SDCWorks: A Formal Framework for Software Defined Control of Smart Manufacturing Systems

Matthew Potok*, Chien-Ying Chen†, Sayan Mitra* and Sibin Mohan†
*Dept. of Electrical and Computer Engineering, †Dept. of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{potok2,cchen140,mitras,sibin}@illinois.edu

***Abstract*—Discrete manufacturing systems are complex cyber-physical systems (CPS) and their availability, performance, and quality have a big impact on the economy. Smart manufacturing promises to improve these aspects. One key approach that is being pursued in this context is the creation of centralized *software-defined control (SDC)* architectures and strategies that use diverse sensors and data sources to make manufacturing more adaptive, resilient, and programmable. In this paper, we present SDCWorks—a modeling and simulation framework for SDC. It consists of the semantic structures for creating models, a baseline controller, and an open source implementation of a discrete event simulator for SDCWorks models. We provide the semantics of such a manufacturing system in terms of a discrete transition system which sets up the platform for future research in a new class of problems in formal verification, synthesis, and monitoring. We illustrate the expressive power of SDCWorks by modeling the realistic SMART manufacturing testbed of University of Michigan. We show how our open source SDCWorks simulator can be used to evaluate relevant metrics (throughput, latency, and load) for example manufacturing systems.**

I. Introduction

The manufacturing industry represents 12% of the US GDP [1]. One of the key performance indicators for a manufacturing system is the Overall Equipment Effectiveness (OEE), capturing the availability, performance and quality of the production system. Worldwide studies indicate that the average OEE in manufacturing plants is 60%, whereas world-class OEE is considered to be greater than 85% [2]. One of the important contributors to this low OEE is unscheduled downtime, caused by random faults, machine degradation, and increasingly, cyberattacks [3], [4], [5], [6], [7], [8].

One way to reduce this downtime and improve the overall effectiveness and management of manufacturing systems is to develop a *global view* of the system. Such a view will enable (i) early detection of disruptions, (ii) timely isolation of affected component(s) and even (iii) potential reprogramming of parts of (or even the whole) system. Importantly, this centralized ability to reprogram the system based on a global view also enables the system operator to quickly react to changing demand, allowing for new parts to be introduced in an existing manufacturing system, for improved profitability. We call our approach *software-defined control* (SDC)[1]. The key idea of SDC is to *separate the logical (control) plane of decision making from the management of the physical components and sensory information* for large-scale discrete manufacturing systems. All the decision-making logic is combined in a single, centralized *control plane*, called the "cyber-physical control plane" or *controller* for short in this paper. The controller has a global view of the entire system allowing it to orchestrate the routing of physical parts and to schedule operations on machines and conveyors on the factory floor. Although SDC is inspired by SDN, there are several major differences between the two approaches: physical components (akin to packets being routed) have different properties – (i) the queue sizes (buffers) are limited, (ii) we cannot just "drop" parts like we do with SDN packets and (iii) the time scales are much different – since manufacturing systems operate in the millisecond to second ranges. Therefore, we cannot directly model/translate SDN concepts to SDC.

Towards this vision, we present SDCWorks—a formal modeling framework that captures different state components of manufacturing systems, demarcated into *plants* and *controllers*. Plants describe the floor plan of the factory – essentially the various physical components that operate on the parts being manufactured (that we call "parts" or "widgets" without loss of generalization). Controllers are cyber components that orchestrate the operations of the plant(s) and the movement of parts[2]. While there exists significant work in the area of modeling manufacturing systems (*e.g.,* [10], [11], [12], [13], [14], [15], [16]), none of them capture the cyber and physical aspects of the system. Modern manufacturing systems have significant complexity due to the interplay between the software and hardware components. SDCWorks captures these aspects at a level of abstraction that makes it both expressive and useful for simulation and analysis. Additionally, with some extensions, SDCWorks can function as a digital twin [17] for a manufacturing system, simulating in parallel with an operational system to optimize performance and detect faults.

SDCWorks has the power to express a wide variety of *discrete* manufacturing systems. Models developed within the SDCWorks framework can reap many benefits of rigorous modeling: (i) computer-aided analysis and verification, (ii) security analysis (detection of threats, attacks, *etc.* and also developing countermeasures), (iii) ensuring that real-time

We thank Kira Barton, Felipe Lopez, James Moyne and Miguel Saez for many detailed discussions and providing valuable inputs on the SDC visions and manufacturing systems. This work was supported by NSF CPS Frontier grant NSF CPS 1544901.

[1]Inspired by the concepts from software-defined networking (SDN) [9].

[2]Section III-B provides more details about each of these.

timing requirements are met, (iv) synthesis of controller code for some (or all parts) of the plant and (v) monitoring parts through given plant layouts. This is akin to the results in the SDN field where the controller (with its global view of the system) can enable verification (*e.g.,* [18], [19]), consistency checks[20], [21], dependency checking [22], synthesizing network updates [23], system updates [24], [25] and security [26] to name just a few. We believe that rigorously developed control architectures and strategies can significantly reduce unscheduled downtime.

We use both a realistic testbed and a synthetic model for evaluation and analysis (Section VII). We are able to record the throughput, end-to-end time and load for varying product manufacturing requirements and plant topologies.

Hence, the high-level contributions of this paper are:

1) SDCWorks, a *formal modeling framework* that can capture the cyber and physical properties of discrete manufacturing systems. [Sections III-B]
2) An illustration of the expressive power of the framework. [Section V-B]
3) Implementation of a baseline controller to illustrate the capabilities of SDCWorks. [Section IV]
4) An open source simulator for SDCWorks models. [Section VI]

We first start with some background material and a description of the system model.

II. Background: Complexity of Modeling Modern Manufacturing

The physical part of a manufacturing system consists of machines and material handling devices (*e.g.,* robots and conveyors). Machines and robots typically have their own low-level controllers (these are denoted MC and RC in Figure 1). Raw or unfinished parts arrive at the system, are transported via material handling and exit the system in a finished state. The system also has multiple *Logic Controllers* (LCs), often implemented on PLC hardware. These LCs read data from sensors and send commands to actuators. LCs are typically local controllers that coordinate a physical region of a plant, called a *cell*. The LCs enable the production process by sending the right commands to the right actuators at the *right times*. Each LC is typically paired with a *Safety Controller* (SC), a special type of logic controller that, instead of enabling production, prohibits the manifestation of unsafe behavior in the system. This is primarily for humans interacting with the physical system. For instance, if an emergency-stop button is pressed then all of the machines, robots and conveyors within the sphere of influence should stop as quickly (and safely) as possible. In current practice, these LCs and SCs are individually programmed by control engineers, using templates and style guidelines. Though these controllers may operate for years or even decades due to exhaustive testing and experienced design, this "distributed" style of programming makes the process of managing and reconfiguring such systems *difficult, error prone and even insecure*. For the purposes of this paper, the MCs, RCs, LCs, SCs and the various machines are all part of the 'plant' while the 'controller' is as described in Section I – the centralized controller that oversees/manages the entire system.

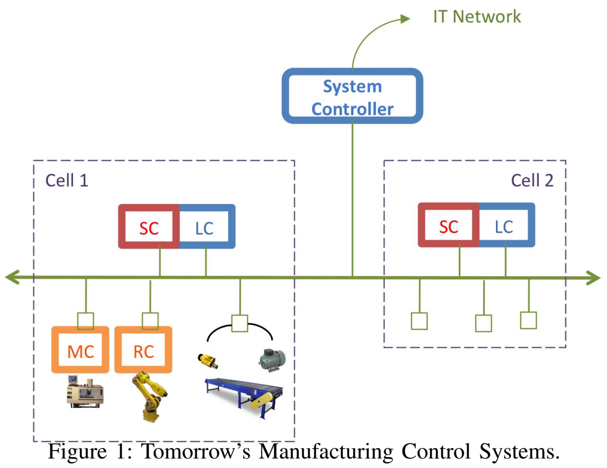


Figure 1: Tomorrow's Manufacturing Control Systems.

III. SDCWorks Modeling Framework

A. Semantic Underpinnings

We will use variables to model different state components of the plant and the controller. Each variable x has an associated type denoted by $type(x)$ that is the set of values that x can take. Let X be a set of variables. A *valuation* for X is a function that maps each variable $x \in X$ to a value in $type(x)$. The set of all possible valuations of X is denoted by $val(X)$. Given a valuation $\mathbf{x} \in val(X)$, its restriction to a set of variables, $S \subseteq X$, is denoted by $\mathbf{x}.S$.

The overall system with a plant and a controller will be modeled as a *discrete transition system*. Formally, a transition system $\mathcal{H}$ is a tuple $\langle X, \Theta, A, D\rangle$ where

(i) X is a finite set of variables partitioned into $X = X_C \cup X_P$, sets of controller and plant variables; the set $val(X)$ of valuations of X is called the set of *states*;
(ii) $\Theta \subseteq val(X)$ is a set of initial states;
(iii) A is a finite set of actions partitioned into $A = A_C \cup A_P$ disjoint sets of controller and plant actions; and
(iv) $D \subseteq val(X) \times A \times val(X)$ is the set of discrete state transitions. An individual transition $(\mathbf{x}, a, \mathbf{x}') \in D$ is written as $\mathbf{x} \xrightarrow{a} \mathbf{x}'$.

For a given state $\mathbf{x} \in val(X)$, $\mathbf{x}.X_P$ and $\mathbf{x}.X_C$ are said to be the plant and controller state at $x\mathbf{x}$. The plant variables (X_P) are read/write variables for the plant actions (A_P) and are only read by the controller actions (A_C). That is, for any $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ with $a \in A_P$, $\mathbf{x}.X_C = \mathbf{x}'.X_C$. Similarly, the controller variables (X_C) are read/write variables for the controller actions (A_C) and are only read by the plant actions (A_P).

An *execution* of $\mathcal{H}$ is an sequence of states and transitions with plant and controller transitions alternating: $\alpha = \mathbf{x_0}, a_1, \mathbf{x_1}, a_2, \mathbf{x_2} \cdots$, where $\mathbf{x_0} \in \Theta$ and $a_i \in A_C$ for odd i and $a_i \in A_P$ for even i in the sequence. A state $\mathbf{x}$ is *reachable* if there exists an execution that ends in $\mathbf{x}$. Any set $I \subseteq val(X)$ that contains all the reachable states of $\mathcal{H}$

is called an *invariant*. Invariants capture properties that must *always* hold for the system and can be stated in terms of plant and controller variables.

B. SDCWorks Modeling Overview

An SDCWorks model is specified by the following three components: (i) a plant that has a collection of cells where each cell can perform certain operations; (ii) a set of parts that move through the cells where each part is associated with a requirement that defines the sequence of operations that need to be performed on the part for it to be completed; and (iii) a controller that orchestrates the operations at cells and movement of parts. We formally describe each of these components and the transition system model defined by these components in the following sections.

C. Plant Description

The plant models the layout of cells and the connections between them, and specifies the subset of operations from the total set of operations, OP, that each cell can perform. There are two special operation corresponding to creation ($op_\top$) and removal ($op_\bot$) of parts. We define $\overline{OP} = OP \cup \{op_\top, op_\bot\}$.

Formally, a *plant* is specified by a tuple $P = \langle G_P, L_P, T_P, Q_P\rangle$, where

(i) $G_P = \langle V_P, E_P\rangle$ is a graph of cells, where V_P is the set of cells and E_P is the set of edges that defines the paths for moving parts
(ii) $L_P : V_P \mapsto 2^{\overline{OP}}$ maps each cell to the set of operations that can be performed at that cell
(iii) $T_P : V_P \times \overline{OP} \mapsto \mathbb{N}$ maps a cell $v \in V_P$ and an operation $op \in L(v)$ to $T_P(v, op)$, the total time (measured in number of transitions) required to complete op at v
(iv) $Q : V_P \mapsto \mathbb{N}$ maps each cell to its queue length $Q(v)$, the maximum number of parts that can be queued at v

For any graph and G_P in particular, for any vertex $v \in V_P$ $next(v)$ and $prev(v)$ denote the set of predecessors and successors of v in G_P.

Cells: A cell is any physical component of a plant where some operation can be performed on a part such as machines, conveyors, and etc. A cell can either be an individual component or a set of multiple components.

Sources and Sinks: A *source* is a cell $v \in V_P$ such that $op_\top \in L_P(v)$ and $prev(v) = \emptyset$. A *sink* is a cell v with $op_\bot \in L(v)$ and $next(v) = \emptyset$. For simplicity, in this paper we assume that there is a single source ($v_\top$) and a single sink ($v_\bot$) in any plant P and that $L(v_\top) = \{op_\top\}$ and $L(v_\bot) = \{op_\bot\}$. A cell in V that is neither a source nor a sink is an *ordinary* cell.

D. Parts and Requirements

For modeling convenience, we assume that there exists a universal set W of unique identifiers for *all* parts that will ever be seen by the manufacturing system. Let $W_\top = \{w \in W | loc(w) = v_\top\}$ denote the set of parts that have not been created yet, $W_\bot = \{w \in W | loc(w) = v_\bot\}$ denote the set of parts that have been completed, and $W_\circ = W \setminus (W_\top \cup W_\bot)$ denote the set of parts that are currently in the system. For all the results presented in the paper, a sliding window of unique identifiers will be adequate. A possible implementation of this is to attach a unique RFID tag on each part [27].

A *requirement* models a sequence of operations in OP that need to be performed on each part for the part to be considered completed. Formally, a requirement specifies a directed acyclic graph (DAG) $R = \langle V_R, E_R\rangle$ and a labeling function $L_R : V_R \to \mathcal{P}(\overline{OP})$. For each requirement, there is a single source, a vertex $v \in V_R$ with no incoming edges and only one operation, $op_\top$, and at least one sink, a vertex $v \in V_R$ with no outgoing edges and only one operation, $op_\bot$. A *path* of R is a sequence of vertices in V_R, $\pi = v_0, \dots, v_k$ such that v_0 is the source, v_k is a sink, and $(v_i, v_{i+1}) \in E_R$ for each i in the sequence. Given a path π, we define $L_R(\pi) = L_R(v_0), \dots, L_R(v_k)$ which is the corresponding sequence of operations in $\overline{OP}$.

Example 1. *Consider a manufacturing system with cells $V_P = \{v_1, v_2, v_3\} \cup \{v_\top, v_\bot\}$. The plant is a graph $G_P = (V_P, E_P)$ as shown in the representation below. The circles (machines) and squares (conveyors) represent individual cells. For notational purposes, conveyors are shown as squares through which the edges pass; however in the actual graph, there are separate edges that lead into and from the conveyors rather than just pass through them. This notation is used in the remainder of the paper.*

The operations that are supported by a cell v can be obtained by $L_P(v)$, e.g., $L_P(v_3) = \{op_2, op_3\}$. Let's further consider two part types being fabricated in this plant. Assuming requirement 1 requires to complete op_1 and op_2 and requirement 2 requires to complete op_1 and op_3 in the given order, then the requirements are denoted by $R_1 = \{op_1 \to op_2\}$ and $R_2 = \{op_1 \to op_3\}$, $\mathcal{R} = \{R_1, R_2\}$.

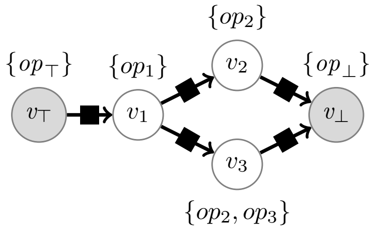


Figure 2: The plant graph of a simple example.

E. Concepts related to plants and requirements

Each $queue(v)$ has the following operations: $head$, returns the element at the front of the queue; len, provides the current length of the queue up to a maximum specified by $Q(v)$; and pop, removes and returns the element at the head of the queue and decrements the len by one. Additionally, we define the following notation: $bag(v) := \{w \mid loc(w) = v\}$; $Bag := \cup_{v \in V} bag(v)$.

F. Discrete Transition System

We now describe the nominal plant and an abstract controller. Additional variables may be added to this model, for example, to implement specific controllers strategies and to track more complex performance metrics.

Plant Variables: Figure 3 gives the names and types of the plant variables (X_P). The variable loc returns the cell that a part is currently located in. Initially, all parts are located at the source, $v_\top$. Upon consumption at a sink, a part's location is set to $v_\bot$. The plant variable pos assigns, for each part, $w \in W_\circ$ a natural number. If $loc(w) \in V$ then $pos \leq Q(loc(w))$ and it is the actual position of w in the queue of $loc(w)$ (otherwise, $w \notin W_\circ$ and $pos(w)$ is meaningless). The variable $queue$ is the queue (array) of parts at that cell. As specified by the plant P, for any cell $v \in V_P$, the length of $queue(v)$ is upper bounded by $Q(v)$. Initially, the queue is empty (*i.e.,* all entries are set to $\bot$). The variable $requirement$ selects and assigns a requirement $R \in \mathcal{R}$ to a part. The selection of R will determine the set of operations that are performed on the part. Finally, $part\_time$ tracks the total time a part spent in the plant starting from $v_\top$ and ending at $v_\bot$. All parts have their $part\_time$ initialized to 0.

```
Plant Variables:
    $loc : W \mapsto V_P \cup \{v_\top, v_\bot\}$
    $pos : W \mapsto \mathbb{N}$
    $queue : V_P \mapsto \mathsf{Array}[W \cup \bot]$
    $requirement : R \mapsto W$
    $part\_time : W \mapsto \mathbb{N}$, init $\forall w \in W part\_time(w) := 0$
```

Figure 3: Plant (X_P) variables and types.

Abstract Controller Variables: Figure 4 shows the list of abstract controller variables: The variable $action$ encodes the actuation decision made by the controller for each cell v for the following plant transition in which the cell v uses the decision to perform some operation. The variable $next\_tr$ specifies for each cell v a neighboring cell to which v should transfer a part when it receives a $transfer$ action controller. The $timer$ keeps track of the current time of the overall system. A concrete controller uses these and additional variables to make decisions and keep track of the overall system state.

```
Abstract Controller Variables:
    $action : V_P \mapsto \overline{OP}$
    $next\_tr : V_P \mapsto V_P$
    $timer : \mathbb{N}$, init $timer := 0$
```

Figure 4: Abstract controller variables.

Plant Transitions: During each plant transition, all cells are updated based on the decisions made by the controller. These decisions are reflected in the controller variables such as $action$ that are read by the plant. The plant assigns one of the following actions to each cell: $move$ moves parts on conveyors, $op \in OP$ perform operation op at cells, $op_\top$ creates new parts at sources, $op_\bot$ removes parts at sinks and $transfer$ moves parts between adjacent connected cells in the plant. We now discuss these transitions in more detail.

If a conveyor cell v is provided with a $move$ action, v decrements the position of each of the parts in $queue(v)$.

If a cell is given the OP action, the cell starts to perform op on the part at the head of its queue. While this presumably changes the local physical state of the plant and the part, in our model, this action does not change any plant variables. Instead, we will see later that once op completes then the controller will update the record for the part.

If a source cell v is given the $op_\top$ action, it selects a part from $w' \in W_\top = bag(v_\top)$, the set of parts at the source, $v_\top$, and assigns it a requirement R. This nondeterministic choice models the uncertainty in the type of requirement that is demanded from the next part. The location of the new part w' is set to be $v_\top$ and its position is set to the end of its queue.

If a sink cell is given the $op_\bot$ action, the cell will remove the part from the head of its queue and the removed part's location is set to $v_\bot$.

If a cell is given a $noop$ action, the cell does not change any of the variables.

If a cell is given a $transfer$ action, the cell removes the part at the head of its queue and reads the $next\_tr$ variable to determine the next cell to transfer the removed part. The cell transfers the part to the next cell by changing the location of the part to that of the next cell and the position to end of the queue of the next cell.

```
Plant Transitions:
  for each $v \in V_P$
    if $action(v) = move$ then
      for each $w \in bag(v)$
        $pos(w) := pos(w) - 1$

    if $action(v) = op \in OP$ then
      $do(op)$

    if $action(v) = op_\top$ then
      $w' :=$ choose $bag(v_\top)$
      $requirement(w') :=$ choose $R \in \mathcal{R}$
      $loc(w') := v$
      $pos(w') := len(queue(v))$
      $part\_time(w') := timer$

    if $action(v) = op_\bot$ then
      $w' := pop(queue(v))$
      $loc(w') := v_\bot$
      $part\_time(w') := timer - part\_time(w')$

    if $action(v) = noop$ then
      pass

    if $action(v) = transfer$ then
      $w' := pop(queue(v))$
      $loc(w') := next\_tr(v)$
      $pos(w') := len(queue(next\_tr(v)))$
```

Figure 5: Plant (X_P) transitions.

IV. Baseline Controller

In this section, we present a basic centralized controller strategy which is a refinement of the abstract controller of Figure 4. Controllers make the following decisions: (a) plan a sequence of operations according to the requirements of a part, (b) map these operations on to appropriate cells, and (c) orchestrate the movement and operation of the parts and cells. In presenting the controller, we first introduce the formal notion of a *plan* that encapsulates the first two pieces.

A. Plans and Feasible Graphs

Recall, given a requirement R and a plant P, $G_P = \langle V_P, E_P \rangle$ is the graph of cells for the plant and $G_R = \langle V_R, E_R \rangle$ is the graph of the requirement. A *plan* C is essentially a forward simulation relation from V_R to V_P [28]. That is, it is a relation $C \subseteq V_R \times V_P$ such that for any $(v_{1R}, v_{1P}) \in C$ and $(v_{1R}, v_{2R}) \in E_R$, there is *a path* $\pi = v_{1P}, \ldots, v_{2P}$ in the plant graph, such that (a) $(v_{2R}, v_{2P}) \in C$, and (b) $L_R(v_{2R}) \in L_P(v_{2P})$. In other words, a plan relates vertices in V_R to vertices in V_P so that each edge (v_{1R}, v_{2R}) in the requirement graph can be simulated by paths in G_P and the first and the last vertices in the path are at cells that can perform the operations required at (v_{1R}, v_{2R}).

The above definition of plan allows each operation in the requirement R to be accomplished by traversing an arbitrary but finite path (i.e., sequence of cells) in G_P. A path corresponding to a single edge in E_R may revisit the same cell in G_P multiple times. We can show this using a standard simulation argument [28]. Therefore, a plan gives a set of feasible paths for achieving the requirement R in the plant P.

Proposition 1. *There exists a plan C for a requirement R and plant P if and only if for every path of G_R there exists a corresponding path in G_P.*

In order to obtain performance guarantees, concrete controllers will restrict the family of paths. For example, for every edge in $(v_{1R}, v_{2R}) \in E_R$, the corresponding path $\pi = v_{1P}, \ldots, v_{kP}$ in G_R may be required to satisfy one of these conditions:

- *k-cell repetitions*: each cell $v \in G_P$ may repeat in π at most k times. For $k = 1$ this implies that a cell may be visited at most once.
- *k-op misses*: the number of cells v such that $L_R(v_{2R}) \in L_P(v)$ is in the path is at most k. For $k = 1$ this implies that there is only one cell v_{2P} in π with $L_R(v_{2R}) \in L_P(v_{2P})$ where the operation $L_R(v_{2R})$ is performed.

Given a C and path π in G_R, the corresponding set of paths in G_P constructed from C are called the *feasible paths* of π. The set of all the feasible paths corresponding to R in G_P is called the *feasible graph* and is denoted by $F_{R,P}$. We now proceed to describe our baseline controller that constructs and uses a particular class of plans represented as feasible graphs.

B. Implementation of Baseline Controller

Controller Variables: Figure 6 gives the names and types of our controller variables (X_p). Any variables that overlap with the abstract controller variables take on the same definition as before unless explicitly stated otherwise.

The controller variable *completed* keeps track of the sequence of operations that have been performed on each part. Initially, it is the empty sequence for every part. The *pointer* variable of a part is a pointer to a vertex in the feasible graph $F_{R,P}$ corresponding to the same requirement as that of the part. This variable keeps track of which operation should currently be performed on a part at a particular cell and which path through the plant to take. The variable *start_time* marks the start of a new operation by a cell and tracks how long a particular operation has been ongoing. The *wait_time* variable monitors the amount of time a cell has been waiting for the *transfer* action to move a part from the head of its queue.

A cell's *status* denotes one of three current conditions of a cell: *idle*, the cell is not working on any part; *operational*, the cell is performing some $op \in OP$ on a part at the head of the cell's queue; and *waiting*, the cell has completed $op \in OP$ and is waiting for a *transfer* action to move the part at the head of its queue further along in the plant. The *cost* variable maps a cell's state to a real number which can be used to choose between feasible paths in order to optimize with respect to a cost metric (for example, energy, reliability, etc.).

```
**Controller Variables**:
  $action : V_P \mapsto \overline{OP} \cup \{move, noop\}$
  $can\_enqueue : V_P \mapsto \{\mathsf{T}, \mathsf{F}\}$
  $completed : W \mapsto OP^*$
  $cost : V_P \mapsto \mathbb{R}$
  $next\_tr : V_P \mapsto V_P$
  $pointer : V_P \mapsto V_{F_{R,P}}$
  $start\_time : V_P \mapsto \mathbb{N}$, **init** $\forall v \in V\, start\_time(v) := 0$
  $status : V_P \mapsto \{idle, operational, waiting\}$
  $timer : \mathbb{N}$, **init** $timer := 0$
  $wait\_time : V_P \mapsto \mathbb{N}$
```

Figure 6: Controller (X_C) variables and types.

Controller Transitions: During each controller transition, the controller first increments *timer* by 1 and performs the following three major steps: (i) determines the shortest path in each feasible graph $F_{R,P}$ according to the chosen cost metric; (ii) updates each of the *next_tr* and *action* variables for each cell; and (iii) possibly updates the *cost* of each cell in the plant. We discuss these steps in some more detail.

From the statically computed feasible graphs $F_{R,P}$, the controller runs a shortest path algorithm utilizing the recently set costs to determine the best paths through the plant. The controller iterates through each $v \in V$ and first sets the *next_tr* variable to the the cell with the lowest cost from the set $next(v)$. Next, the controller determines the action that the cell should take depending on its type. If the $celltype(v)$ is a conveyor cell, a *move* action is done unless there is a part at the head of the queue; otherwise, the cell increments its *wait_time* by 1. If the type is a source cell, the controller checks whether a part already exists in the cell's queue. If so, then the controller tries to transfer the part to the next cell; otherwise, it instantiates a new part. If the type is an ordinary cell, the controller cycles between the *idle*, *operational*, and *waiting* states to determine the correct action. If a cell's status is *idle*, the controller tries to transfer the part immediately in the case of a *noop* operation, performs an *op* action if there is a part at the head of the queue, or does nothing. If the cell's status is *operational*, the controller assigns the action *op* to the cell until the *op* has been completed at which point the controller sets the cell's status to waiting. If the cell's status is $waiting$, the controller tries to transfer the part. If the transfer is successful, the controller changes the cells's status to $idle$ and the cycle repeats. Additionally, regardless of state, if a cell has space on its queue, it will try to enqueue a part from an input conveyor with the largest $wait\_time$. If the type is a sink cell, the controller instructs the cell to either terminate a part at the head of the queue or do nothing.

C. Rigorous Reasoning about Properties

Our SDCWorks framework is designed to support rigorous analysis of correctness and performance properties of the system. The formal framework will support standard reasoning in terms of invariants, abstractions, substitutivity and will be eventually supported by computer-aided verification tools [29]. While a full verification of our baseline controller is reserved for future papers, we present several key assertions and arguments to sketch the outline of a correctness argument. Additionally, the baseline controller we present here uses a subset of the global view available to it and makes decisions based on a local level. Much more complex controllers can be implemented with SDCWorks such as those presented in literature [30].

The following invariant captures the mutual exclusion property and ensures that there is no "pile-up" of parts on cell and conveyors.

Proposition 2. *For any ordinary cell $v \in V_P \setminus \{v_\top, v_\bot\}$, and any two distinct parts $w_1, w_2 \in W$, if $w_1, w_2 \in bag(v)$ then $pos(w_1) \neq pos(w_2)$.*

This invariant is proved by induction on the length of any execution of the $\mathcal{H}$. The base case (an execution of length 0) holds vacuously, because at any initial state of $\mathcal{H}$, all the parts are at the source $v_\top$. For the inductive step, we check for each possible plant and controller transition to verify that the invariant is preserved. This involves a case analysis across all the **if** conditions in Figure 5 and Figure 7. The $can\_enqueue$ variable restricts the $transfer$ action from occurring unless there is an empty space at the end of the queue for any cell.

The following invariant captures the correctness property that ensures that only complete parts appear at the sink.

Proposition 3. *For any sink $v_\bot$, and any part $w \in bag(v_\bot)$, the part is completed, i.e., $completed(v)$ is a path in $requirement(w)$.*

This invariant is derived from a stronger invariant that captures the relationship between the $completed(v)$ variable and the position of the part and the feasible graph. Roughly, a part w with requirement R follows a a feasible path from the feasible graph $F_{R,P}$. From Proposition 1 it follows that, as w traverses the plant from a source $v_\top$ to a sink $v_\bot$ as specified by the feasible path, it must follows a path in G_R; therefore, when it appears at $v_\bot$ the part is complete.

The next proposition captures the property that all parts complete within bounded time for any plant that is a directed acyclic graph (DAG).

```
Controller Transitions:
  $timer = timer + 1$
  for each $v \in V_P$
    $cost(v) := userDefinedCost(v)$

  for each $F_r \in \mathcal{F}$
    ShortestPathFirst($F_r$)

  for each $v \in V_P$
    $next\_tr(pointer(v)) := \underset{v' \in next(pointer(v))}{\operatorname{argmin}} cost(v')$

    if $celltype(v) = conveyor$ then
      if $\exists w \in head(queue(v))$ then
        $wait\_time(v) = wait\_time(v) + 1$
      else
        $action(v) = move$

    if $celltype(v) = source$ then
      if $\exists w \in head(queue(v))$ then
        $try\_transfer(v)$
      else
        $action(v) := op_\top$

    if $celltype(v) = cell$ then
      if $status(v) = idle$ then
        if $\exists w \in head(queue(v))$ then
          if $op(pointer(v)) = noop$ then
            $try\_transfer(v)$
          else
            $start\_time(v) := timer$
            $status(v) := operational$
            $action(v) := \oplus(pointer(v))$
        else
          $action(v) := noop$
      if $status(v) = operational$ then
        if $timer - start\_time(v) >= T(v, op(pointer(v))$
          $completed := completed + op(pointer(v))$
          $status(v) := waiting$
        else
          $action(v) := op(pointer(v))$
      if $status(v) = waiting$ then
        $try\_transfer(v)$
        if $\exists w \notin head(queue(v))$ then
          $status(v) := idle$
      if $can\_enqueue(v)$ then
        $v' := \max wait\_time(prev(v))$
        $action(v') := transfer$

    if $celltype(v) = sink$ then
      if $\exists w \in head(queue(v))$ then
        $action(v) := op_\bot$
      else
        $action(v) := noop$

    if $len(queue(v)) < Q(v)$ then
      $can\_enqueue(v) = \mathsf{T}$
    else
      $can\_enqueue(v) = \mathsf{F}$

  func $try\_transfer(v)$
    if $can\_enqueue(next\_tr(v))$ then
      $action(v) := transfer$
      $next\_tr(v) = next\_tr(pointer(v))$
    else
      $action(v) := noop$
```

Figure 7: Controller (X_C) transitions.

Proposition 4. *For any acyclic requirement and any part $w \in V_o$, there exists a $k > 0$ such that if $part\_time(w) \geq k$ then $w \in bag(v_\bot)$.*

Since G_P has finite depth in this case, the property is established by showing that w traverses an edge from $loc(w)$ in the plant graph G_P in finite time. For plants with cycles, this property may not hold with our baseline controller as parts may deadlock.

V. Modeling a Realistic Manufacturing Testbed

To demonstrate the expressive power of our modeling framework we present a formal model of the System-level Manufacturing and Automation Research Testbed (SMART), a realistic testbed at the University of Michigan.

A. SMART System Overview

SMART is a hybrid serial-parallel line manufacturing testbed at University of Michigan [31] (see Figure 8) with three manufacturing blocks and two conveyor lines connected with a controllable pneumatic diverter. In total, there are three industrial robots and four CNC milling machines spread across the three manufacturing blocks.

Blocks 1 and 2 each contain two CNC machines and a six degree of freedom robotic arm. Block 3 contains the third robotic arm in the system that is used for additive manufacturing. Each of the CNCs are controlled by a PLC and are programmed to perform operations based on the inputs from central controller (the baseline controller in our model). The robotic arms are programmed to transfer parts between the conveyor and the CNCs one at a time. Blocks 1 and 2 are linked with a single conveyor line. The conveyor from Block 2 branches out into two conveyors that lead to either Block 1 or 3. There is a diverter at the conveyor fork that determines which branching conveyor a part should take.

The SMART system has various sensors for gathering plant and part state information: (i) a camera system to capture images of the parts, (ii) RFID transceivers and seven proximity sensors to locate parts, (iii) energy consumption monitors for the conveyors, robotic arms, and one of the CNCs, (iv) state monitors for the robotics arms and CNCs to capture position, velocity, load, and etc. data. Each part is labeled with an RFID tag with a unique ID. The re-write capabilites of the RFID tags allow information to be stored in the tags such as the $completed$ variable.

B. Modeling SMART System

With minor adjustments, the SMART system can be modeled within the SDCWorks framework (refer to Figure 10). We consider Block 3 as the entry and exit point for parts (there are two bins, one with the raw materials and another for completed parts). The robotic arm in Block 3 can transfer raw material from the raw materials bin onto the conveyor. This action corresponds to the creation of a new part at the source node. When a completed part reaches Block 3, the robotic arm transfers the part from the conveyor into the completed bin. This action corresponds to the consumption of a completed part at a sink node. To fit our model constraints, Block 3 is split into two distinct cells, a source cell $v_\top$ and a sink cell $v_\perp$. Each of the CNC machines are modeled as individual cells with their operations listed in Table IV. The robotic arms in Blocks 1 and 2, and the diverter are modeled as cells with only a $noop$ operation

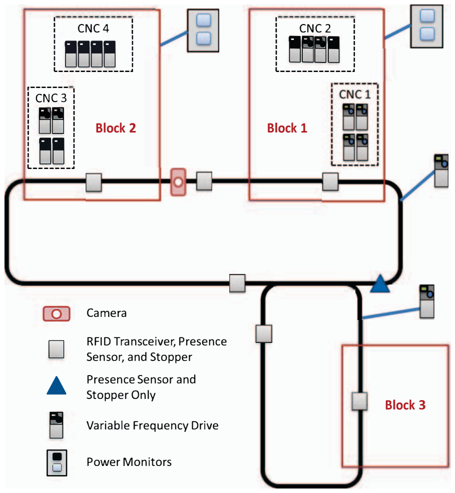


Figure 8: Layout of the SMART testbed.

Figure 9: Machine cells and conveyors.

Table I: Time T_P for each CNC machine to complete operations.

	CNC 1 (v_4)	CNC 2 (v_5)	CNC 3 (v_6)	CNC 4 (v_7)
op_1	20	25	35	–
op_2	50	30	–	10
op_3	25	–	15	30
op_4	–	10	25	30

* all values are scaled to transition time ticks

* "–" indicates that the operation is not supported on this machine

Table II: Operation requirements. Three requirements are considered in the case study in SMART.

	Operations
R_1	$op_\top \to op_1 \to op_2 \to op_3 \to op_4 \to op_\perp$
R_2	$op_\top \to op_1 \to op_3 \to op_1 \to op_\perp$
R_3	$op_\top \to op_2 \to (op_3\ ,\ op_4) \to op_1 \to op_\perp$

We consider three types of parts that can be manufactured

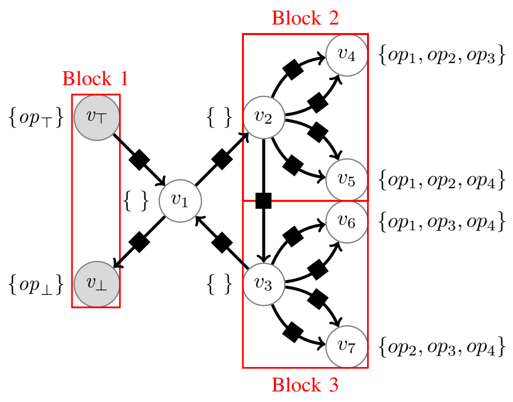


Figure 10: The graph representation of SMART for the formal model.

in the SMART system. The requirements for each part type are given in Table II. The symbol "–" indicates that the operation is not supported by the corresponding CNC machine in that column. All operations specified in the table must be completed in the order they shown.

Requirement 1 (R_1) requires a sequence of four unique operations. Requirement 2 (R_2) only requires two unique operations but demands that operation op_1 is performed twice. Requirement 3 (R_3) requires three operations but allows for the second operation be either op_3 or op_4.

VI. SDCWorks Simulator

We have developed a flexible, open source, discrete event simulator capable of simulating arbitrary SDCWorks models. The simulator is developed in Python3 and uses the Graphviz and Matplotlib libraries to visualize all outputs. The simulator is available for download at https://github.com/SDC-UIUC/synthesis.

The SDCWorks simulator takes as input plant and requirement files specified in the YAML format. The plant input file specifies all the cells, the operations each cell supports, and the time it takes each operation to complete for a particular cell. The requirement input file specifies all the requirements to run against the plant. Each requirement contains a list of nodes with a single operation and a list of edges to link the nodes. Examples of both types of input files can be found in the link above.

During an execution, the simulator first parses the user input and creates graphs of the input plant and requirement graphs in the DOT language. Each graph is written to a PNG file to allow users to visually verify that the simulator constructed the plant and requirement graphs correctly. Next, the simulator executes the system for a specified amount of time using the baseline controller (others can be coded). To make the system run deterministically, the sources assign requirements to parts in a round-robin fashion whenever parts are instantiated. At every time step, the simulator moves parts through the plant, updates the states of each cell and logs various metrics: throughput, end-to-end delay, and the number of live parts in the plant at a given time. Finally, the simulator outputs a log file with the state of every cell in the system and all live parts at every time step and plots all the metrics listed above.

VII. Case Study

In this section we use a synthetic linear model and its variant as examples to demonstrate the use of the SDCWorks modeling framework. We also carry out an analysis using the simulator described in Section VI.

A. A Synthetic Linear Model

Let's consider a simple linear manufacturing system consisting of five cells, $V = \{v_1, v_2, v_3, v_4, v_5\}$. This plant contains a fork that provides multiple path options for dynamic allocation and support for fabricating multiple types of parts. The plant is represented as a graph (Figure 11). The supported operations and the time for each cell to complete designated operations are listed below:

Table III: Time T_P for each cell to complete operations.

	v_1	v_2	v_3	v_4	v_5
op_1	10	–	–	–	–
op_2	–	20	–	–	–
op_3	–	–	40	35	–
op_4	–	–	–	50	–
op_5	–	–	–	–	15

* all values are scaled to transition time ticks

* "–" indicates that the operation is not supported on this machine

Let's consider three types of parts to be fabricated in this plant, each with its own corresponding requirement denoted by $\mathcal{R} = \{R_1, R_2, R_3\}$. These requirements are listed in Table V.

Requirement 1 is a typical linear manufacturing process that requires the parts to visit all the cells in one of the paths in the plant. Requirement 2 shows a different set of operations requiring some cells to be bypassed. Requirement 3 has an option for the system to choose a path that satisfies one of the operation requirements (*i.e.,* either op_4 or op_5 at the second stage). These types of requirements are not uncommon since different models of machines can perform the same work on the parts albeit with different tools.

The above plant and requirements are converted into YAML files as input for the simulator. During simulation, one part is placed at the entrance (*i.e.,* $v_\top$) at every other tick, as introduced in Section VI. Each part appears at the entrance and

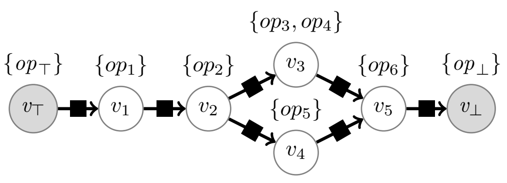
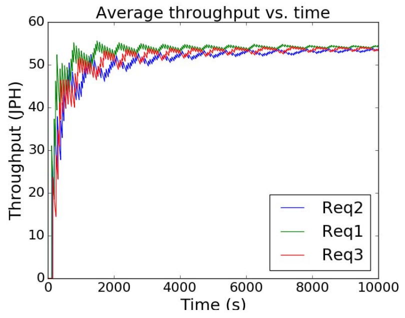
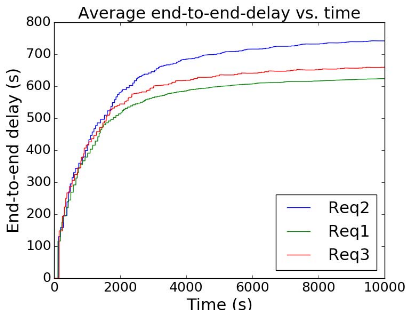


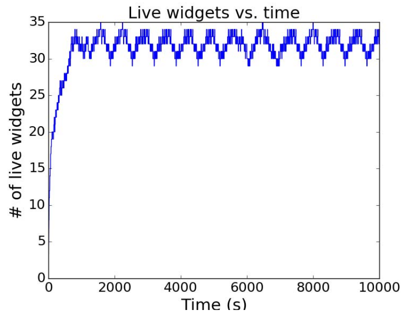
Figure 11: The plant graph of the linear model case.



(a) Throughput of each part (JPH).

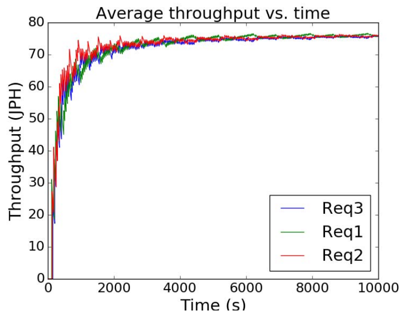


(b) End-to-end time of each part.

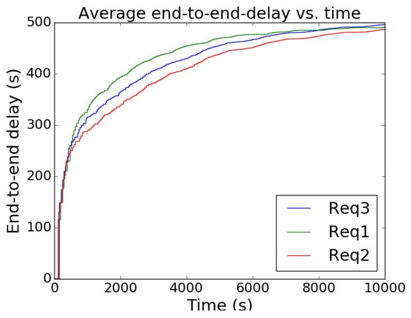


(c) Total alive parts in the plant.

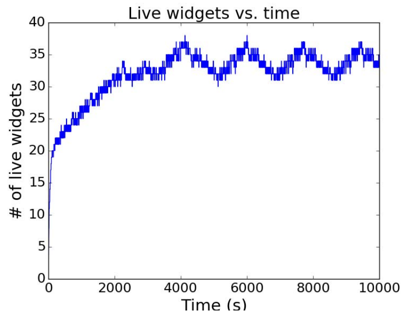
Figure 12: Experiment results for the plant shown in Figure 11.



(a) Throughput of each part (JPH).



(b) End-to-end time of each part.



(c) Total alive parts in the plant.

Figure 13: Experiment results for the plant that has an additional cell, v_6, as shown in Figure 14.

Table IV: Time T_P for each CNC machine to complete operations.

	CNC 1 (v_4)	CNC 2 (v_5)	CNC 3 (v_6)	CNC 4 (v_7)
op_1	20	25	35	–
op_2	50	30	–	10
op_3	25	–	15	30
op_4	–	10	25	30

* all values are scaled to transition time ticks

* "–" indicates that the operation is not supported on this machine

Table V: Operational requirements for the linear model.

	Operations
R_1	$op_\top \rightarrow op_1 \rightarrow op_2 \rightarrow op_3 \rightarrow op_6 \rightarrow op_\bot$
R_2	$op_\top \rightarrow op_2 \rightarrow op_5 \rightarrow op_\bot$
R_3	$op_\top \rightarrow op_1 \rightarrow (op_4 \text{ or } op_5) \rightarrow op_6 \rightarrow op_\bot$

is assigned a requirement from $\{R_1, R_2, R_3\}$ in a round-robin fashion. We run the simulation for 10000 ticks (we assume that each tick represents one second in this experiment). The simulation results are shown in Figure 12.

The results suggest that the throughput stabilizes after 5000 ticks. The throughput for all three requirements converges to around $50JPH$ to $60JPH$ (JPH stands for jobs-per-hour, a common metric in manufacturing systems). Although the arrival rate of the parts can potentially be higher (because one part is introduced into the system at every tick, the maximum arrival rate is $3600/hour$ for three requirements and $1200/hour$ for each requirement), the actual throughput is constrained by the bottleneck at v_4 with op_5 that takes 50s to complete the operation. Hence, the throughput upper bound on this node is $3600/50 = 72JPH$. As it is a mandatory path for R_2, the throughput of the nodes and edges on the path for R_2 before this v_4 is also limited by its throughput $72JPH$. Note that the actual throughput for R_2 is lower than the upper bound throughput of v_4, as shown in Figure 12(a), because v_4 is shared between R_2 and R_3.

B. A Variant Model

From the previous example, we observed that the throughput is constrained by v_4. A reasonable method to increase the throughput is to add another cell that supports the same operation, op_5. To alleviate the bottleneck at v_4, a new cell v_6 is added with the same capabilities as v_4. The updated plant is shown as a graph in Figures 14.

We keep the set of requirements and other configurations the same as the previous case for comparison. The plant with v_6 is then fed into the simulator for experimentation. The results are shown in Figure 13.

From the simulation, we can observe that the throughput for all three requirements increases since the original bottleneck is removed after adding additional supports for op_5. As a result, the end-to-end time for R_2 (*i.e.,* the requirement that requires op_5) decreases significantly. Also, the total alive parts slightly grows in this case because the capacity (*i.e.,* additional space for parts to sit in) of the plant increased with the addition

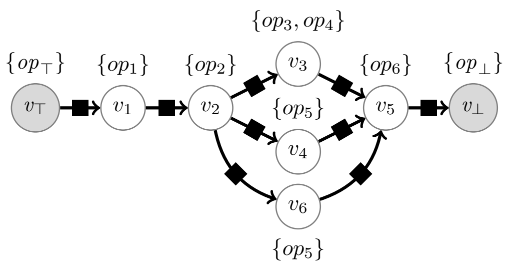


Figure 14: The graph for the plant introduced in Section VII-A with an additional cell, v_6, added.

of another cell. It's worth noting that v_3 becomes the new bottleneck in the modified setup as its support for op_3 has the highest time cost, which causes the congestion on the path containing it.

VIII. Conclusion

Future manufacturing systems will be composed of complex software and manufacturing plants that work in close coordination with each other. They will have to deal with failure, changing requirements, security threats, *etc*. Hence, a system that has a *global* view of the overall system will be invaluable for managing such systems. SDCWorks provides a mechanism for modeling and analysis of such systems. We envision that it will engender a new set of research problems in synthesis, verification, monitoring, fault-tolerance targeting manufacturing and more general software-defined control systems.

References

[1] The Executive Office of the President, "Making in America: U.S. manufacturing entrepreneurship and innovation," White House, Tech. Rep., June 2014.

[2] M. Ahmad and N. Dhafr, "Establishing and improving manufacturing performance measures," *Robotics and Computer-Integrated Manufacturing*, vol. 18, no. 3–4, pp. 171 – 176, 2002, 11th International Conference on Flexible Automation and Intelligent Manufacturing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0736584502000078

[3] National Cybersecurity and Communications Integration Center, "Ics-cert year in review," Department of Homeland Security, https://ics-cert.us-cert.gov/, Tech. Rep., 2014.

[4] T. McDermott and A. Wolfson, "Manufacturing – a persistent and prime cyber attack target," Cohn Reznick, http://www.cohnreznick.com/manufacturing-persistent-and-prime-cyber-attack-target, Tech. Rep., September 26 2014.

[5] K. Zetter, "A cyberattack has caused confirmed physical damage for the second time ever," *Wired*, January 2015.

[6] K. Hon, "Performance and evaluation of manufacturing systems," *CIRP Annals-Manufacturing Technology*, vol. 54, no. 2, pp. 139–154, 2005.

[7] P. Jonsson and M. Lesshammar, "Evaluation and improvement of manufacturing performance measurement systems-the role of oee," *International Journal of Operations & Production Management*, vol. 19, no. 1, pp. 55–78, 1999.

[8] M. M. Ahmad and N. Dhafr, "Establishing and improving manufacturing performance measures," *Robotics and Computer-Integrated Manufacturing*, vol. 18, no. 3, pp. 171–176, 2002.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[10] A. A. Desrochers and R. Y. Al-Jaar, *Applications of Petri nets in manufacturing systems: modeling, control, and performance analysis*. IEEE, 1995.

[11] R. G. Askin and C. R. Standridge, *Modeling and analysis of manufacturing systems*. John Wiley & Sons Inc, 1993.

[12] C. G. Cassandras, *Discrete event systems: modeling and performance analysis*. CRC, 1993.

[13] M. Zhou and K. Venkatesh, *Modeling, simulation, and control of flexible manufacturing systems: a Petri net approach*. World Scientific, 1999, vol. 6.

[14] S. Jain, N. Fong Choong, K. Maung Aye, and M. Luo, "Virtual factory: an integrated approach to manufacturing systems modeling," *International Journal of Operations & Production Management*, vol. 21, no. 5/6, pp. 594–608, 2001.

[15] J. Ezpeleta, J. M. Colom, and J. Martinez, "A petri net based deadlock prevention policy for flexible manufacturing systems," *IEEE transactions on robotics and automation*, vol. 11, no. 2, pp. 173–184, 1995.

[16] A. Negahban and J. S. Smith, "Simulation for manufacturing system design and operation: Literature review and analysis," *Journal of Manufacturing Systems*, vol. 33, no. 2, pp. 241–261, 2014.

[17] M. Grieves and J. Vickers, "Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems," in *Transdisciplinary Perspectives on Complex Systems*. Springer, 2017, pp. 85–113.

[18] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 323–334.

[19] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[20] W. Liu, R. Bobba, S. Mohan, and R. Campbell, "Inter-flow consistency: Novel sdn update abstraction for supporting inter-flow constraint," in *NDSS Workshop on Security of Emerging Networking Technologies (SENT)*, 2015.

[21] W. Liu, R. B. Bobba, S. Mohan, and R. H. Campbell, "Inter-flow consistency: A novel sdn update abstraction for supporting inter-flow constraints," in *2015 IEEE Conference on Communications and Network Security (CNS)*, Sept 2015, pp. 469–478.

[22] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 20.

[23] A. Noyes, T. War, P. Cerny, and N. Foster, "Toward synthesis of network updates." in *Proceedings of Workshop on Synthesis (SYNT)*, July 2013.

[24] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 49–54.

[25] S. Ghorbani and M. Caesar, "Walk the line: consistent network updates with bandwidth guarantees," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 67–72.

[26] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A survey of security in software defined networks," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 623–654, Firstquarter 2016.

[27] G. Q. Huang, Y. Zhang, and P. Jiang, "Rfid-based wireless manufacturing for real-time management of job shop wip inventories," *The International Journal of Advanced Manufacturing Technology*, vol. 36, no. 7, pp. 752–764, 2008.

[28] N. A. Lynch and F. W. Vaandrager, "Forward and backward simulations – part I: untimed systems." *Information and Computation*, vol. 121, no. 2, pp. 214–233, September 1995. [Online]. Available: citeseer.nj.nec.com/lynch95forward.html

[29] N. Lynch and M. Tuttle, "An introduction to Input/Output automata," *CWI-Quarterly*, vol. 2, no. 3, pp. 219–246, September 1989.

[30] T. T. Johnson and S. Mitra, "Safe and stabilizing distributed multi-path cellular flows," *Elsevier*, February 2015.

[31] I. Kovalenko, M. Saez, K. Barton, and D. Tilbury, "Smart: A system-level manufacturing and automation research testbed," vol. 1, p. 20170006, 10 2017.