of *implicit*, in which case, *explicit* and *implicit* semantics coincide.

We say that an analytical sequence $A = (s_1, ..., s_n)$ is a *constraint* analytical sequence if all of its statements are of the form

$$(T_i \quad x_i = e_i) \qquad \text{or} \qquad C_i$$

where e_i is an expression of type T_i and C_i is a constraint. Formally, the semantics of a *well-formed* constraint analytical sequence $(s_1, ..., s_n)$, denoted *Sem* $((s_1, ..., s_n))$, is defined as follows:

- Consider an implicit analytical sequence $(s_{i_1}, ..., s_{i_k})$, which is a sub-sequence of $(s_1, ..., s_n)$ that contains all statements s_i' of the form $T_i \ x_i = e_i$, and its semantics $(T_{i_1} \ x_{i_1} = a_{i_1}, ..., T_{i_k} \ x_{i_k} = a_{i_k})$ (which is an *explicit* analytical sequence), and
- Consider a sequence $(C_{j_1}, ..., C_{j_m})$, which is a sub-sequence of $(s_1, ..., s_n)$ that contains all the constraint statements
  - If there exists $1 \le i \le m$, such that C_{j_i} evaluates to *FALSE* after every variable x_i in it is replaced with the constant a_i, then *Sem* $((s_1, ..., s_n))$ is defined as *INVALID*. Otherwise, *Sem* $((s_1, ..., s_n))$ is defined as the explicit analytical sequence $(T_{i_1} \ x_{i_1} = a_{i_1}, ..., T_{i_k} \ x_{i_k} = a_{i_k})$.

We say that an analytical sequence $A = (s_1, ..., s_n)$ is an *alternative* analytical sequence, if each s_i, $1 \le i \le n$, is of the form

$$(Ti \quad x_i), \quad (T_i \quad x_i = a_i), \quad (T_i \quad x_i = e_i), \qquad or \quad C_i$$

where a_i is a constant of type Ti, and e_i is an expression of type Ti, and C_i is a constraint. Note that an alternative analytical sequence may have repetition of declaration statements for the same variable x. Consider the analytical sequence $(s_1, ..., s_n)$ resulting from A by removing, for every variable x, all declarations except for its first appearance in A. Formally, the semantics of a well-formed *alternatives* analytical sequence $(s_1, ..., s_n)$, denoted *Sem* $((s_1, ..., s_n))$, is defined as follows:

Consider all non-instantiated variables $x_{i_1}, ..., x_{i_k}$ in $(s_1, ..., s_n)$. *Sem* $((s_1, ..., s_n))$ is the set

$$\{ E(a_{i_1}, ..., a_{i_k}) \mid a_{i_1} \text{ in } D(T_{i_1}), ..., a_{i_k} \text{ in } D(T_{i_k}) \wedge E(a_{i_1}, ..., a_{i_k}) \neq INVALID \}$$

where:

- $D(T_{i_1}), ..., D(T_{i_k})$ are the domains of types $T_{i_1}, ..., T_{i_k}$, respectively, and

- *E(* a_{i_1} *, ...,* a_{i_k} *)* denotes *Sem ((s₁,..., sₙ) [* x_{i_1}/a_{i_1} *,...,* x_{i_k}/a_{i_k} *])*, where *(s₁, ..., sₙ) [* x_{i_1}/a_{i_1} *,...,* x_{i_k}/a_{i_k} *]* denotes the constraint analytical sequence *(s₁', ..., sₙ')* that results from *(s₁, ..., sₙ)* by replacing each statement of the form *(*T_{i_j} x_{i_j}*), 1 ≤ j ≤ k*, with the statement T_{i_j} x_{i_j} = a_{i_j}.

We say that an analytical sequence *A= (s₁,..., sₙ, sₙ₊₁)* is a flat optimization sequence if *(s₁,..., sₙ)* is an alternative sequence, and $s_{(n+1)}$ is of the form:

min x_i, *max* x_i, *or sat.*

where x_i, $1 \le i \le n$, is one of the variables in the left hand sides of assignments in

(s₁, ..., sₙ). Assuming without loss generality that, for every variable *x* in *A*, there is a single declaration of *x* (if this is not the case, all declarations of *x* except for its first appearance are removed.) Formally, the semantics of an optimization analytical sequence *(s₁,..., sₙ, s(n+1))*, denoted *Sem ((s₁,..., sₙ, s(n+1)))*, is defined as follows:

If *Sem (s₁, …, sₙ)* = ∅ then we say that *Sem ((s₁,..., sₙ, s(n+1)))* is *INFEASIBLE*. Otherwise, consider an explicit analytical sequence *E* in *Sem ((s₁,..., sₙ))* such that:

- If $s_{(n+1)}$ is *min* x_i, then for all *E'* in *Sem ((s₁, …, sₙ)*, $a_i \le a_i'$, where a_i and a_i', are the analytical model constants in the assignments T_i $x_i = a_i$ of *E*, and T_i $x_i = a_i'$ of *E'*.
- If $s_{(n+1)}$ is *max* x_i, then for all *E'* in *Sem ((s₁, ..., sₙ)*, $a_i \ge a_i'$, where a_i and a_i', are the analytical model constants in the assignments T_i $x_i = a_i$ of *E*, and T_i $x_i = a_i'$ of *E'*.

If *E* does not exist, we say that *Sem ((s₁,..., sₙ, s(n+1)))* is *UNBOUNDED.* Otherwise,

Sem ((s₁,..., sₙ, s(n+1))) is *E*.

Note that if $s_{(n+1)}$ is *sat*, the semantics is just an explicit analytical sequence *E* in *Sem ((s₁, ..., sₙ))*. Also note that the optimization semantics (whether it is minimization, maximization, or satisfiability) are non-deterministic, i.e., there may be more than one explicit model that satisfies the condition in the definition of semantics.

Consider the five layers (types) of analytical sequences (1) explicit, (2) implicit, (3) constraints, (4) alternatives, and (5) optimization. Let *L (1), L (2), L (3), L (4),* and *L (5)* denote sets of analytical sequences that can be expressed by each layer, respectively. We claim that

$$L(1) \subset L(2) \subset L(3) \subset \{L(4)\} \subset L(5)$$

and that the semantics of each layer are consistent with all lower layers. That is, for any two layers *i, j,* $1 \le i < j \le 5$, $i \ne 4$, $j \ne 4$', if an AM *A* is in *L (i)*, then *Sem (i)* of *A* is also *Sem (j)* of *A*.

Semantics of a query *(A, P)* is a pair *(A', P')* constructed as follows:

- For every sequence *S*, either *A* or a sequence *B* in a model *M (Id) {B}* in *P*, *S* is replaced by *S'* as follows.
  - Consider all variables $x_{1, \ldots,} x_n$, declared in their order in *S*, then *S'* is the sequence
    $$(T_1 \quad x_1 = a_1 \ , \ldots, \ T_n \quad x_n = a_n)$$

where T_1 ,…, T_n are the corresponding types of $x_{1, \ldots,} x_n$ respectively, and a_i is the constant instantiated with x_i in the semantics *E* of *flat (A,P)*.

C. SPAF Query Computation

In this section, algorithms (reduction procedures) to perform SPAF analytical query computation are introduced. Figure C.1 shows a commutative diagram for analytical query computation, in which the upper left box indicates the query sequence *A* in model package *P*. The query sequence may have *include* statements . The semantics of *A* is sequence *A'* in package *P'* as shown in the upper right box in Figure

24. Two algorithms are included in the computation – analytical query algorithm and flat optimization sequence algorithm. Through the analytical query algorithm (refer to step (1), (6), and (5)), *(A, P)* can be translated to a flat analytical sequence (middle left box). If the flat analytical sequence can be instantiated, it is an implicit analytical sequence, otherwise, it is an optimization analytical sequence whose semantics is a flat explicit analytical sequence (middle right box). This algorithm calls the flat optimization sequence algorithm (refer to step (2), (3), and (4)) to translate the flat optimization sequence to a standard optimization model such as OPL or AMPL (lower left box). By using an optimization solver, the optimization solution (lower right box) can be derived. All variables can then be instantiated, the sequence becomes a flat explicit analytical sequence (middle right box), which can be translate back to *(A', P')*.

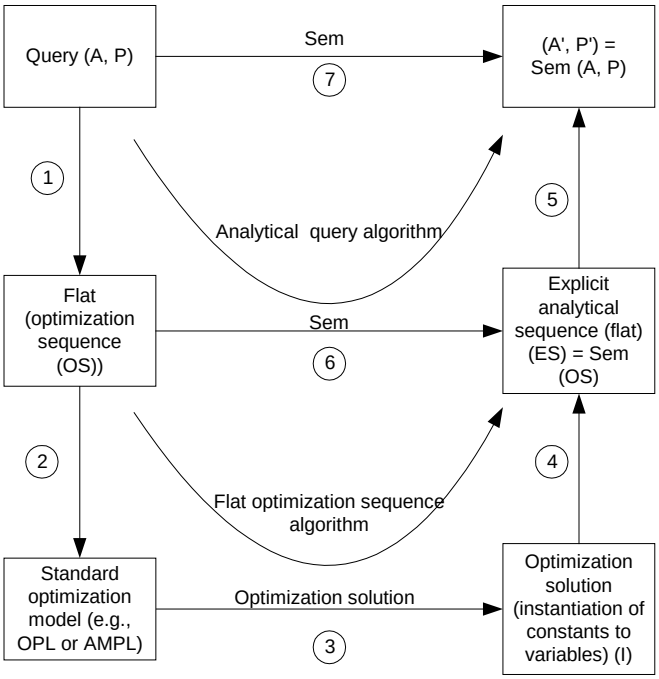


Figure C.1 A commutative diagram for analytical query computation

Figure C.2 presents the algorithm of SPAF Query Computation. The input is an analytical query sequence *A* and a model package *P* that is closed with respect to *A*. The output is *(A', P')* that is the semantics of *(A, P)*. The procedures of the algorithm include:

1. Construct a flat sequence *S1* by replacing all the *include* statements in *A* with corresponding analytical sequences.
2. Construct a new sequence *S2* by removing all the duplicated declarations of *x* except for the first declaration in *S1* for every variable *x* declared in *S1*.
3. If *S2* is instantiated, it must be an implicit analytical sequence. So a new explicit analytical sequence *S3* can be constructed by replacing each variable with a constant that derived from an expression.
4. If *S2* is not instantiated, it must be a flat optimization query. By calling the *OptSeqAlg (S2)* algorithm, it will return the semantics of *S2*.

Input: (A, P) is a well-formed analytical query and P is a closed form model package closed with respect to A.

Output: (A', P') is the semantics of (A, P).

1. *Construct S1 = flat (A, P).*
2. *Construct sequence S2 from S1 as follows:*

   For every variable x declared in S1, remove all declarations of x except for the first declaration in S1.
3. *Check if S2 is instantiated.*
4. *If S2 is instantiated, it must be an implicit analytical sequence of the form (T_1 $x_1 = e_1$, …,T_n $x_n = e_n$). In this case, construct S3 as the explicit analytical sequence (T_1 $x_1 = a_1$,…,T_n $x_n = a_n$),in which each a_i, $1 \le i \le n$, is a constant of type T_i that is computed by expression e_i, where each variable x_j, $1 \le j \le i-1$, is replaced by a_j.*
5. *Otherwise, if S2 is not instantiated, it must be a flat optimization query. Construct S3 by calling the method OptSeqAlg (S2), which returns the semantics of S2.*
6. *Construct the pair (A', P') as follows:*
   For every sequence S, which is either A or a sequence B in a model(Id){B} in P, S is replaced by S' as follows.
   Consider all variables $x_1, \ldots, x_n$, declared in their order in S, then S' is the sequence (T_1 $x_1 = a_1$,…,T_n $x_n = a_n$) where T_1 ,… ,T_n are the corresponding types of $x_1, \ldots, x_n$ respectively, and a_i is the constant to instantiate x_i in S3.

Figure C.2 Algorithm 1: SPAF query computation

Figure C.3 presents the algorithm of Optimization Sequence Algorithm (*OptSeqAlg*). The input is a flat optimization query *S2* generated by the SPAF Query Computation algorithm. The output is the semantics of *S2*.

1. For all variables that are instantiated in every statement, replace the expression with the computed constant.
2. Construct decision variables that are not being instantiated.
3. Construct a set of constraints by replacing decision variables in every statement with its constant. For any variable that is non-instantiated, a constraint is added.
4. Construct the optimization problem with objectives and constraints.

5. Solve the optimization problem using an optimization solver
6. Construct the answer sequence by removing all constraint statements and replacing all the variables using constants computed or the optimization solutions.

Input: Flat optimization query (i.e., $(s_1,\ldots,s_n,s_{n+1})$ where s_{n+1} is of the form min x_i , max x_i $(1 \le i \le n)$ or sat where x_i is not instantiated on $(s_1, \ldots, s_n)$).

Output: Semantics of $(s_1,\ldots,s_n,s_{n+1})$.

1. *Consider all variables $x_{i_1}, \ldots, x_{i_m}$ in $(s_1, \ldots, s_n)$ that are instantiated. For every statement S_{i_j}, $1 \le j \le k$, of the form $T_{i_j}\ x_{i_j} = e_{i_j}$, compute e_{i_j}, and replace e_{i_j} with the computed constant a_{i_j}, i.e., resulting in $T_{i_j}\ x_{i_j} = a_{i_j}$*
2. *Construct the set of decision variable V to be the set of all non-instantiated variables $x_{l_1}, \ldots, x_{l_m}$ in $(s_1, \ldots, s_n)$ ranging over the domains corresponding to types $T_{l_1},\ldots, T_{l_m}$ respectively.*
3. *Construct the set of constraints C as follows:*
   - *3.1 Initially, $C = \emptyset$.*
   - *3.2 For every statement s_i, $1 \le i \le n$ of the form C_i, add to C the constraint resulting from C_i by replacing every instantiated variable x_{i_j} with its constant a_{i_j} from Step 1.*
   - *3.3 For every statement S_i of the form $T_i\ x_i = e_i$, where x_i is non-instantiated, add the constraint $x_i == e_i'$, where e_i' result from e_i by replacing each decision variable x_{i_j} in e_i with its constant a_{i_j} from Step 1.*
4. *Construct the optimization problem O;*

   $\min_V x_n$ *subject to C,* $\max_V x_n$ *subject to C, or* $\underset{V}{sat}\ C$ *according to* s_{n+1}.
5. *Solve the optimization problem O.*
6. *If O is infeasible, return "INFEASIBLE", else if O is unbounded, return "UNBOUNDED."*

7. Otherwise, construct the answer sequence from (s_1, ..., s_n) as follows:

7.1 All non-declaration statements (i.e., constraints) are removed.

7.2 Every declaration statement with type T_i and variable x_i (i.e., of the form T_i $x_i = a_i$ or T_i $x_i = e_i$) be replaced as follows:

7.3 if x_i is instantiated, it is replaced with T_i $x_i = a_i$, where a_i is a constant computed in Step 1.

7.4 if x_i is non-instantiated, the statement is replaced with T_i $x_i = a_i$, where a_i is a constant instantiated into decision variable x_i from the solution of the optimization problem O.

Figure C.3 Algorithm 2: optimization sequence algorithm (OptSeqAlg)

Algorithm correctness: We denote by All-Sem (A, P) the set of all explicit analytical sequences' E that are Sem (A, P).

We denote by All-Ans (A, P) the set of all explicit analytical sequences' E that are possible answers produced by Algorithm: SPAF query computation.

Claim: Algorithm SPAF query computation is CORRECT, i.e., it is:

1. Sound, i.e., for every well-formed analytical query (A, P),
   All_Ans (A, P) ⊆ All_Sem (A, P)
2. Complete, i.e., for every well-formed analytical query (A, P),
   All_Sem (A, P) ⊆ All_Ans (A, P)