Published in final edited form as:

J Intell Manuf. 2018 August ; 29(6): 1287–1301. doi:10.1007/s10845-015-1178-6.

Modeling and Optimization of Manufacturing Process Performance using Modelica Graphical Representation and Process Analytics Formalism

Guodong Shao,

System Integration Division, Engineering Laboratory, National Institute of Standards and Technology, 100 Bureau Drive, MS 8260, Gaithersburg, MD 20899-8260

Alexander Brodsky, and

Department of Computer Science, George Mason University, 4400 University Drive, MS 4A5, Fairfax, Virginia 22030-4444

Ryan Miller

Department of Computer Science, University of Texas at Dallas, 800 W Campbell Rd, Richardson, TX 75080

Abstract

This paper concerns the development of a design methodology and its demonstration through a prototype system for performance modeling and optimization of manufacturing processes. The design methodology uses a Modelica simulation tool serving as the graphical user interface for manufacturing domain users such as process engineers to formulate their problems. The Process Analytics Formalism, developed at the National Institute of Standards and Technology, serves as a bridge between the Modelica classes and a commercial optimization solver. The prototype system includes (1) manufacturing model components' libraries created by using Modelica and the Process Analytics Formalism, and (2) a translator of the Modelica classes to Process Analytics Formalism, which are then compiled to mathematical programming models and solved using an optimization solver. This paper provides an experiment toward the goal of enabling manufacturing users to intuitively formulate process performance models, solve problems using optimization-based methods, and automatically get actionable recommendations.

Keywords

Optimization; Manufacturing Process; Process Analytics; Graphical User Interface

Correspondence to: Guodong Shao.

Introduction

Smart Manufacturing (SM) requires the collaboration of advanced manufacturing capabilities and digital technologies to create highly customizable products faster, cheaper, and greener. According to NIST (2014), Smart Manufacturing Systems (SMS) should enable manufacturers to easily and rapidly (1) respond changing demands and conditions, (2) reconfigure factory production to optimize system performance, (3) deal with uncertainty and learn from past experience for continuous improvement, (4) maintain seamless interoperability between factory processes and supply networks, and (5) allow communication of end-user needs and production instructions (SMLC 2011). Industry 4.0, originated in Germany, promotes intelligent factory (Smart Factory) that is characterized by adaptability, resource efficiency, and ergonomics, as well as the integration of customers and business partners in business and value processes. Its technological foundation is comprised of cyber-physical systems and the Internet of Things (Jacinto 2015).

The concept of SM is very similar to the concept of Industry 4.0. Both concepts are built up based on the intelligent manufacturing systems paradigms, such as flexible manufacturing, reconfigurable manufacturing.

To cope with today's rapid changes and help efficiently achieve the SM goals, manufacturers need more efficient and effective tools and systems for decision support regarding their products, processes, and system design and operations. There is a need for solutions that allow users in manufacturing facilities to (1) easily and intuitively model performance of manufacturing processes, (2) optimize key performance indicators (KPI's), and (3) provide actionable recommendations on how to configure their manufacturing processes for achieving specific SM goals. These solutions should not require end users to have complex programming skills or extensive software tool experience, but enable users to model their problems using the model components they are familiar with, use the data they collected, and ask questions using their own terminology.

How does the work described in this paper contribute to the SM goals? The tasks of developing analytical models and translating analytical results into recommendations are complex, costly, and error prone. One of the fundamental challenges is that modeling and optimization require mathematical knowledge and programming training, which manufacturing engineers or production operators normally do not have. There are Graphical User Interface (GUI)-based tools available for manufacturers to model and analyze their systems; however, these tools are typically customized for special purposes and are difficult to extend (e.g., Kumar et al. 2012). Furthermore, the customized tools do not typically support sharing and reuse of the modeling knowledge and the created models with other systems and analysis tasks.

The two core existing general technologies for modeling and analysis are simulation and optimization. Simulation is highly useful in cases where manufacturing problems are too complex to be expressed in a closed mathematical form. Physics-based simulation modeling can be performed using simulation languages/tools such as Modelica and Simulink (Modelica, 2014; MathWorks, 2014), while Discrete Event Simulation (DES) and agent-

based modelling can be performed using tools such as ARENA and/or AnyLogic (AnyLogic, 2014; Rockwell Automation, 2014). Most of these simulation tools take an object-oriented approach, allowing problems to be described in a modular, extendible, and reusable manner.

However, in cases where the problem can be expressed in a closed analytical form, simulation-based optimization is not competitive, in terms of optimality of results and computational complexity, compared with solvers based on Mathematical Programming (MP) or Constraint Programming (CP). Typically, the convergence is slow, the results are not optimal and, furthermore, there are no guaranteed bounds on how far from optimality the results are (Klemmt et al. 2009). Whereas, in MP and CP, there are broad classes of problems with guaranteed optimality or, at least, an upper bound on how far from optimality the results are. Therefore, in situations where the quality of optimization is critical, and if the problem can be expressed in a closed mathematical form, MP and CP is the technology of choice. However, MP/CP solver-based decision optimization requires significant expertise and special training in mathematics and Operation Research (OR) to formulate models. Furthermore, the models typically lack the modularity of modern object-oriented languages, so they tend to be difficult to modify or extend for reuse. Some examples of existing MP/CP tools include CPLEX, Gurobi, Gecode, JaCoP (CPLEX 2014, Gurobi 2015, Gecode 2015, Jacop 2015).

To bridge the gap between the flexibility of simulation modeling and efficiency of MP/CP solver-based optimization, researchers at the National Institute of Standards and Technology (NIST) have proposed and developed the Sustainable Process Analytics Formalism (SPAF), a framework that incorporates simulation-like features such as modular, extendible, and reusable; and use MP/CP solvers to find optimized solutions for manufacturing process performance metrics (Brodsky, Shao, & Riddick, 2014). However, SPAF still requires the modeler to develop mathematical models and programs; there is no GUI designed specifically for the use of SPAF. To address this issue, a GUI needs to be developed to work with SPAF to enable users to intuitively model the problem, map to the SPAF format, and solve the problem using a commercial optimization solver.

A powerful GUI functionality is provided by a modeling and simulation tool based on Modelica modeling language (Modelica Association, 2012). Modelica is a non-proprietary, object-oriented, declarative, equation based, and multi-domain modeling language developed by the Modelica Association.

To bridge the gap between the capabilities of SPAF and Modelica based GUI, the focus of this paper is a novel design methodology and its demonstration through a prototype system for manufacturing process performance modeling and analysis. The system utilizes the graphical modeling of Modelica and the capability of SPAF to overcome the complexity of formulating solver-based optimization models. More specifically, the contributions reported in this paper are as follows. First, a design methodology of a modeling and analysis and its demonstration via a prototype system for what-if analysis and optimization of operational performance of SMS is proposed. The proposed design enables automatic construction of optimization models based on mathematical programming from the graphical description,

which eliminates the need of dealing with the complexity of formulating these models manually. This is done by leveraging the Modelica language and environment, the SPAF compiler, and a Mixed-integer linear programming (MILP) solver (i.e., IBM CPLEX). The proposed design is based on (1) manufacturing model components' libraries, including processes, process connectors, and metrics using Modelica and SPAF, and (2) a translator that translates Modelica classes into SPAF classes, which are then compiled to mathematical programming models and solved using the MILP solver.

Second, the design is demonstrated by implementing a prototype system that enables users to intuitively model manufacturing processes, and perform user-driven what-if analysis and optimization, leading to actionable recommendations through a GUI. In the prototype system, the Modelica drag-and-drop environment and the manufacturing domain model libraries (for both Modelica and SPAF) enable the easy modeling of manufacturing problems. The use of the SPAF compiler and a MILP optimization solver enable users to effectively perform what-if analysis and optimization. Domain users can formulate the problems and receive actionable recommendations through the Modelica GUI.

The next section briefly reviews related work including Modelica and SPAF. Followed by the introduction of the concept of the modeling and analysis system through an example. Then the system architecture and implementation are discussed in details. Finally, a conclusion and discussion of the future work are provided.

More on related work

There exist many technologies and tools for different analysis purposes, e.g., simulation and optimization. However, to properly identify and use them is not easy, often time, the learning curves for using these tools are steep, and users need to be trained and are required to have specific knowledge and skills for using these tools to model problems. Manufacturing end users lack a tool that enables them to use effective problem-solving techniques, yet allows them to model their problems in designs and operations in an intuitive manner at the same time. This section overviews some related technologies, languages, tools, and environments that are the important pieces of the proposed system.

Modeling, simulation, and optimization: decision optimization has been used to find the best solution by exploring all the trade-offs for various usage scenarios. An OR optimization model typically has (1) decision variables, (2) constraints that have to be satisfied, and (3) an objective function to be optimized. A feasible solution is an instantiation of values from corresponding domains of decision variables that satisfy all the constraints. Among all feasible solutions, an optimal solution is one that makes the objective minimal or maximal, as required. Using decision optimization to analyze the performance of a manufacturing process is challenging because the model abstractions only have indirect connections to the elements of a manufacturing process. For example, within an equation, multiple parameters are not a one-to-one mapping with elements in a manufacturing process. Brodsky and Nash (2005) and Brodsky, Egge and Wang (2012) indicate "the notions of order and timing of events are usually not explicit in OR models and the execution of such a model is typically a black box to the user. For example, the state of machines and buffers from one time interval

to the next would be modeled in an OR model as a set of generic equations and inequality constraints universally quantified over finite sets. This makes debugging of an optimization model a challenging task."

On the other hand, simulation modeling is often used by users to quantify and observe behaviors of processes by analyzing and comparing alternatives. Brodsky and Nash (2005) state: "the elements of a simulation are state variables and state transitions, which could have a clear one-to-one correspondence with elements of a process. Real-world time and sequence of events correspond to time and sequence in the simulation runs, which makes it easier to understand and debug the model. Also, object-oriented software allows modular development of the simulation." DES has been applied to manufacturing applications for what-if analyses of various scenarios to aid decision-making (McLean & Shao, 2001). Researchers have already built intelligence into DES by integrating Expert Systems, Complex Adaptive Systems (CAS), and System Dynamics. For example, Pathak and Dilts (2002) apply the concepts of CAS to their supply chain models and simulate the adaptive and evolving behavior of the models in different environments. AnyLogic simulation modeling software supports all existing modeling methods, i.e., system dynamics, discrete event, and agent based modeling within one modeling language and model development environment (AnyLogic, 2014).

There are varieties of simulation optimization tools that work with DES packages. For example, OptQuest for ARENA, AnyLogic, and SIMUL8 (OptTek 2015, Rockwell Automation, 2014, SIMUL8 2015); SimRunner for ProModel (SimRunner 2015, ProModel 2015); and Optimizer for WITNESS (Optimizer 2015, Witness 2015). These optimization add-ons apply search strategies such as genetic algorithms, neural networks, scatter search, tabu search, and simulated annealing. However, for optimization problems that can be expressed in closed analytical form, simulation-based optimization is not as efficient as MP/CP solver-based optimization in terms of optimality of results and time complexity (Klemmt et al. 2009).

Sustainable Process Analytics Formalism (SPAF)

To take advantage of both simulation and optimization modeling techniques, SPAF is developed to facilitate decision optimization, problem formulation, and execution. It combines concepts from simulation and solver-based optimization, i.e., SPAF not only enables modular, extensible, and reusable modeming, but also enables MP/CP optimization solver to efficiently find solutions. SPAF allows users to represent manufacturing knowledge and formally describe (1) process structure, (2) process data, (3) control variables, and (4) analytical models of metrics and constraints. The process structure includes the hierarchical composition of processes, sub-processes, and resource flows. Process data include production- and sustainability- related information, attributes, and parameters. Control variables can be instantiated by both users and the systems that implement the formalism. The formalism includes mathematical specification for metrics, constraints, and objectives. The SPAF includes three major parts.

1. Generic analytics language: provides unified syntax and semantics for process modeling and support of data querying, what-if analysis, and decision

optimization. For example, given a SPAF model of a manufacturing process involving multiple machines, users can pose (a) data query: retrieve the specification parameters of all turning machines, (b) what-if analysis query: given a specific control setting for each machine, find the total energy consumption, (c) decision optimization query, given the demand of the product to be produced, and find the settings of all machines to minimize the energy consumption while satisfying the demand.

2. Process description and sustainability metrics: (a) enable formal representation of process structure (sub-processes, what they consume and produce), resource flow (e.g., materials/parts/products from one process to another), data, control variables, objectives (e.g., minimize cost), and constraints (e.g., machine capability and demand); (b) support sustainable metrics models (e.g., computations of energy, cost, emissions); and (c) are easy to use by manufacturing and business users on a factory floor.

3. Reduction procedure: enables the translation of a SPAF query against a model into specialized models, e.g., an MILP model, which can be solved by IBM CPLEX optimization solver (CPLEX 2014).

Detailed SPAF syntax, semantics, and examples have been provided in (Brodsky, Shao, and Riddick 2014). The modular design of SPAF enables the built-in process modeling and the creation of a SPAF model library that supports manufacturing metrics definition and reuse. Reuse makes it easier for manufacturing and business users to model their manufacturing processes. However, users still need to be trained to develop SPAF models with basic mathematical programming skill. A domain-specific modeling GUI is necessary to allow manufacturing users to model their problems in a more intuitive manner. In the design proposed in this paper, a Modelica tool is adopted for this purpose and integrated with SPAF.

Modelica

Modelica is an open standard for describing physical models and components. It allows users to model different complex processes. There is a Modelica Standard Library that contains generic model components and functions in various domains (Modelica Association, 2012). Modelica has no compiler or linked software package, rather it is implemented as a standard in various commercial and open-source software tools, such as Open Modelica's OMEdit software or Wolfram with its SystemModeler software (OpenModelica, 2014; WolFram, 2014). Other similar Modelica simulation environments include CATIA Systems, CyModelica, Dymola, LMS AMESim, JModelica.org, MapleSim, SCICOS, and SimulationX, Vertex. Many manufacturing companies are using Modelica for modeling their operations. For example, automotive companies, such as Audi, BMW, Daimler, Ford, Toyota, and VW, use Modelica to design energy efficient vehicles and/or improve air conditioning systems. Other companies that use Modelica for their modeling purpose include ABB, EDF, and Siemens (Modelica, 2014). However, the existing Modelica standard model library is not designed for conveniently modeling the manufacturing process performance in the way amenable to MP/CP-based optimization. Modelica environment is used as a GUI and develop and import an initial prototype of a reusable and extensible manufacturing process model component library into the Modelica environment. Modelica

provides simulation capability for optimization; some researchers have reported the application of Optimica, a JModelica effort for dynamic optimization (Dietl, et al., 2014). Optimica extends Modelica with language constructs that enable formulation of dynamic optimization problems based on Modelica models. Dynamic optimization mainly addresses design problems such as minimum-time problems, parameter estimation problems, and on-line optimization control strategy problems (Åkesson 2008). However, it is complex and requires significant programming and modeling efforts.

System functionality and graphical user interface

Before discussing the design and implementation of the proposed system (i.e., *How* the system works), now the system functionality (i.e., *What* the system does) is provided. Key user roles, system functionality, its semantics, as well as a typical case scenario for each role using the system GUI are discussed. Key system functionality is to enable analytical tasks such as what-if analysis and optimization of performance of manufacturing processes. The idea is that these required analytical tasks will not be implemented from scratch, but will use reusable components of machine and process performance models from a prebuilt model library, these components are used to construct composite process models, against which the predictive (what-if) and prescriptive (optimization) queries can be performed by users with different roles. As depicted in Figure 1, potential user roles of the modeling and analysis system include (1) process modelers who must understand the process plan, machines, and flow of materials and parts and (2) manufacturing end users, who ask declarative analysis queries against previously constructed process models. Of course, multiple manufacturing users can fill the two roles. For example manufacturing process engineers can play both roles, whereas manufacturing operators and business managers can play the role of manufacturing end users. Modelers have at their disposal a pre-built library of model components (representing performance models of machines/processes), as well as higher-level process models that have been previously constructed by the modelers of the manufacturing enterprise. Manufacturing end users can perform various analyses of the manufacturing processes that have been defined by the modelers.

To describe the functionality of the system in more detail, consider a GUI screen capture of the system depicted in Figure 2 using the Wolfram System Modeler, which supports Modelica modeling. The screen is split into three parts. The left part is a window for a library of manufacturing models, organized into folders by modelers and end users. The upper right part is a workspace window used for constructing composite processes and performing what-if analysis and optimization. The lower right part is a window used to display status information. Each of these parts and the corresponding functionality is given below.

Library of Manufacturing Performance Models (left window in Figure 2)

The library includes models of machines available at the manufacturing facility, models of composite manufacturing processes, and commonly used components constructed by manufacturing modelers. Note that manufacturing process models may be complex and involve an arbitrary hierarchy of sub-processes, where each atomic process (an atomic

process is an end process in which there is no sub-process) corresponds to a machine on the manufacturing floor.

In the example in Figure 2 (left window), the library folders include (1) Examples; (2) Built-in library for specific manufacturing machine models; (3) User classes for composite process models constructed and calibrated for machines available at the manufacturing facility, and fully instantiated process models that would typically be from a result of previously run computation or optimizations; and (4) Manufacturing template models' folder, which contains a folder for template machines models. These templates can be used to create specific machine models by instantiating the parameters. In the example folder, there are template models for DieCasting machines, FillingMachines, Lathes, and Turning machines.

While implementation of performance models of machines and processes may be complex, their meaning for manufacturing modelers and end-users is simple. Following the SPAF formalism (Brodsky, Shao, and Riddick 2014), here is what manufacturing users need to understand about the performance models of both atomic and composite process models:

1. *Correspondence between a model and a specific physical machine or process plan* used in the manufacturing facility. For example, the user must know that a particular Milling machine's performance model corresponds to a specific physical milling machining center on the manufacturing floor, which has model A55, by vendor MAKINO, and facility catalog number 015.
2. *Input and output flows of processes.* Output flows correspond to things produced (products, parts, etc.) by the processes and input flows correspond to things consumed by the processes. For example, the input to the Lathe machine is a piece of a metal block and the output of the process is a machined part. A process may have one, multiple or no input and output flows. Each flow has a description (type) of an item that it represents and is associated with a quantity, which can be a real or an integer number. A quantity can either be a variable (to be instantiated later, e.g., by users or through optimization), or a constant.
3. *Process parameters and controls.* Each atomic process model (i.e., a machine) may have its specific parameters that describe the machine or provide coefficients as well its control variables that represent the "knobs" that end users have control over. For example, a milling machine may have control parameters (variables) of feed rate, spindle speed, and depth of cut that can be set to different values. Note that these control variables may not be instantiated at the beginning; they may be instantiated later by users or through optimization. Each composite process, as discussed in more detail below, has indirect control of its sub-processes and the load distribution among them.
4. *Process dependent variables as a function of parameters and controls.* Each process may have dependent variables including metrics and KPIs such as energy consumption, carbon emissions, and cost. The users need to understand what the metrics mean and trust that the system knows how to compute them correctly from the process parameters and controls, but they are not required to understand the mathematics behind it.

5. *Process feasibility constraints.* Each process may have feasibility constraints that capture, for any combination of process controls, whether this combination is within a feasible operation range of the process (e.g., capacity and safety limitations). While the process feasibility constraints in the system are only an approximation of the real-world process/equipment feasibility range, the users need to trust that the system feasibility constraints are "conservative," i.e., that the controls are indeed feasible in the real-world process/equipment if the system indicates so.

Modeling of Composite Processes by Manufacturing Modelers

To explain the modeling process by modelers, an example from Shao et al (2014) is adopt and modified. In the example, parts produced by an injection molding process, arc welding process, and one of the three machine tools for turning process (after a die casting process) are manually fastened together at a threaded fastening station. An analysis of this process (by end-users) is exemplified to determine the allocation of the number of parts to different turning machines that minimizes energy consumption.

Using the manufacturing model library, users can start the model formulation process according to the modeling requirements. The hierarchical model folders of the model library include built-in model template, user-defined machines and composite processes, and user instances. For example, under folder DieCasting, there is a model template "DiecastingTemplate," which corresponds to a generic Die Casting machine in a parametric form; two instance models "Ecoline53" and "Evolution 53" that represent specific Die-Casting machine models by different vendors with all of the parameters instantiated. Below is a typical case scenario to define a new composite process by a modeler (see the left window of Figure 2):

1. The modeler identifies the inputs (i.e., materials, parts, energy, coolant) used by the process and output from the process (i.e., products, parts) produced by the process. The inputs and output are indicated by the yellow (on the left) and white (on the right) triangular icons, which the user can drag and drop from the component library. In the example, all processes use energy. Here are the material inputs and outputs for each process:
    a. Welding process: The inputs include two metal parts and the output is one joined part.
    b. Injection molding process: The inputs are plastic grains and the output is a plastic part.
    c. Die casting process: The input is metal powers and the output is a cast part.
    d. Turning process: The input is a cast part from the die casting process and the output is a machined part.
    e. Fastening process: The inputs are all three parts from the injection molding, the welding, and the turning processes and the output is a final assembled part.

2. The modeler identifies the sub-processes within the process, according to the process plan, selects their appropriate graphical representations from the library, and drags and drops them on the workspace. For example, in Figure 2, seven sub-processes include injection molding, die-casting, arc welding, three turning processes, and fastening. Note that some sub-processes are *atomic,* i.e., an end process that represents a machine and does not have any internal sub-process, whereas some sub-process may be a composite process that includes more subprocesses in it.

3. The user connects sub-processes in the workspace using the *Connector* components (directed arcs). Connectors go to Process Inputs or from Sub-process outputs to other Sub-process inputs or the Process outputs. Each connector signifies the flow of a particular Item (material, part, product, energy, etc.). Implicitly, they also signify the balance of the input and output quantities (i.e., zero-sum constraints). Figure 2 shows the workspace after the completion of this step for the example process.

4. If necessary, the data and parameters of the existing model components (sub-processes or connectors) in the library can be instantiated or modified by clicking on a component and modifying its associated data structure displayed.

5. The modeler can uniquely name the created composite process and store it in the library (under a selected folder) for future use.

What-if Analysis and Optimization

The user can use the created process model (or an existing process model from the library) for what-if analysis and optimization to get actionable recommendations. There are two key functions available to the manufacturing user: *compute* and *optimize.*

Compute—The user invokes the *compute* function against a predefined process (e.g., the models that are newly created or taken from the model library by the user). To perform computation, all parameters and controls in the process and in its sub-processes must be instantiated. For the example in this paper, the only control variables include numbers of parts allocated to each of the turning machines. If more parameters/variables are required, the user can click on the appropriate machine/sub-process icon, a parameter window will appear to allow the users to input or modify existing values, and the instantiated model can be saved under a new unique process name. The *compute* function performs the following actions:

1. Compute each process-dependent variable from the process parameters and controls

2. Evaluate each process feasibility constraint to *True* or *False* from the process parameters and controls. If all feasibility constraints evaluate to *True*, the process instance is *feasible.*

3. Create a copy of the instantiated process where all dependent variables and feasibility constraints are instantiated to constants computed in 1 and 2.

4. Optionally store the resulting instantiated process model under a new unique name in the manufacturing model library under a folder of users' choice.

Optimize—The user invokes the *optimize* function against a predefined process (e.g., the models that are newly created or taken from the model library by the user). To perform optimization, all parameters (but not control variables) in the process (and in its sub-processes, recursively) must be instantiated. The user can click on the appropriate machine/sub-process item, its parameters and controls will be displayed to allow users to instantiate the missing or modify the existing parameters. In addition, every control and dependent variable displayed have the value range for min and max bounds, which by default are negative and positive infinity if the variable is not constrained. The user can choose to put min and/or max bounds as additional constraints on any or all variables. The user can also instantiate any variable (control or dependent) to a constant, which is equivalent to setting both min and max bounds for this variable to the same value. For example, the user may click on the input flow of Turning machine 1 in the composite process in Figure 2, and set the quantity allocated variable to a constant or with the min and max bounds. When the user presses the *optimize* button, the system requests the user to choose MIN or MAX option, and to choose a (dependent) variable to be used as the objective. For example, the user may choose to minimize (MIN) the energy consumption (a dependent variable). Then, the system performs the following:

1. If there does NOT exist a feasible instantiation of process control variables, i.e., an instantiation that would make all feasibility constraint evaluate to *True*, the system reports status *Infeasible* to the user. For example, this can happen if the throughput demand on the output flow (assembled product in Figure 2) is too high for the limited production capacity of the machines used in the composite process.
2. Otherwise, if a feasible instantiation exists, the system finds a feasible instantiation of all control variables, which makes the selected objective value minimal or maximal, as required, among all feasible process instances. For example, for the process in Figure 2, the system would choose the settings of all machines (the control variables) as to minimize all feasibility constraints (including the throughput demand constraint) that minimize the energy consumption, while producing the required demand of the assembled product.
3. The system computes the instantiated process instance with the instantiation of control variables found in 2 as described in the *Compute* function.
4. The user can then store the resulting instantiated (optimal) process instant, under a new unique name, in the manufacturing model library for future use. The instantiated model has the process controls with optimal values.

In the next section, overall system architecture will be discussed, each system component will be explained in detail, and the system implementation will be described.

System architecture and implementation

This section describes (1) the high-level system architecture and components and their interactions, (2) Modelica and SPAF classes to model manufacturing components, (3) a translation of Modelica classes to SPAF classes, and (4) the SPAF model component library.

High-level architecture

Figure 3 depicts a system architecture diagram, which is composed of four main layers:

1. A Graphical User Interface (GUI) based on Modelica tool
2. A Translator from Modelica classes to Process Analytics Formalism (SPAF) classes
3. A SPAF Compiler
4. A Mixed Integer Linear Programming solver such as Optimization Programming Language (OPL) (IBM 2014) CPLEX.

At a glance, each process model in the manufacturing library described in previous section is represented as a model class in the Modelica language. The base machine model classes in the library are built-in using Modelica as part of this system implementation. However, composite process models, such as the one depicted in Figure 2, are automatically constructed by the Modelica tool from the graphical diagram designed by the user. The Modelica tool is the top layer of the system to enable (1) the composition of the models, (2) the parameter input by users, and (3) the automatic construction of Modelica composite process classes. The GUI (a working space, a library of manufacturing models, and action buttons) allows users to perform model formulation and execution. The inputs to the system from the users include conceptual model of the problem, model data collected from the real world, and queries/objectives of the models.

As described in previous section, when an optimization function is requested by an end-user for either an atomic or a composite process, the user annotates an objective function variable. To perform optimization based on MP, an MP solver, the CPLEX MILP solver, is used. It is shown in the lowest layer in Figure 3. However, since the MILP solver does not “understand” the semantics of a process performance model and its optimization, it requires the input be as a formal optimization model, expressed in a modeling language such as OPL. Such a model requires a flat specification of the (1) decision variables, (2) constraints, and (3) an objective function, as opposed to the modular specification of manufacturing processes, which may consist, recursively, of sub-processes.

System implementation must translate an optimization problem formulated against Modelica (composite) process performance models to a “flat” OPL-like language. Direct translation is complex, as it requires representing a recursively-nested composite process into “flat” data and decision variables’ structures. The intermediate Layer 3 of the SPAF compiler is used to simplify this complexity. SPAF is similar to an object-oriented modeling language designed for modeling composition and MP-based optimization. The advantage of using the SPAF layer is that it is much closer to the modular structure of Modelica, which is used to

represent graphically the described manufacturing processes; therefore, translating the Modelica composite process models to SPAF classes is considerably easier than the direct translation into OPL. Once SPAF models are constructed, they are translated, using the SPAF compiler, into an OPL model, which can be solved using CPLEX MILP solver. The key problem addressed by the SPAF compiler is a reduction procedure from SPAF to OPL and from the result of OPL to the result of SPAF. The reduction procedure must be (1) sound, i.e., the result obtained through reduction must be according to the SPAF formal semantics; and (2) complete, i.e., the result according to the SPAF semantics must be obtainable through the reduction. This is described in more detail (Brodsky, Shao, & Riddick, 2014, Alrazgan and Brodsky 2014). To use the SPAF layers, two problems need to be addressed. The first problem is the translation of the Modelica composite process models into SPAF process models. This is exactly the function of Layer 2: Modelica to SPAF translator. This translation must accurately capture the composition of SPAF models. The second problem is the creation of the built-in library of components that mimics the Modelica built-in library and the “constructors” of the composite process models. In the next two subsections, each of these two problems will be discussed respectively. In fact, only the SPAF process models will capture the semantics of manufacturing performance models, including (1) computation of the dependent variables (e.g., metrics and KPIs) from process parameters and control variables and (2) feasibility constraints. Whereas, Modelica model classes only capture the data structures of manufacturing processes, leaving the formal mathematical representation to SPAF classes.

Modelica vs. SPAF classes

At a glance, each process model in the manufacturing library is represented as a model class in the Modelica language. The base machine model classes in the library are built-in using Modelica as part of this system implementation. In Modelica environment, templates for machine model components such as Milling machines or Lathes are prebuilt. The templates can be instantiated to specific model components graphically and stored in the model library. By using drag and drop, the model components from the library can be placed into the workspace to create a composite model graphically. The Modelica system automatically constructs a composite Modelica class. Figure 4 shows an example composite Modelica model class “ManufacturingFloorExample” and one of its component classes. In the composite class, the first part is using class constructors for the relevant machines/sub-processes such as Injectionmodeling process, Lathe machine, ThreadedFastening process, DieCasting process. The second part shows that the *connect* function, which is used to describe the connections between output items/parts and input items/parts. For example, connect (dieCasting1.y, lathe3.u) indicates the *y* output of dieCasting 1 process goes to the u input of the lath3 process.

The created Modelica classes include all the sub-processes and their connections. However, these Modelica classes only contain data, not mathematical equations that reflect the relationships among inputs and outputs, which is required for computational analysis such as optimization. Optimization using Modelica is not a straightforward process. The implementation of Optimica is complex and requires significant programming and modeling efforts. A more effective approach needs to be taken to realize the optimization modeling

and execution. The SPAF classes are used to address this problem. That is, in parallel to the composite Modelica class, a corresponding SPAF composite class is created using pre-built SPAF atomic component classes for the same set of machines. A generic SPAF composite process constructor performs this process composition task. The SPAF classes encode equations and constraints required by the modeling, analysis, and optimization. The SPAF, as opposed to optimization modeling languages such as OPL, is very close to the structure and the modularity of manufacturing process performance models and those of the Modelica modeling language.

Figure 5 depicts a SPAF instance of the manufacturingFloorExample that corresponds to the Modelica instance in Figure 4. In Figure 5, the first part corresponds to the sub-processes of the Modelica model. The second part corresponds to the connector part of the Modelica model. This is done by encoding the Modelica connectors as a graph structure of the SPAF class.

Note that the overall process of the ManucturingFloorExample instance is construcated using the mscComposer class constructor, which is a component in the SPAF library. It is in this mscComoser class, all of the mathematical constraints dealing with all the subprocess and the connections between them are modeled. Some of the constraints are expressed directly in mscComoser, while others are expressed recursively invoked in subprocess constructor such as in the DieCasting, Turning 1, and metalArcWelding constructors.

Figure 6 is the final composite class of the SPAF model for the manufacturing example, a graph of connectors from external, SubProcesses, and Distribution boxes are defined; constraints for every component is added; and metrics are specified.

Modelica to SPAF translatior

To generate SPAF models from the Modelica models, a translator has been developed to translate/map the Modelica composite models to the SPAF composite models classes. A parser iterates through the steps shown in Figure 7 and translates the Modelica syntax and data structure to the corresponding SPAF syntax and structure. The parser removes Modelica annotations, package information, changes or removes the keywords, modifies the models syntax according to SPAF syntax as shown in Figure 5, creates sub-processes definitions, generates connections, and adds constraints if it is for an optimization task, and finally formats the code.

Figure 8 shows a sample of partial codes of the translator developed using Java, Modelica composite model, and the corresponding SPAF composite model (Modelica 2014; Brodsky, Shao, and Riddick 2014).

Once the final SPAF composite class is created, the existing SPAF compiler will then automatically translate the SPAF models to OPL model format, which can be solved directly by commercial optimization solvers such as IBM ILOG CPLEX. Results from CPLEX are instantiated back into the SPAF composite model. The translator will then take these results and instantiate them into the corresponding Modelica composite model as “actionable recommendations,” which will be displayed to the user through the GUI. Users can take

appropriate actions to control and improve their operations. At the same time, the users can also get an instantiated optimized version of their problem and store it back to the library for future reference. Within the GUI, action buttons allow users to execute their models by selecting the process and a metric they wish to computer or optimize, and then display the computation or optimization results.

Since the system uses a MILP solver (CPLEX), the underline assumption/hypothesis of the SPAF compiler is that the equations within the atomic SPAF component models need to be mapped into MILP formulation, i.e., they are linear in variables that range over real and integer numbers. This leads to a limitation on the expressness of the atomic SPAF component models.

Manufacturing models' library

In general, a library of manufacturing models may be organized based on a hierarchical classification, which may include unit processes of casting, joining, forming, machining, molding, and additive manufacturing. In turn, each one of those may be further classified. For example, machining processes can be further classified into drilling, gashing, hobbing, and milling. The detail of designing a classification and structure of the library is out of the scope of this paper.

In the proposed architecture, the core library needs to be maintained is the SPAF model library. For every SPAF model in the library, a corresponding class (model) in Modelica also needs to be created to enable the use of the Modelica's GUI. Composite models created using the Modelica GUI is automatically translated into SPAF composite models. In turn, SPAF models are automatically translated into lower level model required by the underlining tool, e.g., an OPL model for optimization problem to enable the use of the CPLEX MILP solver. Because the executable models are machine-generated by the SPAF compiler, it is not necessary to maintain a library of models for the underlining tools.

To construct the SPAF composite process models such as ManufcturingFloorExample exemplified in Figure 5, a model component library has been developed, components in the library include

- mscComposer: a model constructor for a composite process that encodes the computation of metrics of the process and constraints (flow zero-balance constraints, sub-process constrains, etc.) based on the process graph.
- mscConnector: a model constructor for a set of flows (material, parts, products etc.) that used as inputs or outputs of the processes.
- mscDistribution: a model constructor for describing interconnected flows and their zero-balance constraints.
- mscMetrics: a model constructor for a set of user defined additive metrics (such as power, energy, emission, and cost) and their computation.
- mscProcess: an abstract class defining generic interface of processes including concepts of external connectors and metrics.

- mscMachine: an abstract class defining generic interface of machines including concepts of input/output flows and metrics.
- mscBaseThruMachine: a model constructor for a base machine with metrics defined as a piecewise linear function of throughput.
- Instances of specific machines such as Turning1, DieCasting, InjectionMolding.

Now each of these component models starting with mscComoser in Figure 9 is described. The structures of this msxComposer class include a set of SubProcesses, with their corresponding metrics and constraints as well as structures to represent a graph of connectors. In the graph, the nodes are connectors, and edges connect these connectors. An edge between two connectors indicates that some items within the connectors "flow" from one connector to the other. For example, in Figure 2, connector 2 corresponding to the output from Die casting machine are connected by three edges to connector 3, connector 4, and connector 5 that correspond to the input of Turning machine 1, Turning machine 2, and Turning machine 3. In mscCompoer, the graphs are abstracted by its maximal connected components. For example, in Figure 2, the connected components of the graph include {2, 3, 4, 5}, {6, 7, 8, 9}, {14}, {15} etc. Each maximally connected graph component has a single "representative" connector in it, used to identify the graph component. In mscComposer class, the *GraphCompReprConnectors* structure keeps the set of all representative component connectors, while the *CompConnectors[GraphCompReprConnectors]* array gives, for each reprehensive connector, the set of all connectors in its graph component. For example, from Figure 2, GraphCompReprConnectors = {2, 9, 13, 14}, and CompConnectors [2] = {2, 3, 4, 5}; CompConnectors [9] = {6, 7, 8, 9}; CompConnectors [13] = {10, 12, 13}; CompConnectors [14] = {14}. The external Connectors structure gives the set of the input and output connector of the composite process, e.g. {1, 11, 15} in Figure 2.

The constraint section of mscComposer captures the fact that, for every connected component of the graph and every "flow" (resource such as material, parts, and products) including both input and output, the sum of all quantities must be zero, i.e., the zero-sum balance constraints. Note that some quantities are positive, which means that the corresponding "flow" supply the corresponding resource, while others are negative, which means that the corresponding "flow" consumes the corresponding resource. Connectors are either external process connectors or subProcess connectors. Each connector represents a set of items (i.e., resources) and their quantities. An edge on the graph connecting two components represent a flow of the connectors for flows from external, SubProcesses, and Distribution boxes, ExtConnectors, GraphCompReprConnectors, CompConnectors.

The class mscConnector in Figure 10 simply describes a set of flows (e.g., for materials, parts, products), and their corresponding quantities. Note that some quantities are integers (for discrete flows) and others are reals.

The class *mscDistibution* in Figure 11 is a convenience model (not used in this example) that can be used to simplify the flow graph so that the number of edges will be smaller, and the graph will be easier to understand by end-users.

The class mscMetrics in figure 12 is used to describe a set of user-defined additive metrics (such as power, energy consumption, cost, CO_2 emissions etc.), and their computation.

The abstract classes mscProcess (Figure 13) and mscMachine (Figure 14) are used to define a common interface for processes and machines respectively.

The class baseThruMachine as shown in Figure 15 is a template machine model class for the case when additive metrics are defined as a piecewise linear function of the machine's throughput.

The template machine model classes, such as mscBastThruMachine, can be used to create instances of specific machines such as the DieCasting shown in Figure 16. Using the same template, other specific machines can also be modeled.

Conclusion and future work

In conclusion, this research proposed the concept of a modeling and analysis system for what-if analysis and optimization of smart manufacturing systems. The authors implemented an initial prototype system that integrates Modelica, process analytics formalism, and IBM CPLEX. The roles of these components are as follows: (1) Modelica serves as a domain specific modeling GUI for modeling manufacturing problems, (2) SPAF enables the optimization modeling by encoding equations and constraints required by the optimization models, (3) IBM CPLEX is the optimization solver for the model execution. Key components, connectors, metrics, and a model library have been developed for a manufacturing example using Modelica classes. The Modelica composite classes are translated to a SPAF composite class, in turn, to an OPL model, and then user-driven data queries (i.e., what-if analysis and optimization) are performed through a graphical user interface using a commercial optimization solver.

The proposed system is an initial effort towards the goal of enabling users to easily and intuitively formulate models, solve problems, and get actionable recommendations through a graphical user interface, so the requirement of fully automated system is understood. Future work includes (1) verifying the needs of modeling and analysis systems by industry and identifying appropriate real world case scenarios, (2) developing generic process performance models, (3) automating the model generation process, i.e., starting from user domain-specific modeling to decision guidance feedback, (4) designing a classification and structure of the model component library, and (5) creating standardization plan for knowledge based manufacturing process component libraries.

Acknowledgments

The authors thank Abdullah Alrazgan, a graduate student from George Mason University, for his effort on SPAF compiler development. The work represented here was partially funded through cooperative agreement #70NANB12H277 between George Mason University and NIST.

References

1. Åkesson, J. Optimica—An Extension of Modelica Supporting Dynamic Optimization. 2008. 57–66. Available via <https://www.modelica.org/events/modelica2008/Proceedings/sessions/session1b3.pdf>
2. Alrazgan, A, Brodsky, A. Tech Rep GMU-CS-TR-2014-4, 2014. Department of Computer Science, George Mason University; Fairfax, VA, 22030: 2014. Toward Reusable Models: System Development for Optimization Analytics Language (OAL). [Online]. Available: <http://cs.gmu.edu/tr-admin/papers/GMU-CS-TR-2014-4.pdf> [Accessed Oct. 2015]
3. AnyLogic. [Accessed Oct. 2015] Multimethod simulation software. 2014. Available via <http://www.anylogic.com/>
4. Brodsky A, Nash H. 2005CoJava: a unified language for simulation and optimization. The Conference on Object Oriented Programming Systems Languages and Applications. :194–195.
5. Brodsky, A; Egge, N; Wang, X. Reusing Relational Queries for Intuitive Decision Optimization. Proceedings of the 44th Hawaii International Conference on System Sciences; Hawaii. 2011.
6. Brodsky A, Egge N, Wang X. 2012Supporting Agile Organizations with a Decision Guidance Query Language. Journal of Management Information Systems. :39–68.
7. Brodsky A, Shao G, Riddick F. 2014Process Analytics Formalism for Decision Guidance in Sustainable Manufacturing. Journal of Intelligent Manufacturing. :1–20.
8. CPLEX. [Accessed July 2015] 2014. Available via <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/>
9. Dietl, K; Yances, SG; Johnsson, A; Åkesson, J; Link, K; Velut, S. Industrial application of optimization with Modelica and Optimica uing intelligent Python scripting. Proceedings of the 10th International ModelicaConference; Lund, Sweden. 2014. 778–786.
10. Fritzon, P. Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica. 1. Wiley-IEEE Press; 2011.
11. Fu MC. 2002Optimization for Simulation: Theory vs. Practice (Feature Article). INFORMS Journal on Computin.
12. Gecode. [Accessed Oct. 2015] Generic constraint development environment. 2015. Available via <http://www.gecode.org>
13. Gurobi. [Accessed Oct. 2015] An easier way to make better decisions – The state-of-the-art mathematical programming solver. 2015. Available via <http://www.gurobi.com/>
14. IBM. [Accessed Oct. 2015] Modeling with OPL. 2014. Available via <http://www-01.ibm.com/software/commerce/optimization/modeling/>
15. Klemmt A, Horn S, Weigert G, Wolter K-J. 2009Simulation-based optimization vs. mathematical programming: A hybrid approach for optimizing scheduling problems. Robotics and Computer-Integrated Manufacturing. :917–925.
16. Kumar A, Veeranna V, Durgaprasad B, Sarma B. 2013A MATLAB GUI Tool for Optimization of FMS Scheduling using Conventional and Evolutionary Approach. International Journal of Current Engineering and Technology. :1739–1744.
17. Jacinto, J. [Accessed Oct. 2015] Smart Manufacturing? Industry 4.0? What's It All About?. 2015. Available via <http://www.totallyintegratedautomation.com/2014/07/smart-manufacturing-industry-4-0-whats/>
18. Jacop. [Accessed Oct. 2015] JaCoP - Java Constraint Programming solver. 2015. Available via <http://jacop.osolpro.com/>
19. Law, A, Kelton, W. Simulation Modeling and Analysis. Boston: McGraw-Hill Higher Education; 2000.
20. MathWorks. [Accessed Oct. 2015] Simulation and model-based design. 2014. http://www.mathworks.com/products/simulink/
21. McLean, C, Shao, G. Proceedings of the 2001 Winter Simulation Conference. Piscataway, NJ: Institute of Electrical and Electronics Engineers; 2001. Simulation of Shipbuilding Operations; 870–876.

22. Modelica. [Accessed Oct. 2015] Modelica and the Modelica Association. 2014. <https://www.modelica.org/>
23. Modelica Association. [Accessed July 2104] Modelica® - A Unified Object-Oriented Language for Systems Modeling Language Specification. 2012. Available via <https://www.modelica.org/documents/ModelicaSpec33.pdf>
24. NIST. [Accessed July 2104] Smart Manufacturing Program. 2014. Available via <http://www.nist.gov/el/msid/syseng/>
25. OpenModelica. [Accessed Oct. 2015] OpenModelica. 2014. <https://www.openmodelica.org/>
26. Optimizer. [Accessed Oct. 2015] WITNESS Optimizer. 2015. <http://www.lanner.com/en/media/witness/optimiser.cfm>
27. OptTek. [Accessed Oct. 2015] Opt Quest - Simulation Optimization. 2015. Available via <http://www.opttek.com/OptQuest>
28. Pathak SD, Dilts DM. 2002; Simulation of supply chain networks using complex adaptive system theory. Engineering Management Conference, IEMC '02. IEEE International. 2:655–660.
29. ProModel. [Accessed Oct. 2015] ProModel – Better Decisions – Faster. 2015. Available via <https://www.promodel.com/>
30. Rockwell Automation. [Accessed Oct. 2015] Capabilities: Sustainable Production. 2014. Available via <http://www.rockwellautomation.com/solutions-services/capabilities/sustainable-production/overview.page>
31. Rockwell Automation. [Accessed Oct. 2015] ARENA simulation software. 2014. <https://www.arenasimulation.com/>
32. Shao GA. 2014Decision Guidance Methodology for Sustainable Manufacturing using Process Analytics Formalism. Journal of Intelligent Manufacturing.
33. SimRunner. [Accessed Oct. 2015] SimRunner User Guide. 2015. Available via < ftp://www.web.ith.mx/pub/PROMODEL/Docs/SimRunner.pdf>
34. SIMUL8. [Accessed Oct. 2015] Process Simulation Software. 2015. Available via <http://www.simul8.com/>
35. SMLC. [Accessed Oct. 2015] Implementing 21st Century Smart Manufacturing. 2011. Available via <https://smart-process-manufacturing.ucla.edu/about/news/Smart%20Manufacturing%206_24_11.pdf>
36. Witness. [Accessed Oct. 2015] Witness – The Predictive Simulation Platform for Business Modelers. 2015. Available via <http://www.lanner.com/en/witness/>
37. WolFram. [Accessed Oct. 2015] Computation meets knowledge. 2014. <http://www.wolfram.com/system-modeler/>
38. WolFram Mathematica. [Accessed Oct. 2015] The world's definitive system for modren technical computing. 2014. <http://www.wolfram.com/mathematica/>
39. Arunkumar Y, Patil R, Mohankumar S. 2012Discrete Event Simulation for Increasing. International Journal of Engineering Research and Development. :36–40.

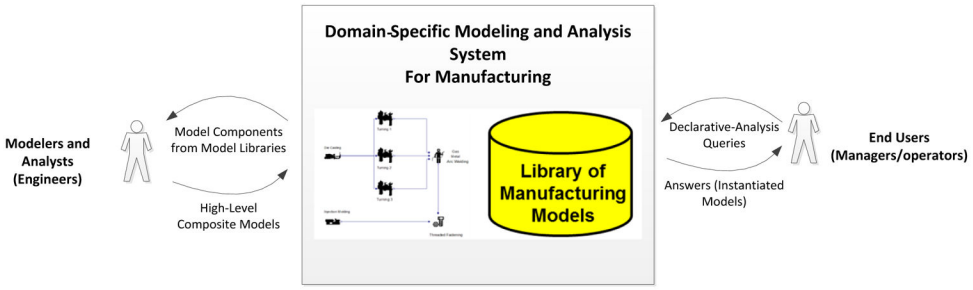

Figure 1.
Potential users of a modeling and analysis system for smart manufacturing

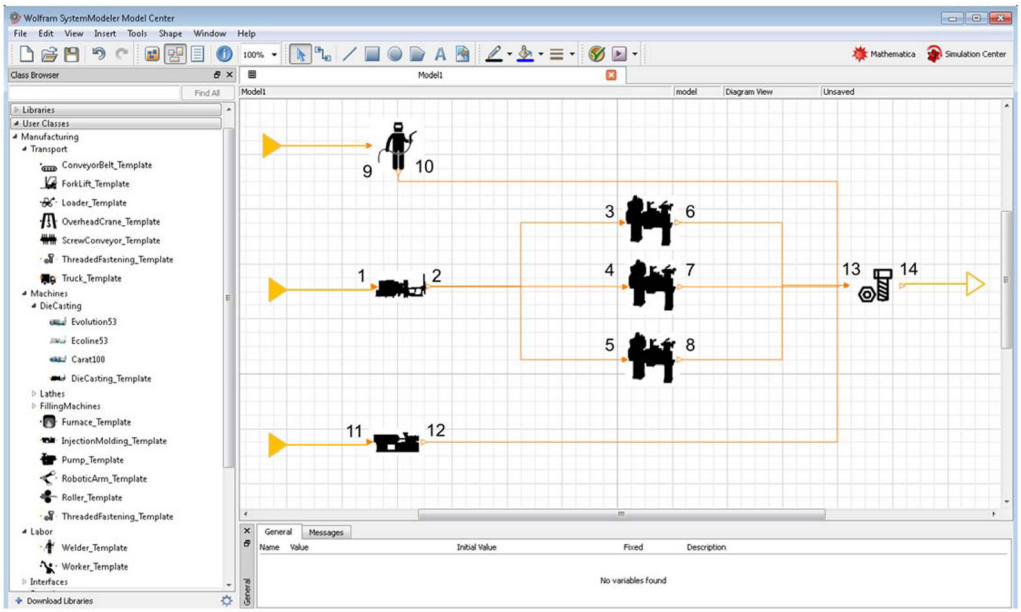

Figure 2.
Drag-and-drop screen capture of the proposed system

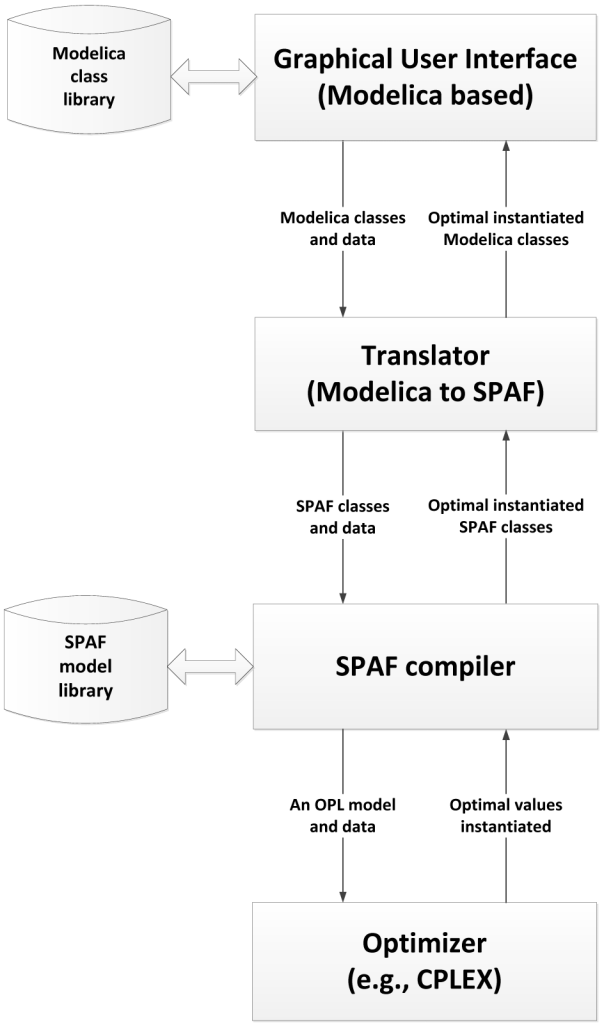

Figure 3.
System architecture

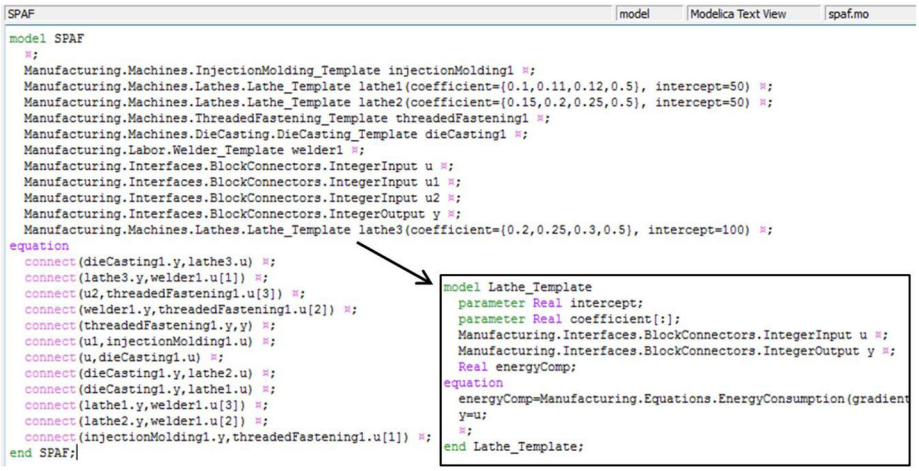

Figure 4.
Modelica composite class and its component class for the example

```
        mscProcess ManufacturingFloorExample= new
mscComposer {
// sub-procesees
        mscProcess dieCastingProc = new DieCasting{};
        mscProcess turning1Proc = new Turning1{};
        mscProcess turning2Proc = new Turning2{};

        mscProcess turning3Proc = new Turning3{};
        mscProcess gasMetalArcWeldingProc = new
GasMetalArcWelding{};
        mscProcess injectionMoldingProc = new
InjectionMolding{};
        mscProcess threadedFasteningProc = new
ThrededFastening{};
// connectors
        mscConnector dieIn = dieCasting.inputConnector;
        mscConnector dieOut =
dieCasting.outputConnector;
        mscConnector tur1In =
turning1Proc.inputConnector;
        mscConnector tur2In =
turning2Proc.inputConnector;
```

```
        mscConnector tur3In =
turning3Proc.inputConnector;
        mscConnector tur1Out =
turning1Proc.outputConnector;
        mscConnector tur2Out =
turning2Proc.outputConnector;
        mscConnector tur3Out =
turning3Proc.outputConnector;
        mscConnector weldIn =
gasMetalArcWeldingProc.inputConnector;
        mscConnector weldOut =
gasMetalArcWeldingProc.outputConnector;
        mscConnector injIn =
injectionMoldingProc.inputConnector;
        mscConnector injOut =
injectionMoldingProc.outputConnector;
        mscConnector fastOut =
threadedFasteningProc.outputConnector;
        mscConnector fastBoltIn =
threaedFasteningProc.threadedBoltInput;
        mscConnector fastPlasticIn =
threadedFasteningProc.plasticPartInput;
        mscConnector fastWeldingIn =
theadedFastingProc.weldingPartInput;
        SubProcesses =  {dieCastingProc, turning1Proc,
turning2Proc, turning3Proc, gasMetalWeldingProc,
injectionMoldingProc, threadedFasteningProc};
        DistrBoxes = {};
        ExtConnectors = {fastOut, fastBoltIn, injIn, dieIn};
        GraphCompReprConnectors = {dieOut, weldIn,
weldOut, injOut};
        CompConnectors[GraphCompReprConnectors] =
[{dieOut,tur1In,tur2In,tur3In}, {weldIn,tur1Out, tur2Out,
tur3Out}, { weldOut, fastIn}, {injOut, fastIn}];
```

Figure 5.
The composite process for the manufacturing example

```
class mscComposer extends mscProcess {
        {process} SubProcesses = ...;
        {mscDistribution} DistrBoxes =...;
        {mscConnector} ExtConnectors = ...;
        {mscConnector}
GraphCompReprConnectors = ...;
        {mscConnector}
CompConnectors[GraphCompReprConnectors] = ...;
// constraints for every connected component:
{string} AllFlows[rc in GraphCompReprConnectors] = union
(c in CompConnectors[rc]) c.Flows;
forall (rc in GraphCompReprConnectors) {
        forall (f in AllFlows[rc])  sum (c in
CompConnectors[rc], cf in c.Flows, cf == f) v[f] == 0.0;
        }
// constraints to express the composite process metrics:
        {string} AllMetricNames = union (p in
SubProcesses) p.metrics.Names;
        {string} AllDescreteNames = union (p in
SubProcesses) p.metrics.DescreteNames;
        mscMetrics metrics = new mscMetrics {Names =
AllMetricNames, DescreteNames = AllDescreteNames} ;
        forall (n in AllMetricNames)
                metrics.v[n] == sum (p in SubProcesses, sn
in p.metrics.Names : sn == n) p.metrics.v[n];
        }
```

Figure 6.
Composite class SPAF model for the manufacturing example

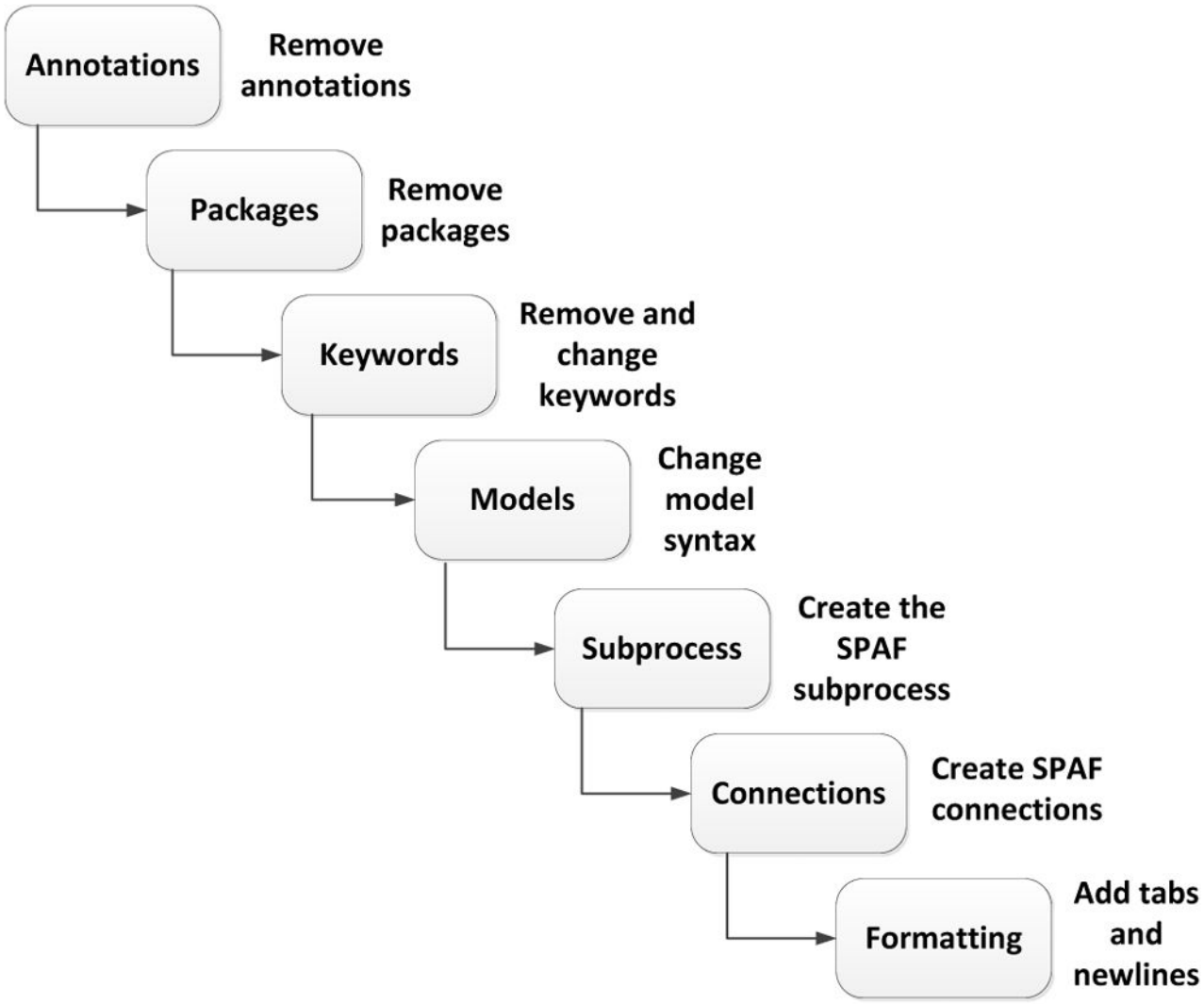

Figure 7.
Steps of translating a Modelica model to a SPAF model

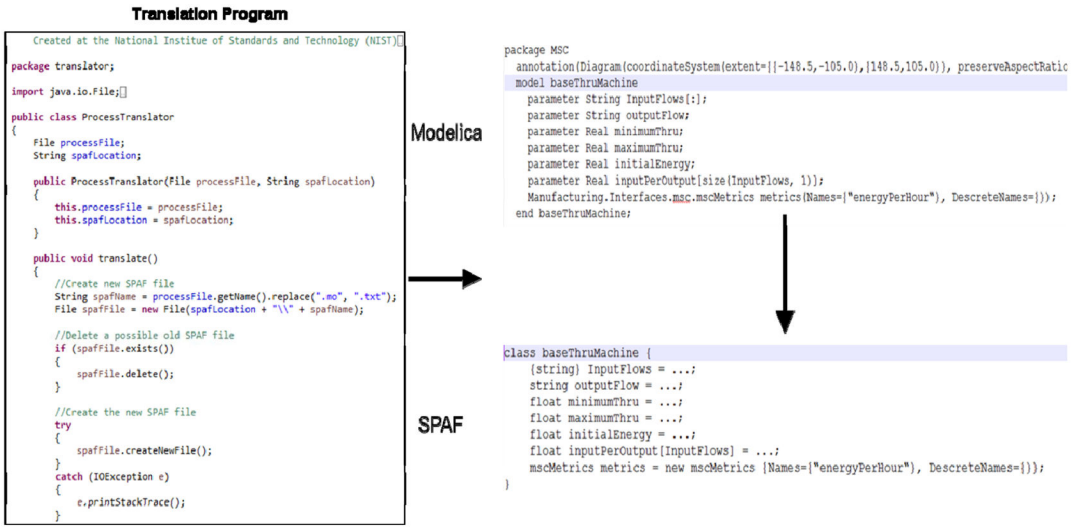

Figure 8.

An example code of the translator program and sections of Modelica model and the corresponding SPAF model

```
class mscComposer extends mscProcess {
        {process} SubProcesses = ...;
        {mscDistribution} DistrBoxes = ...;
        {mscConnector} ExtConnectors = ...;
        {mscConnector} GraphCompReprConnectors =...;
        {mscConnector}
CompConnectors[GraphCompReprConnectors] = ...;

// constraints for every connected component:

        {string} AllFlows[rc in GraphCompReprConnectors]
= union (c in CompConnectors[rc]) c.Flows;
        forall (rc in GraphCompReprConnectors) {
                forall (f in AllFlows[rc])  sum (c in
CompConnectors[rc], cf in c.Flows, cf == f) v[f] == 0.0;
        }

// constraints to express the composite process metrics:

        {string} AllMetricNames = union (p in
SubProcesses) p.metrics.Names;
        {string} AllDescreteNames = union (p in
SubProcesses) p.metrics.DescreteNames;

        mscMetrics metrics = new mscMetrics {Names =
AllMetricNames, DescreteNames = AllDescreteNames} ;
        forall (n in AllMetricNames)
                metrics.v[n] == sum (p in SubProcesses, sn
in p.metrics.Names : sn == n) p.metrics.v[n];
}
```

Figure 9.
Composer class SPAF model for the example

```
class mscConnector {
        {string} Flows = ...;
        float v[Flows] = ?;
        // string unit[Flow] = ;
        {string} DescreteFlows = ...;  // must be a subset of
Metrics
        int dv[DescreteFlows] = ?;
        forall (f in DescreteFlows) v[f] == dv[f];
}
```

Figure 10.
Connector SPAF model for the example

```
class mscDistribution {
        class distrConnector = new mscConnector{};
        sum (f in distConnector.Flows) v[f] == 0.0;
}
```

Figure 11.
Flow distributor SPAF model for the example

```
class mscMetrics {
        {string} Names = ...;
        float v [Names] = ?;
        {string} DiscreteNames = ...; // must be a subset of
Metrics
        int dv [DiscreteNames] = ?;
        forall (n in DiscreteNames) v[n] == dv[n];
}
```

Figure 12.
Metrics SPAF model for the example

```
class mscProcess {
{mscConnector} ExtConnectors = ...; // all flows in external
connectors are assumed to be different from each other
mscMetrics metrics =...;
}
```

Figure 13.
Abstract process class SPAF model for the example

```
class mscMachine {
{string} InputFlows = ...;
string outputFlow = ...;
mscMetrics metrics = ...;
}
```

Figure 14.
Abstract machine class SPAF model for the example

```
class baseThruMachine extends mscMachine {

        {string} InputFlows = ...;
        string outputFlow = ...;
        float thru = ?;
        float minimumThru = ...;
        float maximumThru = ...;
        minimumThru <= thru <= maximumThru;

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

        float energyPerHour = energyFunction(thru);

        float inputPerOutput[InputFlows] = ...;
        float unitPerHour[i in InputFlows] =
inputPerOutput[i] * thru;

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

Figure 15.
Base throughput machine SPAF model for the example

```
class DieCasting extends mscProcess {

        string inputFlow = "dieCastingInput;
        string outputFlow = "dieCastingOutput";

        class machine = new baseThruMachine {
                        InputFlows = {inputFlow};
                        outputFlow = outputFlow;
                        minimumThru = 0.0;
                        initialEnergy = 50.0;
                        maximumThru = 150;
                        slope[] = [0.2, 0.25, 0.30, 0.5];
                        break[] = [10,60,100];
                        inputPerOutput[] = [5.7]
        };
        mscMetrics metrics = machine.metrics;
        mscConnector inputConnector = new mscConnector
{Flows = {inputFlow}};
        mscConnectot outputConnector = new mscConnector
{Flows = {outputFlow}};
        {mscConnectors} ExtConnectors = {inputConnector,
outputConnector}

// constraints defining connectors' values

        outputConnector.v[outputFlow] == machine.thru;
        inputConnector.v[inputFlow]     ==     -1.0     *
machine.unitPerHour[inputFlow];
}
```

Figure 16.
An example of SPAF machine component model – Die Casting