Exploiting Specification Modularity to Prune the Optimization-Space of Manufacturing Systems

João Bastos
Eindhoven University of Technology
Eindhoven, The Netherlands
j.p.nogueira.bastos@tue.nl

Sander Stuijk
Eindhoven University of Technology
Eindhoven, The Netherlands
s.stuijk@tue.nl

Jeroen Voeten
Eindhoven University of Technology
Embedded Systems Innovation
Eindhoven, The Netherlands
j.p.m.voeten@tue.nl

Ramon Schiffelers
Eindhoven University of Technology
Eindhoven, The Netherlands
ASML
Veldhoven, The Netherlands
ramon.schiffelers@asml.com

Henk Corporaal
Eindhoven University of Technology
Eindhoven, The Netherlands
h.corporaal@tue.nl

ABSTRACT
In this paper we address the makespan optimization of industrial-sized manufacturing systems. We introduce a framework which specifies functional system requirements in a compositional way and automatically computes makespan optimal solutions respecting these requirements. We show the optimization problem to be NP-Hard. To scale towards systems of industrial complexity, we propose a novel approach based on a subclass of compositional requirements which we call constraints. We prove that these constraints always prune the worst-case optimization-space thereby increasing the odds of finding an optimal solution (with respect to the additional constraints). We demonstrate the applicability of the framework on an industrial-sized manufacturing system.

CCS CONCEPTS
• Theory of computation → Formalisms; • Computing methodologies → Model development and analysis; • Software and its engineering → System modeling languages; Specification languages; • Design languages; • Computer systems organization → Embedded and cyber-physical systems;

KEYWORDS
Manufacturing systems, Design exploration, Modular system specification, Formal models, Makespan optimization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SCOPES '18, May 28–30, 2018, Sankt Goar, Germany.
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5780-7/18/05... $15.00
https://doi.org/10.1145/3207719.3207728

1 INTRODUCTION
Manufacturing systems are Cyber-Physical Systems which perform operations on batches of products. Production and transport units (the physical part) carry out operations, while a controller (the cyber part) coordinates the order and timing of execution. Examples of these systems include lithography machines or industrial production printers. During their design, functional requirements (such as keeping the order of products in a batch or avoiding collisions) must be satisfied while at the same time demanding performance requirements must be met, for instance regarding productivity.

In [2, 14] a framework is proposed for the specification of both functional and timing requirements for such manufacturing systems. Compared to other frameworks, its novelty lies in the separation of concerns between specification and optimization, together with its compositional support for requirements specification. However, no complexity study is provided with respect to the optimization problem. In this paper we restrict this framework of [14] to batch-oriented manufacturing systems and prove that their optimization is NP-Hard. As a consequence, optimal solutions might take prohibitively long depending on the size of the optimization-space induced by the optimization problem.

When considering real industrial applications, the size of the optimization-space cannot be disregarded [15]. To still support optimization, heuristic approaches can be applied, however, often at the cost of sub-optimal solutions and uncertainty regarding their quality. In this work we propose a novel approach where we further exploit the modularity of the framework as an alternative to heuristic solutions. We introduce classes of compositional requirements, which we call constraints, and show that they effectively prune the (worst-case) optimization-space. Furthermore, this approach still guarantees that optimal solutions can be found when the additional constraints are taken into account.

The approach is inspired by common practices in an industrial setting, where manufacturing systems are typically over-spe
[11] and in which over-specification is used implicitly and unconsciously to deal with complexity. Examples of over-specification that we have encountered in industrial cases are for instance: disallowing multiple mapping possibilities for an operation or enforcing the static ordering of system operations.

Our approach allows system designers to deal consciously with over-specification by an explicit formalization in terms of constraints. This constraint-oriented approach provides control over the worst-case size of the optimization-space of the system. Furthermore, it is guaranteed that functionally correct controllers are obtained.

**Contribution:** Our contributions are summarized below:
- We refine the framework of [14] to deal with batch-oriented systems and exploit the concept of constraints as a formal means to deal with optimization complexity and prove that these constraints effectively prune the (worst-case) optimization space.
- We show that the framework scales to an industrial-sized manufacturing system.

**Overview:** The remainder of this paper is organized as follows. Section 2 summarizes the framework which we refine. The batch optimization problem is explained in Section 3 and proven to be NP-Hard in Section 4. Section 5 elaborates on the modular specification of batch-oriented logistics. Section 6 shows that constraining effectively prunes the (worst-case) optimization-space. Section 7 exemplifies the approach with a real industrial case-study. Section 8 discusses related work and Section 9 concludes.

## 2 SPECIFICATION FRAMEWORK

In this paper, we assume as a starting point the framework of [14]. In this section, we summarize the building blocks of this framework and its essential properties.

A manufacturing system is decomposed into a set of peripherals \( \mathcal{P} \), actions \( \mathcal{A} \) and resources \( \mathcal{R} \). Each peripheral can execute actions. An action describes an atomic behavior of the system, e.g. the movement of a motor or the actuation of an on/off peripheral such as a clamp. The complete set of actions describes all behavior that the system can exhibit. Peripherals are aggregated into resources, which can be claimed and released. Figure 1 a) depicts a synthetic example of a system decomposed into resources R1, R2 and R3, peripherals p1, p2, p3, p4 and p5 and actions x1, x2, x3, x4, x5, x6 ands x7. Using peripheral actions, deterministic functional behaviors of the system can be constructed as Activities. An activity is a Directed Acyclic Graph (DAG), consisting of a set \( N \) of nodes and a set \( \rightarrow \) of dependencies between nodes. Nodes refer to either an action executed by a peripheral (associated with a pair \((x, p) : x \in \mathcal{A} \) and \( p \in \mathcal{P} \)), or a claim \((c)\) or release \((r)\) of a resource (associated with a pair \((r, v) : r \in \mathcal{R} \) and \( v \in \{c, r\}\)). Figure 1 b) depicts two such activities, a and b.

Multiple activities can be sequenced to form a new activity. Resource sharing and concurrency between multiple activities is taken into account by the correct claiming and releasing of resources between activities. Figure 1 c) depicts the result of sequencing activities a and b. This can be formally achieved by applying the semi-colon (:) operator [14], yielding a;b. Intuitively, the releasing and claiming of shared resources must be correctly matched and replaced by a new dependency such that the resulting DAG is itself an activity.

Besides functional specification, the framework requires a temporal specification for performance optimization. A function \( T : \mathcal{A} \rightarrow \mathbb{R}_{\geq 0} \) maps each action to its fixed execution time. Consequently, given an activity, the execution times can be lifted to the level of nodes, where the execution times of claim and releases nodes are assumed to be 0. In Figures 1 b) and c) the timing information is annotated within the nodes (except for the release and claim nodes). Since actions are executed on resources, a *resource time stamp* vector \( \gamma_R : \mathcal{R} \rightarrow \mathbb{R}^{\rightarrow} \) represents the system state in terms of resource availability. For each \( r \in \mathcal{R} \), \( \gamma_R(r) \in \mathbb{R}^{\rightarrow} \) is the availability time of resource \( r \). We use \( 0_R \) to denote the resource time-stamp with all zero-value entries.

Starting from resource vector \( \gamma_R \) the execution of an activity \( a \) leaves the system in a new state \( \gamma'_R \). This new state is determined by computing the completion times of each node in the activity, taken into account the dependencies and execution times of its nodes. For this purpose, \((max, +)\) algebra [1] is used. In [14] it is shown that \( \gamma'_R = M_a \otimes \gamma_R \), where \( M_a \) is the \((max, +)\) matrix of activity \( a \) and where \( \otimes \) is the \((max, +)\) matrix multiplication operator. The new system state \( \gamma'_R \) determines the makespan of the activity execution, represented by mks(\(a, \gamma_R\)). The makespan is given by the largest value in \( \gamma'_R \) which is by definition the \((max, +)\) norm \( \| \gamma'_R \| \). Notice that the makespan of the execution of a sequence \( a_1 a_2 \cdots a_n \) of activities (starting from \( \gamma_R \)) is given by \( M_{a_1} \otimes M_{a_2} \otimes \cdots \otimes M_{a_n} \otimes \gamma_R \) (where \( \otimes \) denotes the activity sequencing operator). In [14] it is shown that this expression is equivalent to \( M_{a_1} \otimes M_{a_2} \otimes \cdots \otimes M_{a_n} \otimes \gamma_R \).

---

**Figure 1:** Overview of the specification framework [14].
3 BATCH MAKESPAN OPTIMIZATION

Having described the basic framework for the specification of manufacturing systems, we can now introduce the batch optimization problem. For this purpose, we start by defining the necessary concepts for the specification of the logistics of batch-oriented manufacturing systems. A sequence of activities can model the complete manufacturing of a product, or batch of products, where a single activity models one manufacturing operation. The collection of all allowed activity sequences of a system defines the logistics behavior. Such a collection is encoded in a logistics automaton.

Definition 3.1 (Logistics automaton). A logistics automaton is a tuple \((\mathcal{S}, \mathcal{A}, \rightarrow, \mathcal{S}_0)\), where \(\mathcal{S}\) is a finite (possibly empty) set of states, \(\mathcal{A}\) is a finite (possibly empty) set of activities, \(\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}\) is a transition relation, and \(\mathcal{S}_0 \subseteq \mathcal{S}\) is a set of initial states, where \(\mathcal{S}_0 = \emptyset\) if \(\mathcal{S} = \emptyset\) and \(\mathcal{S}_0 = \{\mathcal{S}_0\}\) otherwise. Let \(s \xrightarrow{a} s'\) be a shorthand for \((s, a, s') \in \rightarrow\). The following additional properties must hold:

- Acyclicity: there exists no \(s \in \mathcal{S}\) such that \(s \xrightarrow{+} s\), where \(\xrightarrow{+}\) is the transitive closure of \(\rightarrow\), and where \(s \xrightarrow{t} t\) denotes that \(s \xrightarrow{a} t\) for some \(a \in \mathcal{A}\).
- Reachability: if \(\mathcal{S} \neq \emptyset\) then for all \(s \in \mathcal{S}, s \xrightarrow{s} s\), where \(\xrightarrow{s}\) is the reflexive transitive closure of \(\xrightarrow{\cdot}\).

Definition 3.2 (Language of a logistics automaton). Let \(L = (\mathcal{S}, \mathcal{A}, \rightarrow, \mathcal{S}_0)\) be a logistics automaton. The language \(L(L)\) of \(L\) is defined by

\[
L(L) = \begin{cases} 
\emptyset, & \text{if } \mathcal{S}_0 = \emptyset \\
\{a \in \mathcal{A}^* \mid s_0 \xrightarrow{a} s \text{ for some } s \in \mathcal{S} \text{ and } s \neq s'\}, & \text{if } \mathcal{S}_0 = \{s_0\}
\end{cases}
\]

Here \(\mathcal{A}^*\) denotes the collection of all sequences of activities in \(\mathcal{A}\). Each \(a \in \mathcal{A}^*\) is of the form \(a_1 \ldots a_n\), where \(a_i \in \mathcal{A}\) (1 ≤ i ≤ n). For \(n = 0\), \(a\) is the empty activity sequence denoted by \(e\). For states \(s, s' \in \mathcal{S}\) and \(a = a_1, \ldots, a_n \in \mathcal{A}^*\) we let \(s \xrightarrow{a} s'\) denote the existence of \(s_1, \ldots, s_n \in \mathcal{S}\) such that \(s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n = s'\).

Further \(s \xrightarrow{a} s'\) denotes that \(s \xrightarrow{s} s'\) for some \(s \in \mathcal{S}\), \(s \xrightarrow{a} s'\) denotes that \(s \xrightarrow{s} s'\) for some \(s \in \mathcal{S}\) and \(s \xrightarrow{a} s'\) denotes that \(s \xrightarrow{s} s'\) does not hold. Note that \(L(L) = \emptyset\) if \(\mathcal{S} = \emptyset\) and \(L(L) = \{e\}\) if \(\mathcal{S} = \{s_0\}\). Notice also that any sequence in the language should “run to completion”, i.e. finish in a state with no outgoing transitions. As a consequence languages of logistics automata are not prefix closed (except when the language equals \(\{e\}\)).

Figure 1 d) depicts an example logistics automaton \(L_{ex}\). Two activities, \(a\) and \(b\) are used to specify different allowed activity sequences. The activities are annotated on the edges. Each activity sequence is determined by a path starting from the initial state (depicted with an extra circumference and no incoming edges) and finishing on a final state (a state without outgoing edges). In this way the automaton encodes a set of activity sequences, where each sequence represents one possible way to successfully manufacture a product, or batch of products. Hence the logistics automaton encodes a language of allowed activity sequences. For example, \(L_{ex}\) encodes activity sequences \(ab\) and \(ba\) and thus \(L(L_{ex}) = \{ab, ba\}\). Each activity sequence can result in different manufacturing completion times when taking into account the temporal execution of its activities. For example, the completion times of sequences \(ab\) and \(ba\) are 6 and 5.5 respectively. These completion times are obtained as the makespan of the corresponding activity sequences \((mks(ab, 0_R)\) and \(mks(ba, 0_R)\)). Thus the Batch Makespan Optimization (BMO) involves determining the activity sequence in the language of the logistics automaton with minimal makespan.

Problem 1 (Batch Makespan Optimization). Given a Logistics automaton \(L\) determine \(a \in L(L)\) such that

\[mks(a, 0_R) \leq mks(a', 0_R)\]

for all \(a' \in L(L)\). Here we let \(mks(a, 0_R)\) denote \(mks(a_1; \ldots; a_n, 0_R)\) for brevity.

This problem can be solved in two steps. The first is to build a timed expansion of logistics automaton \(L\) using the \((\max, +)\) characterization of its activities as a new automaton which we call the \((\max, +)\) automaton denoted by MaxPlus\((L)\). It can be shown that MaxPlus\((L)\) satisfies Definition 3.1 and is thus a logistics automaton itself.

Definition 3.3 ((\(\max, +\)) automaton). Let \(L = (\mathcal{S}, \mathcal{A}, \rightarrow, \mathcal{S}_0)\) be a logistics automaton. First define MaxPlus\((\mathcal{S}, \mathcal{A}, \rightarrow, \mathcal{S}_0)\) as the smallest set \(\mathcal{V}\) satisfying inference rules (1) and (2):

\[
\begin{align*}
\mathcal{S}_0 &= \{s_0\} \\
(s_0, 0_R) &\in \mathcal{V} \\
(s, \gamma_R) &\in \mathcal{V} \quad \text{if } (s', \gamma_R \otimes M_a) \in \mathcal{V}
\end{align*}
\]

Here \(\gamma_R\) denotes a resource time-stamp vector and \(0_R\) denotes the resource time-stamp vector containing only 0 valued entries. \(M_a\) denotes the \((\max, +)\) matrix corresponding to activity \(a \in \mathcal{A}\) and \(s, s' \in \mathcal{S}\). Then we define MaxPlus\((L)\) as

\[
(\text{MaxPlusStates}(L), \mathcal{A}, \rightarrow', \mathcal{S}_0')
\]

where \(\rightarrow' = \{(s, \gamma_R), a, (s', \gamma_R') \in \text{MaxPlusStates}(L) \times \mathcal{A} \times \text{MaxPlusStates}(L) \mid s \xrightarrow{a} s' \text{ and } \gamma_R' = \gamma_R \otimes M_a\} \) and \(\mathcal{S}_0' = \{s_0, 0_R\}\) otherwise.

As an example consider the \((\max, +)\) automaton of logistics automaton \(L_{ex}\), MaxPlus\((L_{ex})\), depicted in Figure 1 d). The temporal execution of activity sequence \(ab\) and \(ba\) leads to different resource time-stamp vectors and thus the final state of \(L_{ex}\) is duplicated in MaxPlus\((L_{ex})\). Note that the state-space induced by MaxPlus\((L)\) corresponds to the optimization-space of the BMO problem. Therefore, the second step is to explore MaxPlus\((L)\) to find a \(a \in L(L)\) for which the makespan is minimal. This is achieved by finding a final state for which the \((\max, +)\) norm of the resource time-stamp vector is minimal. An activity sequence \(a\) terminating in such a state is a solution to the BMO problem.

4 COMPLEXITY ANALYSIS

To show that BMO is NP-Hard, we first show that its decision version, the Batch Makespan Satisfaction (BMS) problem (whether an activity sequence exists with makespan below a given bound), is NP-Complete. It is well-known that the Weighted Set Partitioning problem (WSP) problem [?] is NP-Complete [6]. Here a set of \(n\)
elements with integer valued sizes, for which the total size is equal to \( S \), can be partitioned into two subsets of equal size \( S/2 \). WSP can be reduced to our BMS problem. We define two resources \( R_1 \) and \( R_2 \) (each with one peripheral \( p_1 \) and \( p_2 \)). For each combination of element and resource we defined an activity, \( a_{x_i}^{e_r} : (R_1, e_r) \rightarrow (x_i, p_1) \rightarrow (R_1, r_l) \) and \( a_{x_j}^{e_r} : (R_2, e_r) \rightarrow (x_j, p_2) \rightarrow (R_2, r_l) \), where \( x_i \) refers to element \( i \) (with \( 1 \leq i \leq n \)) and where the execution time of \( x_i \) corresponds to the size of the \( i \)th element. We further define a logistics automaton with states \( S \), labeled \( s_0, ..., s_n \). The transitions are defined as: \( s_i \xrightarrow{a^{e_r}} s_{i+1} \) for \( 0 \leq i \leq n \) such that all combinations of possible partitions are included in the automaton as activity sequences. Therefore, a partitioning is possible iff an activity sequence exists with a makespan equal to \( S/2 \). Thus, we have reduced a known NP-Complete problem to the BMS problem. Further, it is not possible to see that a solution can be verified in polynomial time. Hence, the BMS problem is NP-Complete. The BMO problem is NP-Hard, since the BMS problem can be solved in polynomial time using a solution of the BMO problem. For assume we have an optimal solution \( q_o \) for the BMO problem. If \( mks(q_o) \) is less than a given bound \( B \), a solution with a lower makespan than \( B \) exists and therefore the answer to BMS problem is positive. Otherwise, no solution exists with makespan lower than \( B \), yielding a negative answer to the BMS problem. Thus, BMO is NP-Hard.

5 MODULAR LOGISTICS SPECIFICATION

Even though a logistics automaton is able to encode the complete manufacturing of a batch of products, for large batch sizes or complex manufacturing jobs a monolithic automaton is not desired. A modular and compositional methodology is more suitable to deal with specification complexity, maintainability and understandability. Therefore, in this work we restrict our framework [14] towards batch-oriented systems and take inspiration from the constraint-oriented specification style of the LOTOS framework [3] and the compositional specification of requirements in CIP [13] by defining a composition operator on logistics automata. The operator allows batches to be specified individually (i.e. each product flow can be specified by an individual automaton) and then composed to obtain an automata that encodes all the logistics requirements for the batch of products.

**Definition 5.1 (Composition of Logistics Automata).** Let \( L_1 = \langle S_1, \text{Act}_1, \rightarrow_1, S_0 \rangle \) and \( L_2 = \langle S_2, \text{Act}_2, \rightarrow_2, S_0 \rangle \) be logistics automata. Before we define the composition automaton \( L_1 \odot L_2 \), we first define relation \( \rightarrow \subseteq S_1 \times (\text{Act}_1 \cup \text{Act}_2) \times S_2 \) as the smallest set \( V \) satisfying the following inference rules:

\[
\begin{align*}
\frac{s \xrightarrow{a} s'}{s, t \xrightarrow{a} (s', t)} \\
\frac{s \xrightarrow{a} s'}{s, t \xrightarrow{a} (s', t)}
\end{align*}
\]

\[
\begin{align*}
\frac{t \xrightarrow{a} t'}{s, t \xrightarrow{a} (s', t')} \\
\frac{t \xrightarrow{a} t'}{s, t \xrightarrow{a} (s', t')}
\end{align*}
\]

where \( s, s' \in S_1 \) and \( t, t' \in S_2 \). Now define the set of states \( S \) of the composition automaton as

\[
\begin{align*}
S &= \begin{cases} \emptyset, & \text{if } S_1 = \emptyset \text{ or } S_2 = \emptyset \\
\{(s, t) \in S_1 \times S_2 \mid (s_0, s_0) \xrightarrow{a^*} (s, t) \} & \text{and for some } (s', t') \in S_1 \times S_2 \text{ with } \\
(s', t') \notin S_1 \times S_2 \text{ with } & \text{if } S_1 = \emptyset \text{ and } S_2 = \emptyset \\
S_0 &= \begin{cases} \emptyset, & \text{if } S = \emptyset \\
\{s_0, s_0\}, & \text{otherwise} \\
\end{cases}
\end{cases}
\]

Further define \( \rightarrow^* = \{(s, t), a, (s', t') \} \rightarrow (s, t) \in S \text{ and } (s', t') \in S \}

Finally, the composition automaton \( L_1 \odot L_2 \) is defined as:

\[
(S, \text{Act}_1 \cup \text{Act}_2, \rightarrow^*, S_0)
\]

As an example consider logistics automata \( L_1 \) and \( L_2 \) depicted in Figure 2 a) and b). Each automata models the manufacturing life-cycle of single product. We can compose them into a new logistics automaton that reflects a two-product system using the composition operator \( \odot \). The resulting composition \( L_1 \odot L_2 \) is shown in Figure 2 c). One complete activity sequence from the start state to a final state in \( L_1 \odot L_2 \) represents the full execution of both product life-cycles. In fact \( L_1 \odot L_2 \) captures all the allowed interleavings of activities satisfying the life-cycles of both \( L_1 \) and \( L_2 \). In this fashion, batches of products can be specified, including batches with different product types and corresponding life-cycles (an important aspect of flexible manufacturing systems [9]).

Using logistics automata we capture the activity flow of a product and ensure the completion of the manufacturing of this product. In addition we can express constraints within the same product, concerning for instance a safe handover between resources. For a complete specification of the manufacturing of a batch of products, we also need to specify constraints across different product flows. We will express such constraints in terms of constraint automata and introduce a constraint operator to compose them with logistics automata. Constraint automata capture constraints between different products such as ordering constraints (e.g. FIFO ordering for a batch), safety constraints (e.g. access to exclusive safety areas), resource capacity constraints (e.g. a resource must be empty before receiving a product) or other constraints that are expressed as dependencies across different product flows.

**Definition 5.2 (Constraint automaton).** A constraint automaton is a tuple \( \langle S, \text{Act}, \rightarrow, S_0 \rangle \), where \( S \) is a finite (possibly empty) set of states, \( \text{Act} \) is a finite (possibly empty) set of activities, \( \rightarrow \subseteq S \times \text{Act} \times S \) is a transition relation, and \( S_0 \subseteq S \) is a set of initial states, where \( S_0 = \emptyset \) if \( S = \emptyset \) and \( S_0 = \{s_0\} \) otherwise. The following additional property must hold:

- Reachability: if \( S \neq \emptyset \) then for all \( s \in S, s_0 \xrightarrow{a^*} s \).

A constraint automaton encodes a language, just as a logistics automaton does.
Finally, the constrained automaton as relation on L (so that be a logistics automaton let a new logistics automaton which is denoted by A restraint operator. A constraint automaton Constraints can be applied to logistics automata through the con- straintoperator. A constraint automaton is defined by

\[ L(C) = \begin{cases} \emptyset, & \text{if } S_0 = \emptyset \\ \{ a \in \text{Act}^* \mid s_0 \xrightarrow{a} s \text{ for some } s \in S \}, & \text{if } S_0 = \{ s_0 \} \end{cases} \]

Constraints can be applied to logistics automata through the constraint operator. A constraint automaton is called a constraint on logistics automaton if \( \text{Act}_2 \subseteq \text{Act}_1 \). Applying a constraint \( C \) to automaton \( L \) yields a new logistics automaton which is denoted by \( L \upharpoonright C \).

Definition 5.4 (Constraint Operator). Let \( L = (S_1, \text{Act}_1, \rightarrow_1, S_0_1) \) be a logistics automaton \( C = (S_2, \text{Act}_2, \rightarrow_2, S_0_0) \) be a constraint on \( L \) (so that \( \text{Act}_2 \subseteq \text{Act}_1 \)). Before we define \( L \upharpoonright C \) we first define relation \( \rightarrow \subseteq S_1 \times \text{Act}_1 \times S_2 \) as the smallest set \( V \) satisfying the following inference rules:

\[
\begin{align*}
\text{(1)} & \quad s \xrightarrow{a_1 s' a \in \text{Act}_1 \setminus \text{Act}_2} (s, t) \xrightarrow{a} (s', t) \\
\text{(2)} & \quad s \xrightarrow{a_1 s' t \xrightarrow{a} t' a \in \text{Act}_1 \cap \text{Act}_2} (s, t) \xrightarrow{a} (s', t')
\end{align*}
\]

where \( s, s' \in S_1 \) and \( t, t' \in S_2 \). Now define the set of states \( S \) of the constrained automaton as

\[
S = \begin{cases} 
\emptyset, & \text{if } S_1 = \emptyset \text{ or } S_2 = \emptyset \\
\{ (s, t) \in S_1 \times S_2 \mid (s_0_1, s_0_2) \xrightarrow{\ast} (s, t) \} & \text{and for some } (s', t') \in S_1 \times S_2 \text{ with}
\end{cases}
\]

Further define \( \xrightarrow{\cdot} = \{ (s, t), (a, s', t') \} \subseteq \rightarrow \} \text{ for } (s, t), (s', t') \in S \}

\[
S_0 = \begin{cases} 
\emptyset, & \text{if } S = \emptyset \\
\{ s_0_1, s_0_2 \} & \text{otherwise}
\end{cases}
\]

Finally, the constrained automaton \( L \upharpoonright C \) is defined as:

\[ (S, \text{Act}_1, \xrightarrow{\cdot}, S_0) \]

As an example consider automata \( C_1 \) and \( C_2 \) in Figures 2 d) and e). \( C_1 \) models a constraint on the order of activities \( c \) and \( d \) of the product modeled by \( L_1 \) and \( C_2 \) a constraint on the input order of both products by ordering activities \( a \) and \( f \) of \( L_1 \) and \( L_2 \). The result of the constraint operation is depicted in Figures 2 f) showing \( (L_1 \circ L_2) \upharpoonright C_1 \) and \( (L_1 \circ L_2) \upharpoonright C_2 \).

In logistics automaton \( L \upharpoonright C \), \( C \) constrains the behavior of \( L \) resulting in a subset of the original language. This is claimed by the following Lemma:

Lemma 5.5 (Language Constraining). Let \( L \) be a logistics automaton and let \( C \) be a constraint on \( L \). Then \( L(L \upharpoonright C) \subseteq L(L) \).

Note that the constraint operator requires the logistics automaton to run to completion, while this is not true for the constraint automaton. In other words constraint automata capture only safety requirements (expressing that nothing bad should happen) while logistics automata capture both safety requirements and liveness requirements (expressing that something good happens eventually, namely the completion of the different products in the batch). On the contrary, the composition operator requires that both logistics automata run to completion. It was not strictly necessary to add the concepts of constraint automata and constraint operator since constraints can in principle be encoded as logistics automata (by unfolding recursive loops sufficiently often). However, the automaton obtained by composing would in many cases have a larger state-space than the constrained automaton, which we try to prevent. In addition logistics automata encoding constraints would be incomprehensible because of their size and also less reusable. For these reasons we decided to introduce new concepts in the form of constraint automata and the constraint operator.

6 Optimisation-Space Pruning

By observing Figure 2 one may conclude that constraining leads to state-space reduction. Indeed the size of \( (L_1 \circ L_2) \upharpoonright C_1 \) is smaller than that of \( L_1 \circ L_2 \) and the size of \( (L_1 \circ L_2) \upharpoonright C_1 \) \( C_2 \) is smaller than that of \( L_1 \circ L_2 \upharpoonright C_1 \). One can even show that the same holds for the \( (\max,+ \upharpoonright C_1) \) \( C_2 \). This observation led to the main idea behind this way of exploiting constraints to prune the optimisation-space. However, even though constrainting leads to a reduction of the encoded language (see Lemma 5.5) it
does not necessarily imply a reduction of the state-space, neither of the logistics automaton nor of the (max,+) optimization-space. For instance, consider logistics automata $L_3$ and constraint automata $C_3$ and the constrained logistics automaton $L_3 \uparrow C_3$ depicted in Figure 3. Here the size of $L_3 \uparrow C_3$ is larger than that of $L_3$.

In this section we establish sufficient conditions for the constraint operation to reduce the worst-case optimization-space (which is one of the main contributions of this paper). To this end we introduce a relation $\subseteq$ on logistics automata which is stronger than language inclusion (i.e. $L_1 \subseteq L_2$ implies $L(L_1) \subseteq L(L_2)$) and further satisfies the property that $L_1 \subseteq L_2$ implies that the size of the state-space of $L_1$ never exceeds that of $L_2$). We further define the worst-case optimization-space of logistics automaton $L$ as a new logistics automaton $\text{Tree}(L)$ and prove that $\text{MaxPlus}(L) \subseteq \text{Tree}(L)$ and that indeed $\text{Tree}(L \uparrow C) \subseteq \text{Tree}(L)$ for any $L$ and deterministic constraint automaton $C$. This implies that the constraining with a deterministic constraint leads to a reduction of the worst-case state-space.

Definition 6.1 (Inclusion). Let $L_1 = \langle S_1, \text{Act}_1, \rightarrow_1, S_0_1 \rangle$ and $L_2 = \langle S_2, \text{Act}_2, \rightarrow_2, S_0_2 \rangle$ be logistics automata. Then $L_1$ is included in $L_2$, written $L_1 \subseteq L_2$, if and only if $\text{Act}_1 = \text{Act}_2$ and either i) $S_1 = \emptyset$ or ii) $S_0_1 = \{ s_0_1 \}$ and $S_0_2 = \{ s_0_2 \}$ and there exists an injective relation $R \subseteq S_1 \times S_2$ satisfying:

1. $(s_0_1, s_0_2) \in R$;
2. For all $(s_1, s_2) \in R$ and $a \in \text{Act}_1$ if $s_1 \rightarrow_1 s'_1$ (for some $s'_1 \in S_1$) then $s_2 \rightarrow_2 s'_2$ (for some $s'_2 \in S_2$) and $(s'_1, s'_2) \in R$;
3. For all $(s_1, s_2) \in R$ if $s_2 \rightarrow_2$ then $s_1 \rightarrow_1$.

It is not hard to prove that if a logistics automaton is included in another logistics automaton, the size of the state-space of the former never exceeds that of the latter. As an example consider the automata of Figure 2 c) and d) and note that $(L_1 \cup L_2) \uparrow C_1 \subseteq (L_1 \cup L_2) \uparrow C_1$ and that the state-space of $(L_1 \cup L_2) \uparrow C_1$ does not exceed that of $L_1 \cup L_2$. Considering the automata of Figure 3, on the other hand, we observe that in this case $L_3 \uparrow C_3 \subseteq L_3$ and that the size of the state-space of $L_3 \uparrow C_3$ is larger than that of $L_3$.

Definition 6.2 (Tree automaton). Let $L = \langle S, \text{Act}, \rightarrow, S_0 \rangle$ be a logistics automaton. We first define $\text{Paths}(L)$ as the smallest set $V$ satisfying:

\[
\begin{align*}
S_0 &\in V \\
\forall s \in V &\quad s \rightarrow s' \quad \forall s \cdot s' \in V
\end{align*}
\]

Here $s, s' \in S$ and $a \in \text{Act}$. $\text{Paths}(L)$ contains sequences of elements in $S$ and $\text{Act}$. Each sequence is of the form $s_0 a_0 s_1 a_1 \cdots s_n$ and encodes the path from starting state $s_0$ to state $s_n$ (via transitions labeled with activities $a_0 \cdots a_{n−1}$) and intermediate states $s_1 \cdots s_{n−1}$). In inference rule (2), $s \rightarrow s'$ refers to a path that ends in state $s$. Note that $q$ can refer to an empty sequence of elements. Remark that $\text{Paths}(L) = \emptyset$ if $S = \emptyset$. We now define $\text{Tree}(L)$ as

\[
\text{Tree}(L) = \langle \text{Paths}(L), \text{Act}, \rightarrow', S_0' \rangle
\]

where $\rightarrow' = \{ (q s, a, q s a s') \in \text{Paths}(L) \times \text{Act} \times \text{Paths}(L) \mid s \rightarrow s' \}$ and $S_0' = \emptyset$ if $S_0 = \emptyset$ and $S_0' = \{ s_0 \}$ otherwise. Note that a Tree automaton is a logistics automaton.

**Lemma 6.3.** Let $L = \langle S, \text{Act}, \rightarrow, S_0 \rangle$ and $T(L)$ be a logistics automaton. Then $\text{MaxPlus}(L) \subseteq \text{Tree}(L)$.

**Proof.** The results follows directly in case $S = \emptyset$. Otherwise $S = \{ s_0 \}$ for which case we define relation $R \subseteq \text{MaxPlus}(S) \times \text{Paths}(L)$ as the smallest set $V$ satisfying the following inference rules:

\[
\begin{align*}
S_0 &\in V \\
((s_0, 0)_R, s_0) &\in V \quad (s, y_R), q s \in V \quad s \rightarrow s' \in V
\end{align*}
\]

It is not hard to show that $R$ satisfies conditions 1., 2. and 3. of Definition 6.1. To prove that $R$ is injective we have to show that any two pairs in $R$ with equal right-hand elements also have equal left-hand elements. We will do this by induction on the depth of the derivation tree of one of the pairs. Notice that the pairs are either both derived by inference rule (1) or both derived by inference rule (2). In the first case their left-hand elements are obviously the same. In the other case the pairs must be of the forms $((s', y_R \otimes M_a), q s a s')$ and $((s'', y''_R \otimes M_a), q s a s')$ and we further know that $((s, y_R), q s) \in R$ and $((s, y'_R), q s) \in R$. By induction we then have $y_R = y'_R$.

Finally, we show that if $C$ is a deterministic constraint automaton, the constraining of a logistics automaton $L$ with $C$ never results in a larger worst-case optimization-space.

**Lemma 6.4.** Let $L = \langle S_1, \text{Act}_1, \rightarrow_1, S_0_1 \rangle$ be a logistics automaton and $C = \langle S_2, \text{Act}_2, \rightarrow_2, S_0_2 \rangle$ a constraint on $L$. If $C$ is deterministic, then $\text{Tree}(L \uparrow C) \subseteq \text{Tree}(L)$.

**Proof.** Let $L = \langle \text{Paths}(L), \text{Act}_1, \rightarrow_1, \text{Tree}(L) \rangle$ and $T(L \uparrow C) = \langle \text{Paths}(L \uparrow C), \text{Act}_1, \rightarrow_1, \text{Tree}(L) \rangle$. Notice that these tree automata have the same alphabets. Now either i) Paths$(L \uparrow C) = \emptyset$ or ii) $\text{Paths}(L \uparrow C) = \{ s_0, c_0 \}$ and $\text{Tree}(L \uparrow C) = \emptyset$ where $s_0 \in S_0$ and $c_0 \in S_0_2$. In case i) $T(L \uparrow C) \subseteq \text{Tree}(L)$ holds by definition. Otherwise, we define $R \subseteq \text{Paths}(L \uparrow C) \times \text{Paths}(L)$ as the smallest set $V$ satisfying the following inference rules:

\[
\begin{align*}
S_0 &\in V \\
((s_0, c_0), s_0) &\in V \\
((q s, c), (s, c) \rightarrow L_{C} (s', c')) &\in V
\end{align*}
\]
Figure 4: Case study automata: logistics (Life-Cycle) and constraints (Capacity, First-In and First-Out, Swap and Assignment).

Recall that $q(s, c)$ and $p s$ refer to a paths that end in states $(s, c) \in S_L, C$ and $s \in S_1$ respectively. We have to show that $R$ is injective and satisfies condition 1., 2. and 3. of Definition 6.1. Now clearly $((S_0, c_0), S_0) \in R$ so condition 1. is satisfied. For condition 2. let $(q(s, c), p s) \in R$ and assume $q(s, c) \xrightarrow{\text{Tree}(L, C)} q'$ (for some $a \in \text{Act}_1$ and $q' \in \text{Paths}(L \upharpoonright C)$). Then $q'$ must be of the form $q(s, c) a(s', c')$ for some $(s', c') \in S_{L, C}$ for which $(s, c) \xrightarrow{L, C}(s', c')$. But then $s \xrightarrow{a_1} s'$ and thus $p s \xrightarrow{\text{Tree}(L)} p s a's$. Consequently $(q(s, c) a(s', c'), p s a') \in R$, which follows from inference rule (2). For condition 3. let $(q(s, c), p s) \in R$ and assume $p s \xrightarrow{\text{Tree}(L)}$. Then $s \xrightarrow{a_1}$ and since $(s, c) \in S_{L, C}$ also $(s, c) \xrightarrow{L, C}$. But then also $q(s, c) \xrightarrow{\text{Tree}(L \upharpoonright C)}$.

Finally to prove that $R$ injective we have to show that any two pairs in $R$ with equal right-hand elements also have equal left-hand elements. Notice that such two pairs are either both produced by inference rule (1) or both by inference rule (2). In the first case obviously both left-hand sides are equal. In the second case the pairs are of the form $(q_1(s, c_1), a(s', c'_1), p s a s')$ and $(q_2(s, c_2), a(s', c'_2), p s a s')$. Then $(q_1(s, c_1), p s) \in R$ and $(q_2(s, c_2), p s) \in R$. By induction we then have $q_1 = q_2$ and $c_1 = c_2$. We further know that $(s, c_1) \xrightarrow{L, C}(s', c'_1)$ and $(s, c_1) \xrightarrow{L, C}(s', c'_2)$. Now since $C$ is deterministic, we also have that $c'_1 = c'_2$.

In conclusion, the addition of any deterministic constraint results in a reduced worst-case optimization space. In addition it results in an upperbound of the optimal makespan corresponding to the unconstrained specification. Constraints therefore allow the conscious use of over-specification (addition of constraints on top of the original system specification) to control the optimization-space and effectively determine upperbounds on the optimal makespan. We show an example of such a realistic case in the following section.

7 CASE STUDY: WAFER LOGISTICS

To demonstrate the applicability of the modular specification approach we use the model of a lithography system from ASML [15]. Lithography systems are manufacturing systems which process integrated circuits (ICs) by exposing reticle images on silicon wafers. Figure 5 depicts the wafer flow, the 5 operational units (the Conditioner (COND), the Discharge Unit (DU), the Pre-Aligner (PA) and two chucks (CH0 and CH01)) and 2 transport units (the Unload and Load Robots (UR and LR)). The arrows in the figure model the possible flows wafer can have in the system. We do not elaborate on the modeling of the resources, peripherals and actions of the example lithography machine and refer to [15] for the details thereof. Instead our starting point is the set of Activities provided by the system. This list is given in Table 1. To avoid clutter in the figures and text we will refer to these activities using bold-case letters as indicated in Table 1 (e.g. activity Track_2_COND is referred to be letter $a$). We denote the corresponding active on a specific wafer with an index $i$ starting from 1 and ending with $n$, where $n$ is the size of the batch (e.g. $a_i$ refers to activity Track_2_COND performed on behalf of wafer $i$).

For every product we define its life-cycle, i.e. the operation flow. Figure 4 depicts the logistics automata Life-Cycle which captures the operation flow of a product. The life-cycle captures the input ($a$), pre-processing ($b, c, d, e, f$), measure/exposure ($g, h, i, j$ or $k, l, m, n$) and output ($o$) of a wafer in the system. The pre-processing activities are necessary to set a ideal wafer temperature for an accurate exposure process and to align the wafer for a correct overlay of multiple exposures. For performance gains the expose stage is composed of two identical wafer chucks (CH0/CH1) with swap.
activities \((p, q, r, s)\) to alternate between the measure/expose activities. Thus this measure/expose operation is captured by two possible sequences of activities \((g, h, p, i, r, j)\) or \((k, l, q, m, s, n)\) each modeling the measure/expose on a specific chuck (visible from the branching in automaton Life-Cycle). For a batch of \(n\) products we specify \(n\) instances of such automata and composed them using the \(\odot\) operator. All wafers follow the same life-cycle in this case study (but our approach is general enough to deal with different operational flows for different products).

Next to the Life-Cycle requirement we specify other system requirements. These requirements are captured by the Capacity, Swap, First-In (FI) and First-Out (FO) constraint depicted in Figure 4. Note that all these requirements are specified as deterministic constraint automata. Capacity constraints ensures that each resource has limited capacity (in this case all resource have unary capacity). The Swap constraint ensures that between two consecutive measure or expose activities the chucks first have to swap. The First-In and First-Out requirements enforce the order of processing of the wafers in a batch. These automata define the system specification as required for a correct manufacturing.

Table 2 shows the results in terms of the number of states (N. States) and transitions (N. Trans.) of the resulting logistics and (max,+)-automaton state-spaces (Logistics Automaton and Optimization-space) as well as the obtained minimal makespan (Makespan). These results are computed using 8 Intel i7 920@2.67Ghz CPUs with 8GB of memory. The models and the composition and constraining operators are implemented using the CIF (Compositional Interchange Format) tooling [13]. The values shown in Table 2 correspond to a batch of 25 wafers. DNF (Did Not Finish) denotes the case where the construction of the logistics automaton (or its optimization-space) failed to finish. The first row (Logistics) represents the unconstrained system (only the Life-cycle is specified). Every row adds an additional constraining on top of the previous ones (indicated by the + symbol). Note that the incremental constraining of the system with requirements Capacity, First-In and First-Out (FIFO) and Swap systematically reduces the size of the worst-case optimization-space (according to the results of Section 6). Note that in this case the reduction occurs even on the state-spaces of the logistics automata and on the actual optimization-space until the point that the optimization-space can be built and explored to find the minimal makespan of 337 seconds. We leave the study to identify the conditions for which the actual optimization-space reduces as part of future work.

To display the use of the approach as a formal means to consciously use over-specification to further prune the optimization-space we use domain knowledge to generate one additional non-essential constraint called Exchange (see Figure 4). The Exchange constraint enforces an activity order between the load of wafer \((i+2)\) to a chuck and unload of wafer \((i)\) from the same chuck. The results are shown in Table 2. Notice that in this case the optimal solution could already be found without over-specification. Because of this we can compare quantitatively the impact of the additional constraint: it effectively reduces the optimization-space by more than 50%. In general a deterministic constraints results in an upperbound of the optimal makespan (see Section 6); in this case the optimal result is actually preserved.

8 RELATED WORK

To the best of our knowledge, this is the first paper in which modular formal requirements are proposed as an effective means to get qualitative grip on state-spaces sizes.

This work combines the ingredients from the modular constraint-oriented approach of the LOTOS framework [3, 7] and of Supervisory Control Theory (SCT) [10] developed in [14] and applied in [4, 5, 12]. We build upon the general concepts of SCT, but refine this framework by allowing only non-recurrsive requirements and by explicitly distinguishing logistics automata from constraint automata.

To get insight in the impact of automatic composition on the state-space sizes we took inspiration from the Calculus of Communication Systems (CCS) [8], in particular from Milner’s simulation relation \(\prec\). This relation captures behavior but abstracts from all structural information of the automata. We strengthen this relation by adding structural information (in the form of an injectivity requirement) to allow qualitative reasoning about state-space sizes. In addition it benefits from the effective proof technique of establishing simulation relations. The strengthening of pre-order \(\prec\) makes \(\subseteq\) into a partial-order. As far as we have been able to verify, this is the first paper in which the inclusion relation \(\subseteq\) (to compare transitions systems in a structural way) is established.

9 CONCLUSIONS

In this paper we introduced a framework for the specification and optimization of batch-oriented manufacturing systems and show that their makespan optimization is NP-Hard. We introduce a composition operator for logistics automata to address the specification of complex manufacturing system in a modular fashion. We further introduce the notion of constraint automata to capture requirements across different products flows such as input orderings, capacity and safety constraint. Using a constraint operator we are able to modularly compose deterministic constraints with logistics automata and show that this operation efficiently prunes the (worst-case) optimization-space. Moreover, we show that our approach gives explicit control over the worst-case size of the optimization-space. We provide proof of these results by defining an inclusion relation between logistics automata which relates qualitatively their state-space sizes. Moreover, obtained solutions are i) makespan optimal with respect to the added constraints and ii) upperbounds with respect to the original optimization problem. As future work, we are
interested in establishing the conditions for which the constraining operation prunes the actual optimization-space.

ACKNOWLEDGMENTS

This research is supported by the Dutch Technology Foundation NWO-AES (Applied and Engineering Sciences), carried out as part of the Robust Cyber-Physical Systems (RCPS) program, project number 12694.

REFERENCES


