Deadlock Avoidance for Multiple Tasks in a Self-Organizing Production Cell

Joseph Hirsch, Martin Neumayer, Hella Ponsar, Oliver Kosak and Wolfgang Reif
Institute for Software & Systems Engineering, University of Augsburg, Germany
E-Mail: joseph.hirsch@student.uni-augsburg.de, {neumayer, ponsar, kosak, reif}@isse.de

***Abstract*—Deadlocks represent situations in which two participants are waiting for each other to finish an activity so that neither of them will ever finish. Deadlocks can occur in complex computer-integrated systems, such as flexible and self-organizing production systems. As deadlocks bring production to halt, methods for deadlock control in production systems are widely studied. Yet most algorithms proposed are not suited for the use in decentral multi-agent systems, as they require central control or can not handle concurrency. Other algorithms can be used in a decentral fashion but assume that only one type of product will be manufactured at a time. But in times of mass customization, where customers choose a product from a variety of options, support for several product types is required. To meet both the requirements of mass customization and decentral multi-agent systems, we present a new decentralized approach for avoiding deadlocks in a self-organizing production cell, where several types of products are being manufactured in parallel. Our approach is based solely on local knowledge and does not assume central control. We evaluate our approach in terms of effectiveness and message overhead to conclude that it avoids starvation and deadlocks with a reasonable message overhead.**

***Index Terms*—deadlock avoidance, decentral mechanisms, self-organization, production systems, multi-agent systems**

I. Introduction

Mass production systems are tailored to produce one product in high quantities. The machinery used is highly specialized for its operation and rigidly linked. As every machine performs only one specific task before the product is handed over to the next machine, cycles are prevented by design, as they may become a bottleneck for throughput or yield hold-and-wait-conditions. While this design enables high throughput and low costs, these traditional production lines are also prone to failure and do not offer the flexibility to manufacture different types of products at once [1]. Due to rigid linkage, often realized with conveyor belts, the breakdown of a single machine or conveyor might bring the whole line to halt.

With mass customization, small lot sizes, and ever-increasing cost pressure, production systems are about to change [2]. Manufacturers strive for flexible automation to provide customers with just the product they need while keeping production costs low [1]. Self-organizing production systems represent an approach to meet these requirements. Modeling production systems as a set of self-organizing agents offers a way to increase autonomy, responsiveness, and openness [3]. Self-organizing production systems also leverage decentral control to ensure scalability [4] and robustness [5]. Connecting production agents with automatically guided vehicles (AGVs) instead of using conveyor belts can further increase the degree of flexibility and automation [6].

Self-organizing production systems are also not tailored to a specific type of product. Instead, self-organizing production systems consist of agents or machines offering capabilities such as milling or drilling and AGVs connecting those agents as needed. We consider machines that process one product[1] at a time while having no buffers. Products are described as a sequence of capabilities and given to the system. The matching between the capabilities needed to manufacture the type of product and the capabilities offered by the agents is termed *task allocation* [2]. Task allocation is formulated as a Constraint Satisfaction Problem (CSP) [7] and solved at runtime. We denote the result of task allocation as *product flow*. Task allocation at runtime has two main advantages:

1) Robustness: A self-organizing production system can deal with partial breakdowns by detecting faults, finding, and implementing a new task allocation.
2) Flexibility: Self-organizing production systems offer flexibility in terms of the product manufactured. As long as the needed capabilities for a new type of product exist in the system, new task allocations can be found by solving the corresponding CSP.

However, the shift towards finding a task allocation at runtime, instead of design time, yields the problem of cycles emerging. Cycles arise if an agent receives a product he has processed before, or if two different product flows are arranged in a circle. Cycles can then overflow with products and result in deadlocks, as demonstrated in Fig. 1. Cyclic arrangements should be avoided in the first place as they may lead to inefficiency and the hazard of deadlocks. However, this is not always possible, assuming limited redundancy of machinery and capabilities per machine. In cases of agent or component failures, there could no longer be any task allocation left that prevents cyclic arrangements. Even in such a state, production should continue to ensure the desired robustness.

In this paper, we present a new approach for preventing deadlocks in a decentralized, self-organizing production cell, where several products are being manufactured in parallel. To prevent deadlocks, the approach detects cycles in the product flow whenever the configuration of the system changes. For

[1]We refrain from using the word resource as it is ambiguous in the manufacturing domain and can serve as a term for a machine as well as for a product [2].

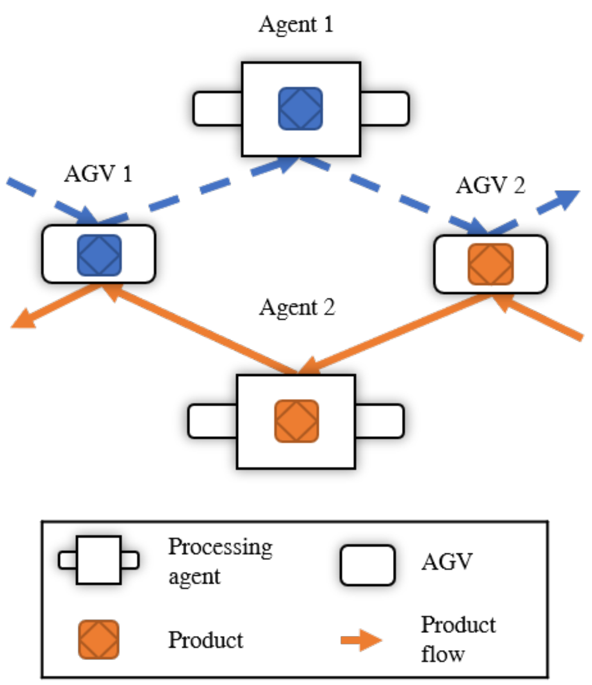


Fig. 1. A production system, consisting of two agents and two AGVs, encounters a deadlock caused by two opposing product flows. As every agent and AGV is holding a product while cyclically waiting for another agent or AGV, the system comes to a halt.

each cycle, the maximum permissible number of products is calculated and then enforced at runtime. This approach extends previous work done in [8]. It supports multiple types of products and is based solely on local knowledge.

The remainder of this paper is structured as follows: In Section II, we introduce a motivating example and review the fundamentals of deadlocks and dealing with deadlocks. Section III examines previous work and other approaches towards the problem. In Section IV our approach is explained. We evaluate our approach in terms of messages sent and effectiveness in Section V. We conclude this paper with future work and further research directions in Section VI.

II. Problem Definition

In this section, we present a motivating example to illustrate how deadlocks can emerge in self-organizing production systems. To better understand the phenomenon of deadlocks, a formal introduction into deadlocks and deadlock control is given. Finally, we lay out the requirements for a possible solution and match these requirements with the methods for deadlock control presented before.

A. Motivating example

Consider the motivating example depicted in Fig. 1: Assume a production system consisting of two processing agents and two AGVs, where there are two types of products to be manufactured. While the blue products (dashed product flow) require processing by Agent 1, the orange products (solid product flow) have to be processed by Agent 2. In the situation presented AGV 1 and Agent 1 hold blue products, while Agent 2 and AGV 2 hold orange products.

This arrangement is problematic: AGV 1 cannot pass on the product it holds to Agent 1, as Agent 1 also holds a product. Yet Agent 1 cannot pass on its product to AGV 2, as AGV 2 holds a product. This pattern continues until we reach AGV 1 again. As every participant is waiting for another participant in a way that none ever finishes, a deadlock emerges and the production halts.

B. Deadlocks

Formally, deadlocks are situations where two or more participants are waiting for another to finish in a way that no one ever finishes. Coffman et al. describe four conditions that have to be met for a deadlock to occur [9]:

1) *Mutual exclusion*: Two participants can't use the same resource at the same time.
2) *No preemption*: Participants keep their resources until their computing finishes. Under no circumstances, resources are removed from the participants forcibly.
3) *Hold and wait*: Participants can wait for a resource if they already hold another one.
4) *Circular wait*: The participants wait relationships are arranged in a circle so that each participant waits for another one to release a resource.

We assume the first three criteria are met in self-organizing production systems. Circular wait situations might arise depending on the arrangement of agents, as pointed out in the motivating example in Fig. 1. Especially after partial failures of the system, there may be no arrangement of agents left, which is both free of cycles and able to keep up production. Consequently, a mechanism to deal with deadlocks is needed.

C. Dealing with deadlocks

Coffman et al. [9] distinguish between three approaches to deal with deadlocks:

a) Deadlock prevention: The approach of designing a system in a way that no circular wait situations can emerge because no product flow contain any cycles and product flows are not arranged in cycles. Prevention of deadlocks results in a system that is deadlock-free. But the design requires global knowledge about the product flows and the arrangement of agents within it. In scenarios, where only task allocations exist that yield cyclic arrangements, no task allocation can be implemented. This contradicts our motivation to continue production as long as possible.

b) Deadlock detection and recovery: This approach does not prevent deadlocks in the design of the system. Instead, the system detects deadlocks at runtime and recovers from them. In general, this approach yields a higher utilization of the system. Yet, deadlock recovery is a non-trivial challenge, especially when not only computer processes and data but also physical objects such as products are affected. Therefore, we concentrate on a third approach.

c) Deadlock avoidance: The goal of deadlock avoidance is to prevent the system from reaching a deadlock at runtime. This is achieved by monitoring the product flow and using the knowledge about the future behavior of each agent to detect circular wait situations before they emerge. The advantage of this approach is a system that can adapt to different situations while having a high utilization of the available agents. There is no need for the recovery of deadlocks, but communication between the agents might be required.

D. Problem Statement

In summary, deadlocks can appear in production systems, when there are circular wait situations. These circular wait situations are the result of cyclic arrangements of agents. Yet, in self-organizing production systems, cyclic arrangements cannot be ruled out, as the product flow is determined at runtime. Therefore, an approach for dealing with deadlocks is required.

As deadlock prevention approaches inhibit cyclic arrangements of agents, these approaches rule out configurations that could keep up production. This contradicts our motivation to manufacture products as long as possible. Deadlock recovery approaches are not further considered in this paper, as recovering from deadlocks involving physical products is beyond the scope of this work. Therefore, to solve the problem presented, a decentral deadlock avoidance approach inhibiting circular wait relationships is suited.

Besides the functional requirement of avoiding deadlocks in production with different types of products, the approach has to match the decentral nature of multi-agent systems and allow for concurrency. Lastly, the message overhead to provide the aforementioned functionality should be as low as possible.

III. Related Work

A. The Organic Design Pattern (ODP)

Self-organizing production systems in this paper are specified and modeled using the Organic Design Pattern (ODP), described in [10] and [11]. The relevant parts of the pattern are described in the following paragraphs. For more information, please refer to the cited sources.

a) Definitions: The active participants of the system are called *agents*, while *products* are processed by the agents. In contrast to previous work, we refrain from using the term *resource* as it is ambiguous in the context of manufacturing systems [2]. The blueprint on how to manufacture a product is called *task*. A task consists of a set of capabilities in a specified order. The task's state specifies which capabilities were already applied and which capability has to be executed next. To meet the required capabilities of a product, each agent has a set of *capabilities* it can provide. The concept of a *role* encompasses three aspects:

1) A *precondition* that includes an agent to take products from, called *port*, as well as the product's task and state.
2) A set of *capabilities* to apply to the product.
3) A *postcondition* that comprises an agent to pass products on to (*port*), as well as the product's task and state.

An adequate system configuration that satisfies all given tasks need to include corresponding roles for each capability of each task. The roles must be distributed among the agents in a way that each capability of the task is met and that a valid product flow is represented by matching pairs of pre- and postconditions.

As an example, let's consider the system lined out in Fig. 1: Agent 1 might have a role, that tells him to take products for a given task and state from AGV 1 (its precondition port), perform one or several capabilities on the product and hand the product over to AGV 2 (its postcondition port). AGV 2 then must have the corresponding role that has Agent 1 as its precondition port.

b) Specification of a system: In the ODP a system is specified using sets: Each system consists of a set of agents. An agent has a set of capabilities that it can apply and a set of roles allocated defining the function of the agent in the system. The tasks are defined by the conditions that are stored in the roles of the agents.

c) Processing of products: To keep the system running and process products according to their tasks, each agent checks two things periodically:

1) If it has any *input requests* meaning another agent finished applying its capabilities to a product and wants to hand it over to the agent. In this case, the agent chooses a role which precondition fits one of the input requests and executes this role.
2) If it has any *producer roles*. A producer role has a capability list that consists of or starts with a so-called produce capability, which adds a product to the system, e.g., by taking it out of storage. If a producer role exists, the agent chooses one randomly and executes it.

B. Deadlock control in production systems

Dealing with deadlocks in production systems is a well-known and widely studied problem. Therefore, only a fraction of the available literature is covered in this paper. For a comprehensive survey of graph-theoretic, automata-based, and Petri Net approaches for controlling deadlocks in production systems refer to [12]. For a survey on detecting deadlocks in distributed systems, see [13].

In [14] and [15], Petri Nets are used to generate a restriction policy to avoid deadlocks in Flexible Manufacturing Systems (FMS). Both approaches require a global view of the system to generate these restrictions, therefore they are not suitable for distributed systems.

Another approach is presented in [16] and [17]. The authors introduce an algorithm that leverages global knowledge about the system to detect cycles in the graph representing the working procedures (tasks). For each event in the system (i.e. a new product is created or transmitted to another agent) a central controller unit determines which transactions are save to execute in the next step. Due to the required central control, this approach is also not applicable to distributed systems.

A distributed cycle detection algorithm is presented in [18]. It makes use of messages, traveling along the edges of a graph.

Messages are forwarded until they return to the agent that has sent the message or they hit a sink and can't be forwarded further. Although the authors present a well-designed and lightweight algorithm, it is only capable of deciding if a given agent is in a cycle. The algorithm cannot determine the agents in the cycle or the cycle's size, which is essential for deadlock avoidance.

In [8], the authors discuss deadlock avoidance for systems modeled with the Organic Design Pattern (ODP). An algorithm for distributed systems without central control or knowledge is introduced. The algorithm is based on cycle detection and controlling the resource flow at runtime. However, systems with multiple tasks being processed at the same time are not considered and left as future work.

IV. Approach

In [19] and [20], Wysk et al. study the detection of deadlocks in manufacturing systems. The authors elaborate on the idea of circular wait relationships in production systems. They prove that the following two conditions must be met for a deadlock to occur:

1) There exists at least one cycle in the product flow graph
2) Each agent in the cycle has to be occupied by a product

In the introduction, we have pointed out that cycles cannot be completely avoided, therefore in this section, we propose a decentral deadlock avoidance approach, that addresses the second condition and ensures that the number of products in each cycle is lower than the number of agents in the cycle at any time. To realize this behavior, our approach consists of two steps.

1) Whenever the configuration of the system changes, e.g., because a new type of product enters or an agent breaks down, **cycle detection** will be performed. If a cycle is detected, the algorithm determines how many products are allowed to enter.
2) At runtime, the agents keep track of the number of products that are currently in each cycle in a product counter (pc). They enforce the limits calculated in cycle detection by coordinating the agents that are entrances and exits of the cycles. In the following, this step is referred to as **enforcing the limits for products in cycles**.

Before cycle detection and enforcing the limits for products in cycles are explained in detail, some basic definitions are introduced.

a) Cycles: Cycles are groups of agents, each of which can only contain a certain number of products to avoid deadlocks. This also includes cycles that are not directly identifiable in the production plan, but are detected by the combination of cycles. Further detail will be given later on. For cycles the following information is stored: The agents that are part of the cycle, the tasks that are responsible for the cycle, the maximum number of products that the agents in the cycle can handle without overloading it ($maxProducts$) and if the cycle was detected by combining other cycles.

b) Entrance agents: An entrance agent (of a cycle) has at least one role that places a new product into the cycle. Agents can detect whether they are an entrance agent locally. An agent is a potential entrance agent, if the following condition does apply:

$$\begin{aligned} &\exists (r_1, r_2) \in roles \times roles : \\ &\qquad r_1.precondition.port \neq r_2.precondition.port \\ &\qquad \vee\ r_1.postcondition.port \neq r_2.postcondition.port \end{aligned}$$

An agent is an entrance agent to a cycle c if the following condition applies:

$$\begin{aligned} \exists r \in roles : r.&postcondition.port \in c \\ &\wedge r.precondition.port \notin c \end{aligned}$$

The other agents in the cycle, however, don't know the agent's roles so they need to be informed, that the agent is an entrance agent.

A. Cycle detection

Each agent stores the set of cycles it is involved in along with the entrance agents. To reduce the message overhead, cycle detection is split into two steps.

```
Data:
$roles$ : the roles assigned to agent $a$
$M_R$: the cycle detection messages that $a$ has to receive
$cycles$ : the cycles stored by $a$ in its context
for $(r_1, r_2) \in roles \times roles$ do
    if $r_1.postcondition.port = r_2.precondition.port$
     then
        $c \leftarrow$ new Cycle
        $c.agents \leftarrow [a, r_1.postcondition.port]$
        $c.maxProducts \leftarrow 1$
        Add $c$ to $cycles$
        Call Alg. 3
    end
end
if $a$ is a potential entrance agent then
    $M_R \leftarrow []$
    for $r \in roles$ do
        $m \leftarrow$ new cycle detection message
        $m.originalSender \leftarrow a$
        $m.isCycle \leftarrow True$
        $m.forks \leftarrow []$
        $m.cycle \leftarrow$ new cycle
        $m.cycle.agents \leftarrow [a]$
        $m.cycle.maxProducts \leftarrow 0$
        Add $m$ to $M_R$
        Send $m$ to $r.postcondition.port$
    end
end
```

Algorithm 1: Start of the cycle detection.

```
Data:
$m$ : the cycle detection message received by agent $a$
$M_R$ : the cycle detection messages that $a$ has to receive
$roles$ : the roles of agent $a$
$cycles$ : the cycles stored by $a$ in its context
if $a = m.originalSender$ then
    if $m.isCycle$ then
        Add $m.cycle$ to $cycles$
        Call Alg. 3
    end
    Add all $m.forks$ to $M_R$
    if received all $M_R$ then
        finish cycle detection and start production
    end
end
else
    $nextAddressees \leftarrow []$
    for $r \in roles$ do
        if $r.postcondition.port \notin$
          $m.cycle \vee r.postcondition.port =$
          $message.originalSender$ then
            add $r.postcondition.port$ to
              $nextAddressees$
        end
    end
    if $nextAddressees$ is empty then
        $m.isCycle \leftarrow False$
        Send $m$ to $m.originalSender$
    end
    else
        Add $a$ to $m.cycle.agents$
        $m.cycle.maxProducts \leftarrow$
          $m.cycle.maxProducts + 1$
        $copies \leftarrow$ Create $|nextAddressees| - 1$ copies
          of $m$
        Add all $copies$ to $m.forks$
        Send $m$ to first agent of $nextAddressees$
        Send $copies$ to other agents of
          $nextAddressees$
    end
end
```

Algorithm 2: Receive and forward a *cycle detection message*. These steps are performed whenever agent a receives a *cycle detection message*.

1) Cycle detection based on local knowledge: With its local knowledge, an agent detects all cycles it is involved in consisting of only two agents. If the agent receives a product from another agent, it is also giving a product to, the agent is in a cycle with the other agent, meaning that only one of them can accept a product at a time. Fig. 2 shows a cycle like this. Both agents are entrance agents to the cycle. These cycles are based on opposing tasks, so they are referred to as encounter cycles in the following.

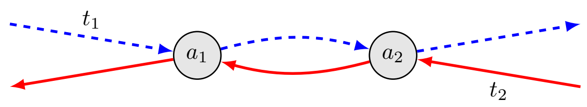


Fig. 2. Cycle with two involved agents. This cycle can be detected without message overhead, based on local knowledge.

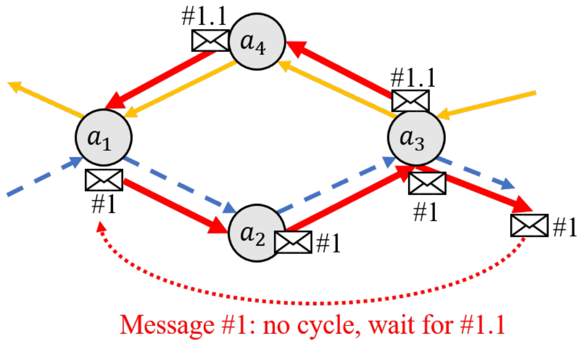


Fig. 3. Cycle detection messages. Agent α_1 is possibly an entrance agent to a cycle because of its roles of the two tasks which are depicted in yellow and blue. It initiates a *cycle detection message* with the identifier #1. This message is forwarded along the product flow. Agent α_3 splits the message to message #1 and message #1.1. Message #1 eventually returns to agent α_1 with the information that it does not represent a cycle but that this message has been split and that the sender has to wait for message #1.1. As agent α_1 receives message #1.1 it knows about the cycle since it was the origin of the message.

2) Cycle detection based on the analysis of the product flow graph: Cycles with more than two agents are detected by sending messages along the product flow. Fig. 3 gives an overview of this process. Agents send out and forward *detect cycle messages*. Alg. 1 shows how and when messages are created, Alg. 2 explains how agents react when receiving a message. These messages include a set of agents that forwarded the message. Each agent forwarding the message adds itself to this set, before passing the message on. If the message returns to its sender, the sender is part of a cycle, and the set of forwarding agents is equal to the agents in the cycle.

a) Agents that send out cycle detection messages: To further reduce the number of messages sent, only potential entrance agents send out *cycle detection messages*. To check if an agent is a potential entrance agent, it analyzes its roles in pairs. If the agent has two roles with different pre- or postcondition ports, it is a potential entrance. In contrast, an agent with only one role or the same pre- and postcondition port for all roles can't be an entrance-agent. An entrance agent has to have at least two different pre- and postcondition combinations thus two roles: One role has to be transporting products within the cycle or out of the cycle and one has to be transporting products into the cycle. If the agent is a potential entrance agent, it is sending out a *detect cycle message* to every postcondition port of every role.

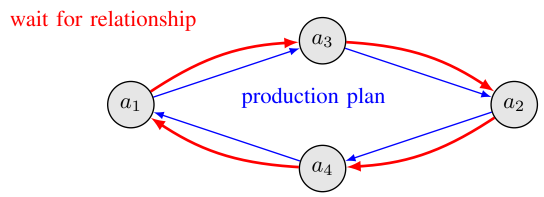
b) Receiving and forwarding detect cycle messages: If a *detect cycle message* is received, the agent checks if it is the origin of the message. If the agent is not the origin of the message, it forwards the message to all of its successive agents according to the product flow. To prevent infinite looping messages, they are only forwarded under the following condition: Either the postcondition port of the role is the message's original sender or the postcondition port is not yet in the set of agents of the cycle stored in the message. If the message has to be forwarded to more than one agent, it is split into multiple messages which we refer to as forks. The identifiers of the forks are stored in the original message so the agent that initiated the message knows for which forks to wait. When forwarding a message, the agent adds itself to the set of agents stored in the message because if the message returns to the original sender, the agent is part of the deadlock cycle the message represents. If all the postcondition ports of the agent's successive roles are in the set of agents already, the message is running in a loop and the corresponding cycle has been detected before, so the message is returned to the original sender with this information.

If the agent is the source of the message it receives, it stores the message and checks if it received all messages and all forks of the messages it has initiated.

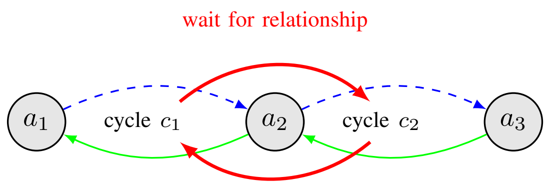
3) Processing the cycles: Whenever a cycle is stored, a series of tests, which are described in the following, is applied. Alg. 3 summarizes these tests.

```
Data:
$c$: the new cycle detected
$cycles$ : the cycles stored by $a$ in its context
if nested cycles are contained in $c$ then
    Inform every agent of $c$ about all nested cycles
      inside $c$
end
if agent is entrance agent of $c$ then
    Inform every agent of $c$ about the entrance agent
end
if another cycle exists, which the agents transports
  products into when transporting them out of $c$ then
    Combine the cycles
end
for $c' \in cycles$ do
    Recalculate $c'.maxProducts$
end
Remove unneccessary cycles from $cycles$
```
Algorithm 3: Tests when agent a detects a new cycle.

1) **Check for cycles contained in the new cycle**
If a cycle c_{in} is contained in another cycle c it is important that all the agents of c know c^{in} for step 5 (calculating $maxProducts$). If an inner cycle is detected, the agent informs all the agents in c that aren't in c^{in} about c^{in}.
2) **Check if the agent is an entrance agent of the new cycle**
The agent checks if it is an entrance to the cycle. If this is the case, the agent sends a *inform entrance message* to all agents in the cycle to inform them about it.
3) **Check for combined cycles**
The agent checks its stored cycles and the new cycle in pairs if a role exists, which transfers a product from one to another cycle. If a role like this exists, the cycles need to be combined to a bigger cycle because both cycles depend on each other. To remove a product of one cycle, the other one has to have space for the product. Fig. 4 shows two different types of cycles, where the second one is detected by combining cycles. Whenever two cycles are combined, the set of agents, as well as the set of tasks, are merged. The capacity $maxProducts$ of the new cycle is calculated in step 4. Therefore, the new combined cycle is stored without removing the old cycles and the agent proceeds to do all the checks of this list for the new cycle.
4) **Calculate the maximum number of products of each cycle**
The calculation of the maximum number of products in the cycles is mainly done when detecting them: If an agent creates a *detect cycle message*, it initiates the cycle with $maxProducts = 0$. Each agent that is added to the cycle then increases $maxProducts$ by one. Encounter cycles are stored with $maxProducts = 1$. Following this algorithm, the maximum number of products for each cycle c is $|c|-1$, where $|c|$ is the number of agents in the cycle. If some cycles are nested, $maxProducts$ has to be adjusted. To calculate the final value for $c.maxProducts$ it is necessary to check if other cycles are contained in c, meaning that the cycle's agents are a subset of the agents of c. If there is a product in a cycle c^{in} that is inside cycle c, all agents of c^{in} are reserved for that product, while the present calculation assumes that a product only occupies one agent. If that is the case, $c.maxProducts$ has to be decreased: Let c be the cycle observed and C^{in} the cycles inside of c that are not combined cycles. Then following formula applies:
$$c.maxProducts = |c.agents| + \sum_{c^{in} \in C^{in}} c^{in}.maxProducts - \left| \bigcup_{c^{in} \in C^{in}} c^{in}.agents \right| - \left| \bigcup_{(c_i^{in}, c_j^{in}) \in C^{in} \times C^{in}} c_i^{in}.agents \cap c_j^{in}.agents \right|$$
This calculation is evaluated for every cycle whenever a new cycle is detected. The detection of a new cycle could have an impact on already known cycles e.g., if the new cycle is nested inside a known cycle.
5) **Check for unnecessary cycles**
The agent checks for all of its cycles if a combination of cycles (c^{in}, c^{out}) exist so that c^{in} is contained in c^{out} and $c^{in}.maxProducts \geq c^{out}.maxProducts$. Cycle c^{in} then gets removed from the cycle list because c^{out} is more restrictive.



(a) A cycle of wait for relationships because of a cycle in the product flow. The number of agents in the wait-for cycle is equal to the number of agents in the product flow cycle.



(b) Agent a_2 has to wait for cycle c_2 to have a space for a product before it can hand over a product from agent a_1 (out of the cycle c_1) to agent a_2 (into the cycle c_2). The number of agents in this wait for relationship is 3 while the number of participants is 2. The number of products that is allowed to enter this configuration is 1 in order to avoid a deadlock. The algorithm would combine c_1 and c_2 to a new cycle and find $maxProducts = 1$.

Fig. 4. Different types of cycles.

B. *Enforcing the limits for products in cycles*

The following section describes the part of the mechanism executed at runtime when a role is chosen to be executed or when other agents send announcements and *take back messages*:

a) Choosing role: Before an agent accepts an input request to execute a capability or produces a product, it checks if it can execute the corresponding role without producing a deadlock. If the agent is an entrance-agent to a cycle, the agent checks if the locally stored number of products in the cycle is less than $maxProducts$ by at least one. If so, it sends an *announce product message* to every other entrance-agent of the cycle and waits for all of them to accept the product. If it receives a *declining message*, it will not execute the role but instead, notify all agents that already accepted the product (*take back message*). Thus, the notified agents know the product wasn't processed and adjust their product counter respectively. Fig. 5 shows an exemplary communication sequence of an agent sending out *announce product messages* and handling the decline of a product. *Take back messages* are necessary since the agents accept a product and update their product counter optimistically. If the product can't be processed, this update has to be rolled back. If all agents accept an announced product, the announcing agent can safely execute the role.
After executing a role, an agent that is an exit of a cycle decreases its product counter and informs the entrance agents of the cycle to decrease their product counter for this cycle as well.

b) Handling announce product messages: An agent receiving an *announce product message* locally checks if the cycle is capable of handling another product ($maxProducts > pc$) and sends back either an *accepting message* or a *declining message*. The necessity of *take back messages* is rooted in the concurrency of the system. Ideally an announcement should not be declined since the announcing agent should not have asked for permission in the first place because of its local knowledge. Yet, problems arise, if two agents announce products at the same time. Then the local counter was already updated and the product is declined. If the agent sends back an *accepting message* it will assume that the sending agent will put a new product into the cycle and increase the corresponding product counter.

Apart from checking whether the product can be accepted regarding deadlocks, fairness between the tasks is brought into account as well: Each agent keeps track of how often it executed each role. If the agent has an input request of a role that has been executed less frequently than the average of all roles, it declines products belonging to other roles to execute the underrepresented role in the next choose role iteration.

c) Handling a take-back message: If an agent receives a *take back message* for a cycle it will decrease its local product counter, as it optimistically increased its product counter for a product that never entered the cycle. Although take-backs are very unlikely to occur, this fallback mechanism is needed due to the concurrent nature of the system: Two or more agents may decide to execute a role adding a product to a cycle that can only handle one more product at the same time. First, they decide locally, and due to the lack of knowledge about the other agents' intentions, they assume the role to be executable. Then they announce the product at the respective other agents and get a declining response. If an additional agent is also an entrance-agent to the concerning cycle, this agent also received a product announcement of the other agents and has already accepted the product of one of the agents and optimistically updated the product counter. This additional agent has to decrease the product counter again.

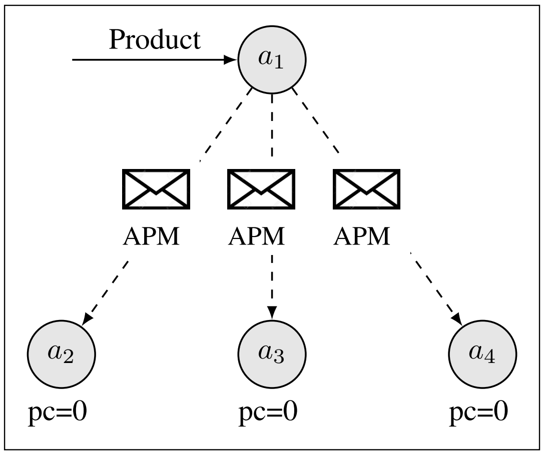
V. Evaluation

In this section, we experimentally evaluate, if our approach presented in Section IV is capable of avoiding deadlocks under varying conditions. Therefore, we run several simulations with different system configurations and measure the following properties:

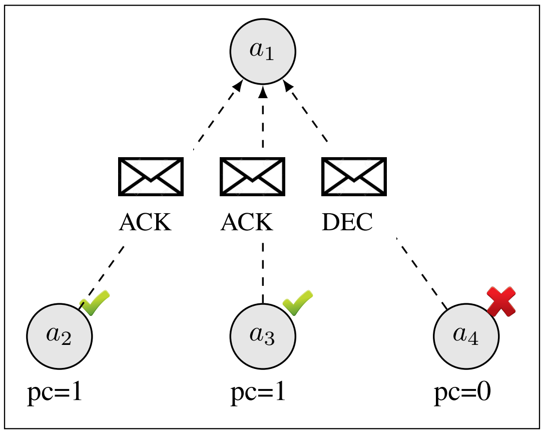
1) Runtime in seconds
2) Number of deadlocks encountered
3) Number of messages sent

Additionally, we divide the number of manufactured products by the runtime to calculate the system's throughput.

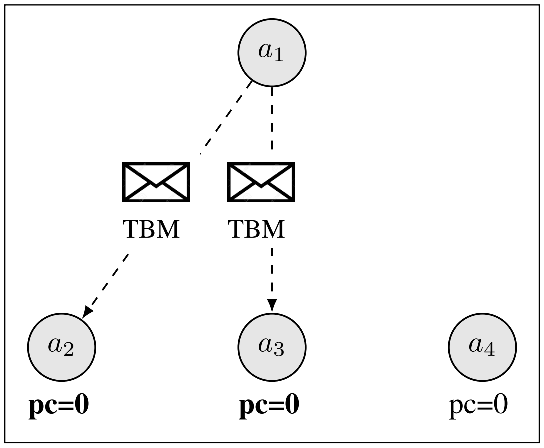
This allows us to compare our approach to a conservative locking algorithm. The conservative locking algorithm uses the cycle detection mechanism described earlier but does not consider the combination of cycles. We announce a product whenever it enters any cycle. That means the agent sends a message alongside the product flow for every product and waits until all agents that will eventually handle the product accept it (they only do if there is space in all of their cycles).



(a) Agent a_1 wants to insert a product to a cycle of which the Agents a_2, a_3 and a_4 are the other entrance agents, so it sends a *announce product message* (APM) to all of them.



(b) Agents a_2 and a_3 accept the product of a_1 and send back a *acknowledge message* (ACK) after increasing their product counter. Agent a_4 declines the product of a_1, does not increase *pc* and sends a *declining message* (DEC) to a_1.



(c) Agent a_1 sends a *take back message* (TBM) to Agents a_2 and a_3 who update their product counters correspondingly.

Fig. 5. Communication when announcing a product.

If a product is processed completely, the previously reserved capacity in all cycles is released at once. The difference between this and our approach described in Section IV is that, once a product is announced, it has to be processed completely before the cycles are considered to have capacity again. Our approach detects combined cycles and thus gets by with fewer messages. We also assume our approach to have a higher utilization as it frees reserved capacity earlier. As another benefit, we expect fewer declines for announced products.

We also run the simulations without any deadlock avoidance. In this case, only the deadlocks encountered are measured. The other properties are not comparable as we cannot quantify the cost for deadlock recovery. Measuring the deadlocks encountered gives the reader an order of magnitude on how prone to deadlocks a configuration is and finally stresses the need for deadlock control.

We formulate the following hypotheses to test in our evaluation:

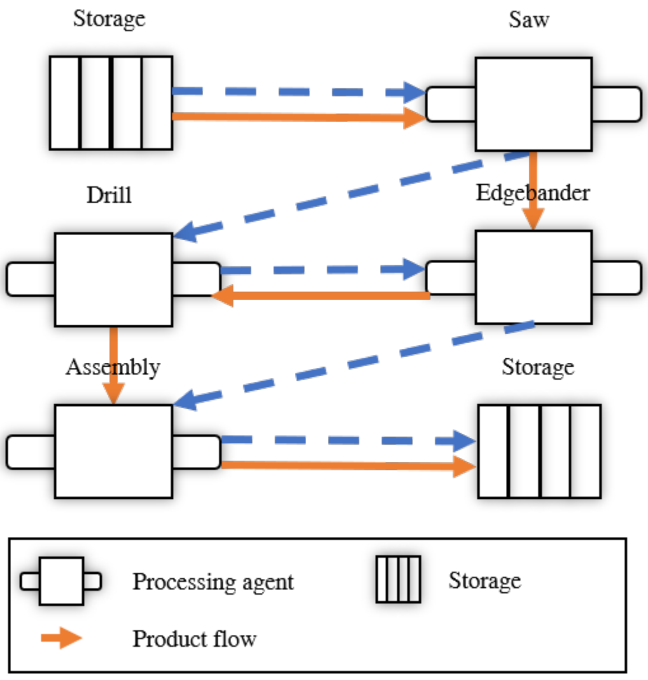
- Hypothesis 1: Both the conservative locking algorithm and our approach avoid deadlocks effectively in different configurations with more than one type of product to be manufactured.
- Hypothesis 2: To do so, our approach requires fewer messages than the conservative locking approach.
- Hypothesis 3: Our approach outperforms the conservative locking algorithm in terms of throughput.

A. Experimental setup

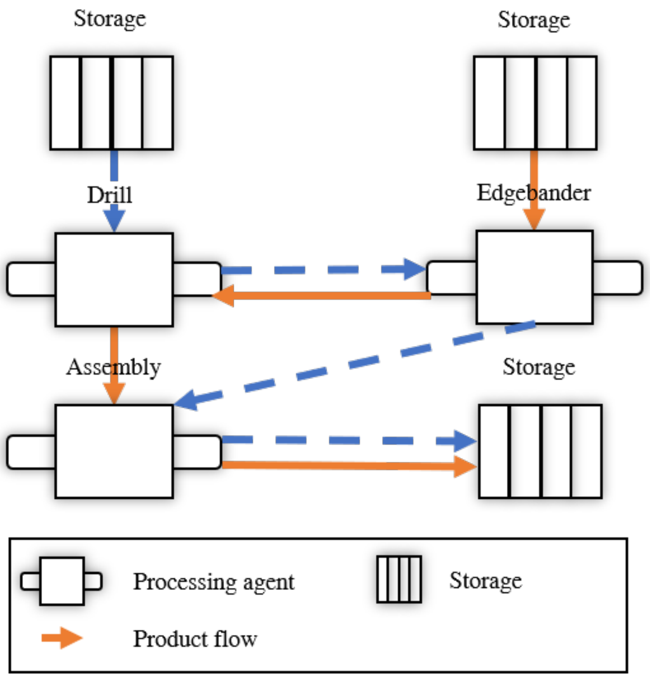
To verify our hypotheses experimentally, we examine two systems configurations depicted in Fig. 6. For each system configuration, we perform 100 runs and average the results. Each run simulates the manufacture of 100 products, while products are split equally among the existing types of products.

Both configurations draw inspiration from manufacturing furniture, where wooden panels are first taken from storage and sawn into parts. Then holes for connectors, such as dowels, are drilled. Afterward, edges are applied to cover the exposed sides. This process is termed edgebanding. Finally, the product is assembled and stored, e.g., for shipping. However, often the order of drilling and edgebanding can be interchanged. Therefore in Fig. 6a, we present a system with two such types of furniture products: The first product type requires drilling before edgebanding (dashed product flow), while the second type requires edgebanding before drilling (solid product flow). For each required capability, there is one agent, alongside two storages.

Fig. 6b introduces Configuration 2 as a variation of Configuration 1. In Configuration 2, we assume the manufacturer not to cut parts himself, but instead, buy cut parts for two different types of products from suppliers. Parts for different types of products are stored in different storages. Again, the first type of product requires drilling before edgebanding (dashed product flow), while the other product requires edgebanding before drilling (solid product flow). After assembly, all products are



(a) Configuration 1



(b) Configuration 2

Fig. 6. System configurations for the experimental evaluation. AGVs are omitted for the sake of simplicity. Note the cyclic arrangement of the two tasks between the drilling and the edgebanding machine.

stored in a third storage. Note the cyclic arrangement between the drill and the edgebander in both configurations.

B. *Experimental results*

The results for Configuration 1 in Table I support our hypotheses: While we identified 6 deadlocks on average without deadlock avoidance, conservative locking and our approach did avoid deadlocks effectively (Hypothesis 1). For this, our approach required about 70% of the runtime, the conservative locking algorithm needed, also resulting in higher throughput (Hypothesis 3). Besides, the difference in terms of message overhead is notable: While with conservative locking 915 messages were sent, our approach required only 319 messages with a standard deviation of 0 messages (Hypothesis 2).

Compared to Configuration 1, in Configuration 2 deadlocks occur more frequently if deadlock avoidance is omitted. We measured about 28 deadlocks on average with a standard deviation of 14 deadlocks. The main reason is that the two storages simultaneously add products to the cycle, while in Configuration 1 the saw could only add one product at a time. Therefore, the results for Configuration 2 are slightly more scattered, yet the analysis in Table I and Fig. 7 confirms our hypotheses as well: Our approach and the conservative locking algorithm prevent any deadlocks, reinforcing Hypothesis 1. Again, our approach requires significantly fewer messages (Hypothesis 2). While the conservative locking algorithm required 1065 messages on average, our approach got by with 416 messages on average. As for Hypothesis 3, our approach outperforms the conservative locking algorithm though the results are not as conclusive as in Configuration 1.

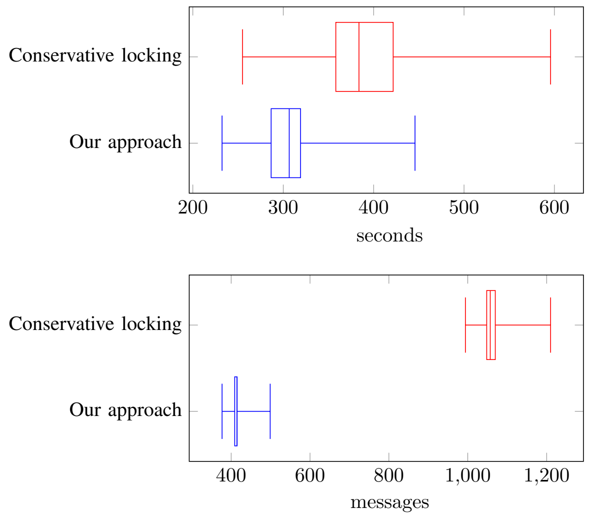


Fig. 7. In-depth comparison of conservative locking and our approach for Configuration 2. The plots show the result of 100 runs in terms of runtime (above) and messages sent (below) for the production of 100 products. Whiskers denote minimum and maximum values.

We, therefore, conclude that our hypotheses hold in the configurations considered. While the configurations considered are typical for furniture production, further investigations must show whether our results also apply to other domains.

TABLE I. EXPERIMENTAL RESULTS OF THE COMPARATIVE EVALUATION OF THE CONSERVATIVE LOCKING AND OUR APPROACH. FOR EACH CONFIGURATION AND STRATEGY, THE MEASUREMENTS OF 100 RUNS, EACH SIMULATING THE MANUFACTURING OF 100 PRODUCTS, ARE AVERAGED.

Configuration	Strategy	No. of deadlocks: mean (std)	Runtime in seconds: mean (std)	Throughput in products/s: mean (std)	No. of messages sent: mean (std)
1	No deadlock avoidance	6.11 (3.77)	-	-	-
	Conservative locking	0 (0)	218 (25)	0.46 (0.05)	915 (0)
	Our approach	0 (0)	155 (14)	0.65 (0.06)	319 (0)
2	No deadlock avoidance	27.80 (14.14)	-	-	-
	Conservative locking	0 (0)	393 (61)	0.26 (0.04)	1065 (36)
	Our approach	0 (0)	312 (44)	0.33 (0.04)	416 (21)

VI. CONCLUSION

In this paper, we consider the problem of deadlocks in self-organizing production systems. Deadlocks can occur in production systems if there is a cycle in the product flow, and each agent in this cycle is occupied by a product. Therefore, we present an approach that detects cycles in the system's product flows and controls the number of products in each cycle. Detecting cycles and controlling the number of products is realized by sending messages, hence our approach works without central knowledge or control. Furthermore, our approach detects cycles arising from the combination of several product flows and thus is suitable for production systems with several types of products to be manufactured in parallel.

To demonstrate the effectiveness of our approach, we conduct several experiments measuring the time to manufacture a fixed number of products, the number of deadlocks encountered, and the number of messages sent. Our evaluation indicates that our approach avoids deadlocks in various system configurations. Compared to a conservative locking algorithm, our approach requires considerably less message and time overhead. Therefore, systems using our approach can achieve higher throughput.

In future experiments, we plan to investigate the effect of adding buffers in front of and after every agent. Buffers are well studied in literature and known to increase the decoupling of machines [21]. Another approach towards the problem is considering deadlocks in task allocation. Adding hard constraints to the CSP to prevent cyclic arrangements is not desirable. But the idea of adding soft constraints to prefer configurations without cycles seems like a promising way to relax the problem.

Finally, we strive for a theoretical proof that verifies the deadlock avoidance property of our approach and confirms our simulation results.

REFERENCES

[1] Y. Koren, *The global manufacturing revolution: product-process-business integration and reconfigurable systems*. John Wiley & Sons, Ltd, 2010.

[2] Y. Chevaleyre, U. Endriss, J. Lang, P. Dunne, M. Lemaitre, N. Maudet, J. Padget, S. Phelps, J. Rodriguez-Aguilar, and P. Sousa, "Issues in multiagent resource allocation," *Informatica*, vol. 30, pp. 3–31, 2006.

[3] L. Monostori, J. Váncza, and S. Kumara, "Agent-based systems for manufacturing," *CIRP Annals*, vol. 55, no. 2, pp. 697 – 720, 2006.

[4] A. Dorri, S. S. Kanhere, and R. Jurdak, "Multi-agent systems: A survey," *IEEE Access*, vol. 6, pp. 28 573–28 593, 2018.

[5] P. Leitão, J. Barbosa, and D. Trentesaux, "Bio-inspired multi-agent systems for reconfigurable manufacturing systems," *Engineering Applications of Artificial Intelligence*, vol. 25, no. 5, pp. 934 – 944, 2012.

[6] L. Ribas-Xirgo, J. M. Moreno-Villafranca, and I. F. Chaile, "On using automated guided vehicles instead of conveyors," in *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, 2013, pp. 1–4.

[7] S. C. Brailsford, C. N. Potts, and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," *European Journal of Operational Research*, vol. 119, no. 3, pp. 557 – 581, 1999.

[8] J.-P. Steghöfer, P. Mandrekar, F. Nafz, H. Seebach, and W. Reif, "On deadlocks and fairness in self-organizing resource-flow systems," in *Architecture of Computing Systems - ARCS 2010*, C. Müller-Schloer, W. Karl, and S. Yehia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 87–100.

[9] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Computing Surveys (CSUR)*, vol. 3, no. 2, pp. 67–78, 1971.

[10] H. Seebach, F. Ortmeier, and W. Reif, "Design and construction of organic computing systems," in *2007 IEEE Congress on Evolutionary Computation*, Sep. 2007, pp. 4215–4221.

[11] H. Seebach, F. Nafz, J.-P. Steghöfer, and W. Reif, "How to design and implement self-organising resource-flow systems," in *Organic Computing—A Paradigm Shift for Complex Systems*. Springer, 2011, pp. 145–161.

[12] M. P. Fanti and M. Zhou, "Deadlock control methods in automated manufacturing systems," *IEEE Transactions on systems, man, and cybernetics-part A: systems and humans*, vol. 34, no. 1, pp. 5–22, 2004.

[13] M. Singhal, "Deadlock detection in distributed systems," *Computer*, vol. 22, no. 11, pp. 37–48, 1989.

[14] Z. A. Banaszak and B. H. Krogh, "Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows," *IEEE Transactions on robotics and automation*, vol. 6, no. 6, pp. 724–734, 1990.

[15] N. Wu and M. Zhou, "Modeling and deadlock avoidance of automated manufacturing systems with multiple automated guided vehicles," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 35, no. 6, pp. 1193–1202, 2005.

[16] M. P. Fanti, B. Maione, S. Mascolo, and A. Turchiano, "Event-based feedback control for deadlock avoidance in flexible production systems," *IEEE Transactions on Robotics and Automation*, vol. 13, no. 3, pp. 347–363, 1997.

[17] M. P. Fanti, "Event-based controller to avoid deadlock and collisions in zone-control agvs," *International Journal of Production Research*, vol. 40, no. 6, pp. 1453–1478, 2002.

[18] A. Boukerche and C. Tropper, "A distributed graph algorithm for the detection of local cycles and knots," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 8, pp. 748–757, 1998.

[19] R. A. Wysk, N.-S. Yang, and S. Joshi, "Detection of deadlocks in flexible manufacturing cells," *IEEE Transactions on robotics and automation*, vol. 7, no. 6, pp. 853–859, 1991.

[20] H. Cho, T. Kumaran, and R. A. Wysk, "Graph-theoretic deadlock detection and resolution for flexible manufacturing systems," *IEEE Transactions on Robotics and Automation*, vol. 11, no. 3, pp. 413–421, 1995.

[21] Y. Dallery and S. B. Gershwin, "Manufacturing flow line systems: a review of models and analytical results," *Queueing Systems*, vol. 12, no. 1, pp. 3–94, Mar 1992.