On deadlocks and fairness in self-organizing resource-flow systems

Jan-Philipp Steghöfer, Pratik Mandrekar, Florian Nafz, Hella Seebach, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Steghöfer, Jan-Philipp, Pratik Mandrekar, Florian Nafz, Hella Seebach, and Wolfgang Reif. 2010. "On deadlocks and fairness in self-organizing resource-flow systems." In *Architecture of Computing Systems - ARCS 2010: 23rd International Conference, Hannover, Germany, February 22-25, 2010, proceedings*, edited by Christian Müller-Schloer, Wolfgang Karl, and Sami Yehia, 87–100. Berlin: Springer. https://doi.org/10.1007/978-3-642-11950-7_9.

On Deadlocks and Fairness in Self-organizing Resource-Flow Systems*

Jan-Philipp Steghöfer[1], Pratik Mandrekar[2], Florian Nafz[1], Hella Seebach[1], and Wolfgang Reif[1]

[1] Universität Augsburg, Institute for Software- & Systems-Engineering, Augsburg, Germany
{steghoefer,nafz,seebach,reif}@informatik.uni-augsburg.de
[2] Birla Institute of Technology and Science Pilani, Goa, India
pratik.mandrekar@gmail.com

Abstract. Systems in which individual units concurrently process indivisible resources are inherently prone to starvation and deadlocks. This paper describes a fair scheduling mechanism for self-organizing resource-flow systems that prevents starvation as well as a distributed deadlock avoidance algorithm. The algorithm leverages implicit local knowledge about the system's structure and uses a simple coordination mechanism to detect loops in the resource-flow. The knowledge about the loops that have been detected is then incorporated into the scheduling mechanism. Limitations of the approach are presented along with extension to the basic mechanism to deal with them.

1 Introduction

In a resource-flow system agents handle resources by receiving them from another agent, processing them and handing them over to another agent. An instance of this are flexible manufacturing systems. If agents can only process one resource at a time and there are no buffers available, such systems are prone to deadlocks. If the agents are arranged in a way that an agent can receive a resource it had already processed before, the resource has been processed by agents arranged in a loop. This loop can fill up with resources, in a way that the agent can not give its currently held resource to another agent and can thus not accept a new resource. Such a situation is depicted in Fig. 1.

This paper introduces a deadlock avoidance mechanism that leverages the implicit knowledge available to the agents in self-organizing resource-flow systems modelled with the Organic Design Pattern (ODP) [16] and incorporates the mechanism into a fair scheduler for this system class. The local knowledge of the agents along with the structure that is given by the definition of the resource flow allows the algorithm to be efficient both in terms of messages sent and in computational time required while effectively avoiding deadlocks and preventing

* This research is partly sponsored by the priority program "Organic Computing" (SPP 1183) of the German research foundation (DFG) in the project SAVE ORCA.

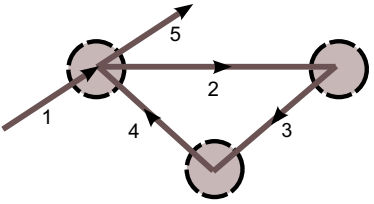

Fig. 1. A situation prone to deadlocks induced by cyclic resource processing

starvation without the need to globally analyse the system or trace the state of all agents during runtime.

The paper is structured as follows: Sect. 2 defines the terms associated with deadlocks and introduces how deadlocks can be dealt with. Sect. 3 shows how liveness hazards are dealt with in literature and Sect. 4 describes the system class the proposed approach is applicable to. Sect. 5 then introduces a fair role-selection algorithm that is extended with a deadlock avoidance mechanism in Sect. 6. Limitations of this mechanism and ways to overcome them are outlined in Sect. 7 and Sect. 8 concludes the paper.

2 Background and Definitions

There are a number of different liveness hazards that can occur in concurrent systems. The definitions of these hazards often vary a great deal between different authors and domains. Therefore, the following brief definitions introduce the terms as they are used in this paper.

Deadlocks. A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does. Coffman et al. [10] give four conditions that must hold for a deadlock to occur:

1. "Mutual exclusion" i.e., agents claim exclusive control of the resource they require;
2. "No preemption" i.e., resources cannot be forcibly removed from the agents holding them until processing of the resources is completed;
3. "Wait for condition" i.e., Agents hold resources allocated to them, while waiting for additional ones;
4. "Circular wait" i.e., a circular claim of agents exists, such that each agent holds one or more resources that need to be given to another agent in the claim.

Livelock. A liveness failure is referred to as a livelock when an agent although not blocked cannot proceed further and keeps retrying. As an example consider two mobile agent's who obstruct each others paths. Both decide to give way to the other at the same time and move to the right leading again to an obstruction. Now both decide at the same time to move to the left and the process keeps repeating leading to no agent moving forward. Another example is the 'after you after you' politeness protocol.

Starvation. Starvation occurs when an agent is continuously denied access to a resource. Bustard et al. illustrate the starvation problem using the dining philosophers problem in the field of autonomic computing [8]. It can also be characterized as a special case of livelock [21].

Fairness. Fairness in its most general definition guarantees that a process will eventually be given enough resources to allow it to terminate [26]. This is usually achieved by implementing a scheduling mechanism. Especially in distributed systems that share resources, the ability to terminate may depend on other components or on interacting with them. Fairness is classified into weak and strong fairness where weak fairness implies that a component will eventually proceed if it can almost always proceed and strong fairness implies that a component which can proceed infinitely often does proceed infinitely often [11].

There are three different ways on how potential deadlocks can be dealt with. Their applicability depends on the structure of the systems under consideration and the assumptions that can be made about them.

Deadlock Prevention prohibits circular waits among agents while the system is not running. The strength of the approach is also its greatest weakness: a prevention mechanism requires global knowledge of the system and all its reachable states which often yields a straightforward often-simple control law but acquiring this knowledge can be difficult and computationally intractable. Additionally, limiting concurrency to prevent any deadlock state from occurring can be overly conservative, potentially leading to low utilization of system resources.

Deadlock Avoidance suitable governs the resource flow to prevent circular waits. This is a dynamic approach that can utilize the knowledge of the current allocation of resources and the future behaviour of processes to control the release and acquisition of resources. The main characteristic of deadlock avoidance strategies is that they work during runtime and base their decisions on information about the resource and agent status. More precisely, a deadlock controller inhibits or enables events involving resource acquisition or release by using look-ahead procedures.

Deadlock Detection and Resolution monitors the agents and/or the flow of resources and detects deadlocks at runtime. Resources can then be removed from the system or put into buffers to resolve the deadlock. This strategy can in general allow higher resource utilization than that prevention or avoidance methods can offer. However, it should only be used when deadlock is rare and detection and recovery are not expensive and much faster than deadlock generation.

3 Related Work

Deadlocks have been dealt with in several domains that are concerned with software-systems, e.g. deadlocks which are inherent in databases that use locking

mechanisms [5] and involve detecting cycles in the wait-for graphs when the dependency relations between agents are known locally by using serialization techniques or resource allocation logic in operating systems [10]. However, the primary realm of related work that is relevant for the self-organizing resource-flow systems considered in this paper stems from the domain of manufacturing control and Flexible Manufacturing Systems (FMS). For an overview of such approaches, see, [14].

Several proposed **Deadlock Prevention** strategies are based on the use of transition systems, mostly Petri nets. [19] provides a mechanism to generate an automated supervisor to check for deadlocks in the given Petri nets and hence prevent them from occurring. A survey of techniques using Petri nets has been made by Zhi Wu Li et al. [23] with respect to AGV (Automated Guided Vehicle) systems in manufacturing. Siphon based policies are shown to achieve a good balance of the best of all properties of the Petri net based methods [18]. However all these techniques rely on known interactions between the agents (mostly the AGVs in the referred cases) and restrict transitions to states that could lead to a deadlock based on a universal control policy generated during Petri net analysis.

The methods discussed above all require a global view of the system, offline computation or computation during the design phase. The deadlock prevention approach establishes the control policy in a static way, so that, once established, it is guaranteed that the system cannot reach a deadlock situation if the system is not changed [13]. Serial execution of the processes on the resource like scheduling in the case of software processes or threads involves assumptions or knowledge of the time of execution which might not be available. The computations required can become infeasible for large scale systems.

Deadlock Avoidance mechanisms using Petri nets [33,17] restrict the maximal number of parts that can be routed into the system. The Petri nets are either constructed upfront or constructed dynamically which involves a communication overhead. A Petri net allows to check at each state whether the next state would lead to a deadlock or not. This analysis can be used to obtain suitable constraint policies for the system [3,33]; a similar mechanism has also been utilized for deadlocks in software programs [30] using supervisory control of Petri nets. A resource-allocation graph can be used to construct a local graph to check for deadlocks based on a global classification of agents into deadlock risk and deadlock free agents and combines the restrictions of deadlock prevention with the flexibility of deadlock avoidance [34]. [6] also uses a resource-allocation graph in which resource allocation and release is modelled by the insertion and deletion of edges. Deadlock avoidance is achieved by prohibiting actions that would lead to the insertion of an edge that would make the graph cyclic. If the components of a system know the nature of all the jobs they will be performing and all the components they will be communicating with in advance, deadlocks can be avoided in a distributed fashion [27].

Deadlock Detection and Resolution [28] often involves construction of a wait-for graph [7]. A wait-for graph is a graph to track which other agent an agent is currently blocking on. Maintaining such a global picture or agent state involves

a lot of communication overhead while locally distributed deadlock detection schemes like Goldman's Algorithm [15] pass information locally to agents in the form of tables or other data sets. Ashfield et al. [2] provide dedicated agents for deadlock initiation, detection and resolution in a system with mobile agents. Deadlock Recovery Mechanisms using buffers [32] or rollback propagation [31] have been proposed.

Livelocks are closely related to deadlocks [1] but have not been dealt with as rigorously as deadlocks have been. A combination of different formal methods is used in [9] to check a system for livelock freedom at design time. Similarly, [20] introduces a type system that can be used to prove that processes and their communication are deadlock- and livelock-free.

In [22], the author examines different characterizations of **fairness** in trace-based systems and explores the practical usefulness of generalized fairness properties. [4] introduces least fairness, a property that is sufficient to show freedom of starvation as well as the concept of conspiracy that describes how concurrent processes interact to induce starvation. Parametric fairness, a fairness measure based on a probabilistic model is used in [29] to show that most executions of concurrent programs are fair. Markov Models are used to study unfairness and starvation in CSMA/CA protocols [12].

4 Self-organising Resource-Flow Systems

In order to analyse, specify, model, and construct self-organising resource-flow systems, the Organic Design Pattern (ODP) has been developed [16]. It describes self-organizing systems in terms of agents that process resources by applying capabilities according to roles. Roles are data structures that contain a precondition, a list of capabilities to apply and a postcondition ODP is part of a methodology that contains a design guideline, the pattern description with its static and dynamic parts as well as methods and tools for formally verifying properties of systems modelled with ODP. In the following, only the relevant parts of the pattern will be described. More information can be found in the cited resources.

4.1 Set-Based Description of ODP

An ODP system consists – among other things – of a set *agents* of the agents in the system, a set *capabilities* which is the set of all capabilities in the system and a set *roles* which denotes the set of all possible roles in the system. An agent $ag \in agents$ consists of two sets of agents ($input_{ag}$ and $output_{ag}$), the former one defining the agents from which ag may accept resources and the latter one to which ag may give resources, as well as a set cap_{ag} of capabilities the agent can apply and a set $roles_{ag}$ which determine the function of the agent in the system. A condition c is an element of the set *conditions*, containing an agent to exchange a resource with, a sequence of capabilities[1] that have been applied to a

[1] In this definition, capabilities is treated as an alphabet whose symbols can form words which describe the state, task and capabilities to apply to a resource.

resource and a sequence of capabilities that have to be applied to an agent (also called the task to be performed on the resource). A role $r \in roles$ is composed of a precondition that describes where a resource comes from, its state and task when it is accepted by the agent, a sequence of capabilities to apply on the resource, and a postcondition that describes to which agent the resource has to be given and its state and task after processing.

ODP systems are reconfigured by changing the allocation of roles to agents. Whenever an agent can not fulfil its capabilities any more or is no longer available, a new allocation of roles to the agents is calculated (e.g. by employing constraint satisfaction techniques [25]) and the newly calculated roles are adopted by the agents. This ensures that as long as all the capabilities that are required to process resources are still available in the system, a valid configuration can be found.

The role that has been applied to process a resource by agent ag_{proc} is also used to determine which agent ag_{next} it has to be given to. ag_{proc} sends ag_{next} a message informing it of the intent to transfer a resource. The message contains information about the resources status that can be used by the ag_{next} to determine the role whose precondition fits the resource status and the sending agent. The role that has been derived is then stored in a map in which the ag_{proc} is the key and the role is the value. The set of values in this map is used at a later point to select the next role to be executed by ag_{next}. Sect. 6.2 describes the message reception and role selection process formally.

4.2 Liveness in Self-organising Resource-Flow Systems

Preventing starvation of an agent is straight-forward: it has to be ensured that the resource the agent holds is eventually taken by another agent. This can be ensured by a role-selection mechanism that will eventually select each applicable role. Dealing with deadlocks, however, is a more complicated matter.

Systems with a flow of physical resources are prone to deadlocks whenever they are in a configuration that establishes a cycle in the resource-flow graph. In Sect. 2, four different characteristic conditions for deadlocks were mentioned. The first three of these (mutual exclusion, no preemption, and wait-for condition) are always true in ODP systems as resources can not be shared between agents but are claimed exclusively by an agent for processing, processing of a resource is always completed before it is available to another agent, and an agent has to wait for a subsequent agent to take the resource away before it can process another one. Thus, the only way to deal with deadlocks in ODP systems is to break the "circular wait" property. A circular wait occurs when two or more agents are arranged in a loop and each agent waits for the subsequent agent to take the resource it has been offered.

All centralized approaches of deadlock prevention and avoidance suffer from the same limitations: pre-calculating all states a system can reach is very expensive and in many cases computationally infeasible even for relatively small cases. Supervising the state of all agents involves a massive amount of communication and limits the agents' autonomy considerably. Additionally, ensuring a

consistent state to make decisions about applicable state transitions is by no means simple and may require the introduction of synchronization between agents, thus severely reducing system performance. A centralized solution also always suffers from standard problems: it is a bottleneck and a single point of failure. Finally, a distributed solution can be employed regardless of an existing centralized entity, that, e.g., handles reconfiguration.

Deadlock detection might be feasible in resource-flow systems as there are distributed mechanisms available (see [28] for a survey). However, the resolution mechanisms proposed are hardly adaptable to resource-flow systems. Rollback propagation [31] is not applicable since resources undergo an irreversible set of operations. Use of buffers [32] has to be avoided as in many applications agents can not store resources and it can not be assumed that there are spare agents to temporarily hold resources. A conceivable mechanism for deadlock resolution is to let an agent dump the resource it currently holds, which is undesirable because resources are potentially valuable and should be processed to completion.

5 A Fair Role-Selection Mechanism

The main purpose of a fair role-selection mechanism is to avoid starvation. As described above, starvation occurs if an agent is waiting for a resource to be taken by another agent and this never happens. In ODP terminology, the agent ag_{rec} that is the receiver of the resource never executes the role r_x that would take the resource from the sending agent ag_{sen}. The scheduling algorithm described below thus ensures that each role that is applicable is eventually executed.

Definitions and Initialization: An agent contains the following data structures: a map of roles associated to agents that sent a request to the current agent $requests \subseteq agents \times roles$, a map $applicationTimes \subseteq roles \times \mathsf{N}$ that stores the value of the counter at the point in time when a role has last been executed and contains one value per role $(\forall(r_1, t_1), (r_2, t_2) \in applicationTimes : r_1 = r_2 \rightarrow t_1 = t_2)$, and a number $counter \in \mathsf{N}$ that counts the number of times the agent executed a role. These structures are initialized as $requests = \emptyset$, $applicationTimes = \{\forall r \in roles|(r, 0)\}$, and $counter = 0$.

Reception of Message: Whenever an agent ag_{sen} is ready to transfer a resource to an agent ag_{rec}, ag_{sen} sends a message $m = (ag_{sen}, c)$ where c is the postcondition of the role that ag_{sen} executed. ag_{rec} executes a function $getRole : agents \times conditions \rightarrow roles^+$ as $appRoles = getRole(ag_{sen}, c)$ where

$$\begin{aligned} getRole(ag_{sen}, c) = \{r : r \in roles_{ag_{rec}} \wedge \\ r.precondition.from = ag_{sen} \wedge \\ r.precondition.state = c.state \wedge \\ r.precondition.task = c.task\} \end{aligned}$$

and updates the $requests$ set: $requests = requests \cup \{(ag_{sen}, r)\} \forall r \in appRoles$. Each tuple in $requests$ is also associated with the timestamp of its reception which can be retrieved by $ts : agents \rightarrow \mathsf{N}$.

Choose a Role: Whenever ag_{rec} has to decide which role to execute next, it goes through *requests* and applies for each of the roles stored in it a fitness function $f : Roles \rightarrow \mathsf{N}$. In its simplest form, f is instantiated as $f(r) = (counter - t)$ where $(r, t) \in applicationTimes$. This will yield the highest value for the role that has not been executed the longest, thus ensuring that each role that is applicable will eventually be executed if $f(r)$ is maximized.

Formally, the selection of the role to execute is done by evaluating a function $max : (agents \times roles)^+ \rightarrow (agents \times roles)$ which yields $(a, r) = max(requests)$ where

$$max(requests) \;=\; (ag, r_{next}) \text{ if } \forall (ag\,, r\,) \in requests : \\ f(r\,) < f(r_{next}) \vee (f(r\,) = f(r_{next}) \;\wedge\; ts(ag) \leq ts(ag\,))$$

The agent then chooses r_{next} as the next role to execute. The incorporation of ts ensures that even if two or more requests evaluate to same fitness value, max yields an unambiguous result, namely the oldest request.

Update: Once a role has been selected for execution, the agent needs to update its structures to reflect the changes by setting $t = counter$ where $(r_{next}, t) \in applicationTimes$, $requests = requests \setminus \{(ag, r_{next})\}$, $counter = counter + 1$.

6 Distributed Deadlock Avoidance

The basic principle of the deadlock avoidance algorithm is very simple: If a loop is detected, determine the size of the loop n and allow only $n - 1$ resources to be processed by the n agents that are part of the loop at any one time.

The algorithm described below, however, constitutes a kind of local simulation of the surroundings of an agent and works in a distributed fashion. It is therefore applicable for systems with a centralized **and** a distributed configuration mechanism. Thus, it is more versatile than a centralized loop detection mechanism, but suffers some limitations because of its distributed nature and the limited knowledge that is available at the agents.

6.1 Distributed Loop Detection

The algorithm is split in two parts: The first part is run locally on each agent after it has received a new role allocation (i.e. after it has been reconfigured). It determines whether the agent **potentially** is the entry of a loop. The second part then verifies this assumption by sending a message that will eventually reach a sink – meaning that the agent is not part of a loop – or return to the sending agent – meaning of course that the agent is part of a loop.

Loop Estimation: Define a set $S : roles \times roles$ as follows:

$$S = \{(r_i, r_j) \mid r_i.precondition.state \sqsubseteq r_j.precondition.state \wedge \\ r_i.precondition.task = r_j.precondition.task \wedge \\ r_i, r_j \in roles \wedge r_i \neq r_j\}$$

where '$\sqsubseteq$' is the list prefix operator. Also define a function min as

$$min(S) = abs(|p_q| - |p_r|), \text{ if } \forall\, (p_k, p_l) \in S : abs(|p_q| - |p_r|) \;\leq\; abs(|p_k| - |p_l|)$$

where $p_i = r_i.precondition.state$, for $i \in \{k, l, q, r\}$. This allows to determine whether a loop exists and give an estimate of the length of the loop:

$$|S| \geq 1 \Rightarrow loop = \texttt{true} \wedge n_{est} = min(S)$$

This algorithm also yields an estimate n_{est} for the number of agents in the loop that is an underestimate when we assume that each agent applies at most one capability.

Loop Determination: To determine the actual size of the loop and to eliminate potential loops the agent ag_{org} creates a message m that has a list of agent identifiers m_{agents} initialized with the identifier of ag_{org}, a boolean flag m_{sink} that determines if the message has reached a sink and that is initialized with `false`, the postcondition of role r_j in $m_{condition}$, and the tuple $(r_i, r_j) \in S$ in m_{roles}. The message is sent to the agent in the output of the condition[2].

When an agent ag_i receives a loop-detection message, it uses the $getRole()$ function to find the role that would be chosen if the sending agent had delivered a request to take a resource. ag_i then adds its own identifier to m_{agents}. If ag_i is not a sink, i.e. the role chosen by ag_i does contain a postcondition, it replaces $m_{condition}$ by the postcondition of the selected role and forwards the message to the agent that is set as the output in the chosen role. In case ag_i is a sink, it sets m_{sink} to true and returns the message to its originator, i.e., the first entry in m_{agents}[3].

Eventually, the message will return to the original sender ag_{org}, either because it passed through the loop (in which case m_{sink} will be `false`) or because it reached a sink and was returned. If a sink was reached, a loop does not exist and S can be updated: $S = S \setminus \{m_{roles}\}$. Otherwise, the actual length of the loop is determined by counting the elements in m_{agents} and updating the agent-global n that contains the number of agents in the smallest loop the agent is part of. n is updated only if the number of elements in m_{agents} is less than the current n.

The loop determination part of the algorithm terminates when all messages send out by ag_{org} have returned to ag_{org}.

6.2 Extension of the Scheduling Mechanism to Avoid Deadlocks

If a minimal loop size n has been calculated, the role-selection algorithm presented in Sect. 5 can be extended as follows:

[2] We assume that agents operate in a stable communication environment and message loss is thus not considered.

[3] In case direct communication is not possible, the message is passed back along the way it reached the sink. If an agent ag_j receives a message with m_{sink} set to `true`, it looks up the agent that is in front of ag_j's agent identifier in m_{agents} and sends the message to this agent.

1. Define $executions : roles \times roles \times \mathsf{N}$
2. Initialize $executions = \{\forall s \in S | (s, 0)\}$
3. Choose:
$$f\ (r) = \begin{cases} 0, & \text{if } \exists((r_i, r_j), x) \in Executions : r_i = r \wedge x \geq n-1 \\ f(r), & \text{otherwise} \end{cases}$$
4. Update:
$$\forall((r_i, r_j), x) \in executions : \begin{cases} x = x + 1 & \text{if } r_{next} = r_i \\ x = x - 1 & \text{if } r_{next} = r_j \end{cases}$$

The fitness function $f\ (r)$ ensures that role r_i is only executed if the difference between the number of times r_i and r_j have been executed is less than $n-1$. If role r_i is not executed because of this condition, the entry in *requests* will be evaluated again the next time a role is selected. If a resource leaves the loop (i.e., r_j is executed), the difference becomes less than $n-1$ and r_i is eligible for execution again. Loop-induced deadlocks are thus avoided while the fairness guarantees are still upheld.

If $\forall(ag, r) \in requests : f\ (r) = 0$ (i.e., no role can be executed at the moment without causing a deadlock), the agent stays idle until *requests* is updated by a new request and the role-selection algorithm is executed again.

7 Alternatives and Extensions

The basic mechanism described above works for many systems. However, there are situations that can not be dealt with as easily. This section describes these situations and how the mechanism can be adapted to accommodate more complex system structures and concurrent, distributed reconfiguration.

7.1 Dealing with Distributed Reconfiguration

Usually, the cycle detection algorithm would be executed whenever an agent receives a new configuration. However, in some systems with a distributed reconfiguration algorithm, the agents of a cycle might be reconfigured without the agent at the entry of the cycle even noticing. The process might break the cycle, change the number of agents in it or alter the cycle's structure in other, unforeseen ways. If the agent at the entry point upholds its deadlock avoidance strategy, there might thus be deadlocks occurring.

To counter this kind of situation, two strategies are conceivable. Firstly, after a reconfiguration occurred, information about this can be distributed to adjacent nodes and the algorithm can be run after receiving this information. Secondly, the agent at the entry point just runs the cycle detection periodically. As the algorithm requires only few messages and very little computational effort, this approach seems more efficient and also avoids problems with consistency and information dissemination in large-scale systems.

7.2 More Than One Entry Point into a Loop

A situation where a loop has several entry points is depicted in Fig. 2. It is most likely to occur when some agents of the loop do not apply a capability but just forward the resource and if there are two independent upstream resource processing lines that produce resources with the same state.

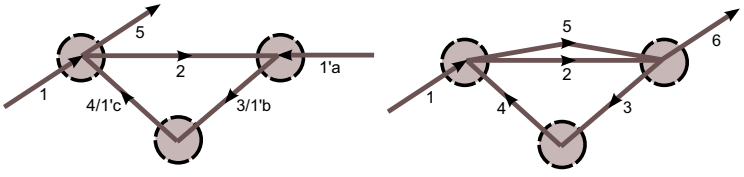

Fig. 2. Examples of loops of three agents and two entry points

It is possible that resources enter the loop at two different points thus causing a congestion. Such a situation can only be remedied if the two agents that let resources into the loop are cooperating to limit the number of resources processed in the loop. The requirement can be described as follows: if a loop consists of n agents and has $n - k$ entries where $n - 1 < k \geq 0$, the agents at the entries have to coordinate to ensure that a maximum of $n - 1$ resources are within the loop.

If an agent ag_i sent out a loop detection message and at a later point receives such a message originating from another agent ag_j, ag_i can suspect that a_j is an entry point to the same loop. After ag_i and ag_j determined they are part of a loop, they can exchange information about the loops they are part of and thus establish their relationship. During productive phases, the agent at the exit of the loop (say, ag_i) can award the other agent (ag_j) a quota of resources a_j may allow into the loop. After a resource has been allowed into the loop, ag_j will inform ag_i. When ag_j's quota is exhausted and ag_i has detected that resources exited the loop again, ag_j will be granted a new quota.

Although it is suspected that this simple coordination mechanism scales well for loops with more than two entry points, an investigation of such situations as well as a detailed description of the mechanism will be left as future work.

7.3 Bifurcations

An agent can have two roles with the same precondition but different postconditions, e.g. to balance the load for the successive agents. In such a case, the resource-flow graph bifurcates and the loop-detection message has to be sent to both outputs of both postconditions. As the originating agent ag_{org} now has to expect more than one message as a reply to its cycle detection message, it has to be informed about the bifurcation. ag_{org} then waits until the original message

and all bifurcated message return. It is then able to establish the length of the minimal loop it is part of by the mechanism described above.

This theoretically rather simple procedure becomes tedious when implemented. The main problem is that it is not certain when a message will arrive at a_{org}. If the message that went through a cycle arrives before the message that indicates a bifurcation, the agent might have finished its cycle determination already. Special cases might have to be introduced to deal with such a situation.

7.4 Multiple Tasks

If resources with several tasks are processed in a system simultaneously, circular waits can now be induced by roles that are dealing with different tasks as shown in Fig. 3. However, such waits do not necessarily occur in every system with more than one task. If resources flow only in one direction through the system or if each agent is configured to handle only one task, the proposed approach is still applicable. In the general case, an extended loop detection mechanism would be able to detect circular waits induced by the processing of resources with different tasks and reduce this problem to the case of loops with several entry points. This, however, is left as future work.

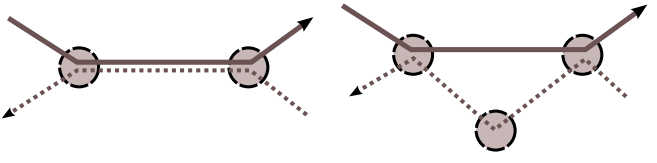

Fig. 3. Cycles induced by roles dealing with multiple tasks

8 Discussion and Conclusion

This paper described mechanisms for fair scheduling and deadlock avoidance based on local knowledge and minimal communication that are suitable for large scale dynamic system with a changing structure. In comparison to the approaches in literature this method is not limited by the computational time that is required to construct and simulate a global graph, works during runtime of the system and specifies the concrete application of the knowledge about potential deadlocks in the context of the scheduler. It also does not involve a massive amount of message passing and does not undermine the self-organizing and autonomous properties of a system and its entities respectively.

Although some details of extensions for more complex system structures are not yet fully elaborated, the approach already proves useful and has been implemented in the ODP Runtime Environment [24] on a case study of an adaptive production cell. The preliminary results are very encouraging and the open issues will be evaluated and solved with the help of this implementation.

References

1. Abate, A., Innocenzo, A.D., Pola, G., Di Benedetto, M.D., Sastry, S.: The Concept of Deadlock and Livelock in Hybrid Control Systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 628–632. Springer, Heidelberg (2007)
2. Ashfield, B., Deugo, D., Oppacher, F., White, T.: Distributed Deadlock Detection in Mobile Agent Systems. In: Hendtlass, T., Ali, M. (eds.) IEA/AIE 2002. LNCS (LNAI), vol. 2358, pp. 146–156. Springer, Heidelberg (2002)
3. Banaszak, Z.A., Krogh, B.H.: Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows. IEEE Transactions on Robotics and Automation 6(6), 724–734 (1990)
4. Bandyopadhyay, A.K.: Fairness and conspiracy concepts in concurrent systems. SIGSOFT Softw. Eng. Notes 34(2), 1–8 (2009)
5. Barbosa, V.C.: Strategies for the prevention of communication deadlocks in distributed parallel programs. IEEE Transactions on Software Engineering 16(11), 1311–1316 (1990)
6. Belik, F.: An efficient deadlock avoidance technique. IEEE Transactions on Computers 39(7), 882–888 (1990)
7. Bracha, G., Toueg, S.: Distributed deadlock detection. Distributed Computing 2(3), 127–138 (1987)
8. Bustard, D., Hassan, S., McSherry, D., Walmsley, S.: GRAPHIC illustrations of autonomic computing concepts. Innovations in Systems and Software Engineering 3(1), 61–69 (2007)
9. Buth, B., Peleska, J., Shi, H.: Combining Methods for the Livelock Analysis of a Fault-Tolerant System. In: Haeberer, A.M. (ed.) AMAST 1998. LNCS, vol. 1548, p. 124. Springer, Heidelberg (1998)
10. Coffman, E.G., Elphick, M.J.: System deadlocks. Computing Surveys (1971)
11. Costa, G., Stirling, C.: Weak and strong fairness in CCS. Information and Computation 73(3), 207–244 (1987)
12. Durvy, M., Dousse, O., Thiran, P.: Self-Organization Properties of CSMA/CA Systems and Their Consequences on Fairness. IEEE Transactions on Information Theory 55(3), 1 (2009)
13. Ezpeleta, J., Colom, J.M., Martinez, J.: A Petri net based deadlock prevention policy for flexible manufacturing systems. IEEE Transactions on Robotics and Automation 11(2), 173–184 (1995)
14. Fanti, M.P., Zhou, M.C.: Deadlock control methods in automated manufacturing systems. IEEE Transactions on Systems, Man and Cybernetics, Part A 34(1), 5–22 (2004)
15. Goldman, B.: Deadlock detection in computer networks. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1977)
16. Güdemann, M., Nafz, F., Ortmeier, F., Seebach, H., Reif, W.: A Specification and Construction Paradigm for Organic Computing Systems. In: Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, pp. 233–242. IEEE Computer Society, Washington (2008)
17. Hsieh, F.S.: Fault-tolerant deadlock avoidance algorithm for assembly processes. IEEE Transactions on Systems, Man and Cybernetics, Part A 34(1), 65–79 (2004)
18. Huang, Y.S., Jeng, M.D., Xie, X., Chung, D.H.: Siphon-based deadlock prevention policy for flexible manufacturing systems. IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans 36(6), 1249 (2006)

19. Kim, K.H., Jeon, S.M., Ryu, K.R.: Deadlock prevention for automated guided vehicles in automated container terminals. OR Spectrum 28(4), 659–679 (2006)
20. Kobayashi, N.: Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In: van Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D., Ito, T. (eds.) TCS 2000. LNCS, vol. 1872, pp. 365–389. Springer, Heidelberg (2000)
21. Kwong, Y.S.: On the absence of livelocks in parallel programs. In: Kahn, G. (ed.) Semantics of Concurrent Computation. LNCS, vol. 70, pp. 172–190. Springer, Heidelberg (1979)
22. Lamport, L.: Fairness and hyperfairness. Distributed Computing 13(4), 239–245 (2000)
23. Li, Z.W., Zhou, M.C., Wu, N.Q.: A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 38(2), 173–188 (2008)
24. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.-P., Reif, W.: A generic software framework for role-based Organic Computing systems. In: SEAMS 2009: ICSE 2009 Workshop Software Engineering for Adaptive and Self-Managing Systems (2009)
25. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.-P., Reif, W.: A universal self-organization mechanism for role-based Organic Computing systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) ATC 2009. LNCS, vol. 5586, pp. 17–31. Springer, Heidelberg (2009)
26. Park, D.: On the Semantics of Fair Parallelism. In: Bjorner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 504–526. Springer, Heidelberg (1980)
27. Sanchez, C., Sipma, H.B., Manna, Z., Gill, C.D.: Efficient distributed deadlock avoidance with liveness guarantees. In: Proceedings of the 6th ACM & IEEE International conference on Embedded software, pp. 12–20. ACM, New York (2006)
28. Sfinghal, M.: Deadlock detection in distributed systems. Computer 22(11), 37–48 (1989)
29. Wang, J., Zhang, X., Zhang, Y., Yang, H.: A Probabilistic Model for Parametric Fairness in Isabelle/HOL. In: Theorem Proving in Higher Order Logics: Emerging trends Proceedings, pp. 194–209 (2007)
30. Wang, Y., Kelly, T., Kudlur, M., Mahlke, S., Lafortune, S.: The application of supervisory control to deadlock avoidance in concurrent software. In: 9th International Workshop on Discrete Event Systems, WODES 2008, pp. 287–292 (2008)
31. Wang, Y.M., Merritt, M., Romanovsky, A.B.: Guaranteed deadlock recovery: Deadlock resolution with rollback propagation. In: Proc. Pacific Rim International Symposium on Fault-Tolerant Systems (1995)
32. Wu, N., Zhou, M.C.: Deadlock resolution in automated manufacturing systems with robots. IEEE Transactions on Automation Science and Engineering 4(3), 474–480 (2007)
33. Wu, N., Zhou, M.C., Li, Z.W.: Resource-oriented Petri net for deadlock avoidance in flexible assembly systems. IEEE Transactions on Systems, Man and Cybernetics, Part A 38(1), 56–69 (2008)
34. Zajac, J.: A deadlock handling method for automated manufacturing systems. CIRP Annals-Manufacturing Technology 53(1), 367–370 (2004)