NISTIR 8378

Model of Operational Control of Discrete Event Logistics Systems (DELS)

Timothy Sprock
Conrad Bock

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8378

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

NISTIR 8378

Model of Operational Control of Discrete Event Logistics Systems (DELS)

Timothy Sprock
Conrad Bock
Engineering Laboratory
Systems Integration Division

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8378

July 2021

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
James K. Olthoff, Performing the Non-Exclusive Functions and Duties of the Under Secretary of Commerce for Standards and Technology & Director, National Institute of Standards and Technology

National Institute of Standards and Technology
Interagency or Internal Report 8378
Natl. Inst. Stand. Technol. Interag. Intern. Rep. 8378, 40 pages (July 2021)

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8378

Abstract

Operations management systems manage flows of work, resources, and information through production and logistics systems, ensuring the correct and efficient execution of requested work. They can leverage recent advances in data availability, analytical capabilities, and industrial automation to substantially improve overall system performance. They can streamline flows of work, resources, and information across functionally heterogeneous components of production and logistics systems, which often have their own planning, execution, and data management systems. These smart manufacturing capabilities can improve system performance, but add complexity its design and operation.

This paper proposes a model of operational control enabling an integrated approach to design, analysis, and operation across heterogeneous systems, classified as discrete event logistics systems. The model identifies and formalizes operational control decision-making and actuation functions required by smart systems. The model is applied to define logical components that can be reused, specialized, and assembled into operational control system models. These components can be linked and integrated with other views of the system as part of an overall model-based engineering effort.

Key words

Operations Management; Control; Discrete Event Logistics Systems (DELS); Production and Logistics; Smart Manufacturing; System Modeling.

Table of Contents

1 Introduction **1**

2 Modeling Framework **2**

2.1 Operational Control Functions 2

2.2 Pattern For Modeling Operational Control Functions 3

2.3 Defining Control Functions 5

2.4 Decision Support Interfaces to Analysis Methods 6

2.5 Libraries of Reusable Model Components 7

2.6 Actuator Modeling for Operational Control 7

3 Model of Operational Control **10**

3.1 Which tasks to serve? (Admission) 10

3.2 When, or in what order, is an admitted task is serviced? (Sequencing) 13

3.3 Which resource(s) is assigned to serve a task? (Assignment) 14

3.4 Which process does the task required next? (Dynamic Process Planning) 18

3.5 Which state should a resource be in? (Changing State) 20

3.6 Joint Control Decisions 22

3.6.1 Which task next, and which resource will service it? (Scheduling) 23

3.6.2 For a task: which process next and which resource will execute it? (Routing) 24

4 Operational Control Model Libraries for Modeling DELS **29**

5 Conclusion **31**

List of Figures

Fig. 1 A canonical set of operational control questions defines a comprehensive functional specification of all decision-making mechanisms that a controller needs to manage the flow of tasks and resources through the system. An abbreviated sketch of this controller architecture can be found in [27] and a longer discussion in [25]. 3

Fig. 2 Graphical views of a control function model in SysML (`Admit`). 9

Fig. 3 Model views of control actuator using Admission Gate as an example. 10

Fig. 4 Example of specializing admit behavior and implementing it by specifying a sequence of steps required to bring an accepted task into the system. 12

Fig. 5 Specialized resource acquisition behaviors 17

Fig. 6 The control processes and actuators are extended from the process and resource elements also used to model the system. 29

Fig. 7 This pattern demonstrates one way that the actuators (components of the system) can be configured to control the flow of tasks and resources through the system. 30

Fig. 8 The decision support component of the controller has access to methods for each control function, each supported by its own interface definition. 31

1. Introduction

Discrete Event Logistics Systems (DELS) transform discrete things flowing through a network of interconnected resources [17, 29]. DELS include systems such as supply chains, manufacturing plants, transportation networks, and warehouses. Traditionally, each kind of DELS relies on dedicated research and development and specialized software systems. Improved performance requires increased integration and coordination among a variety of systems. For example, manufacturing plants are composed of production systems integrated with storage, fulfillment, and material handling systems. Supply chains seamlessly integrate flows between warehouses, transportation systems, and manufacturing.

DELS depend on operational control systems to coordinate the production of products or provision of services by manipulating the flow of items through the system, such as raw materials, work in progress, and other resources. Recent technological advances, such as inexpensive sensing and increased automation, are an opportunity for operational control systems to significantly improve DELS performance and functionality. However, doing this requires operational control systems to support higher levels of automated decision-making and actuation, operational flexibility, and component interoperability.

Addressing these needs in operational control requires methods and technologies to manage system complexity, maintain consistent control systems specifications, and integrate those specifications into the overall system design. Model-based system engineering (MBSE) methods are well-positioned to help with these challenges by translating stakeholder requirements for system functionality and performance into designs that can be linked to testing and verification methods [9]. Model libraries and reference architectures capture best practices, reusable artifacts, and design patterns that simplify assembly of new system models [8]. Model libraries supporting discrete event logistics systems build upon product, process, and resource abstractions [29].

This paper proposes a DELS operational control model library that aligns and integrates a controller's decision support analysis models with actuators executing prescribed control actions. These models extend the classification of control functions in a companion survey paper [28] by representing them as modeling components that can be assembled into new system specifications. The models complement design methodologies, such as the production control design methodology in [4] and more general MBSE methods [9]. Section 2 summarizes the operational control decisions identified in the survey, introduces a modeling pattern linking decision support for each kind of control decision to the execution of

the corresponding control action in the base system, and describes a way to represent the pattern in SysML. Section 3 applies the pattern to each control decision.

2. Modeling Framework

This section introduces a computer-interpretable representation for operational control functions to address the needs described in section 1. Section 2.1 reviews these functions, as detailed in [28]. Section 2.2 introduces a conceptual pattern for describing these functions and section 2.3 discusses ways to define them. Section 2.4 covers the relationship of these definitions to decision support. Section 2.5 reviews model-based methods for reusable components and section 2.6 applies them to actuator functions and actuators.

2.1 Operational Control Functions

DELS share a common abstraction of systems being controlled (*base systems*) — *products* transformed by *processes* executed by *resources* arranged in a *facility* (PPRF) [29]. *Tasks* define a unit of work by authorizing execution of a *process*, such as jobs, orders, etc. DELS specifications define what's flowing and where things can flow, as the basis for specifying control of those flows. Model-based operational control specifications must address four aspects: 1) what kinds of operational control decisions must be made, 2) where, or when, in the system control decisions need to be made, 3) how decisions aremade, or which decision support methods are selected and configured to support decision-making, and 4) how control choices are executed (desired effect realized). This paper and its companion survey paper [28] argue that DELS share an abstraction of operational control functions that manage the flow of tasks and resources through the system.

This paper proposes a library of model components that can be assembled to specify aspects (2)-(4), building upon the operational control decisions identified in the companion paper. These control decisions are summarized as: (1) 'should a task be served?' (*admission*); (2) if so, then 'when should the task be serviced?' (*sequencing*); and, (3) 'by which resource?' (*assignment*); (4) finally, 'what process step does the task require next?' (*dynamic process planning*); (5) as well as the resource-related 'in which state does a resource need to be?' (*change-state*). The order of questions indicate a typical decision-making sequence for tasks flowing through a DELS (figure 1), but are not prescriptive. In this framework, *Scheduling* and *Routing* are modeled as joint, rather than atomic control decisions.

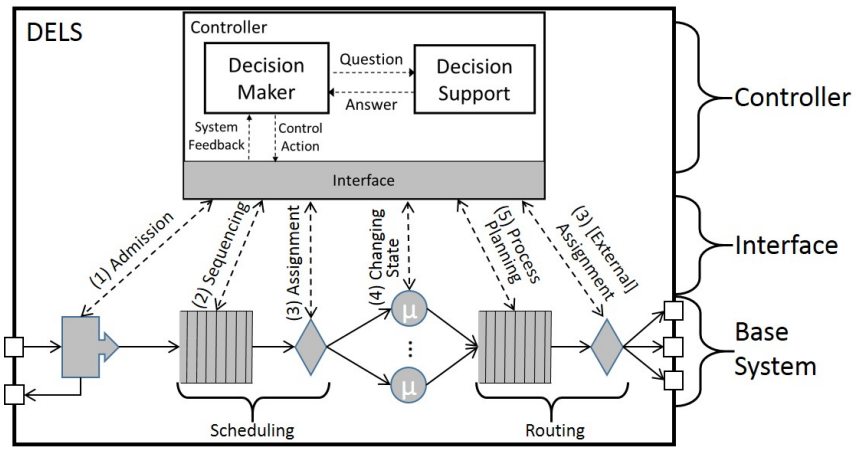


Fig. 1. A canonical set of operational control questions defines a comprehensive functional specification of all decision-making mechanisms that a controller needs to manage the flow of tasks and resources through the system. An abbreviated sketch of this controller architecture can be found in [27] and a longer discussion in [25].

The operational control models in section 3 are organized around these "what should I do?" questions decision-makers pose to decision-support (figure 1), the answers to which are appropriate control actions. The next section introduces a pattern for organizing the related elements used to describe each control question or function, including decision support, control actions, and actuators. These elements are combined to formalize an operational control model for the operational control decisions summarized above. These models, one for each operational control decisions, are described in section 3.

2.2 Pattern For Modeling Operational Control Functions

The operational control model described here links a controller's decision support to actuators in a base system that execute prescribed control actions. A formal definition of each control function threads together multiple aspects of control described in the following modeling pattern. The patterns support a modeling methodology implementing system modeling best practices, such as separating function, behavior, and structure [12]; as well as, base system behavior from controller functions. The pattern is populated for each control function in subsequent sections of the paper. The resulting control function descriptions are an unifying abstraction for building conceptual models and logical architectures of DELS operational control. These abstract model library components provide a base description for creating concrete implementations in systems.

1. **Name:** Colloquial identifier of the control function being addressed, though the literature uses various names for control decisions.

2. **Question:** Domain-independent, informal "what should I do?" kind of question that a decision-maker poses to decision-support and the answer to which is an appropriate control action.

3. **Decision Function:** Mapping of system objects and their state data (input space) to control action alternatives (action space). This transformation formalizes the informal control question/answer interface between decision-making and decision support in a controller.

4. **Actuator Function:** Expected effect of an actuator in the base system, or desired outcome of executing a control behavior. This is how the control function (action selected by controller) is carried out by the base system (actuator).

5. **Decision Expression:** An exemplar mathematical formulation of the decision function in terms of one or more decision decision variables representing the decision, for use in an optimization analysis model.

6. **Decision Support Interface:** The functional interface, or signature, of conforming analysis models that answer this particular control question. These are modeled as UML/SysML classes in this paper.

7. **Actuator Function - Model Library Component:** `Control Process` that embodies the actuator function of actuator. The actuator function is implemented with a behavior that specifies how the actuator function ("expected effect") is accomplished by the actuator. It is abstract and implemented during the design process. These components are modeled as SysML *activities*, which are an extension of UML activities, one of its ways of specifying behavior that is most suitable for flows between actions.

8. **Actuator - Model Library Component:** System `Resource` capable of carrying out the actuator function. These components are modeled as SysML blocks, an extension of UML classes, its way of specifying structure.

2.3 Defining Control Functions

It is a challenge to discover decision-support analysis models for operational control and integrate them with system specifications. The operational control literature organizes analysis models and solution methods by the name of the control function being investigated. However, the same function or method may go by many names, often domain-specific. In addition, existing terms are vague and ambiguous. Some control problems, such as scheduling, involve multiple decisions and require executing multiple actions to achieve the desired effect. In this paper, we select a single identifier for each control function. This section explores alternative ways to identify and define each control function.

One approach to resolve ambiguous terminology is to define each control function without naming it. Instead, we propose identifying each control function using informal "what should I do?" questions. This question, and its answer, begin to define an interface between the decision-maker and decision support. It is a convenient natural language approach to identify the required decision. However, formal definitions for each control function are needed to resolve the remaining ambiguity.

Formal definitions for each control function articulate the decision that must be made or action that can be taken (*decision function*), and the effect of that choice, or prescribed action, on the system (*actuator function*). The decision function is about decision support in a controller. The actuator function is related to behavior of the base system. Decision functions map system objects, such as tasks and resources and their state data, to executable control actions [4, 20, 21, 31, 33]. They are an abstraction of decision variables, policies, rules, and other formulations for making control decisions. The range of a control decision function is the set of alternative control actions (decision or action space). The actuator function maps input objects and their state to the expected effect on those objects in the base system.

The decision function is implemented with decision support analysis methods (section 2.4). These methods can be designed to take advantage of the structure of each decision problem. That is, analysis methods should conform to the functional definition and are expected to output the same type of control action. The actuator function is implemented by concrete behaviors allocated to and executed by system components (see actuators in section 2.6).

2.4 Decision Support Interfaces to Analysis Methods

Interfaces to decision support methods are necessary for interoperable operational control decision support. Software engineering best practices recommend putting decision support methods behind standard interfaces in order to make the algorithms interchangeable. Analysis algorithms for each kind of control function support an interface conforming to decision function definitions. This simplifies use of decision support by putting a variety of specific algorithms behind a small number of predefined definitions. For example, placing many scheduling algorithms under a single definition of scheduling. Systems engineers configure operations management controllers by selecting from libraries of algorithms based on their strengths and weaknesses, such as trade-offs between run-time and guaranteed solution quality. A controller selects the most appropriate algorithm depending on its decision-making circumstances. Each **decision support interface** defines inputs to the algorithms for its particular control function, how the algorithms can access system information, and expected output (decision choice).

Analysis algorithms that implement decision support interfaces can be formulated with one or more decision variables (in expressions) that also conform to the decision function, ensuring that the algorithm can be used via the interface. A **decision expression** implements the decision function by specifying decision variables used by optimization formulations of the operational control decision problem, guiding construction of analysis models that conform to the decision function. Explicit decision expressions enable solutions from analysis models to be mapped directly into executable actions, by using single unique variable for each control decision and each required action. For example, an optimization model may specify the expression $x_l^R = 1$ if resource $R \in \mathscr{R}$ is assigned to execute the next process step of task l, or $x_{l,t}^R = 1$ if the assignment is for future time period $t \in T$.

Explicit decision expressions contrasts with implicit analysis formulations commonly used in practice to reduce solution time. For example, policies and decision rules, such as production switching curves and (s,S) inventory policies, are implicit formulations created from analysis models structured to conform to decision functions. In these kinds of models, the solution to an aggregate decision problem is used to make dynamic, on-line choices for each decision. For example, off-line analysis may determine the percentage of production capacity that should be dedicated to producing product 1. A controller uses that output to guide dynamic acceptance of requests to produce product 1. Another approach addresses control functions indirectly. For example, if a resource is assigned to a task to produce product 1, but is setup to produce product 2, then another setup or change-over is implied.

However, the output of analysis models needs to be actionable. This can be achieved by transforming solutions to implicit formulations into explicit decisions.

2.5 Libraries of Reusable Model Components

Libraries of reusable model components and design patterns enable system designers to produce candidate system designs by assembling predefined model components according to the patterns. This reduces the burden of (re-)design efforts. Object-oriented and model-based systems engineering methods (OOSEM, MBSE) guide the definition of system components and their behavior, which can be refined and composed to execute required system functions [9, 10]. Model-based system specifications developed using general-purpose modeling languages such as the Systems Modeling Language (SysML) are more expressive and less constrained by analysis-specific concepts or artifacts [18].

SysML provides many modeling capabilities essential to designing operational control systems. These include: modeling hardware and software components; separating component function, structure, and behavior; and creating model libraries. Model-based system specifications can be integrated into more comprehensive system architecture models. The system design can be linked to business cases, stakeholder needs, and design requirements. We use SysML in this paper to define model library components for software decision support (section 2.4) and abstract actuator functions, behavior, and structure (section 2.6).

2.6 Actuator Modeling for Operational Control

Actuators for operational control define requirements, constraints, properties, and interfaces that are implemented by concrete system designs. Abstractions for these in model libraries provide reusable aspects of behaviors and structural elements associated with each control function.

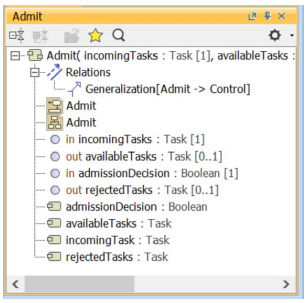
Actuator functions define expected effects of an actuator on their base systems. They are modeled as SysML activities that type (are used by) *actions* (steps) in other activities), as described in this section. **Actuators** are modeled as SysML blocks defining structural features to handle flows when executing the corresponding control function. Actuators in this paper draw inspiration from discrete event simulation and discrete event dynamic system modeling components [6].

Modeling Actuator Functions as SysML Activities

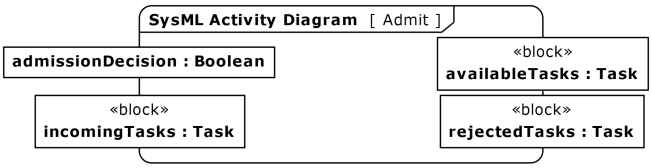
SysML includes multiple, not exactly equivalent, graphical views for showing model library components for **actuator functions** (figure 2). The model tree view available in most SysML tools (showing `Admit` in figure 2a) displays an actuator function activity with its parameters and parameter nodes indented under it, as well as relationships with other activities, such as the abstract `Control` activity. An activity diagram (figure 2b) is associated with each actuator function, showing parameter nodes as rectangles on the border of the diagram (each corresponding to an activity parameter). It is empty inside to act as template filled in or implemented with system specific actions (or instructions) on how to execute the function, as shown in the `Accept Job into Workcell` activity in figure 4 (section 3.1). Finally, the actuator function activity (`Admit`) is used as the type of actions in other activity diagrams (figure 2c, showing only an action, omitting the the activity it is in, see example referred to above). Actions are sometimes called "usages" of an activity that is defined elsewhere (as in the 2b diagram). Usages of parameters are shown as small squares (pins) on the border of actions. Pins are connected by flows to other actions in the same activity, as in the example above. The rest of the paper shows this view for actuator function model library components corresponding to control functions, because the view is similarity to discrete event simulation model blocks (not to be confused with SysML blocks, see, e.g., Arena, SimEvents, etc.).

Modeling Actuators as SysML Blocks

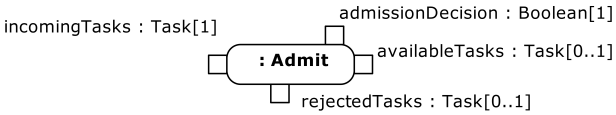
Similar SysML views apply to model library components for **actuators** (figure 3). The model tree view in most SysML tools (showing `AdmissionGate` in figure 3a) displays an actuator block with its properties, ports, and operations indented under it, as well as relationships to other blocks. A more graphical view of blocks is available in block definition diagrams (BDD). These show blocks as rectangles marked with «block »at the top (showing `AdmissionGate` in figure 3b, omitting the diagram frame), with ports as small squares on the border, and compartments for properties and operations (properties omitted here). Finally, the actuator block `Admission Gate` is used as the type of parts in other blocks (figure 3c showing only a part, omitting the block it is in). Parts are sometimes called "usages" of a block defined elsewhere (as in figure 3b), to construct other system component specifications. This applies the same definition-usage pattern as activities and actions for **actuator functions**, but the rest of the paper shows block definition views for actuator



(a) Admit activity in a model tree view available in most SysML tools.



(b) Admit activity diagram view with parameter nodes displayed on the diagram frame.

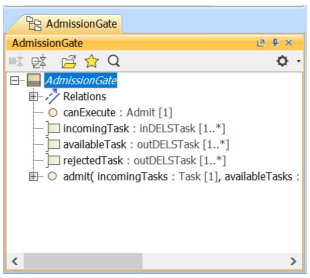


(c) Admit activity used as the type of a SysML action, with usages of its parameters displayed as pins on the action.

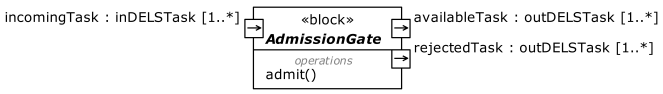
Fig. 2. Graphical views of a control function model in SysML (`Admit`).

model library components corresponding to each control function (as in figure 3b). The block definition and usage views are both similar to discrete event simulation modeling, but the definition view is preferred in SysML to define a model library elements.

Operational control system modeling starts with architecture by selecting abstract actuators and actuator functions in model libraries, then implements specialized, concrete system components. This elaborates actuator functions with concrete behaviors and allocates them to system components implementing the function. In some systems, the concrete actuator implementing a control function might be software. For example, resource assignment actions might be implemented by generating an auxiliary task requesting another resource, such as an automated guided vehicle (AGV), to move workpieces, raw materials, etc. to the assigned resource. In other cases, one structural resource might be allocated multiple functions. For example, a robotic arm might implement acceptance or resource assignment decisions through a sequence of get/move/put behaviors that place a workpiece in a machine.



(a) Admission Gate represented in the model tree, the model without graphics.



(b) Admission Gate viewed as a block with ports for handling flows into and out of the block.

(c) Admission Gate object "usage" as a part of another block.

Fig. 3. Model views of control actuator using Admission Gate as an example.

3. Model of Operational Control

Decision support methods should be designed to produce answers (recommended control actions) that base systems can execute. To ensure this, each control function, defined by decision and actuator functions, is linked to its decision support, actuator behavior, and actuator. This establishes consistency among disparate software and hardware components that act together to effect optimal change in the system. The output from decision support (a selected action) is communicated to the actuator. The actuator has a behavior that carries out the selected action. Modeling these three pieces of operational control — decision support, actuator behavior, and actuator — from consistent functional definitions, such as those proposed in this paper, enables them to be designed separately with more assurance that the pieces will work together when assembled. Sections 3.1 through 3.5 do this for the control functions outlined in section 2.1, while section 3.6 shows how to combine them into more complex functions.

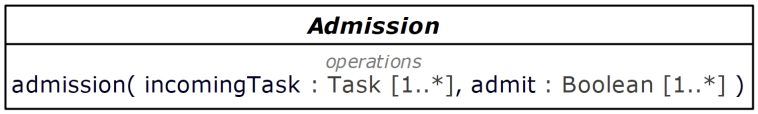
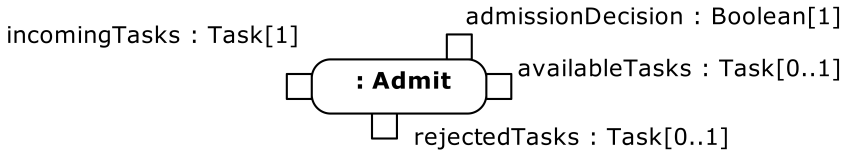
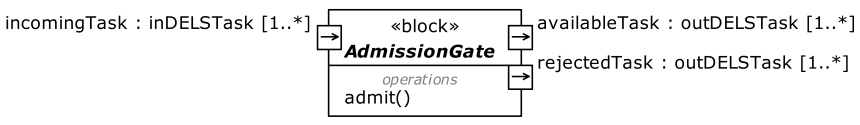
3.1 Which tasks to serve? (Admission)

When a request for a service (task) arrives at a system, the controller must decide whether to serve it. This is the admission decision. This decision controls the rate of work introduced into the system by accepting or rejecting tasks [13]. However, rejecting tasks may incur

monetary penalties, lost sales, or lost goodwill. Admission decision-making processes may also evaluate system state, including available production capacity, raw material inventory on-hand, operator availability, etc.

The *Admission* **decision function** maps ($\mapsto$) a task to a Boolean value (true/false), providing a yes/no answer on whether to admit the task into the system (table 1). The **decision support interface** (`Admission`) captures this decision function in the signature of its *admission*() operation, which defines the types of inputs and outputs to the behavior implementing the function, in this case, an analysis method. Decision-support models can be formulated with binary **decision variables** (x_l), where $x_l = 1$ if task $l \in \hat{\mathscr{L}}$ is admitted into the system. However, some analysis methods may implement the decision function indirectly. For example, an indirect policy may give conditions for acceptable tasks, such as posting a menu of prices and lead-times, and admit any task that agrees to the conditions. These policy-based or rule-based methods must conform to the decision function to ensure they recommend actionable control choices.

Table 1. 'Which tasks to serve?' (Admission)

Decision Function	$Admission:\ \text{Task} \mapsto Bool$
Actuator Function	$Admit(Task) := System.taskSet \leftarrow System.taskSet \cup Task$
Decision Expression	$x_l = 1,\quad$ if arriving task $l \in \hat{\mathscr{L}}$ is accepted
Decision Support Interface	
Actuator Function - System Model Library Component	
Actuator - System Model Library Component	

The **actuator function** (*Admit*) takes the *admission* decision from the controller as input to transform *incoming tasks* into either *available tasks* (into the system) or *rejected tasks* (out of the system). The corresponding model library component models the **actuator function** as a SysML activity `Admit`. The admission control decision and incoming, available, and rejected tasks are modeled as input/output activity parameters. This activity serves as a template for creating conforming behaviors capable of implementing the actuator function. For example, specialized material handling behaviors that implement `Admit` specify the sequence of steps required to bring an accepted *incoming task* into the system and add it to the system's collection of *available tasks* (figure 4). Likewise, alternative steps might specify what to do with *rejected tasks*.

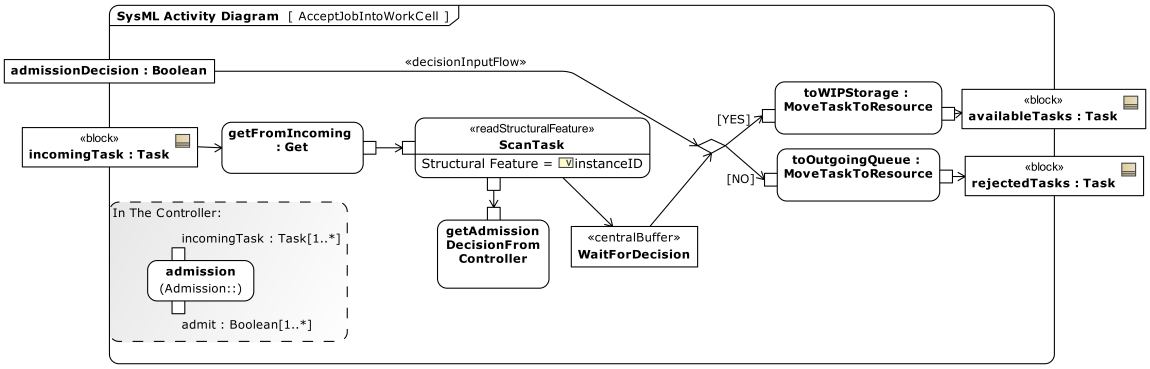


Fig. 4. Example of specializing admit behavior and implementing it by specifying a sequence of steps required to bring an accepted task into the system.

Behaviors implementing the actuator function `Admit` are executed by an `Admission Gate` resource, modeled as a SysML block. It is the structural component associated with the **Actuator Function**. It conforms to the functional definition by defining interfaces for handling the flow of *incoming*, *available*, and *rejected tasks*. The interfaces are modeled as SysML ports, the white boxes with arrows on the edge of the `Admission Gate` block. The ports are typed by `inDELSTask` and `outDELSTask`, which represent abstract placeholders for structural features that support the flow of tasks. The actuator for the *admit* control function could be more appropriately named "Admitter", but Admission Gate seems to be more familiar.

Like the actuator function model library component above, the `Admission Gate` is an abstraction (or template) for specifying concrete system components capable of implementing the Admit function. The `Admission Gate` actuator might be implemented, for example, by a robotic arm that retrieves the *incoming task* from the material handling sys-

tem (MHS), such as a conveyor or automated guided vehicle (AGV). This robotic arm is a specialization of the abstract `Admission Gate` and implements the *admit*() operation with a specialized *getMovePut*() behavior suited to moving tasks into a queue. Alternatively, the material handling system might bring tasks to the system via a conveyor, whereupon the `Admission Gate` might be implemented by a pneumatic pusher that moves tasks from the centralized conveyor onto the system's local conveyor (or local queue).

Conceptually, an *incoming task* arrives to the system, flows into the `Admission Gate`'s *incoming task* port, and triggers the `Admit` actuator function. The actuator function interacts with the `Admission` decision support, which returns a Boolean *admit* decision. This decision is used by the actuator function `Admit`() and structural actuator `Admission Gate`, which implements the behavior and physical structure/capabilities, respectively, required to turn the *incoming task* into either an *available task* or *rejected task* flowing out of the gate.

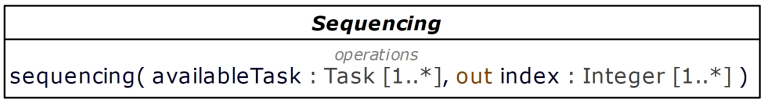
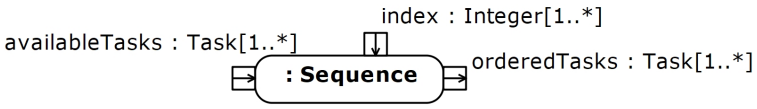
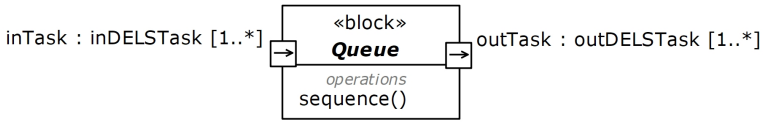
3.2 When, or in what order, is an admitted task is serviced? (Sequencing)

Sequencing decisions specify the order, or partial order, that available tasks are serviced by the system. Sequencing includes decisions such as *prioritizing* some customers' tasks over others, *coordinating* tasks for things sent to the same customer, *batching* similar tasks together for efficient processing or transport, *delaying service* of a task until a future period (back-ordering), and *splitting* tasks into smaller lots to be processed over time.

The sequencing **decision function** (*Index*) maps each task to its position (in $\mathbb{N}$) to be served, providing an order in which to serve waiting tasks. Its associated **decision expression** specifies decision variables ($x_{l,j}$) denoting whether a particular task l is served j^{th} (is in the j^{th} position). The **decision support interface** `Sequencing` captures this decision function in the signature of its *sequencing* operation. Decision support methods implementing this function might be exact and heuristic. Heuristic methods, including priority rules such as earliest due date (EDD) or shortest processing time (SPT), are common decision support methods for sequencing work in a queue (mapping tasks to service position).

The **actuator function** `Sequence` sorts the *available tasks* (physically or virtually) by the *index* output by decision support. The sequencing actuator returns an *ordered set of tasks* or the just the next task. These sequencing behaviors are executed by `Queue`-like **actuators** where the available tasks are waiting for service. This is denoted by the *sequence*() operation on the `Queue` actuator block. The `Queue` defines by two ports: one for handling the input of tasks (*inTask*) and another for handling the return of selected tasks (*outTask*). The actuator for the *sequencing* control function could be more appropriately

Table 2. 'When, or in what order, an admitted task is serviced?' (Sequencing)

Decision Function	$Index : \text{Task} \mapsto \mathbb{N}$
Actuator Function	$Sequence(Index) := \text{System.taskSet} \leftarrow sort(\text{System.taskSet}, Index)$
Decision Expression	$x_{l,j} = 1, \quad$ if task l is served j^{th}
Decision Support Interface	
Actuator Function - System Model Library Component	
Actuator - System Model Library Component	

named "Sequencer", but queue seems to be more familiar.

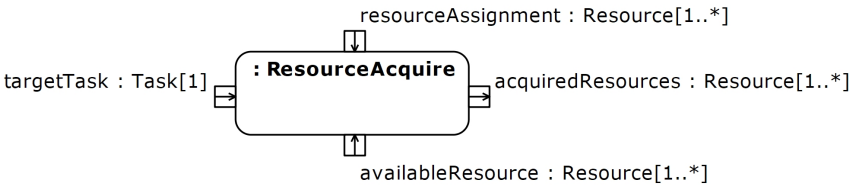
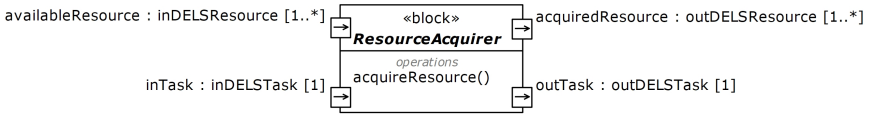
A range of technologies with varying capabilities might sequence tasks in complex control behaviors. Some non-automated storage solutions might only be capable of simple control behaviors. For example, a gravity-fed conveyor is only be capable of enforcing a First In First Out (FIFO) discipline. Some technologies are not be capable of enforcing any sequencing discipline at all. For example, a simple storage rack requires the operator to execute the desired sequencing discipline, possibly with the aid of pick lights or other simple technology. Simple storage technologies might be augmented with automated technology, such as a robotic arm capable of picking specific items, creating an automated storage and retrieval systems (ASRS).

3.3 Which resource(s) is assigned to serve a task? (Assignment)

Resource assignment refers to many closely related decisions focused on matching scarce resources to tasks (units of work) based on resource capacity and capability. In manufac-

turing systems, resources might include labor, critical processing equipment, or material handling equipment. Tasks might also require auxiliary resources such as tools, fixtures, and storage locations to enable process execution.

Table 3. 'Which resource is assigned to serve a task?' (Assignment)

Decision Function	$Assign : Task \times Resource(s) \mapsto Resource(s)$
Actuator Function	$Acquire(Task, Resource(s)) :=$ $Task.nextProcessStep.requiredInputResource \leftarrow$ Resource(s)
Decision Expression	$x^R_{l,j} = 1$, if resource $R \in \mathscr{R}$ is assigned to execute process step $O_{l,j}$, the j^{th} process step of task l
Decision Support Interface	
Actuator Function - System Model Library Component	
Actuator - System Model Library Component	

The **decision function** *Assign* maps a `Task` (l) and set of `Resources` ($\mathscr{R}$) to the subset of resources ($R \in \mathscr{R}$) that will serve the task (table 3). Resources required to serve a task are defined by the `Process` authorized by the task. When a resource is assigned to a task, in most cases, that resource is assigned to execute the *next process step* of that task's required process. However it may be advantageous for the decision support to assign resources to multiple tasks and process steps at once to minimize conflict for scarce resources, but this is an extension of the base function. *As with most decisions, the more choices made concurrently, the better the overall solution.*

This *resourceAssignment* choice is executed by an **actuator function** `Acquire` that acquires the assigned *availableResources* to satisfy the *requiredInputResources* of the task's *nextProcessStep*. The actuator function reflects the association between Tasks and Re-

sources, and both objects are affected by the assignment.

In this framework, `Tasks` authorize the execution of a `Process` which is composed of *processSteps* (typed by a `Process`). In this notation, *Task.nextProcessStep.requiredInputResource* denotes that *nextProcessStep* is a property of `Task` and *requiredInputResource* is a property of `Process`, which types the property *nextProcessStep*. The variable assignment operator ($\leftarrow$) indicates that *Resource* is being assigned to satisfy the requiredInputResource slot of nextProcessStep.

The `Resource Assignment` **decision support** takes an *available task* and *available resources* and then returns a *resourceAssignment* specifying the selected resources assigned to serve the task. This is indicated by `Resource [1..*]`, the type and multiplicity of the output *resourceAssignment*. The decision variable $x^R_{l,j}$ specifies if resource(s), or resource group, $R \in \mathscr{R}$ is assigned to execute the j^{th} process step of task l ($O_{l,j}$). In this notation, $O_{l,j}$ is the a process step of process plan $\mathscr{P}_l$. Explicit assignment formulations define decision variables for each possible match between resources and tasks. This enables the solution to the optimization problem ("which specific, identifiable resource should be acquired?") to be mapped directly to an executable action.

In many cases, process steps require multiple resources to be assigned in order to coordinate resources' availability and reserve capacity. The decision expression $x^R_{l,j}$ incorporates the capability to assign resources to multiple process steps at once, as well as, assigning resource groups (multiple resources) to one or more process steps. For multiple resources, R is defined as a resource group containing multiple resources and $\mathscr{R}$ is a set of resource groups, which can be constructed, for example, by enumerating valid combinations of resources and identifying each group as a single resource group that can be assigned to a process (Mati and Xie [16]). Second, the assignment can be made for any process step in the task's process plan. Özgüven et al. [19], for example, allows resources to be assigned to any step in a flexible process plan.

Many assignment formulations abstract, or simplify, the control problem by only assigning the bottleneck resource and/or only the next process step required by a task (dynamic assignment). For example, $x^m_l = 1$ if resource $m \in \mathscr{M}$ is assigned to execute the next process step of task l. In these simplified assignment problems, $\mathscr{M} \subseteq \mathscr{R}$ is a subset of resources, such as critical machines, bottleneck resources, etc. Practically, *Task.nextProcessStep* either stores the index (j) of the next process step or stores a pointer/reference to $O_{l,j}$, however you want to view it.

The output from the **decision support** (*resourceAssignment*) is passed to the `Resource`

`Acquire` **actuator function** which is executed by the `Resource Acquirer` **actuator** to acquire the required / assigned resources. The `Resource Acquire` «Activity» uses the *resourceAssignment* choice and specifies an implementation for how the assignment is executed to transform *available resources* into *acquired resources*, resources that are ready to execute the require process of the input task.

Selecting and implementing appropriate `Resource Acquirer` actuators usually depends on factors such as the relationship between the requesting and requested resource, the types of resources assigned, and whether the task is being brought to the resource or if the resource is being brought to the task. Note that implementing actuators also encompasses specifying actuator behaviors, and both conform to the actuator function definition. Common approaches to acquire or seize a resource include: move/flow task to resource (stationary resources), seize (consumable or passive), or asynchronous request (mobile autonomous). When the resource is stationary, the assignment can be executed by a switch that directs the *flow of the task to the resource* using material handling to bring workpiece to machine. Another approach is to *seize the resource and bring it to the task*. Resources such as fixtures or inventory can be executed by seizing the resource from a pool/stocking point. Finally, for resources that are autonomous and mobile, the system can *request that resource come to task*. These acquisition behaviors have a big impact on the way system and analysis models are constructed. These three kinds of acquire behavior can be modeled either as specializations of `Resource Acquire` and/or as optional steps within a Resource Acquire behavior (figure 5).

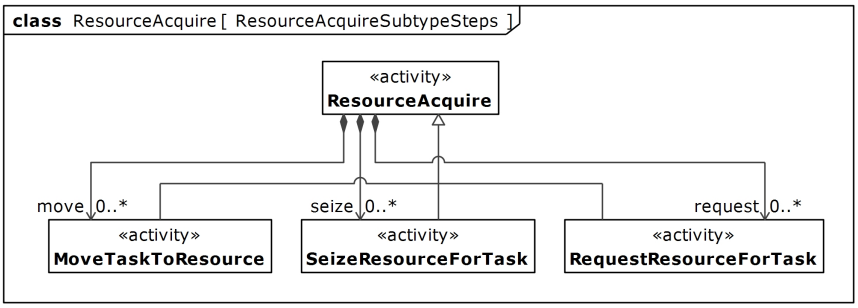


Fig. 5. Specialized resource acquisition behaviors

Most systems must implement a combination of these behaviors and actuators. For example for stationary equipment in a work cell, the assignment mechanism might direct a task into the equipment's queue via pneumatic switch on a conveyor. For resources in a central pool of available units, e.g., input materials, fixtures, or tools, the assignment

actuator might be implemented as a robotic arm or automated guided vehicle (AGV) that "seizes" the resource and transports it from the central buffer to the work station. For autonomous resources such as human operators or resources belonging to another system, the resource may be requested, but then may require assistance to get to the desired location.

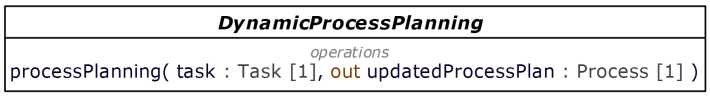
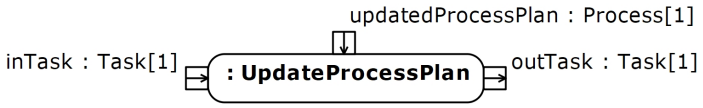
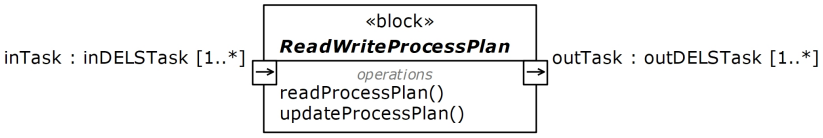
3.4 Which process does the task required next? (Dynamic Process Planning)

Process plans are processes that organize the execution of other processes as process steps connected by sequencing and timing constraints. Process plans often specify multiple ways that a requested process may be executed. These flexible process plans contain alternative paths, or sequences, of process steps that result in the same outcome. Examples of these flexibilities include alternative sequences to put parts into an assembly or create features on a part, alternative processing methods, alternative paths to move and/or store work in process (WIP). They may also simply contain multiple independently-crafted plans that can be selected from prior to the start of processing. Dynamic, or near-real-time, process planning functions are related to tactical process planning functions defining how products are made or orders are assembled. Process planning does not need to be the last control function, and it may not require a designated physical actuator.

The **decision function** *Dynamic Process Planning* maps a *Process Plan* ($\mathscr{P}_l$) to the updated process plan (*Process Plan'*) (table 4). The dynamic process planning control function focuses on resolving flexibility in the process plan 'on-the-fly', pruning unnecessary alternative paths and augmenting process plans with additional steps as needed. Process plans for a product typically only contain the primary transformation (*Make*) process steps. This function may also augment declarative process plans, containing only primary transformation (*Make*) process steps, with auxiliary steps, such as moving tasks (*Move*) to a new location or placing it in temporary storage (*Store*). Additional details on process plan representation and notation are discussed in section 6 of [26], but an overview of required background is provided here.

The base process planning model consists of a set of tasks $\mathscr{L}$, each having a process plan $\rho_l = (O_{l,1}, \ldots, O_{l,p_l})$ defined as an sequence of p_l process steps required to complete the task. Linear process plans specify a single, complete sequence of process steps. They can be extended to include process plans that specify precedence constraints between process steps and allow the analysis method to define execution sequences [14]. Network-based representations of flexible process plans capture execution options (flexibility) using AND/OR digraphs [32].

Table 4. 'Which process does the task required next? (Dynamic Process Planning)

Decision Function	$DynamicProcessPlanning:$ ProcessPlan $\mapsto$ ProcessPlan$'$
Actuator Function	$UpdateProcessPlan(Task, ProcessPlan') :=$ $Task.processPlan \leftarrow ProcessPlan'$
Decision Expression	$x_{l,j} = 1$, if process step $O_{l,j} \in \mathscr{P}_l$ is the next process step of task l
Decision Support Interface	
Actuator Function - System Model Library Component	
Actuator - System Model Library Component	

The **decision expression** may produce an updated process plan (implement the decision function) by selecting just the next process step or resolving every flexibility, or option, stored in the process plan. When the set of linear process plans is represented as a single flexible process plan, the decision variable ($x_{l,j}$) selects the next process step ($O_{l,j}$) from process plan $\mathscr{P}_l$ for task l. Selecting the next process step results in an updated process plan (and/or marking of the original) containing the selected and/or pruned path and augmented process steps. The **decision support interface** `Dynamic Process Planning` implements the **decision function** in its *processPlanning*() operation, accepting a *task* and returning an *updated process plan*.

The **actuator function** *Update Process Plan* implements the process plan selection (*ProcessPlan'* by "marking" the process plan on the task denoting selected and pruned paths. As noted above, *Task.nextProcessStep* either stores the index (j) of the next process step or a reference to the process step $O_{l,j}$. The actuator function is modeled with `UpdateProcessPlan` and `ReadWriteProcessPlan`. The physical actuator `ReadWriteProcessPlan`

reads the process plan from the task (*readProcessPlan*()) and triggers decision support *processPlanning*(), an operation on `DynamicProcessPlanning`). The decision support returns an *updatedProcessPlan* to the actuator function `UpdateProcessPlan`. The actuator function **model library component** `UpdateProcessPlan` "writes" the updated process plan to the task.

Often, process planning is combined with solicitation and selection/assignment of resources, including external ones, to create the more familiar routing function discussed in section 3.6.2.

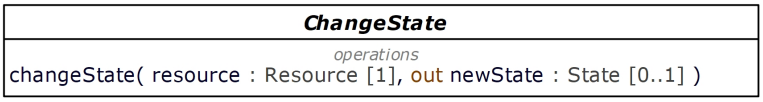
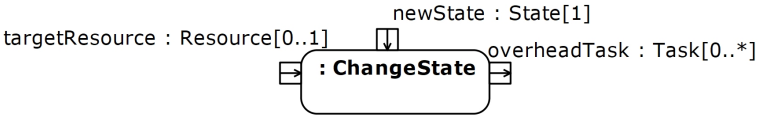
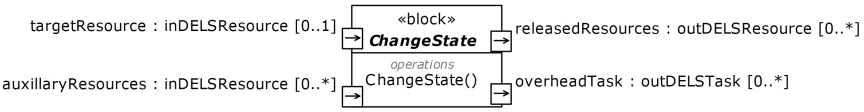
3.5 Which state should a resource be in? (Changing State)

Resource assignment (section 3.3) introduced the idea that executing the resource-to-task assignment may depend on the kind of resource being assigned, with how much work may be assigned to and executed by a resource also depending on the kind of resource. For example, there are many differences between assigning an item in inventory vs a machine or operator to a task. One classification that captures this distinction is discrete state vs capacitated resources [24]. For discrete state resources, work can only be assigned if the resource is in the correct state, e.g. *available* and set-up to execute a particular *process* (provide a particular Service). For capacitated resources, resource objects can only be assigned to a task when one is available, e.g. tools or parts in inventory can only be assigned if one is in stock.

State-based abstractions are common in analysis modeling. The control function for changing the current capability or capacity of a resource uses an abstraction of state that unifies this class of control decisions. Resources might provide multiple services (capabilities), but only one at a time. For example, a machine could be capable of executing multiple types of process or producing several types of parts, but only one type at a time. Resource capability state describes the function, process, or service that a discrete state resource is currently configured to execute and/or a geographic location that the resource can provide its service. A discrete-state resource's capability set defines one state space, where its capability it can currently offer (its capability state) is a component (subset) of its current state (`Resource.`*state*). Changing the capability of a resource includes actions such as changing set-ups or tooling for machines; anticipatory movement and pre-positioning of inventory, tools, or vehicles; and maintenance to preserve availability and capability. Resource capacity describes the amount of work that can be assigned to a particular resource. Changing the capacity of resources includes inventory replenishment of input ma-

terial stocks, maintenance to preserve availability, and turning on additional machines or increasing the processing rate.

Table 5. 'Which state should a resource be in? (Changing State)

Decision Function	$ChangeState:$ Resource.state $\mapsto$ State
Actuator Function	$ChangeState(\text{Resource}, \text{NewState}) :=$ Resource.state $\leftarrow NewState$
Decision Expression	$x_j^R = 1,$ if resource R is changed to state j
Decision Support Interface	
Actuator Function - System Model Library Component	
Actuator - System Model Library Component	

The **decision function** *ChangeState* maps the state of the resource (`Resource`.*state*) to States that it can transition to (capability or capacity) (table 5). The **decision expression** $x_j^R = 1$ specifies a choice to transition resource R to state j, where state is an abstraction for capability or capacity. Here we assume that to some degree that the ability to transition to the new state is not dependent on the current state, but that for execution purposes the current state of the resource (`Resource`.*state*) can be determined. The **decision support interface** `ChangeState` implements the **decision function** in its *changeState*() operation, taking a *resource* as input and returning a *newState*.

The **actuator function** (*ChangeState*) then changes the resource's state (`Resource`.*state*) to the prescribed *newState*. The actuator function is implemented in the model library as the `ChangeState` activity. In cases where additional resources are needed to support the transition of the resource from one state to the prescribed state, the controller may generate one

or more *overhead Tasks* authorizing those resources to execute the `ChangeState` process, including the target resource receiving or accepting the state change. Examples of overhead tasks include inventory orders, maintenance tasks, set-up tasks, etc. These overhead tasks must be accepted, scheduled, and executed by their respective systems, e.g. maintenance, material handling, or procurement. The model library actuator function `ChangeState` activity can be specialized to directly support changing capability and capacity. These specialized behaviors are modeled after set-up (*changeService*) and replenishment behaviors (*increaseCapacity*), respectively.

The actuator `ChangeState` provides an interfaces (ports) for inputting *auxiliaryResources* and, when execution is complete, outputting those resources as *releasedResources*. The `ChangeState` actuator function may be implemented as a behavior of the resource itself, e.g. machine setup/repair, or as an external actuator, e.g. a maintenance system. This is modeled as an optional *targetResource* input (which is `self` when implemented as a behavior of the target resource). Finally there is a port for outputting *overhead Tasks* when they are required to summon auxiliary resources to support executing the state change.

3.6 Joint Control Decisions

The previous sections modeled atomic control decisions that typically only have one effect on the system. However, some common operational control problems are in fact joint control decisions. That is, joint control functions require making two (or more) separate decisions and executing two (or more) separate actions. For example, flexible jobshop scheduling problems with process plan flexibility (FJSP-PPF) are modeled as the composition of the following decision sub-problems [19]: (i) *select a process plan* for each task from a pre-determined set of process plans, (ii) *assign a machine* to each process step in the plans, and (iii) *sequence* tasks assigned to each machine. The three atomic functional components described above (sequencing, assignment, process planning) are obvious from the decomposed sub-problems. While the choices may be selected by the same decision support and executed by the same actuator, this is not necessarily always true. For example, the decision-maker may make scheduling decisions, then sequence work into the factory, but later make dynamic resource assignment decisions to accommodate changing shop floor conditions.

This section models two common joint control decisions: scheduling and routing. Scheduling determines "Which task will be serviced next (or the order of available tasks), and which resource will service it?". It is modeled as joint control decision of *sequencing*

and *resource assignment*. Routing determines "what is the next required process step, and which resource will execute it?". It is modeled as joint control decision of *process planning* and *resource assignment*.

3.6.1 Which task next, and which resource will service it? (Scheduling)

As resources complete processing tasks, scheduling methods choose which tasks will be processed next given available resources and/or which available resource should be assigned to the next waiting task. Scheduling decisions combines sequencing (section 3.2) and resource assignment (section 3.3) Scheduling and its many variants have been studied extensively. However, one challenge that remains is linking decision-making to shop floor execution; that is, specifying how schedules actually get executed.

The **decision function** *Scheduling* is composed of the sequencing and resource assignment decision functions, denoted $Index \circ Assign$ (table 6). The decision function maps a set of tasks and resources to a task sequence and their resource assignments. While the decisions can be made in either order or concurrently, the operations are not necessarily commutative. The scheduling **decision expressions** capture the resource assignment and task sequencing formulations from [2], [30], and [15], respectively.

The **decision support interface** `Scheduling` is a subtype of both `Sequencing` and `Resource Assignment` decision support interfaces. The `Scheduling` interface inherits the *sequencing*() and *assignment*() operations from the atomic decision support interfaces. Inheritance is denoted by the $\wedge$ next to the operation. The scheduling operation outputs both a (sequencing) *index* as well as *resource assignment* choices. This approach has the benefit of solving both problems simultaneously, potentially achieving a better overall solution.

Scheduling is the joint decision between two control decisions. Likewise, it requires executing two separate behaviors: one to sort the tasks and retrieve the "next" one, and a second behavior to move the task to the resource or the resources to task. The **actuator function** `Execute Schedule` and **actuator** `Scheduler` model library components are modeled as both a composition and subtype of the atomic actuator functions and components. Composition is a whole-part relationship, shown in SysML by black diamond associations between blocks, with the whole on the black diamond end and the part on the other end. The generalization relationship denoted in SysML using a hollow-headed arrow directed from the more specialized class, the subtype, to the more general class, the supertype.

Subtyping requires the scheduling component to provide the interfaces of both atomic

components, with some reasonable expectation that the joint component can be substituted for the individual components. For example, the `Scheduler` may act as a `Queue` in some contexts and a `Resource Acquirer` in others. Composition implies that one way to achieve the required functionality and interface is construct the `Scheduler` out of `Queue` and `Resource Acquirer` components and coordinate their individual behaviors to achieved the required overall behavior.

The `Execute Schedule` activity combines the `Sequence` and `Resource Acquire` activities. Likewise, the `Scheduler` is composed of *Queue* and the *Resource Acquirer*. The `Scheduler` inherits the interface definition which includes the task and resources ports and the *sequence*() and *acquireResource*() operations (inheritance denoted by $\wedge$). These interfaces enable the `Scheduler` to continue playing both `Queue` and `Resource Acquirer` roles independently, when necessary. Effectively any component or set of components can be considered a `Scheduler` that can `Execute Schedules` as long as they implement these behaviors.

While the logical modeling appears to be a straight-forward composition of the two decision functions, there may be more subtle concerns about implementing the actuation. These concerns arise when designing how the actuators access the sequencing and assignment choices separately (as needed for actuation), repeatedly, and over time. Addressing these concerns is deferred to future work focused on implementation and deployment.

3.6.2 For a task: which process next and which resource will execute it? (Routing)

After completing a processing step, tasks have additional process steps and may need to flow to another resource for the required processing. Selecting the task's destination is a function of both what needs to be done next ("which process type?") and who is going to provide that service ("which resource?"). The resource is often another DELS. *Routing* decisions combine dynamic process planning (section 3.4) with resource assignment (section 3.3) [1, 5, 23]. In flexible manufacturing systems (FMS), process plan selection for a task is done in conjunction with assignment of operations and tools to each machine [3, 7, 11, 22].

The *Routing* **decision function** reflects the composition of two functions *Dynamic Process Planning* and *Assign* (table 7). This function maps a task's *Process Plan* and available *Resources* to a new or updated *Process Plan*' and resources to perform the next process step in *Process Plan*'.

The **decision expression** specifies two separate variables reflecting joint selection of process planning and resource assignment. Some formulation use only the $x^m_{l,j}$ variable and

assume that if $O_{l,j}$ is not assigned to machine, then it isn't selected at all. The formulation in [19] defines process plan selection variables explicitly. As mentioned previously, explicit formulations simplify the relationship between the decision support and the actuator function.

The **decision support interface** `Routing` is a subtype of both `Resource Assignment` and `Dynamic Process Planning` interfaces. It inherits the *assignment*() and *process-Planning*() operations and defines a *routing*() operation reflecting the functional composition of the two inherited operations.

Routing is the joint decision between two control decisions and requires executing two separate behaviors: one to update the task's process plan to the reflect the selections and one to acquire the required or assigned resources. The **actuator function** (*Route*) reflects this functional composition of the *Dynamic Process Planning* and *Resource Assignment* actuator functions. Composition here is denoted by the wedge/and ($\wedge$) operator.

The actuator function *Route*'s expected effect is that the actuator will direct a task towards the resource that will execute the next required process step. These two functions, process planning and resource assignment, are not necessarily executed in any particular order. For example, candidate DELS can be solicited to perform each potential process before resolving alternative paths in the process plan, or alternatives or "flexibilities" can be resolved and then seek suitable DELS. In practice (and analysis) there are several practical methods for the actuator function to determine which DELS (resource assignment). For example, this can be done using pre-determined static lookup tables, querying global data sources, or soliciting DELS via contract net.

The actuator function `Route` is executed by a `Router` **actuator** that outputs the task to a particular flow interface (*outTask*), connected to the selected target DELS. `Routers`, like the other abstract actuators, may be implemented with complex components and behaviors, e.g. which truck to place an order on, or which conveyor.

The `outDELSTask` interface block may be implemented very differently depending on the system: a single outgoing order rack from which MHS selects what to move or a dock with many dock doors from which the "switch" must select which dock door /truck to place the task. Though in some cases the "routing/switch" block isn't necessary, the DELS simply places the task on its interface (or an outgoing queue) and 'some how' the task is taken to its next destination.

Colloquially, routing seems to imply two things: the resource is an active resource (DELS or equipment) and the assignment actuator executes a *moveTaskToResource* behav-

ior. The *moveTaskToResource* can be passive (as is common in many simulation abstractions) or can be actively facilitated by material handling equipment.

Routing tasks from the system [the resource assignment component involves independent DELS (specialized resources)] complements admission control and might rely on technologies similar to those that bring tasks into the system. However, in material handling systems where an AGV (or non-automated worker) deliver the task, the routing behavior must first summon an AGV to the system. Then a robotic arm, or similar mechanism to the admission actuator, can place the task onto the AGV.

Table 6. 'Which task next, and which resource will service it?' (Scheduling)

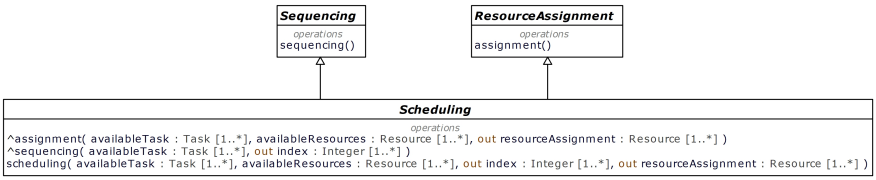
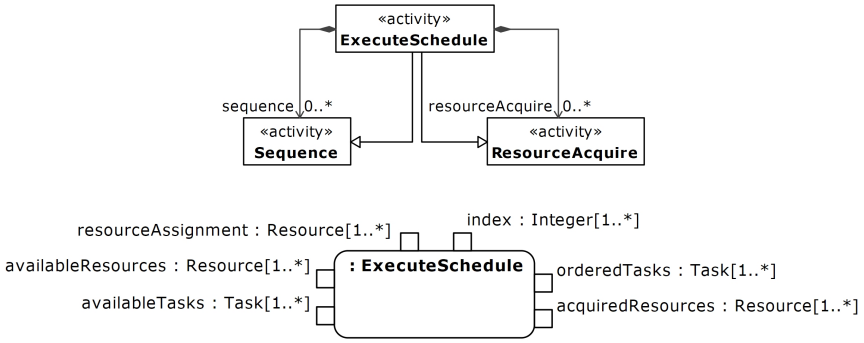
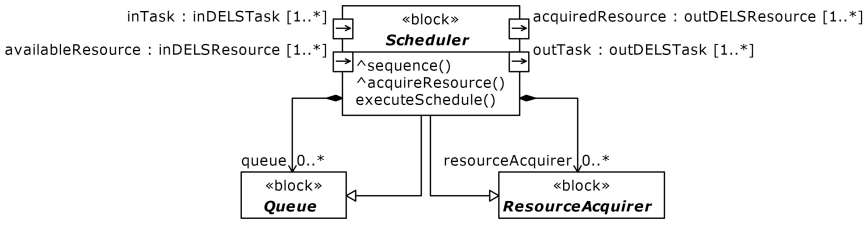
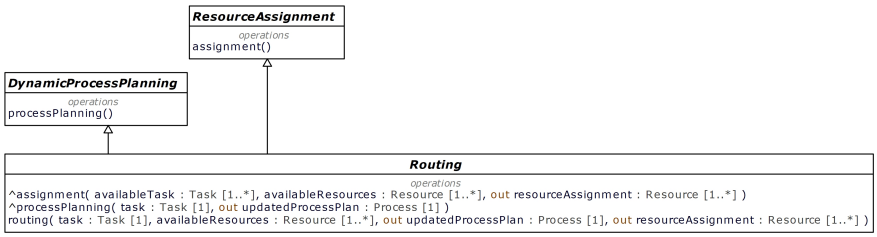
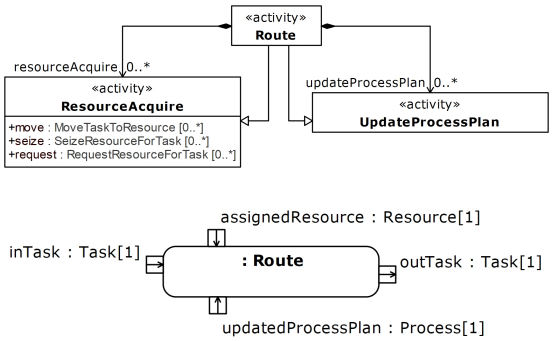
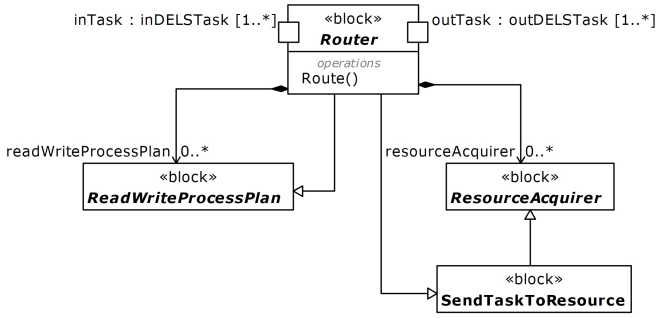
Decision Function	$Scheduling := Index \circ Assign$: Task(s) $\times$ Resource(s) $\mapsto$ $\mathbb{N} \times$ Resource(s)
Actuator Function	$ExecuteSchedule$(Task(s),Resource(s)) := $Sequence(Index$(Resource.availableTasks)) $\wedge$ $Acquire$(Task, Task.requiredInputResources)
Decision Expression	$y_{lt}^{m} = 1$, if task l is processed by machine m during period t [2] $y_{lj}^{m} = 1$, if task l is scheduled in the j^{th} position for processing on machine m [30] $y_{ll'}^{m} = 1$, if task l precedes task l' (not necessarily immediately) on machine m [15]
Decision Support Interface	
Actuator Function - Model Library Component	
Actuator - Model Library Component	

Table 7. For a task: 'which process is next and who will execute it?' (Routing)

Decision Function	$Routing := DynamicProcessPlanning \circ Assign$: ProcessPlan $\times$ Resource(s) $\mapsto$ ProcessPlan$'$ $\times$ Resource(s)
Actuator Function	$Route := Task.processPlan \leftarrow ProcessPlan' \wedge$ $Task.nextProcessStep.requiredInputResource \leftarrow Resource(s)$
Decision Expression	$x^m_{l,j} = 1$, if machine m is selected for process step j of task l $(O_{l,j})$ $z_{l,k} = 1$, if process step $O_{l,j} \in \mathscr{P}_l$ is selected next for task l
Decision Support Interface	
Actuator Function - System Model Library Component	
Actuator - System Model Library Component	

4. Operational Control Model Libraries for Modeling DELS

The model library components are used to develop control system specifications, which are part of the broader system models. To effectively integrate the models, this section briefly describes the link between the operational control model library and the DELS model library described in [29]. The DELS model library builds on product-process-resource (PPR) ontology.

The actuator functions and actuators are linked to the PPR models by specializing them from `Control Process` and `Active Resource`, respectively (figure 6). The operational control components are modeled as specialized process and resources to enable an integrated description of the base system. The goal is an integrated and consistent model of products flowing through processes executed by resources, including control processes and actuators controlling flows through the system.

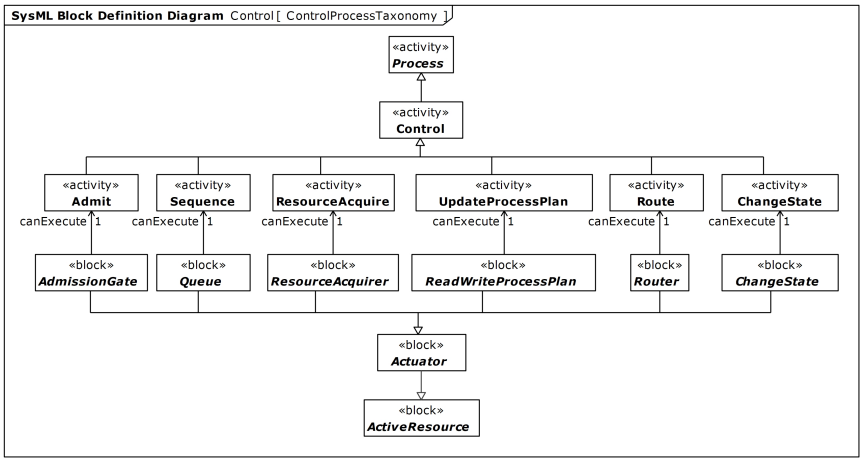


Fig. 6. The control processes and actuators are extended from the process and resource elements also used to model the system.

Model libraries often contain patterns for assembling the library components into system models. The control flow pattern in figure 7 illustrates one way that actuator components can be assembled into a logical model of a system's flow control. This pattern can be reused and extended in a few ways. First, the abstract actuator components described in this paper need to be implemented by selecting concrete system components that can execute the required actuation behaviors. For example, an automated storage and retrieval system can be modeled as a specialized store and queue executing not only storage and retrieval behavior but also sequencing behaviors. Second, the control behaviors are not required to

appear in this rigid sequence or even constrained to appear only once. For example, a system model may include additional control steps for handling flows of tasks that remain in the system but require different processing and resources. Finally, control actuators may be physically distributed across the system despite the logical model depicting them as "close" to one another.

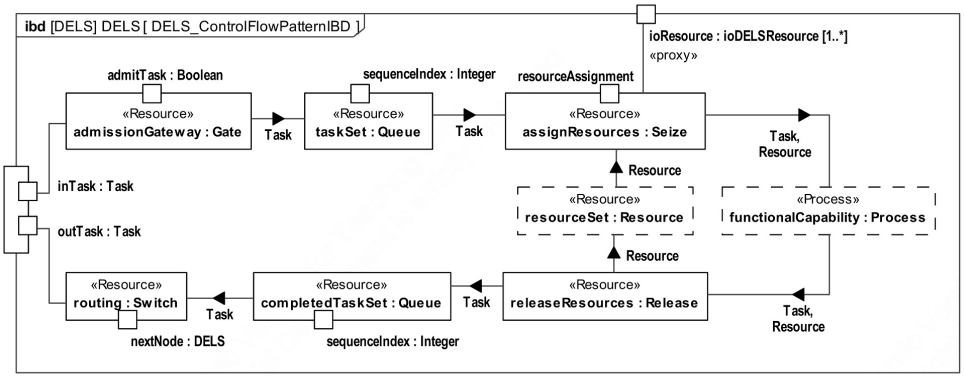


Fig. 7. This pattern demonstrates one way that the actuators (components of the system) can be configured to control the flow of tasks and resources through the system.

The decision support interface for each control function is integrated into the DELS controller's `Decision Support` component (figure 8). The decision support for a particular system is implemented by selecting the most appropriate analysis method(s) for each control function. The controller accesses these analysis methods through a consistent, standardized interface. For example when an admission decision is required, the controller can call *admissionInterface.admission()* without having to know the details of the admission algorithm.

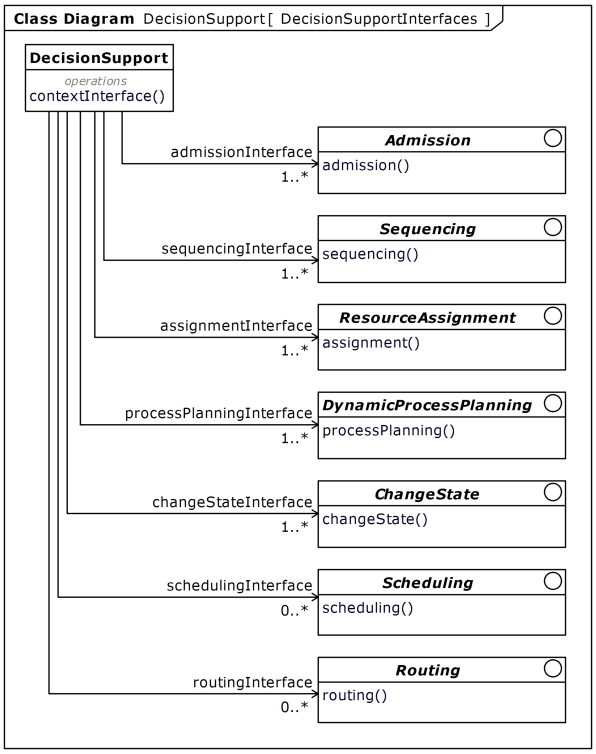


Fig. 8. The decision support component of the controller has access to methods for each control function, each supported by its own interface definition.

5. Conclusion

The model library described in this paper provides a foundation for improving operational control specifications of discrete event logistics systems. It is designed to explicitly link controller's decision support to system (plant) actuation. Logical models built up from these libraries provide consistent specification of operational control across functionally heterogeneous systems. Ultimately, the model library along with suitable design and analysis methods leveraging it enable more efficient and higher quality design and operation of integrated production and logistics systems.

Further work is required to demonstrate how to specialize model library elements to cover specific cases. This also includes further development of methods for elaborating logical model components into software and hardware components of robust cyber-physical production systems. Also, opportunities remain for additional formalization that could bridge current gaps between optimal control theory in modeling real-time systems and operational control models of shop floor and material flow. Effective formal verification

would support integrated design and testing from enterprise to equipment-level control. Finally, there is an opportunity to refine analysis methods and tools supporting control behaviors required by smart, automated production and logistics systems. That is, design and analysis tools need greater capabilities to provide required fidelity to design, test, and operate complex systems.

Acknowledgements

Commercial equipment and materials might be identified to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the U.S. National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

[1] Ümit Bilge, Murat Fırat, and Erinç Albey. A parametric fuzzy logic approach to dynamic part routing under full routing flexibility. *Computers & Industrial Engineering*, 55(1):15–33, 2008.

[2] Edward H Bowman. The schedule-sequencing problem. *Operations Research*, 7(5): 621–624, 1959.

[3] Paolo Brandimarte. Exploiting process plan flexibility in production scheduling: A multi-objective approach. *European Journal of Operational Research*, 114(1):59–71, 1999.

[4] Stefan Bussmann, Nicolas R Jennings, and Michael Wooldridge. *Multiagent systems for manufacturing control: a design methodology*. Springer Science & Business Media, 2013.

[5] Mike D Byrne and Parames Chutima. Real-time operational control of an fms with full routing flexibility. *International Journal of Production Economics*, 51(1-2):109–113, 1997.

[6] Christos G. Cassandras and Stephane Lafortune. *Introduction to discrete event systems*. Springer, 2008.

[7] FTS Chan, TC Wong, and LY Chan. Flexible job-shop scheduling problem under resource constraints. *International Journal of Production Research*, 44(11):2071–2089, 2006.

[8] Robert Cloutier, Gerrit Muller, Dinesh Verma, Roshanak Nilchiani, Eirik Hole, and Mary Bone. The concept of reference architectures. *Systems Engineering*, 13(1): 14–27, 2010.

[9] Jeff A Estefan. Survey of model-based systems engineering (mbse) methodologies. *Incose MBSE Focus Group*, 25:8, 2007.

[10] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2014.

[11] Mansour Abou Gamila and Saeid Motavalli. A modeling technique for loading and scheduling problems in fms. *Robotics and Computer-Integrated Manufacturing*, 19 (1):45–54, 2003.

[12] John S Gero and Udo Kannengiesser. The situated function–behaviour–structure framework. *Design studies*, 25(4):373–391, 2004.

[13] Mikhail Yu Kitaev and Vladimir V Rykov. *Controlled queueing systems*. CRC press, 1995.

[14] Jan Karel Lenstra and AHG Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

[15] Alan S Manne. On the job-shop scheduling problem. *Operations Research*, 8(2): 219–223, 1960.

[16] Yazid Mati and Xiaolan Xie. Multiresource shop scheduling with resource flexibility and blocking. *IEEE transactions on automation science and engineering*, 8(1):175–189, 2011.

[17] Lars Mönch, Peter Lendermann, Leon F McGinnis, and Arnd Schirrmann. A survey of challenges in modeling and decision-making for discrete event logistics systems. *Computers in Industry*, 62(6):557–567, 2011.

[18] OMG SysML 1.5. OMG Systems Modeling Language (OMG SysML) version 1.5. Standard, Object Management Group (OMG), 2017. URL http://www.omg.org/spec/SysML/1.5/.

[19] Cemal Özgüven, Lale Özbakır, and Yasemin Yavuz. Mathematical models for job-shop scheduling problems with routing and process plan flexibility. *Applied Mathematical Modelling*, 34(6):1539–1548, 2010.

[20] David L Poole and Alan K Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[21] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[22] SC Sarin and CS Chen. The machine loading and tool allocation problem in a flexible manufacturing system. *International Journal of Production Research*, 25(7):1081–1094, 1987.

[23] Nanua Singh and Bhaba Krishna Mohanty. A fuzzy approach to multi-objective routing problem with applications to process planning in manufacturing systems. *The International Journal of Production Research*, 29(6):1161–1170, 1991.

[24] Stephen F Smith and Marcel A Becker. An ontology for constructing scheduling systems. In *Working Notes of 1997 AAAI Symposium on Ontological Engineering*, pages 120–127, 1997.

[25] Timothy Sprock. *A Metamodel of Operational Control of Discrete Event Logistics Systems (DELS)*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 2015.

[26] Timothy Sprock. Patterns for modeling operational control of discrete event logistics systems (dels). In *Disciplinary Convergence in Systems Engineering Research*, pages 875–884. Springer, 2018.

[27] Timothy Sprock and Leon F McGinnis. A conceptual model for operational control in smart manufacturing systems. *IFAC-PapersOnLine*, 48(3):1865–1869, 2015.

[28] Timothy Sprock, Conrad Bock, and Leon F McGinnis. Survey and classification of operational control problems in discrete event logistics systems (dels). *International journal of production research*, 57(15-16):5215–5238, 2019.

[29] Timothy Sprock, George Thiers, Leon F. McGinnis, and Conrad Bock. Theory of Discrete Event Logistics Systems (DELS) Specification. NIST Interagency/Internal Report (NISTIR) 8262, National Institute of Standards and Technology, 2020. URL https://doi.org/10.6028/NIST.IR.8262.

[30] Harvey M Wagner. An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly*, 6(2):131–140, 1959.

[31] Abraham Wald. Basic ideas of a general theory of statistical decision rules. In *Proceedings of the International congress of Mathematicians*, volume 1, pages 308–325, 1950.

[32] Richard A Wysk and Jeffrey S Smith. A formal functional characterization of shop floor control. *Computers & Industrial Engineering*, 28(3):631–643, 1995.

[33] Nevin Lianwen Zhang and David L Poole. Stepwise-decomposable influence diagrams. In Bernhard Nebel and Charles Rich, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR '92)*, pages 141–152, 1992.