A Colored Petri Net for Regional Synchronization in Situated Multi-Agent Systems

Danny Weyns and Tom Holvoet

AgentWise, DistriNet, Department of Computer Science,
K.U.Leuven, B-3001 Leuven, Belgium
{danny.weyns,tom.holvoet}@cs.kuleuven.ac.be

Abstract. Interaction is central to multi-agent systems. In this paper we look at agents that interact by performing simultaneous actions in their environment. Simultaneous actions are interfering actions that are executed together and that produce a compound result. To act simultaneously agents need to synchronize their actions. Synchronization of actions is typically established by one centralized synchronizer that ensures for each action cycle that the actions of all agents are treated simultaneously. Centralized synchronization is simple, however, its drawbacks are centralized control and poor scalability. We present a Colored Petri Net (CPN) for regional synchronization. With regional synchronization agents synchronize with each other locally, resulting in independent groups of synchronized agents. Regional synchronization is established in a distributed way and ensures that only agents that are able to perform simultaneous actions are synchronized. The algorithm for regional synchronization, that is based on a two-phase commit protocol combined with a logical clock, is not trivial. For a clear explanation of the algorithm, we discuss the CPN for regional synchronization at great length. We also formally prove that for one agent the various steps of the algorithm are executed correctly, and we prove the correctness of the algorithm for two agents. Finally we discuss simulation results of the algorithm for four agents.

1 Introduction

Interaction is central to multi-agent systems. Through interactions agents are able to setup social organizations. In this paper we look at a particular kind of interactions where situated agents interact by performing simultaneous actions in their environment.

Situated multi-agent systems. In situated multi-agent systems (situated MAS), agents as well as objects have an explicit position in the environment. Situatedness reflects the local relationships between agents and objects. Through its situatedness, a situated agent is placed in a local context that it is able to perceive and in which it can act. For actions, we use a model that is based on the theory of influences and reactions to influences, proposed by J. Ferber [3]. Roughly spoken, this theory separates what an agent wants to perform from

what actually happens: agents produce influences into the environment and subsequently the environment reacts by combining the influences to deduce a new state of the world from them. The reification of actions as influences enables the environment to combine simultaneously performed activity in the MAS.

Simultaneous actions. With simultaneous actions, we mean a set of interfering actions that are executed together and that produce a compound result. We distinguish between three kinds of simultaneous actions: *joint actions*, *influencing actions* and *concurrent actions*. Joint actions are actions that must be executed together in order to produce a successful result. An example of joint actions is two or more agents that pick up an object that none of them can pick up by itself or agents that carry such an object to a certain location together. Influencing actions are actions that positively or negatively affect each other. An example of influencing actions is two agents that push the same object together. When the agents push the object in different directions, the object moves according to the resultant of the two actions. Finally, concurrent actions are simultaneously performed actions that conflict. An example of concurrent actions is two or more agents that try to pick up the same object at the same time. When only one of the involved agents can get the object, a non-deterministic selection can be used to assign the object to one of the agents. For a classification and a detailed study of simultaneous actions we refer to [10].

Supporting simultaneous actions. Support for simultaneous actions involves two aspects: (1) agents must be able to act together and (2) the outcome of a combination of simultaneously performed actions must be in accordance with the domain that is modelled. The focus of this paper is on the first aspect.

To act simultaneously, the agents need to synchronize their actions. In situated MASs synchronization of actions is typically dealt with by one centralized synchronizer that ensures that for each action cycle the actions of all agents are treated simultaneously [4]. Centralized synchronization is simple, however it forces all the agents to act at one global pace. This is a hard requirement and it implies a number of drawbacks. First, the scalability of the model with respect to the number of agents in the MAS is limited. Centralized synchronization does not differentiates between the influences of agents that may interfere (i.e. influences of potential simultaneous actions) and influences that are not. The influences of *all* agents are treated as if they happened together. Therefore, the synchronizer must verify for each influence whether it interferes with any other influence. This makes the costs for calculating reactions in the order of square to the number of agents in the MAS. And second, centralized control conflicts with the distributed nature of MAS. All agents are globally synchronized and that implies centralized control of the evolution of the MAS.

To resolve the drawbacks of centralized synchronization we present an algorithm that allows agents to synchronize with other agents within their perceptual range. The result of the algorithm is the formation of independent groups of synchronized agents. The composition of these groups is based on the locality of the agents and dynamically changes when agents enter or leave each others perceptual range. In this approach, only the actions of regionally synchronized agents

(i.e. potentially simultaneously acting agents) are treated simultaneously. Since regional synchronization is established by the agents themselves, there is no longer a central entity that controls the evolution of the MAS. The price for decentralization of synchronization is the communication cost to set up the groups.

The algorithm for regional synchronization, that is based on a two-phase commit protocol combined with a logical clock, is not trivial. In [9] we have given a high level outline of the algorithm. In this paper, we zoom in and present a detailed formal model of the algorithm by means of a Colored Petri Net (CPN) [6]. For a clear explanation of the algorithm, we discuss the CPN for regional synchronization at great length. In addition, the CPN also allows to formally prove the correctness of the algorithm. In the paper we formally prove that for one agent the various steps of the algorithm are executed correctly. We also prove the correctness of the algorithm for two agents. In addition, we discuss simulation results of the algorithm for four agents. The CPNs presented in this paper are designed with the Design/CPN tool [2].

Overview. This paper is structured as follows. Section 2 discusses the Colored Petri Net for regional synchronization in detail. In section 3 we prove that, from the viewpoint of one agent, the various steps in the algorithm are executed correctly. Subsequently we prove the correctness of the algorithm for two agents. Section 4 discusses a simulation of the algorithm for four agents. Finally, we conclude and look at future work in section 5.

2 Colored Petri Net for Regional Synchronization

In this section we discuss the algorithm for regional synchronization. First we give a high level overview of the algorithm. Then we present the CPN and discuss the subsequent steps of the algorithm in detail.

2.1 High level overview of the regional synchronization algorithm

Regional synchronization realizes logical simultaneity of agent activity based on the locality of the agents. The settlement of simultaneous actions follows two major phases: the phase of *synchronization setup* and the *acting phase*.

With regional synchronization each agent is equipped with its own local synchronizer. Each synchronizer is responsible to establish synchronization for its associated agent with the synchronizers of other agents. Synchronization setup starts when the agent receives the *view set* together with the *synchronization time* from the environment. The view set is the initial set of candidates for synchronization, i.e. the names of the agents which whom the agent starts synchronizing. Since the goal of synchronization is to handle simultaneous actions, the range for an agent to synchronize with other agents should be in accordance with the range for such interfering actions. In the context of situated agents it is quite natural to limit this range to the perceptual range of the agents them-

selves[1]. The synchronization time is the value of the logical clock at the time when the agent's view set was composed. This logical clock is a counter maintained by the environment. Each time a region of synchronized agents concludes the acting phase, the value of the logical clock is incremented and new view sets for the agents are composed[2]. When no other agent is within the perceptual scoop, the agent can act asynchronously with respect to all other agents. Otherwise the synchronizer starts executing the synchronization algorithm.

The algorithm for regional synchronization integrates a two-phase commit (2PC) protocol and a logical clock (LC). The goal of a standard 2PC (see e.g. [1]) is to reach a full agreement between a set of processes (participants) whether or not to perform some action. The protocol is initiated by one process, i.e the coordinator. The coordinator collects votes from the participants and decides about the outcome of the interaction. Logical clocks were first proposed by Lamport [7] to capture numerically causal ordering of events within process groups. In the algorithm for regional synchronization, synchronizers are peers and can play both the role of participant as well as coordinator during one ongoing synchronization setup. Which role a synchronizer plays with respect to the others depends on the comparison of the values of their synchronization-time, i.e. the value of the logical clock they received when they entered synchronization setup.

During synchronization setup, the agent blocks its activity until all agents it is synchronizing with have finished synchronization setup. While executing the algorithm, synchronizers progressively try to synchronize with the synchronizers of the agents inside their perceptual range by means of sending messages back and forth. During this interaction, the synchronizers pass two phases. During the first phase they decide whether they agree about synchronization and subsequently during the second phase they mutually commit to the agreement. When all agents within their perceptual range have concluded synchronization they act together during the acting phase. We call such a group a region of synchronized agents, in short a region. Each acting agent invokes a tuple ($influence$, $names\_of\_synchronized\_agents$) into the environment. The environment collects the influence tuples of the agents per region. Since agents have only a limited view on the environment it may be the case that two agents positioned inside each others perceptual range see different candidates to synchronize with. So at the end of synchronization setup an agent typically knows only a limited number of the agents of the synchronized region to which it belongs. As such the environment is responsible for composing regions on the basis of only subsets of synchronized agents that are passed with the influence tuples.

When all influences of a region are available, the environment calculates the effects of the simultaneously acting agents and that concludes the acting phase. For a detailed study of influence processing by the environment we refer to [11].

[1] In the paper we associate perceptual range with 'visual' perception. However this is not a restriction of the algorithm, other approaches can be used as well as e.g. the agents' acquaintances can be used to determine locality.

[2] Notice that the value of the logical clock is not a global variable, each physical or logical zone of the MAS environment maintains its own logical clock.

Summarizing, the algorithm for regional synchronization combines a two phase commit protocol with a logical clock. During synchronization setup each agent synchronizes with the agents inside its perceptual range and hence also with all agents within the perceptual range of the latter, and so on. Subsequently during the action phase, synchronized agents act together, allowing them to perform simultaneous actions. In comparison to centralized synchronization, regional synchronization avoids centralized control of the evolution of the multi-agent system and improves scalability.

2.2 Colored Petri Net for the Algorithm

Fig. 1 depicts a hierarchical CPN for the regional synchronization algorithm, hereafter denoted as the *synchronization net*. The transitions *handle_mail*, *scheduler* and *simulation_verification* denote substitution transitions that stands for entire subnets. The *simulation_verification* subnet (in short the *test net*) models a simulation or verification net that is connected to the synchronization net. In the next two sections we use different test nets for verification and simulation purposes. Each agent that starts synchronization setup injects a token from the test net into the synchronization net. We call such token a *synchronization token*. The *scheduler* subnet (*scheduler net*) allows to schedule the synchronization tokens according to a specific scheduling algorithm. The standard algorithm schedules all tokens concurrently, however, to control state space explosion during verification, we apply a round-robin scheduling algorithm. When synchronization setup is finished the synchronization token returns to the test net. Depending on the kind of applied test, the test net uses the synchronization token to continue the simulation of a MAS (i.e. regional synchronized agents perform simultaneous actions) or the test net compares the state of the synchronization token with an expected state to verify the correctness of the executed step in the algorithm. Finally, *handle_mail* is an integral part of the synchronization algorithm. This subnet (hereafter called the *mail net*) is depicted in Fig. 2.

To explain the synchronization algorithm, we follow one synchronization token that runs through the algorithm. Each agent synchronizes before each action. Synchronization setup starts when the agent receives the view set together with the synchronization time from the environment. As explained in the previous section, the view set contains the names of the agents within the perceptual range of the agent and the synchronization time is the value of the logical clock maintained by the environment at the time when the agent's view set was composed. With the view set and the synchronization time the agent composes a synchronization token that is put in the *start_sync* place, see Fig. 1. Synchronization tokens are of the color *SyncSet* that is defined as follows:

$$SyncSet = record \;\; name : Name * memberset : MemberSet * clock : Clock;$$

name is the unique name of the synchronizing agent, *memberset* is a data structure in which the current state of the synchronization process for each agent (member) of the view set is registered and *clock* denotes the synchronization time of the agent. The color *MemberSet* is defined as follows:

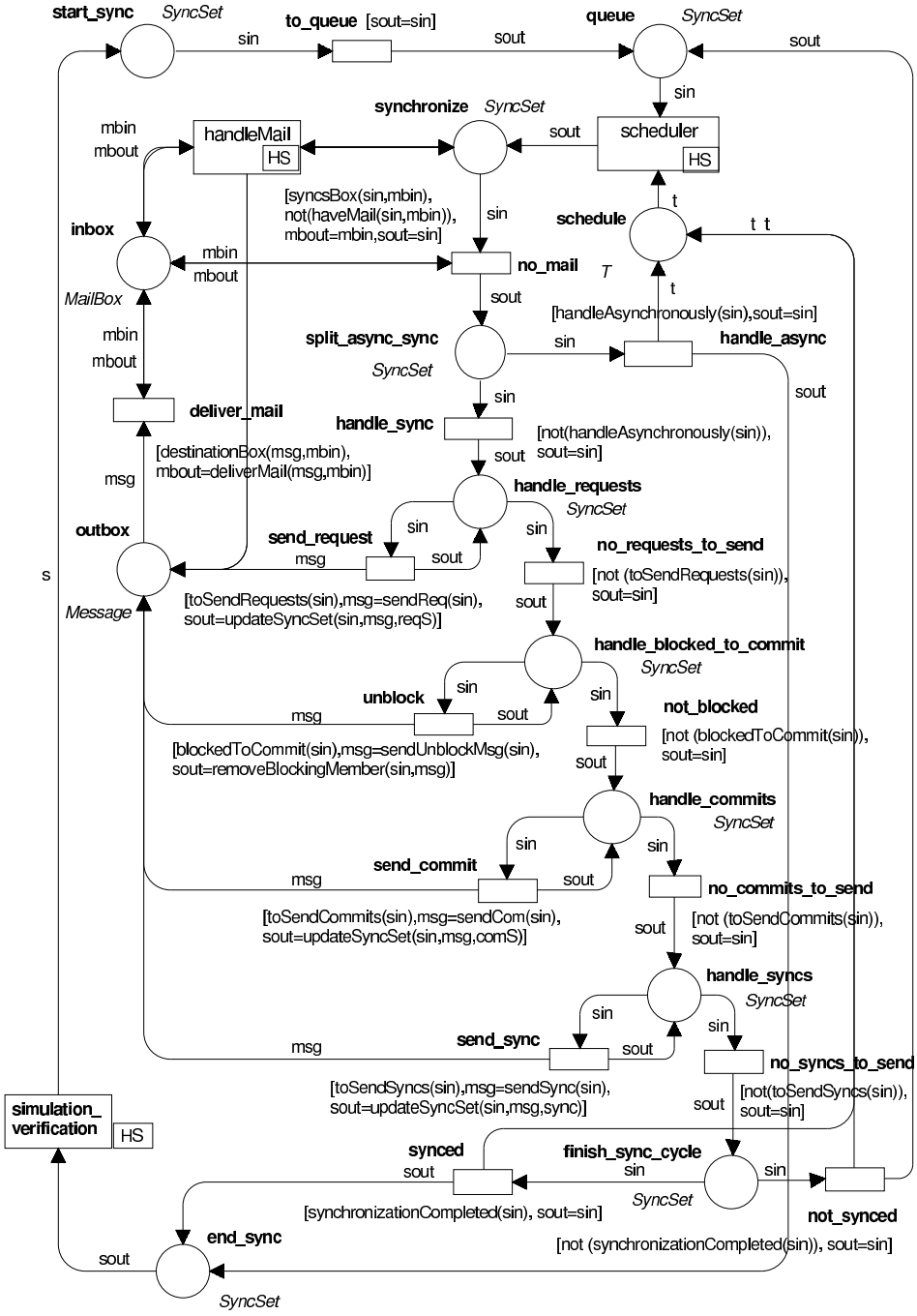

Fig. 1. Main CPN for regional synchronization.

$MemberSet = list\ Member\ with\ 0 \ldots max\_agents;$
$Member = record\ member : Name * state : State * clock : Clock;$
$State = with\ \ ini \mid reqS \mid reqR \mid ackS \mid ackR \mid comS \mid comR \mid sync;$

The *memberset* contains one $Member$ for each agent which whom a synchronizer is synchronizing. Each $Member$ is characterized by the name of the associated agent (*member*), a *state* and a *clock* value. Initially, for each name in the view set, a member is added to the *memberset* with the *state* set to *ini* and the *clock* set to zero. During the algorithm *clock* values are exchanged and the *state* of each member evolves from *ini* to *sync*. The comparison of *clock* values determine the rights of a synchronizer to initiate particular steps in the synchronization algorithm. When synchronization with a member fails, that member is removed from the *memberset*.

For the explanation of the algorithm, we follow the synchronization setup of an agent with name 3 with a view set $\{2, 4, 1\}$ and a synchronization time 7, resulting in the following initial synchronization token:

$s = \{name = 3, memberset = [\{member = 2, state = ini, clock = 0\},$
$\{member = 4, state = ini, clock = 0\}, \{member = 1, state = ini, clock = 0\}], clock = 7\}$

As soon as the synchronization token is scheduled, i.e. when the token moves from the *queue* place to the *synchronize* place, the synchronizer checks its mailbox, verifying whether there are pending requests. Pending requests are request messages received by the synchronizer during the previous acting phase. If the synchronizer has received mail, the *pick_mail* transition in the mail net (see Fig. 2) is enabled. When this transition fires, it takes a synchronization token from the *synchronize* place and a message from the synchronizer's *inbox*. The color of *inbox* is $MailBox$ that is defined as follows:

$MailBox = record\ name : Name * box : Box;$
$Box = list\ Message\ with\ 0 \ldots max\_msgs;$
$Message = record\ from : Name * to : Name * perform : Performs * clock : Clock;$
$Performs = with\ \ reqM \mid ackM \mid nackM \mid comM \mid syncM;$

The condition to enable the *pick_mail* transition is defined as follows:

$[syncsBox(sin, mbin), haveMail(sin, mbin),$
$sout = sin, msgin = pickMsg(mbin), mbout = removeTopMsg(mbin)]$

$syncsBox(sin, mbin)$ selects the mailbox $mbin$ of the synchronization token sin, while $haveMail(sin, mbin)$ verifies whether $mbin$ contains any message. If this latter condition holds and the *pick_mail* transition fires, the synchronization token is passed ($sout = sin$) from the *synchronize* place to the $syncset\_buffer$ place. Simultaneously the top message of the mailbox $mbin$ is removed from the box ($mbout = removeTopMsg(mbin)$) and put in the msg_buffer place ($msgin = pickMsg(mbin)$). For pending requests the performative ($perform$) of the message in msg_buffer is $reqM$. Now there are two possibilities. If the agent associated with the requesting synchronizer belongs to the *memberset* of the syn-

chronization token, the function $reqToAccept(sin, msgin)$ returns true and that enables the req_accept transition. On the other hand, if $reqToAccept(sin, msgin)$ returns false (i.e. when the requesting synchronizer does not belongs to the $memberset$) the req_reject transition is enabled.

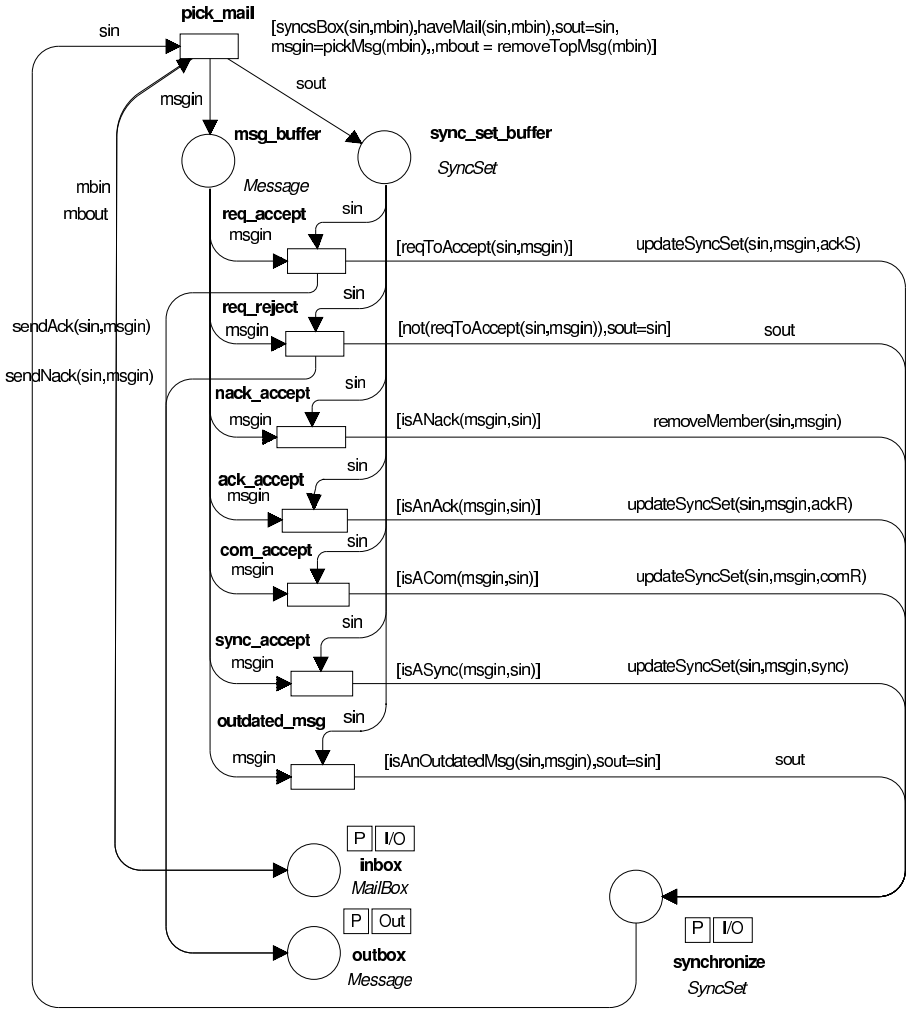

Fig. 2. The *handle_mail* subnet of the main CPN for regional synchronization.

If the req_accept transition fires, an acknowledgement is sent to the requesting agent ($sendAck(sin, msgout)$) and the state of the $memberset$ is updated ($updateSyncSet(sin, msgin, ackS)$), i.e. the state of the corresponding member is set to $ackS$ and its $clock$ is set to the $clock$ value of the requesting message ($clock$ in $msgin$). However, if the req_reject transition is enabled, a 'nack'

message is sent to the requesting agent ($sendNack(sin, msgout)$) and the synchronization token is put in the *synchronize* place without changes ($sout = sin$). All pending requests are handled in a similar way. For the example synchronizer we assume the following content of its mailbox:

$$mbin = [\{from = 6, to = 3, perform = reqM, clock = 6\},\\ \{from = 1, to = 3, perform = reqM, clock = 6\}]$$

Since agent 6 does not belong to the $memberset$ of agent 3 its request is rejected. The request of agent 1 is accepted. Thus, the synchronizer sends two messages:

$$msg_1 = \{from = 3, to = 6, perform = nackM, clock = 7\}$$
$$msg_2 = \{from = 3, to = 1, perform = ackM, clock = 7\}$$

The state of the synchronization token is updated as follows:

$$s = \{name{=}3, memberset{=}[\{member{=}2, state{=}ini, clock{=}0\},\\ \{member{=}4, state{=}ini, clock{=}0\}, \{member{=}1, state{=}ackS, clock{=}6\}], clock{=}7\}$$

After mail handling we return to the synchronization net, see Fig. 1. The acknowledge messages in the *outbox* of the synchronizer enable the *deliver_mail* transition. Each time this transition fires, a message is delivered in the mailbox of the addressee. $destinationBox(msg, mbin)$ selects the appropriate mailbox and $mbout = deliverMail(msg, mbin)$ effectively delivers the message.

When all messages in the *inbox* have been handled and the synchronization token arrives at the *synchronize* place, the transition *no_mail* is enabled ($not(haveMail(sin, mbin))$ holds). When the transition *no_mail* fires, the synchronization token is put in the *split_async_sync* place. At this point, the algorithm can evolve in two possible directions. If the condition $handleAsynchronously(sin)$ holds, i.e. the $memberset$ of the synchronization token is empty, the transition *handle_async* is enabled. When the *handle_async* transition fires, an anonymous token t of color T is put in the *schedule* place, enabling the scheduler to schedule the next waiting synchronization token. Simultaneously, the synchronization token is put in the *end_sync* place ($sout = sin$) where the test net will pick it up and that finishes synchronization setup for the synchronizer. During the following acting phase, the associate agent will act asynchronously with respect to the agents in the MAS. On the other hand, if $handleAsynchronously(sin)$ does not hold (the $memberset$ of the synchronization token is not empty) the *handle_sync* transition is enabled. When this transition fires, the synchronization token is put in the *handle_requests* place. Since the $memberset$ of the synchronization token in the example is not empty the token follows this latter route through the net.

As long as the $memberset$ of the synchronization token contains members in the state ini, the *send_request* transition is enabled (i.e. $toSendRequests(sin)$ holds). Each time this transition fires a request message is sent to a particular member in the initial state. For the synchronization token in the example the following two request messages are sent:

$msg_1 = \{from = 3, to = 2, perform = reqM, clock = 7\};$
$msg_2 = \{from = 3, to = 4, perform = reqM, clock = 7\};$

For each request that is sent, the *state* of the corresponding member in the synchronization token is updated to $reqS$. Applied to the example, the state of the synchronization token becomes:

$s = \{name = 3, memberset = [\{member = 2, state = reqS, clock = 0\},$
$\{member = 4, state = reqS, clock = 0\}, \{member = 1, state = ackS, clock = 6\}], clock = 7\}$

When all requests have been sent, the condition $toSendRequests(sin)$ no longer holds and that enables the *no_requests_to_send* transition. As soon as this transition fires the synchronization token is put in the *handle_blocked_to_commit* place. For the moment we do not elaborate on the next sequence of transitions, i.e. the synchronization token now passes along the chain of transitions *not_blocked* – *no_commits_to_send* – *no_syncs_to_send* – *not_synced*. When this latter transition fires, a new token for the scheduler is put in the *schedule* place and the synchronization token is put in the *queue* place.

We take up again when the synchronization token is scheduled for the next pass through the synchronization net and arrives in the *synchronize* place. First the *inbox* is inspected for new messages (see Fig. 2). The reactions to the acknowledgements and requests may now have been arrived. As discussed above, a request can be answered by an acknowledgement or a rejection, depending on the membership of the requesting synchronizer. An acknowledgement enables the *ack_accept* transition (i.e. the condition $isAnAck(msgin, sin)$ holds). When this transition fires the synchronization token is updated, i.e. the state of the corresponding member in the $memberset$ is set to $ackR$ and the *clock* is set to the value received in the acknowledge message. A rejection enables the *nack_accept* transition (the condition $isANack(msgin, sin)$ holds). When this transition fires the corresponding member is removed from the $memberset$ in the synchronization token. For our example, we suppose that two messages have been received:

$msg_1 = \{from = 2, to = 3, perform = nackM, clock = 8\};$
$msg_2 = \{from = 4, to = 3, perform = ackM, clock = 8\};$

The nack message of agent 2 indicates that after agent 3 has perceived agent 2 (synchronization time 7), agent 2 has moved out of the perceptual range of agent 3. Agent 4 on the other hand (that has acted simultaneously with agent 2, both have the same synchronization time 8) is still within the perceptual scope of agent 3. After mail handling the synchronization token is updated to:

$s = \{name = 3, memberset = [\{member = 4, state = ackR, clock = 8\},$
$\{member = 1, state = ackS, clock = 6\}], clock = 7\}$

After handling the acknowledge messages, we return to the synchronization net where the synchronization token is located in the *synchronize* place, see Fig. 1. The synchronization token now passes the transitions *no_mail*, *handle_sync* (at least if we suppose that the $memberset$ of the synchronization token is not

empty) and *no_requests_to_send* (*send_request* is no longer enabled since the synchronizer has sent all requests during the previous pass through the algorithm). We further suppose that the synchronization token passes the *not_blocked* transition and arrives at the *handle_commits* place, we discuss the selection between *unblock* and *not_blocked* below. From the *handle_commits* place a synchronizer is able to send commits depending on the condition $toSendCommits(sin)$. This condition holds if the $memberset$ of the synchronization token contains members in the state *ackR* or *ackS* of which the *clock* value is smaller or equal to the *clock* value of the synchronization token itself. Each time the *send_commit* transition fires a commitment message is sent to such a member ($msg = sendCommit(sin)$) and simultaneously the *state* of the member is updated to *comS* ($sout = updateSyncSet(sin, msg, syncS)$). For our example, only member 1 meets the requirements to send a commit (the *clock* value of member 4 is greater then the *clock* value of the synchronizer itself). When the *send_commit* transition fires, a commitment message is sent to the synchronizer of agent 1 and the state of the synchronization token is updated:

$msg = \{from = 3, to = 1, perform = comM, clock = 7\};$
$s = \{name = 3, memberset = [\{member = 4, state = ackR, clock = 8\},$
$\{member = 1, state = comS, clock = 6\}], clock = 7\}$

After commitments have been sent, the condition $toSendCommits(sin)$ no longer holds and that enables the *no_commits_to_send* transition. As soon as this transition fires the synchronization token is put in the *handle_syncs* place. For the moment we suppose that *no_syncs_to_send* is enabled. When the synchronization token passes this transition and the *not_synced* transition, it is ready to pass the synchronization net once more.

After the synchronization token is scheduled, the mailbox is checked, see Fig. 2. The synchronizer now expects commitment messages. If the mailbox contains commitment messages the condition $isACom(msgin, sin)$ holds and the *com_accept* transition is enabled (of course, after the *pick_mail* transition has fired). When the *com_accept* transition fires the state of the member in the $memberset$ that has sent the commitment message is updated to *comR*. For our example we suppose that the commitment message of member 4 has now been arrived. The received message and the updated synchronization token are:

$msg = \{from = 4, to = 3, perform = comM, clock = 8\};$
$s = \{name = 3, memberset = [\{member = 4, state = comR, clock = 8\},$
$\{member = 1, state = comS, clock = 6\}], clock = 7\}$

After mail handling, we return to the synchronizer net, see Fig. 1. Starting from the *synchronize* place, the synchronization token now passes along a chain of transitions until it arrives in the *handle_syncs* place. There the condition $toSendSyncs(sin)$ is verified. This condition holds when every member in the $memberset$ is in the state *comR* or *sync*. If this is the case, a synchronization message is sent to every member in the state *comR* and the state of these members are updated to *sync*. Since member 1 in the example is in the state *comS*,

the $toSendSyncs(sin)$ does not hold for the example synchronization token and thus the synchronizer has to wait for a confirmation of the commitment sent to the synchronizer of agent 1. As a consequence the *no_syncs_to_sent* transition is enabled and the synchronization token returns back to the *queue* place.

During a next run through the synchronization net, the synchronizer likely will find the expected confirmation message in its mailbox, see Fig. 2. When a synchronization message has arrived the *sync_accept* transition gets enabled. When this transition fires, the state of the sender in the $memberset$ of the synchronization token is set to *sync*. For the example, we suppose that the confirmation of member 1 has arrived. After the *pick_mail* and *sync_accept* transitions have fired, the state of the synchronization token becomes:

$s = \{name = 3, memberset = [\{member = 4, state = comR, clock = 8\},$
$\{member = 1, state = sync, clock = 6\}], clock = 7\}$

During the next pass through the synchronization net, the condition to send a synchronization message to member 4 now holds (the function $toSendSyncs(sin)$ of the *send_sync* transition returns true). When the *send_sync* transition fires, a synchronization message is sent to member 4 and the updated state of the synchronization token becomes:

$msg = \{from = 3, to = 4, perform = syncM, clock = 7\};$
$s = \{name = 3, memberset = [\{member = 4, state = sync, clock = 8\},$
$\{member = 1, state = sync, clock = 6\}], clock = 7\}$

This enables the *no_syncs_to_send* transition. When this transition fires the synchronization token is put in the $finish\_sync\_cycle$ place. Here the condition $synchronizationCompleted(sin)$ is verified. This condition holds if all members of the $memberset$ are in the state *sync*. Since this is the case for the synchronization token in the example, the synchronization token reaches the *end_sync* place (and a new token is put in the *schedule* place to schedule the next synchronization token) and that concludes synchronization setup.

Now we come back to the selection between *unblock* and *not_blocked*. The condition to enable the *unblock* transition is *blocked_to_commit*. This condition deals with deadlocked situations in the algorithm that may arise when a sequence of overlapping perceptual ranges of synchronizing agents form a cycle.

We illustrate an example of this problem in the Packet-World [5][8]. The Packet-World is a simple MAS application we use as a test case in our research, see 3. In the Packet-World agents have to bring colored packets (rectangles) to corresponding colored destinations (circles). Agents can perform different kinds of actions such as making a step to a neighboring field or pass a packet to a neighboring agent. Agents have only a limited view on the world, the right part of Fig. 3 illustrates the perceptual scope of agent 8. In this case agents can perceive their environment two fields from their current location.

Suppose that in the depicted situation of Fig. 3 agents 8, 1, 7 and 2 are all executing the acting phase. Now agent 7 makes a step North West, entering the perceptual range of agents 1 and 2. S_7, the synchronizer of agent 7, starts

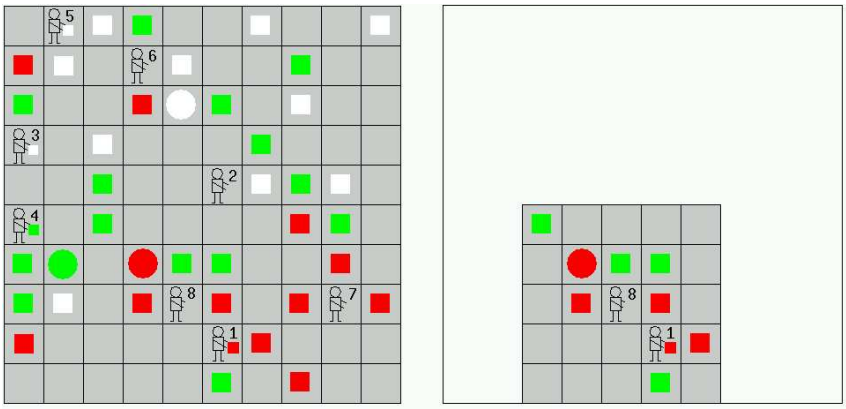

Fig. 3. Blocking problem of regional synchronization in the Packet–World.

synchronization setup by requesting S_1 and S_2 to synchronize. Subsequently, agent 1 and 8 conclude their action (suppose they transferred a packet) and enter synchronization setup. S_8 starts to synchronize with S_1; S_1 starts to synchronize with S_8 and S_7. After a while agent 8 finishes synchronization. We suppose that S_1 and S_7 are still busy synchronizing when agent 2 makes a step to the South and enters synchronization setup. S_2 confirms the pending request of S_7. Subsequently, S_2 requests S_8 to synchronize. While S_1 and S_7 can conclude synchronization setup, S_2 (and as such also the whole region) is blocked since S_8 (that already finished synchronization setup) is unable to respond to S_2 synchronization request.

To deal with such potential deadlock situations, a synchronizer can 'unblock' itself. The condition $blockedToCommit(sin)$ that enables the *unblock* transition holds if (1) at least one member of the *memberset* of a synchronization token is in the state *comR*, *comS* or *sync*; (2) at least one member of the *memberset* is in the state *reqS* and (3) no member is in the state *reqR*, *ackR* or *ackS*. When a synchronization token reaches the *handle_blocked_to_commit* place and the $blockedToCommit(sin)$ condition holds, the *unblock* transition is enabled. Each time this transition fires a nack message is sent ($msg = sendUnblockMsg(sin)$) to a blocking member, i.e. a member of the *memberset* in the state *reqS*. Simultaneously the blocking member is removed from the *memberset* in the synchronization token ($sout = removeBlockingMember(sin, msg)$). When the synchronization token no longer contains a blocking member ($blockedToCommit(sin)$ returns true) the transition *not_blocked* is enabled and that liberates the synchronizer from the deadlock situation.

Unblocking has a side effect. Since messages are delivered indirectly (via the *deliver_mail* transition), an acknowledge message may reach the mailbox of a requested synchronizer with delay. Therefore a blocked synchronizer can not distinguish whether it is effectively blocked due to a synchronizer that is not able to react to a request (because it has finished synchronization setup and

waits for the conclusion of synchronization setup of the other members of the region to which it belongs) or whether the acknowledgement to a request message is underway. As soon as a synchronizer unblocks and removes a blocking member from its *memberset* at least direct synchronization with that member is no longer possible[3]. An acknowledge message of the removed member may reach the mailbox of the unblocked synchronizer after this latter has concluded synchronization setup. Such outdated messages should not interfere with the next synchronization setup of a synchronizer. Outdated messages are therefore removed from the mailbox via the *outdated_msg* transition in the mail net, see Fig.2. When a new synchronization token (i.e. a token with all its members in the state *ini*) enters the synchronization net, first the mailbox of the synchronizer is checked. When the mailbox contains outdated messages, the $isAnOutdatedMsg(sin, msg)$ transition becomes enabled. When this transition fires, an outdated message is discarded. As soon as all outdated messages are removed from the mailbox, the *oudated_msg* transition is no longer enabled and the synchronizer can follow the usual synchronization setup. Note that unblocking due to a delay of message delivering is rather exceptional since it is very unlikely that a synchronizer passes the complete synchronization process with some members of its *memberset* while the acknowledgements of other members did not had the change to reach its mailbox.

3 Verification

In this section we discuss formal verification of the regional synchronization algorithm. First we prove that for one agent the various steps of the synchronization algorithm are executed correctly. Then we prove the correctness of the algorithm for two agents. Together this is only a partial proof of the correctness of the algorithm, e.g. the blocking situations where at least four agents are involved are not covered here.

3.1 Formal verification of the algorithm for one agent

We prove the correct execution of the various steps of the algorithm for one agent by enumeration. We developed a test net that feeds the synchronization net with a set of synchronization tokens (hereafter denoted as *test tokens*) accompanied with possible received messages (*test messages*). This set represents all possible situations a single synchronizer can be faced with during one pass through the synchronization algorithm. For each situation we only took into account a minimal needed *memberset* to verify the step. After a test token has passed through the synchronization net, the test net (1) verifies whether a possible test message has been processed correctly; (2) compares the correctness of the updated test

[3] The agents may still get synchronized indirectly. E.g. in the Packet-World example, agent 2 who unblocks agent 8 is at the end of synchronization setup still synchronized with agent 8 via agents 7 and 1.

token with an appropriate *control token* and (3) compares the correctness of a possible new sent message with a corresponding *control message*.

Fig.4 depicts the CPN of the test net. Initially the *test_sets* place contains a set of *composite test tokens* of color $TestSet$. Each composite test token holds the data needed to verify the correct execution of a particular step of the algorithm. We identified 14 different steps in the algorithm, including steps to verify

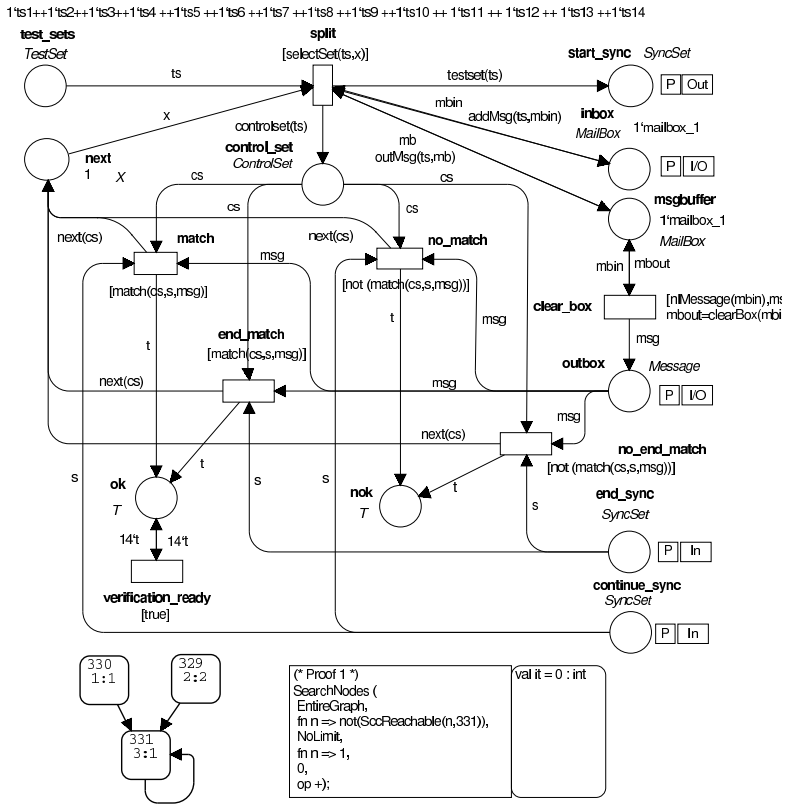

Fig. 4. Verification net for one agent.

the reaction to received messages and steps to verify whether progress in the algorithm is made when the appropriate conditions hold. As an example, we illustrate the composite test token that is used to verify whether a request is correctly accepted:

$is2 = \{name = 1, memberset = [\{member = 2, state = ini, clock = 0\}], clock = 1\} : SyncSet;$
$im2 = \{from = 2, to = 1, perform = reqM, clock = 2\} : Message;$
$os2 = \{name = 1, memberset = [\{member = 2, state = ackS, clock = 2\}], clock = 1\} : SyncSet;$
$om2 = \{from = 1, to = 2, perform = ackM, clock = 1\} : Message;$
$ts2 = \{number = 2, inset = is2, inmsg = im2, outset = os2, outmsg = om2\} : TestSet;$

$is2$ is the initial synchronization token (test token) and $im2$ the accompanied request message (test message) for the test. $os2$ (the control token) represents the expected state of the synchronization token after $is2$ has passed through the synchronization net and $om2$ (the control message) is the expected acknowledge message that has to be sent. $ts2$ aggregates all this data in one composite test token.

The *split* transition selects the composite test tokens one by one, based on their *number* (e.g. in the example above the number of $ts2$ is 2). The composite test token is then split into a synchronization token (test token) that is put in the *start_sync* place, a message token that is put in the *inbox* place and a control token combined with a control message that is put in the *control_set* place. The test token then passes one time through the synchronization net after which it reaches the *end_sync* or the *continue_sync* place depending on its updated state. Possibly sent messages are put in the outbox that, for verification purposes, is not connected to the mailboxes of the addressees. Subsequently, one of the transitions *match*, *end_match*, *no_match* or *no_end_match* is enabled depending the result of the *match* functions of the guards of the transitions. If *match* or *end_match* fires, an anonymous token (of color T) is put in the *ok* place indicating that the test has succeeded. On the other hand if *no_match* or *no_end_match* fires a token is put in the *nok* place, indicating a failure.

The proof of the correct execution of the test steps is based on the liveness property of the $verification\_ready$ transition connected to the *ok* place. As soon as the *ok* place has collected an anonymous token for each verified step of the algorithm (i.e. 14 tokens) the $verification\_ready$ transition becomes enabled and from then on stays live forever. To prove the correct execution of the various steps of the algorithm, we generate an occurrence graph for the net and prove that a path exists from each node in the graph to the node that represents the final marking, i.e the state of the *ok* place with 14 collected tokens. That particular node, the leaf node of the occurrence graph (node number 331) is shown in Fig.4. The proof is straightforward. The $SearchNodes$ function (see Fig.4) searches the nodes that have no path to the leaf node. Since this number is zero we have proven that all test steps are executed correctly.

3.2 Formal verification of the algorithm for two agents

To prove the correct execution of the algorithm for two agents, we developed a test net that simulates a simple MAS. In this MAS two agents perform a sequence of actions that represent all possible synchronization situations for two agents in the algorithm. Then we generate an occurrence graph for this simulation and formally prove that the algorithm works properly.

The simple MAS we used is depicted in Fig. 5. During the simulation the agents make 4 moves according to the path indicated in the figure. The agents synchronize with one another when they perceive each other. We suppose that an agent can only perceive the other agent if it is located at a neighboring field. The verification includes the following four situations. After the first move agent 1 requests agent 2 for synchronization, however agent 2 who has left the perceptual

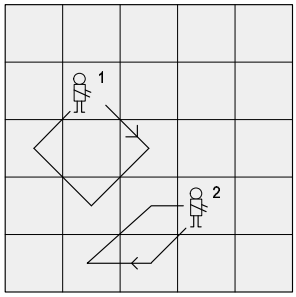

Fig. 5. Subsequent actions of the agents to verify the algorithm for two agents.

range of agent 1 rejects. After the second move agent 2 accepts the request, as a result the third move is made simultaneously. Finally the fourth move is made separately, without any synchronization request.

Fig. 6 depicts the CPN of the test net. The *world* token of color $World$ located in the *world* place represents the MAS environment, i.e. a 5x5 grid, with the agents on their locations. The color $World$ and the initial *world* token are defined as follows:

$color\ World = list\ Item\ with\ 0 \ldots worldsize * worldsize;$
$color\ Item = record\ name : Name * coord : Coordinate;$
$val\ world = [\{name = free, coord = (1,1)\}, \ldots, \{name = a1, coord = (2,2)\}, \ldots$
$\ldots, \{name = a2, coord = (4,4)\}, \ldots, \{name = free, coord = (5,5)\}] : Wold;$

The initial sequence of test moves for the agents are located in the $test\_moves1$ and $test\_moves2$ places. $TestMove$ is a list of $Move$, each $Move$ representing an action of an agent. The initial synchronization tokens of both agents are located in the *start_sync* place. Since in the initial situation no agent is able to perceive the other, both *memberset*'s are empty. When an agent finishes synchronization setup its synchronization token is put in the *end_sync* place. The *compose_regions* transition selects a matching region for the agent from the *colect_regions* place and add the agent as well as the members of its *memberset* set to that region. *Regions* is a list of *Region*, each region is defined as:

$color\ Region = list\ RegionMember\ with\ 0 \ldots max\_agents;$
$color\ RegionMember = product\ Name * Bool;$

When a region is expanded, the entry of the acting agent with name n is set to $(n, true)$, while a member m of this agent is set to (m, s) with s the previous state of the member as registered in the region, or $s = false$ if the member was not yet registered. If no matching region exists, a new region is added to the list of regions in *collect_regions*. The transition *merge_regions* ensures that regions with overlapping agents are merged into one composite region. As soon as the state of all agents of a region becomes *true* the region is ready to be handled, i.e. the agents of the region then make a move simultaneously. In the test we force

regions to be handled in a predefined order. The *test_regions* place contains a list of *Names* each element defining a set of names of agent that have to act simultaneously. When the region that corresponds with the names of the top element

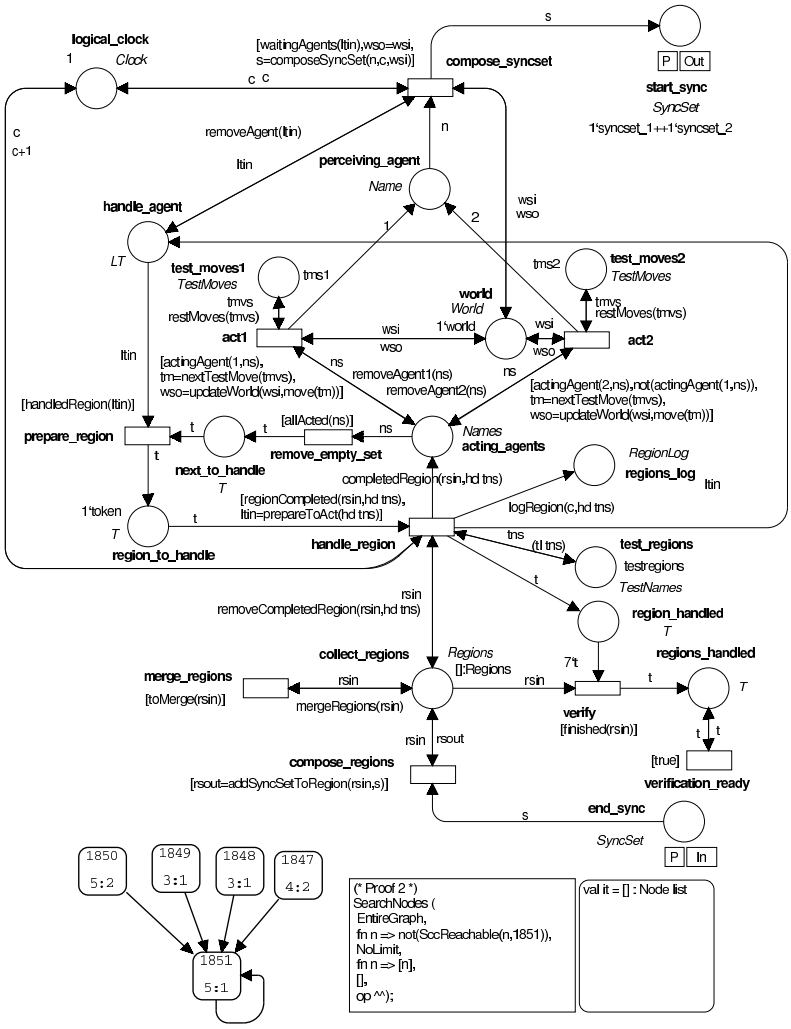

Fig. 6. The test net to prove the correctness of the algorithm for two agents.

of the list in *test_region* have finished synchronization setup, the *handle_region* transition fires. This results in the following effects: (1) the *Names* of the acting agents are put in the *acting_agents* place ($completedRegion(rsin, hdtns)$), (2) an anonymous token is passed to the *region_handled* place, (3) the value of *Clock* in the *logical_clock* place is incremented and (4) a token of color *LT* is

put in the *handle_agent* place. The *LT* token contains an anonymous token for each member of the region that is handled. From the *acting_agents* place agent 1 enables the *act*1 transition and agent 2 *act*2. When one of these transitions fires, it takes the next test move from the *test_moves*1 or *test_moves*2 place and uses this to move the corresponding agent to the next position in the world ($tm = nextTestMove(tmvs)$ and $wso = updateworld(wsi, move(tm))$). Simultaneously, the name of the acting agent is put in the *perceiving_agent* place. This enables the *compose_syncset* transition. When this transition fires a new synchronization token for the agent is composed and put it in the *start_sync* place. Together one anonymous token is collected from the *handle_agent* place. As soon as all agents of the region have acted, the token in the *handle_agent* place will enable the *prepare_region* transition. When this transition fires, an anonymous token is put in the *region_to_handle* place. This prepares the test net to handle the next region.

As soon as all predefined regions have acted and the agents have finished their final synchronization setup, the $verify$ transition is enabled. When this transition fires, an anonymous token is put in the $verification\_ready$ place. This makes the $verification\_ready$ transition live forever. Similar as for the correctness proof for one agent, to prove the correctness of the algorithm for two agents we have to prove that there exists a path from each node in the occurrence graph to the node that represents the final marking. The leaf node of the occurrence graph (node number 1851) is shown in Fig.6. The $SearchNodes$ function searches the nodes that have no path to the leaf node. Since the set of nodes that have no path is empty, we have proved that the synchronization algorithm works correctly for two agents.

4 Simulation

In addition to the formal proofs discussed in the previous section, we developed a test net that simulates the algorithm for four agents. Obviously, a simulation does not prove the correctness of the algorithm. However, the positive results from this simulation support the hypothesis that the algorithm is indeed completely correct. A complete formal proof should further substantiate this claim.

The set-up of the test net for the simulation is similar to the net we used to verify the algorithm for two agents. For the MAS, we used a 8x8 grid environment with four agents. The actions of the randomly scheduled agents are limited to move randomly. We performed simulations for agents with perceptual ranges of 1, 2 and 3 fields. The simulation ends when the agents together have performed 1000 moves. To interpret the simulation results, we added several extra logging places to the net such as places to register the number of times regions of different size have acted, a place to register the number of times regions have merged and a place to register the number of sent messages. The main results of the simulation are depicted in the table below.

The *Clock value* represents the value of the logical clock of the environment. Since the clock value is incremented each time a region acts, its value is equal

to the number of regions that have acted. For a perceptual range of 1 field, 768 regions have acted, i.e. the average region size is $1000/768 = 1,3$. For a perceptual range of 3 fields, 355 regions have acted, i.e. the average region size is $100/355 = 2,8$. These results confirm that agents with a larger perceptual scope form larger regions. The *Region size* values reflect the number of times each particular region size has occurred. E.g. for a perceptual range of 1 field, 584 agents have acted asynchronously (region size = 1) while only 12 regions of 4 agents have acted simultaneously. On the other hand, for a perceptual range of 3 fields, 156 agents have acted asynchronously and just as much regions of four agents have acted simultaneously. The *Region merges* values denote the number of times two subsets of synchronized agents are merged into one compound region. The communication overhead is denoted by the *Number of messages* that are sent. For a perceptual range of 3 fields about three times as much synchronization messages are sent as for a perceptual range of 1 field. :

Perceptual range	Clock value	Region size				Region merges	Number of messages
		1	2	3	4		
1	768	584	146	29	12	6	1849
2	439	162	62	104	114	8	3976
3	355	156	39	107	156	15	5282

Summarizing, larger perceptual ranges result in larger regions. Larger regions improve the possibility for simultaneous actions, however they also increase the communication overhead to setup synchronization.

5 Conclusions and Future Work

In this paper we presented a CPN for regional synchronization. Regional synchronization allows agents to perform simultaneous actions with other agents within their perceptual range. Regional synchronization avoids centralized control of the evolution of the MAS and improves scalability in comparison to central synchronization.

The algorithm combines a distributed two phase commit protocol with a logical clock. The detailed discussion of the CPN for regional synchronization shows that the the algorithm for regional synchronization is not trivial. In practice however, the algorithm can be implemented as a separate reusable module that, once correctly implemented, hides the complexity for the MAS designer.

We formally proved that the various steps of the algorithm for one agent are executed correctly. We also proved the correctness of the synchronization algorithm for two agents. To avoid state space explosion we had to schedule the agents round-robin for verification. In addition to the formal proofs, we discussed simulation results of the algorithm for four agents.

As future work it is our intention to extend the formal verification discussed in this paper to complete the proof of correctness of the algorithm for regional synchronization.

References

1. K. Birman *Building Secure and Reliable Network Applications*, Cornell University, Ithaca NY, 14853, 1995.
2. *Design/CPN*, http://www.daimi.aau.dk/designCPN/
3. J. Ferber, *Un modele de l'action pour les systemes multi-agents*, Journees sur les systemes multi-agents et l'intelligence artificielle distribue, Voiron, 1994.
4. J. Ferber, *Multi-Agent Systems, An Introduction to Distributed Artificial Intelligence*, Addison-Wesley, ISBN 0-201-36048-9, Great Britain, 1999.
5. M. N. Huhns and L. M. Stephens, *Multi-Agent Systems and Societies of Agents*, in G. Weiss ed., Multi-agent Systems, ISBN 0-262-23203-0, MIT press, 1999.
6. K. Jensen, *Coloured Petri Nets*, in Lecture Notes Computer Science, vol. 254, Advances in Petri Nets, Bad Honnef, Springer Verlag, 1986.
7. L. Lamport *Time, clocks and the ordering of events in a distributed system*, ACM, vol. 21, no. 7, pp.558-65, 1978.
8. D. Weyns and T. Holvoet, *A Colored Petri Net for a Multi-Agent Application*, Modeling Components, Objects and Agents, MOCA'02, Aarhus, Denmark, 2002.
9. D. Weyns and T. Holvoet, *Regional Synchronization for Simultaneous Actions in Situated Multi-Agent Systems*, 3th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS'03, Prague, Czech Republic, Proceedings in LNAI vol. 2691, 2003.
10. D. Weyns and T. Holvoet, *Model for Simultaneous Actions in Situated Multi-agent Systems*, First German Conference on Multi-Agent System Technologies, MATES'03, Erfurt Germany, Proceedings in LNCS vol. 2831, 2003.
11. D. Weyns and T. Holvoet, *Formal Model for Situated Multi-Agent Systems*, Formal Approaches for Multi-agent Systems, Special Issue of Fundamenta Informaticae, Eds. B. Dunin-Keplicz, R. Verbrugge, 2004. (to appear)