Dynamic Process Synchronization Using BPMN 2.0 to Support Buffering and (Un)Bundling in Manufacturing

Konstantinos Traganos[1](✉), Dillan Spijkers[1], Paul Grefen[1], and Irene Vanderfeesten[1,2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
{k.traganos,p.w.p.j.grefen,i.t.p.vanderfeesten}@tue.nl, dillanspijkers@gmail.com

[2] Open University of the Netherlands, Heerlen, The Netherlands
irene.vanderfeesten@ou.nl

Abstract. The complexity of manufacturing processes is increasing due to the production variety implied by mass customization of products. In this context, manufacturers strive to achieve flexibility in their operational processes. Business Process Management (BPM) can help integration, orchestration and automation of these manufacturing operations to reach this flexibility. BPMN is a promising notation for modeling and supporting the enactment of manufacturing processes. However, processes in the manufacturing domain include the flow of physical objects (materials and products) apart from information flow. Buffering, bundling and unbundling of physical objects are three commonly encountered patterns in manufacturing processes, which require fine-grained synchronization in the enactment of multiple process instances. Unfortunately, BPMN lacks strong support for this kind of dynamic synchronization as process instances are modeled and executed from a single, isolated point of view. This paper presents a mechanism based on BPMN 2.0 that enables process modelers to define synchronization points by using the concept of recipes. The recipe system uses a dynamic correlation scheme to control many-to-many interactions among process instances to implement required inter-instance synchronizations. We formally describe the involved BPMN patterns, implement and evaluate them in a manufacturing scenario in the high-tech media printing domain.

Keywords: BPMN patterns · Dynamic process instances synchronization · Manufacturing · Buffering · (Un)bundling

1 Introduction

In discrete manufacturing, processes get more complex and organizations strive to manage and orchestrate their operations. Activities on a factory shop floor should also be integrated with business functions for a seamless, end-to-end process management [1]. Business Process Management (BPM) is a paradigm that is often employed to help with process orchestration and improve cross-functional integration. While BPM has proven

© Springer Nature Switzerland AG 2020
D. Fahland et al. (Eds.): BPM Forum 2020, LNBIP 392, pp. 18–34, 2020.
https://doi.org/10.1007/978-3-030-58638-6_2

its strength in business sectors where information processing is dominant, e.g. finance [2], it has also been extensively applied in healthcare [3] and transportation [4], where physical entities are included as well. Without surprise, the application of BPM in manufacturing has increasingly gained attention [5], especially in the Industry 4.0 era, where many advanced robots and automated guided vehicles (AGV) have been introduced. That is because the need for orchestration of the activities of all the versatile actors is more imperative.

Modelling and supporting the execution of processes, is a core part in applying BPM concepts. BPMN, as the de-facto standard for business process modelling [6], is widely used for business processes [7]. Its interdisciplinary understandability [8, 9] and the expressiveness with respect to integration to execution [10], together with the need of integration of business processes and manufacturing operations [1, 11, 12], make it a promising candidate for use in the discrete manufacturing domain. Various extensions of BPMN for manufacturing processes have already been proposed [13, 14] and the comparison to other languages [15] shows the language's strengths.

Despite its maturity and the recent interest of applying the notation in manufacturing, BPMN has inherent limitations. One of these is the fact that process models in BPMN are designed from a single, isolated process instance perspective, disregarding possible interactions among instances during execution [16, 17]. Often, process instances need to interact and collaborate based on information that is outside of the scope of one single instance. This collaboration is more important in manufacturing processes, where physical objects, and not only data information, are under consideration. Think of example of buffering points, where inventory is kept at an intermediate stage of a process, or the situation of bundling or batching products (multiple entities) for further processing as single entity (e.g. placing a number of items in a box for transporting). There should be hence, synchronization points where a process instance, representing the flow of activities of entities, waits or sends information regarding the state from or to other instances, commonly from different process definitions. BPMN provides basic synchronization with elements such as Signals or Messages. But the former is a broadcast message without any payload while the latter sends a payload message (with e.g. process instance identifiers or process definition keys) to only one instance. There is a lack of dynamic synchronization expressibility and functionality in the sense that the synchronization of the control flow of process instances cannot currently be decided based upon runtime state and content information of other process instances.

Buffering of entities and (un)bundling of entities and activities are constructs frequently encountered in the physical world of manufacturing processes. Using BPMN for manufacturing processes, entails explicit support for these constructs. Thus, we present in this paper an approach, called recipe system, to address the dynamic synchronization issue described in the previous paragraph. The approach uses standard BPMN 2.0 elements to form a dynamic controller that works as a correlation mechanism for synchronization points amongst independent process instances.

In Sect. 2 we discuss related work of the synchronization shortcoming of BPMN. In Sect. 3, we discuss the characteristics of the manufacturing concepts that we aim to support in terms of modeling and execution. In Sect. 4, the correlation mechanism is described and formalized. The implementation of the mechanism, its application and evaluation are discussed in Sect. 5. Finally, we conclude and reflect on the presented work in Sect. 6.

2 Background and Related Work

In the introduction, we briefly discussed the increasing interest of applying BPMN in manufacturing. More studies make it prominent [18–22]. The fact though that BPMN originates from the service industry, indicates that the notation cannot fully support concepts occurring in the physical, manufacturing world and consequently in the manufacturing sector. Various extensions have been proposed to capture the specific manufacturing characteristics, e.g. manufacturing activities, resource containers and material gateways [13], sensory event definitions, data sensor representations and specific smart manufacturing task types [23], sensing and actuating tasks [24], assets, properties and relationships among these entities [25].

However, all the aforementioned studies do not touch the synchronization problem that BPMN lacks to support. This shortcoming of the language has already been studied, but rather as a general problem, not targeting at the physical and manufacturing world. In general, we see two different paradigms; activity-centric ones (e.g. what BPMN follows) focusing on describing the ordering of activities, and artifact-centric ones focusing on describing the objects that are manipulated by activities [26–30]. From a BPMN perspective, artifact-centric modeling support is limited, though extension elements to support the artifact-centric paradigm have been defined [31]. Fahland et al. [32] approach the process synchronization from a dualistic point of view, both from the activity-centric and the artifact-centric paradigm perspectives. The study argues that processes are active elements that have agents, actors that execute activities. These actors drive the processes forward. Artifacts, on the other hands, are passive elements that are object to the activities. The activities are performed on these objects. While Petri nets are used as a means of process specification, Fahland argues that locality of transitions, which synchronize by "passing" tokens, are at the core of industrial process modeling languages, just like BPMN. Steinau et al. [33] also consider many-to-many process interactions in their study, proposing a relational process structure, realizing many-to-many relationship support in run-time and design-time. Earlier work on process interactions by van der Aalst et al. [34] (e.g. proclets), allowed for undesired behavior in many-to-many relations [35]. Pufahl et al. [36] put forward the notion of a "batch activity", which is an activity that is batched over multiple process instances of the same process definition. The batch is activated upon the triggering of an activation rule. The concept is similar to the approach presented in this paper, but our study includes a strong focus on the correlation of process instances of different process definitions, that typically contain different activities. Finally, Marengo et al. [37] study the interplay of process instances and propose a formal language, inspired by Declare [38], for process modeling in the construction domain.

The importance of BPMN and the still ongoing research on the issue of multi-instance synchronization, is the motivation of our work on supporting frequent manufacturing patterns with this specific language. Our presented approach enriches BPMN and allows practitioners to use the notation for modeling and enactment of their manufacturing processes. In the next section, we first discuss the characteristics of the manufacturing constructs that this paper aims to support with BPMN, namely buffering, bundling and unbundling.

3 Characteristics and Limitations of Manufacturing Constructs

This section introduces the buffering, bundling and unbundling constructs. Due to their similar but inverse relationship, the bundling and unbundling constructs are jointly discussed. Limitations of BPMN to express them is also discussed.

3.1 Manufacturing Constructs Characteristics

Buffering
From an operations management perspective, buffering is considered as maintaining excess resources to cover variation or fluctuation in supply or demand [39]. The concept is also referred to as decoupling inventory between process steps, as these can be performed independently from each other [40]. Buffering, as an operations type of storage, is covered by the manufacturing operations taxonomy, under the Inventory operations category [41]. From [42], we can define buffering as "a form of (temporary) storage with the intention to synchronize flow material between work centers or production steps that may have unequal throughput". From the five types of inventory from [43], we focus on the decoupling inventory/buffers in this study.

In the BPM field, van der Aalst [44] had already argued that places in Petri nets correlate to physical storage locations, in his effort to use high-level Petri nets to describe business processes. Thus, from a process management perspective the notion of a buffer can be explained as follows. An instance enters the buffer and is kept in a holding state. Once a condition is met (e.g. capacity becomes available in the downstream production step), one or more entities are released. The selection of which entity to be released can be based on multiple queuing policies, e.g. the First-In-First-Out (FIFO) policy. Once an entity is released, control flow continues as normal.

The above explanation though, considers the buffering from a single process instance perspective, leading to the process instance isolation issue we described in the previous sections. There is a need to approach the construct from a process control perspective, as such that buffer-level attributes and information from many process instances are captured and managed, as illustrated in Fig. 1.

Bundling and Unbundling
Manufacturing operations literature recognizes operations that bundle, merge, unitize and package entities, as well as their inverse counterparts, but to the best of our knowledge no literature exists that describes how these entities are selected during operations. This is assumed to be described by the modelers in another part of the models or in different models. In this paper we define bundling as "the synchronization of instances that are grouped in some way, either physically or virtually, whose control flow shall continue or terminate simultaneously as a group". Note this is a process-oriented definition and caution should be taken for generalization.

Examples of bundling are commonly encountered when physical entities need to be grouped into some sorts of a container. Imagine for instance products being produced and put in a packaging box. Once the capacity of the box is reached, the box can be transported as a single entity. Upon arrival of the box to a distribution center, entities are

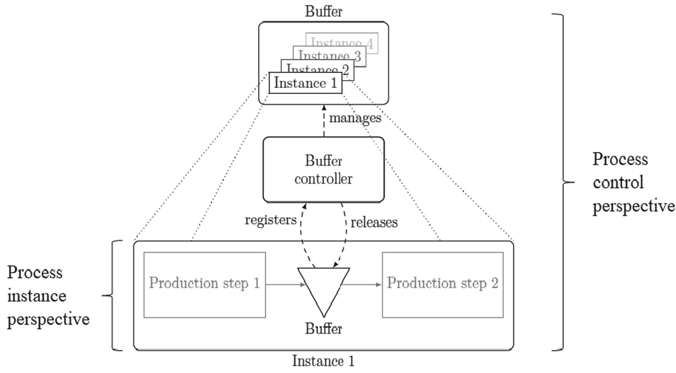

Fig. 1. Buffering construct from both process control and process instance perspective.

unbundled again. We use the term bundling, as a more generic term than batching, since the latter normally refers to putting together entities of the same type, while in bundling we can merge entities of different types. Bundling is often encountered together with buffering, as quite often, (sub-)entities are buffered before the bundling operation can take place, to ensure all (sub-)entities are present.

3.2 BPMN Limitations

A buffering point between two activities (or process fragments), as shown with the triangle element in Process Instance 1 in Fig. 1, could be naively modeled in BPMN 2.0 with the use of conditional or (intermediate) message catching events. These elements can offer the "holding" state of the control flow. However, none approach is suitable. Conditional events use local-instance variables, ignoring information of other process instances. Message events are targeted to a specific, pre-defined instance, missing dynamic correlation information.

Bundling and unbundling constructs can be probably modeled with AND-gateways. But these gateways (un)merge control flows that can be modelled on the same definition, which is not always possible. In many scenarios, different processes have to be correlated and gateways cannot perform this. Erasmus et al. [41] discuss also the use of multi-instance activities (among the other BPMN patterns proposed for modeling manufacturing processes) for unitizing, as they call it, entities. The spawning of repeated instances can serve (un)bundling functionality. However, the isolation problem appears here as well. Each child process instance is unaware of the information of the rest child instances.

4 Concept and Functionality of a Recipe System

In this section we first present the approach to overcome the synchronization issues. The approach is later formalized.

4.1 The Recipe Controller

Having provided the definitions of the constructs under consideration, along with their attributes, and discussed the BPMN limitations to support them (in Sect. 3), we present here the solution of a recipe control system. The work of Spijkers [45] discusses also a list of functional and non-functional requirements for designing such a system.

Recipe
The central notion of the system is the *recipe*. It corresponds to a synchronization (or integration) point, where (previously uncorrelated) control flows in independent process instances may be synchronized. It consists of a set of input rules and output rules. A recipe is fulfilled once all input rules are satisfied. We link two important concepts in a recipe. The *instance type* and the *selector attribute*. The first is used to group process instances of the same type in a *pool*. Think for example a car assembly process. It requires a number of wheels, a number of doors and a chassis. Each of these elements are produced independently according to their process models. Thus, we can have three pools, one with “CarWheel” instance type, one with “CarDoor” type and one with “CarChassis” type. The *selector attribute* is used for discriminating instances that are of the same type, yet of a different variant. For example, the “CarDoor” instance type can have the color (e.g. blue/red) as attribute. A pool is a virtual “container” to keep homogenous process instances; homogeneous from an instance type perspective, as these can have different attributes. All the concepts are illustrated in Fig. 2. Process instances are denoted as shape figures.

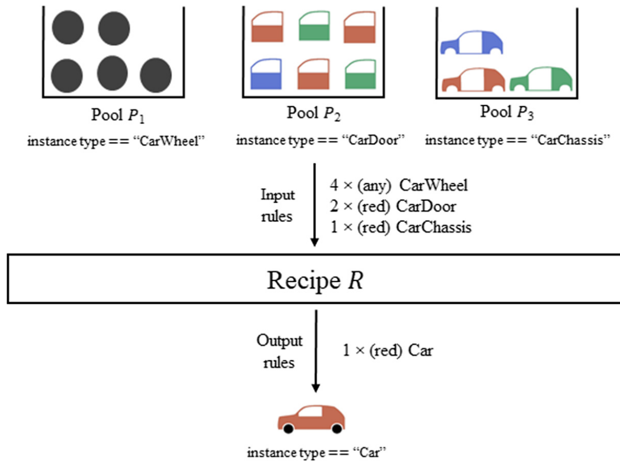

Fig. 2. Illustration of the recipe concepts with an example (Color figure online)

The configuration of each pool plays a crucial role for the fulfillment of a recipe. The following options are considered:

- **Genericity.** A pool can be either *generic* or *specific*. In the first case, the pool does not consider the selector attribute of the buffered process instances (e.g. in Pool $\mathcal{P}_1$

of Fig. 2). In the latter case, recipe fulfillment candidates are nominated based on the selector attribute (e.g. on the color in pools $\mathcal{P}_2$ and $\mathcal{P}_3$ of **Fig.** 2).

- **Availability mask.** Pools can represent physical buffers but as such should account for physical availability, i.e. how instances/objects are accessed. This paper considers three availability masks:

– ALL: All instances are available (e.g. in a virtual or physical pool that we do not care about the physical layout).
– FIRST: The instance that was first placed in the pool is considered as available. Subsequent instances are marked as available if and only if they share the selector attribute value of the first instance, in one sustained sequence.
– LAST: The instance that was placed last in the pool is considered as available. Subsequent instances are marked as available if and only if they share the selector attribute value of the last instance, in one sustained sequence.

- **Release policy.** The release policy ranks instances for recipe fulfillment (and thus "release" from the pool). This paper considers three policies:

– FIFO: instances that have been in the pool the longest are released first.
– LIFO: instances that have been in the pool the shortest are released first.
– ATTR: instances are released based on a selector attribute value.

Fulfillment Cardinality. The fulfillment cardinality determines how many instances of a pool are needed to lead to recipe fulfillment. It can be a single value, i.e. all instances are nominated for fulfillment or it can take a minimum (n) and a maximum (m) value, i.e. the pools needs at least n and less than m.

With the configuration options described above, the recipe can be specified with the following notation, shown in Fig. 3 (same example as in Fig. 2). Upon a recipe fulfilment, a process may continue its flow after the respective synchronization points or a new process instance (mainly from a different process definition) can start.

Recipe name: Final car assembly
Selector attribute: `ordernumber`

Input instance type	min	max	gen	relpol	mask	rel
CarWheel	4	4	•	FIFO	LAST	○
CarDoor	4	4	○	LIFO	ALL	○
CarChassis	1	1	○	LIFO	ALL	•

num	Start process definition key (output)
1	Final_Car_Assembly_Process

Fig. 3. Specification of a recipe (proposed notation)

4.2 Formalization

Recipes ($\mathcal{R}$) are treated as sequences that contain Pools ($\mathcal{P}$) that are treated as sequences that contain instances. The notation $|\mathcal{P}|$ is used to denote the number of instances currently in pool $\mathcal{P}$. The notation $\mathcal{P}(i)$, with $i \in \{1, \ldots, |\mathcal{P}|\}$, refers to the i-th instance in

the pool. Not to be confused with the powerset notation $\mathcal{P}(A)$, referring to the powerset of set A. Note that this instance indexing is based on the time at which an instance was added to the pool. In other words, from a mathematical perspective, a pool is an array of instances that is sorted on arrival timestamp. In general, the symbol i is used to either denote an array index (like in the $\mathcal{P}(i)$ notation) or a process instance, like $i \in \mathcal{P}$. The latter should be read as *instance i in pool $\mathcal{P}$*. The mathematical model, which extends the content presented in the previous section, uses the following symbols:

$\mathcal{R}$ a recipe.
$\mathcal{P}$ a pool. Is a member of a recipe, i.e. $\mathcal{P} \in \mathcal{R}$.
$\mathcal{S}$ the (abstract) set of possible selector attributes.
$s_{\mathcal{P}}$ the selector attribute for pool $\mathcal{P}$.
$\mathcal{V}_s$ the (abstract) set of possible selector attribute values for selector attribute $s \in \mathcal{S}$.
v_i the selector attribute value for instance $i \in \mathcal{P}$.
$c_{\mathcal{P}}^-$ the minimum fulfillment cardinality for pool $\mathcal{P}$.
$c_{\mathcal{P}}^+$ the maximum fulfillment cardinality for pool $\mathcal{P}$.
$\alpha_{\mathcal{P}}(i)$ availability mask function for pool $\mathcal{P}$. $\alpha_{\mathcal{P}}(i) \in \{0, 1\} | \forall i \in \mathcal{P}$.
$\rho_{\mathcal{P}}(i)$ release policy ranking function for pool $\mathcal{P}$. $\rho_{\mathcal{P}}(i) \in \{1, \ldots, |\mathcal{P}|\} | \forall i \in \mathcal{P}$.
$g_{\mathcal{P}}$ boolean whether pool $\mathcal{P}$ is generic (1) or specific (0). $g_{\mathcal{P}} \in \{0, 1\}$.
$\mathcal{S}(\mathcal{P})$ the set of selector attribute values for which at least $c_{\mathcal{P}}^-$ instances exist in pool $\mathcal{P}$.

Formally defined as

$$\mathcal{S}(\mathcal{P}) \equiv \left\{ v \in \left\{ v_p : p \in \mathcal{P} \right\} : \left| \left\{ v_p : p \in \mathcal{P} \wedge v_p = v \right\} \right| \geq c_{\mathcal{P}}^- \right\} \quad (1)$$

Note that, by definition, $\mathcal{S}(\mathcal{P}) \subseteq \mathcal{V}_{s_{\mathcal{P}}}$ holds.

$m(\mathcal{P})$ a map that maps an attribute value to a sequence of fulfillment candidate instances (of the same attribute value) in pool $\mathcal{P}$.

$$m(\mathcal{P}) : v \rightarrow I \quad (2)$$

with $v \in \mathcal{S}(\mathcal{P})$ and set of instances $I \subseteq \mathcal{P}$.

Later in the discussion, Fig. 4 introduces an example of such a mapping.

Availability Mask Functions

Availability masking uses a boolean mask to indicate whether an instance is available for recipe fulfillment. The mask $\alpha_{\mathcal{P}}(i)$ equals to 1 if and only if the instance argument i is available for recipe fulfillment (otherwise 0). Consequently, an instance may only be nominated for a fulfillment if $\alpha_{\mathcal{P}}(i) = 1$ holds for instance $i \in \mathcal{P}$. There are three flavors of availability masks. First, there is the `ALL` mask, which means that all instances are available. Alternatively, there is the `FIRST` mask, which marks the first element as available. Subsequent instances are available if and only if they share the selector attribute value of the *first* instance, in one sustained sequence (as is often the case in physical stacks only accessible from the stacking direction). Somewhat inversely, there is the `LAST` mask. As the name suggests, this mask marks the last element as available. Preceding instances are available if and only if they share the selector attribute of the

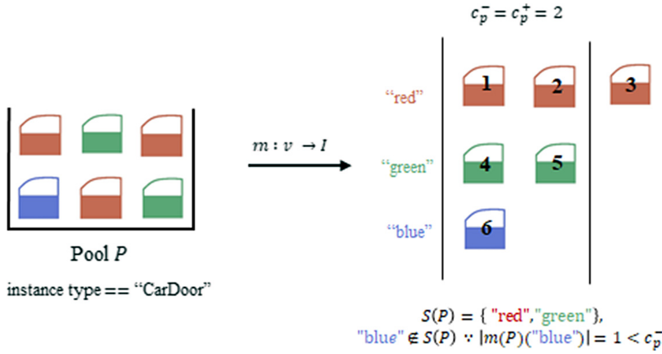

Fig. 4. Map generation $m(\mathcal{P})$ example

last instance, in one sustained sequence. All three masks are defined with the following equations:

$$\alpha_{\mathcal{P}}^{\mathrm{ALL}}(\mathcal{P}(i)) \equiv 1, \ \ \forall i \in \{1, \ldots, |\mathcal{P}|\} \tag{3}$$

$$\alpha_{\mathcal{P}}^{\mathrm{FIRST}}(P(i)) \equiv \begin{cases} 1 \text{ if } i = 1 \vee \left(v_{\mathcal{P}(i)} = v_{\mathcal{P}(i-1)} = \ldots = v_{\mathcal{P}(1)}\right) \\ 0 \qquad\qquad \text{otherwise} \end{cases} \ \forall i \in \{1, \ldots, |\mathcal{P}|\} \tag{4}$$

$$\alpha_{\mathcal{P}}^{\mathrm{LAST}}(P(i)) \equiv \begin{cases} 1 \text{ if } i = |\mathcal{P}| \vee \left(v_{\mathcal{P}(i)} = v_{\mathcal{P}(i+1)} = \ldots = v_{\mathcal{P}(|\mathcal{P}|)}\right) \\ 0 \qquad\qquad \text{otherwise} \end{cases} \ \forall i \in \{1, \ldots, |\mathcal{P}|\} \tag{5}$$

Release Policy Functions
Release policies use a ranking function to prioritize instances for fulfillment. A lower rank means the instance is preferred. First off, there is the First-In-First-Out (FIFO) release policy, which orders instances based on the timestamp t at which they were added to the recipe pool.

$$i_1 \prec i_2 \Leftrightarrow t_{i_1} \leq t_{i_2}, \ \ \forall (i_1, i_2) \in \mathcal{P} \times \mathcal{P} \tag{6}$$

Instance i_1 is preferred over i_2 for release, if and only if the time added to the pool of i_1, t_{i_1} is smaller than or equal to that of i_2, t_{i_2}. In other words: the instances are ranked such that their timestamps are non-decreasing. The ranking function, $\rho_{\mathcal{P}}^{\mathrm{FIFO}}$ is therefore defined simply as the instance index of the time-sorted sequence of instances in a pool:

$$\rho_{\mathcal{P}}^{\mathrm{FIFO}}(\mathcal{P}(i)) \equiv i, \ \ \forall i \in \{1, \ldots, |\mathcal{P}|\} \tag{7}$$

Secondly, there is the inverse of FIFO, Last-In-First-Out (LIFO), again based on timestamp t.

$$i_1 \prec i_2 \Leftrightarrow t_{i_1} \geq t_{i_2}, \ \ \forall (i_1, i_2) \in \mathcal{P} \times \mathcal{P} \tag{8}$$

Notice that Eq. (8) results in the reverse ranking of Eq. (6). The resulting ranking function, $\rho_{\mathcal{P}}^{\mathrm{LIFO}}$ is therefore the inverse ranking of Eq. (7):

$$\rho_{\mathcal{P}}^{\mathrm{LIFO}}(\mathcal{P}(i)) \equiv 1 + |\mathcal{P}| - i, \ \ \forall i \in \{1, \ldots, |\mathcal{P}|\} \tag{9}$$

Lastly, there is the attribute based policy (**ATTR**), which sorts instances based on some attribute, denoted by #. As an example instantiation of this policy, one could think of a priority based policy.

$$i_1 \prec i_2 \Leftrightarrow \#_{i_1} \geq \#_{i_2},\ \ \forall(i_1, i_2) \in \mathcal{P} \times \mathcal{P} \tag{10}$$

To define the **ATTR** release policy ranking function, we first define the sequence $\text{sort}^{\downarrow}(A, \#) \subseteq A$ to be the result of sorting sequence A on some attribute # in descending order (i.e. the result is nonincreasing). Furthermore, we define $\text{index}(i, A) \in \{1, \ldots, |A|\}$ to return the index at which element i occurs in sequence A. Using these intermediate definitions, we can arrive at the final definition:

$$\rho_{\mathcal{P}}^{\text{ATTR}}(\mathcal{P}(i)) \equiv \text{index}\big(\mathcal{P}(i), \text{sort}^{\downarrow}(\{\mathcal{P}, \#\})\big),\ \ \forall i \in \{1, \ldots, |\mathcal{P}|\} \tag{11}$$

where # refers to the priority attribute to be sorted.

Given the properties of these functions, the discussion above can be generalized to

$$i_1 \prec i_2 \Leftrightarrow \rho_{\mathcal{P}}(i_1) \leq \rho_{\mathcal{P}}(i_2)\ \ \forall(i_1, i_2) \in \mathcal{P} \times \mathcal{P} \tag{12}$$

This generalized form is used in the subsequent implementation. The function definition denoted by $\rho_{\mathcal{P}}$ is to be replaced with an appropriate release policy function variant.

Note that, since output rules are released instantaneously once a recipe is fulfilled, the effect of these release policies is only observable if there is a choice which instances should remain in the pool. This choice is only there if there are more instances in the pool than the maximum fulfillment cardinality, i.e. $|m(v \in \mathcal{S}(\mathcal{P}))| > c_{\mathcal{P}}^{+}$. Otherwise, exactly $\min\big(c_{\mathcal{P}}^{+}, |\mathcal{P}|\big)$ instances are selected in the fulfillment and the ordering is irrelevant, as becomes apparent in the following algorithmic discussion.

The Pool Algorithm

As mentioned before, a pool can produce a mapping $m : v \in S(P) \rightarrow I \subseteq P$ upon request. This mapping maps an attribute value v to a sequence of fulfillment candidate instances I. A visual example that explains how that mapping works, can be found in Fig. 4. In this figure, the "CarDoor" pool from Fig. 2 is used as an example.

The pool's mapping algorithm is listed in Fig. 5.

The Recipe Algorithm

The recipe algorithm collects and analyzes pool maps to determine fulfillment feasibility. If a fulfillment can be achieved for a particular selector attribute value, the algorithm releases the appropriate instances from the pools and returns them in a list. The algorithm is listed in Fig. 6.

5 Prototype Implementation, Demonstration and Evaluation

This section discusses the technical implementation of the proposed synchronization approach, as it was prototyped, demonstrated and evaluated in a real use case.

```
Algorithm: Pool's mapping algorithm, i.e. m(P).
  Input: Pool P.
  Output: Mapping of attribute values to sequence of fulfillment candidate
          instances, m : v ∈ S(P) → I ⊆ P.
  /* Map generation phase.                                                    */
  m1 ← ({} → {});                                 /* Initialize empty map m1. */
  foreach i ∈ {i ∈ P : α_P(i) = 1} do   /* For every available instance i in the pool. */
    if m1(v_i) = ∅ then                       /* If value v_i not in map m1 yet. */
      m1(v_i) ← {};                            /* Add new value v_i to map m1. */
    end
    m1(v_i) ← m1(v_i) ∪ {i};                      /* Add instance i to map m1. */
  end
  /* Map pruning phase.                                                       */
  m2 ← ({} → {});                                 /* Initialize empty map m2. */
  foreach v ∈ m1 do                         /* For every key value in map m1. */
    if |m(v)| ≥ c⁻_P then              /* At least c⁻_P instances exist for value v. */
      m1(v) ← sort↑(m(v), ρ_P);       /* Rank instances based on release policy. */
      l ← {};                              /* Initialize empty candidate list. */
      x ← min(|P|, c⁺_P);          /* Determine how many instances to nominate. */
      for (i ← 1; i ≤ x; i ← i + 1) do         /* For every nominated instance. */
        l ← l ∪ {m1(v)(i)};          /* Add instance m1(v)(i) to list of candidates. */
      end
      m2(v) ← l;                  /* Place list of candidates in pruned map m2. */
    end
  end
  return m2;                                      /* Return the pruned map m2. */
```

Fig. 5. Pool's mapping algorithm

5.1 Technical Implementation

The aforementioned mathematical model and algorithms are implemented in a digital artifact using the Java programming language. The classes of the model are also represented by a technical data model. The Java code interacts with the process engine of a BPM System that executes the process models. The code is embedded in a typical process model definition, which offers the functionality of the recipe controller. The high-level internal implementation of the controller in BPMN 2.0 is shown in the bottom part of Fig. 7. It receives Submit (or Cancel) messages from specific synchronization points from the main process definitions (e.g. after Production task A1 of Production Process A and at the end of Production Process B), evaluates the recipes based on the messages' content and releases (via Release messages) the continuation of control flow once recipes are fulfilled.

5.2 Demonstration and Evaluation

The implemented recipe system was demonstrated in a real-world use case in the manufacturing printing domain, within the European EIT OEDIPUS[1] project. The scenario consisted of several printers, binding and trimming machines, and a robotic arm mounted on an AGV to grasp and transport paper and books between the devices and storage places. Activities performed by all these agents were modelled and enacted by a BPMS.

[1] https://www.eitdigital.eu/innovation-factory/digital-industry/oedipus/.

```
Algorithm: Recipe's fulfillment algorithm.
  Output: Sequence of buffered instances that are part of the fulfillment. Empty
          sequence if recipe cannot be fulfilled.
  /* Map analysis phase.                                                        */
  v ← ∅;                          /* Initialize sequence of potential fulfillment values. */
  foreach p ∈ R do                                        /* For each pool in recipe. */
  |  m_p ← m(p);                                 /* Query and store the pool's map. */
  |  if |keys(m_p)| = 0 ∧ c_p^- ≠ 0 then          /* If this pool cannot be fulfilled. */
  |  |  v ← ∅;                                  /* A global fulfillment is infeasible. */
  |  |  break;
  |  end
  |  if v = ∅ then                          /* If this is the first pool to analyze. */
  |  |  /* Take the first pool's potential fulfillment values as starting point.    */
  |  |  if c_p^- = 0 then
  |  |  |  v ← {∅};          /* Add generic null value as potential fulfillment value. */
  |  |  else
  |  |  |  v ← keys(m_p);           /* Add potential fulfillment values to sequence. */
  |  |  end
  |  end
  |  if g_p = 0 ∧ c_p^- ≠ 0 ∧ |p| ≠ 0 then        /* If this pool should be accounted for in
  |   fulfillment feasibility. */
  |  |  if v = {∅} then      /* If the previous pool was a generic pool (or was a satisfied
  |  |   pool with 0 candidates), but this pool is not. */
  |  |  |  v ← keys(m_p);                   /* Overwrite potential fulfillment values. */
  |  |  else
  |  |  |  v ← v ∩ keys(m_p);                    /* Prune potential fulfillment values. */
  |  |  end
  |  end
  end
  /* Fulfillment feasibility analysis phase.                                     */
  if v = ∅ ∨ |v| = 0 then                           /* If no fulfillment is feasible. */
  |  return {};                                           /* Return empty sequence. */
  end
  f ← v(1);                        /* Pick the or a fulfillment value and store it in f. */
  r ← {};                                 /* Initialize sequence of released instances. */
  /* Note: f = ∅ can hold true by design, in case of a generic fulfillment.          */
  /* Data restructure phase.                                                          */
  foreach p ∈ R do                                                  /* For each pool. */
  |  if ¬(c_p^- = 0 ∧ |m(p)| = 0) then                     /* Skip empty optional pools. */
  |  |  foreach i ∈ m(p)(f) do                   /* For every to-be-released instance. */
  |  |  |  release(p, i);                               /* Release instance from pool. */
  |  |  |  r ← r ∪ {i};                   /* Add instance i to sequence of released. */
  |  |  end
  |  end
  end
  return r;                                    /* Return sequence of released instances. */
```

Fig. 6. Recipe's fulfillment algorithm

Various synchronization points existed in the scenario, mapping to the buffering and un(bundling) constructs described in this paper. Recipes, using the notation of Fig. 3, were described and configured for supporting these points. The points were modeled in the process models as well. One such point was the output tray of a printer. There, sequentially produced books were placed and were ready for transportation to a binder. The corresponding recipe took care to synchronize the activities for bundling (and unbundling) of the books from the tray, onto the AGV, and then into the binder. For the sake of brevity, the complete process models are not presented here. Figure 8 shows a simpler example of independent process models interacting with the Recipe controller.

The recipe controller system was primarily evaluated on its functionality to support the modeling in BPMN 2.0 of physical manufacturing constructs (i.e., buffering and (un)bundling). It was also evaluated in terms of usability by asking practitioners, through

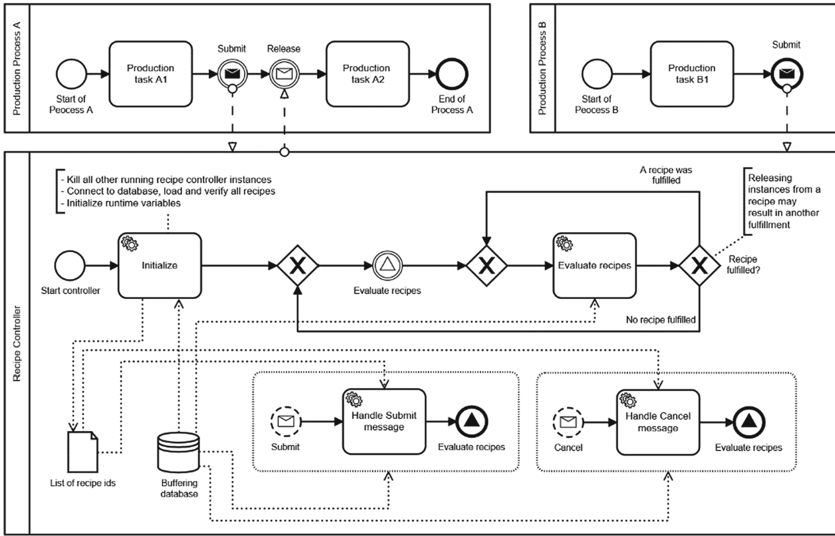

Fig. 7. The recipe controller (BPMN 2.0 process model)

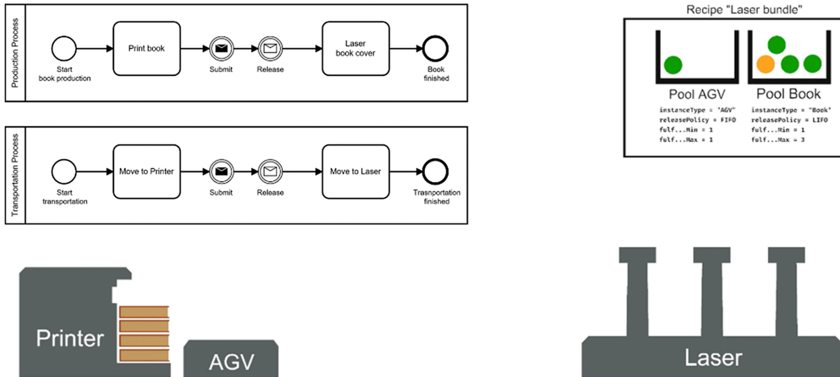

Fig. 8. Interaction of recipe controller with manufacturing processes

a structured survey, to define recipes and model synchronization points. In general, they find the approach useful.

6 Conclusion

This paper presents a solution for realizing dynamic process synchronization and correlation through a structured use of BPMN messages, with the goal to support manufacturing constructs, more specifically, buffering, bundling and unbundling. The solution is a novel approach to address the general dynamic synchronization issue, stemming from the process instance isolation, that the language fails to support. The recipe system that

we propose is formally described, implemented and demonstrated in a real-world case study at a large, international firm in the printing industry. While this work does not claim complete suitability of the solution to all cases, the demonstration of the solution proves its feasibility. The usability was also evaluated by practitioners, engineers and researchers that modeled processes with the recipe system approach, who perceived it as useful.

However, there are limitations in the current work that are opportunities for further research. Assumptions in the definition of the models, such as that pools have infinite capacity or their cardinality is only expressed in units of process instances, may need to be relaxed. Workaround solutions exist, such as using an external knapsack problem solving engine, which passes group information to the recipe system in the form of selector attribute values, so that the recipe system can perform the appropriate bundling operations. Similarly, the assumption that a selector attribute is shared across all pools should be addressed by giving unique object identifiers to each pool. Furthermore, to make the system more dynamic and flexible, the recipes should be (re)configured during runtime and the fulfillment conditions should be variable instead of static. Not forget to mention that new BPMN elements can be crafted as extension to the notation for representing buffer and synchronization points.

References

1. Erasmus, J., Vanderfeesten, I., Traganos, K., Grefen, P.: The case for unified process management in smart manufacturing. In: IEEE Computer Society Digital Library (2018)
2. Brahe, S.: BPM on top of SOA: experiences from the financial industry. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 96–111. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75183-0_8
3. Reichert, M.: What BPM technology can do for healthcare process support. In: Peleg, M., Lavrač, N., Combi, C. (eds.) AIME 2011. LNCS (LNAI), vol. 6747, pp. 2–13. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22218-4_2
4. Baumgraß, A., Dijkman, R., Grefen, P., Pourmirza, S., Völzer, H., Weske, M.: A software architecture for transportation planning and monitoring in a collaborative network. In: C-Matos, L.M., Bénaben, F., Picard, W. (eds.) PRO-VE 2015. IAICT, vol. 463, pp. 277–284. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24141-8_25
5. Janiesch, C., et al.: The Internet-of-Things meets business process management: MutualBenefits and challenges. arXiv:1709.03628 (2017)
6. Decker, G., Barros, A.: Interaction modeling using BPMN. In: ter Hofstede, A., Benatallah, B., Paik, H.-Y. (eds.) BPM 2007. LNCS, vol. 4928, pp. 208–219. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78238-4_22
7. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M., Russell, N.: On the Suitability of BPMN for Business Process Modelling. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 161–176. Springer, Heidelberg (2006). https://doi.org/10.1007/11841760_12
8. Rosa, M., ter Hofstede, A., Wohed, P., Reijers, H., Mendling, J., van der Aalst, W.: Managing process model complexity via concrete syntax modifications. IEEE Trans. Ind. Inform. **7**(2), 255–265 (2011). https://doi.org/10.1109/TII.2011.2124467
9. Witsch, M., Vogel-Heuser, B.: Towards a formal specification framework for manufacturing execution systems. IEEE Trans. Ind. Inform. **8**(2), 311–320 (2012). https://doi.org/10.1109/TII.2012.2186585

10. Ko, R., Lee, S., Wah Lee, E.: Business process management (BPM) standards: a survey. Bus. Process Manag. J. **15**(5), 744–791 (2009)
11. Pauker, F., Mangler, J., Rinderle-Ma, S., Pollak, C.: Centurio.work - modular secure manufacturing orchestration. In: Proceedings of the Dissertation Award, Demonstration, and Industrial Track of the 16th International Conference on Business Process Management (BPM), CEUR-WS.org, Sydney, Australia (2018)
12. Prades, L., Romero, F., Estruch, A., García-Dominguez, A., Serrano, J.: Defining a methodology to design and implement business process models in BPMN according to the standard ANSI/ISA-95 in a manufacturing enterprise. Procedia Eng. **63**, 115–122 (2013). https://doi.org/10.1016/j.proeng.2013.08.283
13. Zor, S., Schumm, D., Leymann, F.: A proposal of BPMN extensions for the manufacturing domain. In: Proceedings of the 44th CIRP International Conference on Manufacturing Systems (2011)
14. Abouzid, I., Saidi, R.: Proposal of BPMN extensions for modelling manufacturing processes. In: 2019 5th International Conference on Optimization and Applications (ICOA), Kenitra, Morocco, pp. 1–6 (2019). https://doi.org/10.1109/icoa.2019.8727651
15. García-Domínguez, A., Marcos, M., Medina, I.: A comparison of BPMN 2.0 with other notations for manufacturing processes. In: AIP Conference Proceedings, Cadiz, vol. 1431, pp. 593–600 (2012). https://doi.org/10.1063/1.4707613
16. Van der Aalst, W., Artale, A., Montali, M., Tritini, S.: Object-centric behavioral constraints: integrating data and declarative process modelling. In: Description Logics (2017)
17. Leitner, M., Mangler, J., R-M, S.: Definition and enactment of instance-spanning process constraints. In: Wang, X.S., Cruz, I., Delis, A., Huang, G. (eds.) WISE 2012. LNCS, vol. 7651, pp. 652–658. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35063-4_49
18. Kim, B.H., Park, S.B., Lee, G.B., Chung, S.Y.: Framework of integrated system for the innovation of mold manufacturing through process integration and collaboration. In: Gervasi, O., Gavrilova, M.L. (eds.) ICCSA 2007. LNCS, vol. 4707, pp. 1–10. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74484-9_1
19. Cadavid, J., Alférez, M., Gérard, S., Tessier, P.: Conceiving the model-driven smart factory. In: ACM International Conference Proceeding Series, August 2015, vol. 24-26, pp. 72–76. Association for Computing Machinery (2015). https://doi.org/10.1145/2785592.2785602
20. Jasiulewicz-Kaczmarek, M., Waszkowski, R., Piechowski, M., Wyczółkowski, R.: Implementing BPMN in maintenance process modeling. In: Świątek, J., Borzemski, L., Wilimowska, Z. (eds.) ISAT 2017. AISC, vol. 656, pp. 300–309. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-67229-8_27
21. Kavka, C., Campagna, D., Milleri, M., Segatto, A., Belouettar, S., Laurini, E.: Business decisions modelling in a multi-scale composite material selection framework. In: 4th IEEE International Symposium on Systems Engineering (2018). https://doi.org/10.1109/syseng.2018.8544386
22. Knoch, S., et al.: Enhancing process data in manual assembly workflows. In: Daniel, F., Sheng, Q.Z., Motahari, H. (eds.) BPM 2018. LNBIP, vol. 342, pp. 269–280. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11641-5_21
23. Yousfi, A., Bauer, C., Saidi, R., Dey, A.K.: uBPMN: A BPMN extension for modeling ubiquitous business processes. Inf. Softw. Technol. **74**, 55–68 (2016). https://doi.org/10.1016/j.infsof.2016.02.002
24. Petrasch, R., Hentschke, R.: Process modeling for industry 4.0 applications: towards an industry 4.0 process modeling language and method. In: 13th International Joint Conference on Computer Science and Software Engineering, JCSSE (2016)

25. Lindorfer, R., Froschauer, R., Schwarz, G.: ADAPT - a decision model-based approach for modeling collaborative assembly and manufacturing tasks. In: Proceedings of the IEEE 16th International Conference on Industrial Informatics, INDIN, pp. 559–564 (2018)
26. Cohn, D., Hull, R.: Business artifacts: a data-centric approach to modeling business operations and processes. IEEE Data Eng. Bull. **32**, 3–9 (2009)
27. Lohmann, N., Wolf, K.: Artifact-centric choreographies. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 32–46. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17358-5_3
28. Meyer, A., et al.: Data perspective in process choreographies: modeling and execution. Techn. Ber. BPM Center Report BPM-13-29. BPMcenter. org. (2013)
29. Meyer, A., et al.: Automating data exchange in process choreographies. In: Jarke, M., et al. (eds.) CAiSE 2014. LNCS, vol. 8484, pp. 316–331. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07881-6_22
30. Meyer, A., Weske, M.: Activity-centric and artifact-centric process model roundtrip. In: Lohmann, N., Song, M., Wohed, P. (eds.) BPM 2013. LNBIP, vol. 171, pp. 167–181. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06257-0_14
31. Lohmann, N., Nyolt, M.: Artifact-centric modeling using BPMN. In: Pallis, G., et al. (eds.) ICSOC 2011. LNCS, vol. 7221, pp. 54–65. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31875-7_7
32. Fahland, D.: Describing behavior of processes with many-to-many interactions. In: Donatelli, S., Haar, S. (eds.) PETRI NETS 2019. LNCS, vol. 11522, pp. 3–24. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21571-2_1
33. Steinau, S., Andrews, K., Reichert, M.: The relational process structure. In: Krogstie, J., Reijers, H.A. (eds.) CAiSE 2018. LNCS, vol. 10816, pp. 53–67. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91563-0_4
34. Van der Aalst, W., Barthelmess, P., Ellis, C., Wainer, J.: Proclets: a framework for lightweight interacting workflow processes. Int. J. Coop. Inf. Syst. **10**, 443–481 (2001). https://doi.org/10.1142/S0218843001000412
35. Fahland, D., De Leoni, M., Van Dongen, B., Van der Aalst, W.: Many to-many: some observations on interactions in artifact choreographies. ZEUS **705**, 9–15 (2011)
36. Pufahl, L., Weske, M.: Batch activity: enhancing business process modeling and enactment with batch processing. Computing **101**(12), 1909–1933 (2019). https://doi.org/10.1007/s00607-019-00717-4
37. Marengo, E., Nutt, W., Perktold, M.: Construction process modeling: representing activities, items and their interplay. In: Weske, M., Montali, M., Weber, I., vom Brocke, J. (eds.) BPM 2018. LNCS, vol. 11080, pp. 48–65. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98648-7_4
38. Pesic, M., Schonenberg, H., Van der Aalst, W.: DECLARE: full support for loosely-structured processes. In: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, pp. 287–300. IEEE (2007)
39. Nahmias, S., Olsen, T.: Production and Operations Analysis, 7th edn. Waveland Press, Long Grove, Ill (2015). (OCLC: 935795578)
40. Cachon, G., Terwiesch, C.: Matching Supply with Demand: An Introduction to Operations Management. McGraw-Hill/Irwin, Boston (2009). (OCLC: ocn191732546)
41. Erasmus, J., Vanderfeesten, I., Traganos, K., Grefen, P.: Using business process models for the specification of manufacturing operations. In: Computers in Industry (to appear)
42. Defense Acquisition University: Integrated Product Support (IPS) Element Guidebook. Defense Acquisition University, Fort Belvoir (2011)
43. De Groote, X.: Inventory theory: a road map. teaching note. Department of Decision Sciences, The Whanon School (1989)

44. Van der Aalst, W.: Putting high-level Petri nets to work in industry. Comput. Ind. **25**(1), 45–54 (1994). https://doi.org/10.1016/0166-3615(94)90031-0
45. Spijkers, D.: Expressing and supporting buffering and (un)bundling in the manufacturing domain using BPMN 2.0. Master's thesis, Eindhoven University of Technology, Eindhoven (2019)