View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Research Papers in Economics

DynCNET: a negotiation and coordination protocol for dynamic task assignment

Willem De Roover
Nelis Boucké
Danny Weyns
Tom Holvoet

Report CW 566, October 2009

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

DynCNET: a negotiation and coordination protocol for dynamic task assignment

Willem De Roover
Nelis Boucké
Danny Weyns
Tom Holvoet
Report CW 566, October 2009

Department of Computer Science, K.U.Leuven

Abstract

Task assignment in Multi-Agent Systems is a complex coordination problem, especially in systems that operate under dynamic and changing conditions. Adaptive task assignment is used to handle these dynamic and changing circumstances. This technical document describes an adaptive task assignment protocol, DynCNET which is an extension of the Contract Net Protocol. In this document, the DynCNET protocol will be build step by step, starting from the Contract Net protocol. We will add dynamic task assignment, synchronization of abort messages and scope handling. The final result will be the DynCNET protocol with support for synchronization of abort messages and scope handling.

Keywords : task assignment, multi agent system, coordination, contract net
CR Subject Classification : I.2.11, I.2.8, I.2.1

DynCNET

a negotiation and coordination protocol for dynamic task assignment

By

Willem De Roover (willem.deroover@student.kuleuven.be)

Nelis Boucké (nelis.boucke@cs.kuleuven.be)

Danny Weyns (danny.weyns@cs.kuleuven.be)

Tom Holvoet (tom.holvoet@cs.kuleuven.be)

Katholieke Universiteit Leuven

Task assignment in Multi-Agent Systems is a complex coordination problem, especially in systems that operate under dynamic and changing conditions. Adaptive task assignment is used to handle these dynamic and changing circumstances. This technical document describes an adaptive task assignment protocol, DynCNET which is an extension of the Contract Net Protocol. In this document, the DynCNET protocol will be build step by step, starting from the Contract Net protocol. We will add dynamic task assignment, synchronization of abort messages and scope handling. The final result will be the DynCNET protocol with support for synchronization of abort messages and scope handling.

Contents

1. Introduction 3
2. AGV Transportation System 3
 2.1. AGV's And Transports 3
 2.2. Architecture 3
3. DynCNET 5
 3.1. CNET 6
 3.2. DynCNET 7
  3.2.1. DynCNET protocol for a Transport Agent 7
  3.2.2. DynCNET protocol for an AGV Agent 9
 3.3. Synchronization of abort messages 11
  3.3.1. DynCNET protocol for a Transport Agent with abort synchronization 12
  3.3.2. DynCNET protocol for an AGV Agent with abort synchronization 15
 3.4. Synchronization of scope handling 18
4. Conclusion 23
References 24

1. Introduction

The goal of this technical document is to explain the DynCNET protocol. DynCNET is a negotiation and coordination protocol for task assignment in Multi-Agent Systems. In DynCNET agents use explicit selection protocols and can negotiate about task assignment. We will describe DynCNET in the context of an Automated Transportation System. The information about DynCNET and the Automated Transportation System is based on the following paper: [1] and master thesis: [2] .

DynCNET is an extension for the contract net protocol (CNET) [3]. In this technical report we will build the DynCNET protocol step by step, starting from the CNET protocol. In the first step, we add dynamic task assignment. In the second step, we take in account synchronization of messages. In the last step, the solution is completed with scope handling.

Overview. This paper is structured as follows: section 2 presents the AGV Transportation system. In section 3 we will describe the DynCNET protocol. We conclude in section 4.

2. AGV Transportation System

In this section we present the AGV Transportation System. In section 2.1 the basic elements are explained. Section 2.2 discusses the architectural elements that are relevant for the rest of this paper.

2.1. AGV's And Transports

DynCNET is applied to an Automated Transport System that uses several AGVs to perform transports. An AGV is an unmanned, computer-controlled vehicle capable of taking a load, driving it around and putting it down. The AGV's are used to perform transports. A transport is a task to pick up a load from a location and drop it at another location. Transport tasks are generated by the warehouse management system. Transports won't be performed immediately since an AGV has to drive to the pickup location first. While the AGV is driving to the load all kind of changes can happen to the system. New AGV's can become available to perform the tasks, other tasks can become available that are more suitable for the AGV, etc.

2.2. Architecture

Figure 1 shows the architecture for the AGV Transportation system. The main components in the architecture are the Transport Agent and the AGV Agent.

Every transportation task will have a Transport Agent who is responsible for the task. The main objectives for the Transport Agent are the communication of its task to the AGV's within the scope of the task and the assignment of the task to an AGV. The decomposition of the Transport Agent component in Figure 1 is shown in Figure 2. The Communication component is responsible for the communication with other agents. Perception perceives the local environment based on request coming from the Communication component. Everything

the Transport Agent knows at a certain point in time is stored in the Current Knowledge repository. Table 1 defines the Transport Agent's interfaces that are relevant for this paper.

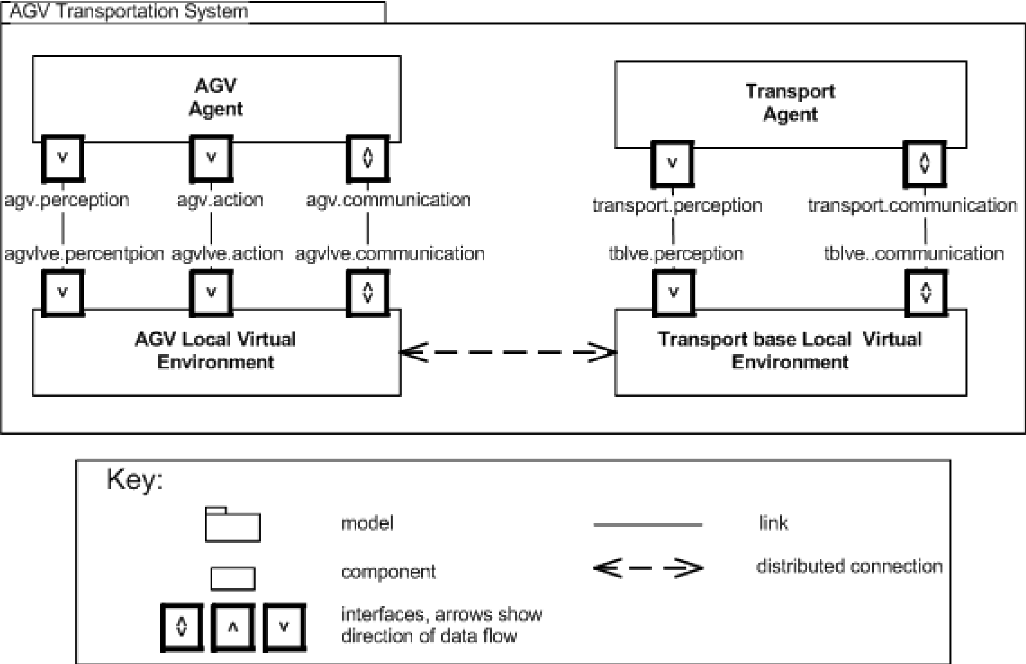


FIGURE 1: AGV TRANSPORTATION SYSTEM ARCHITECTURE

Each AGV has a single AGV Agent. The main responsibility of an AGV agent are obtaining tasks from Transport Agents, handling the tasks and reporting its completion. Figure 2 presents the decomposition for the AGV Agent component in Figure 1. As with the Transport Agent, the Communication component is responsible for the communication with other agents. The Decision Making component is responsible for selecting actions. The Perception component perceives the local environment based on request coming from the Communication component and the Decision Making component. The AGV's knowledge is stored in the Current Knowledge repository. Table 1 defines the interfaces for an AGV Agent that are relevant for this paper.

The Local Virtual environments are software entities that represent and maintain the relevant state of the physical environment and offer distributed communication between other Local Virtual Environments. Important for this paper is that it offers a 'network' infrastructure through which the agents can communicate with each other.

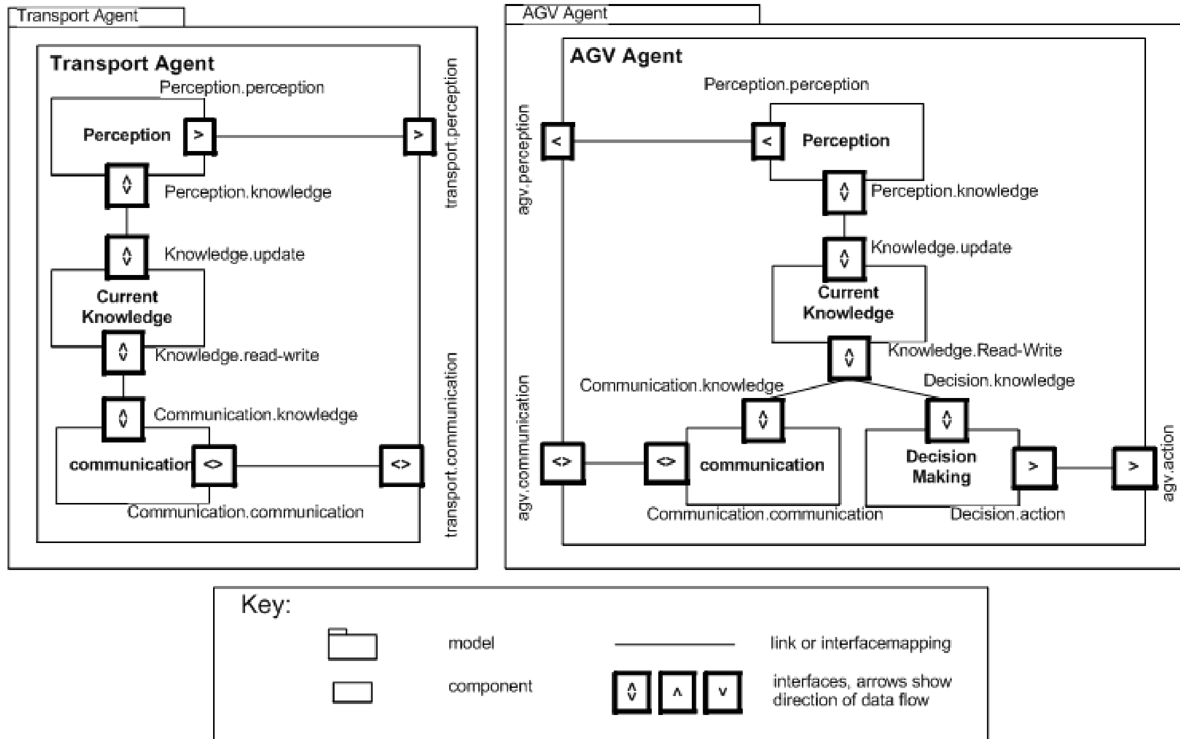


FIGURE 2: TRANSPORT AGENT AND AGV AGENT ARCHITECTURE

Transport Agent	AGV Agent
Communication.communication	
• out: void sendCallForProposal(Cfp cfp) • out: void sendProvisionalAccept(Accept accept) • out: void sendAbort(Abort abort) • in: void receiveProposal(Prop prop) • in: void receiveRetracted(Retract retract) • in: void receiveBound(Bound bound) • in: void receiveAcceptAbort(Abort abort) • in: void receiveRefuseAbort(Abort abort)	• in: void receiveCallForProposal(Cfp cfp) • in: void receiveProvisionalAccept(Accept accept) • in: void receiveAbort(Abort abort) • out: void sendProposal(Prop prop) • out: void sendRetracted(Retract retract) • out: void sendBound(Bound bound) • out: void sendAcceptAbort(Abort abort) • out: void sendRefuseAbort(Abort abort)
Knowledge.update	
• in: void ParticipantInScope(Id id) • in: void ParticipantOutOfScope(Id id)	• in: void TaskInScope(Id id) • in: void TaskOutOfScope(Id id)

TABLE 1: INTERFACE DEFINITIONS FOR A TRANSPORT AGENT AND AN AGV AGENT

3. DynCNET

DynCNET is an extension of the CNET protocol, with “Dyn” referring to the dynamic approach of task assignment. In DynCNET Agents use explicit selection protocols and can negotiate about task assignment. Agents regularly re-analyze the current environment and adapt the assignment of tasks when circumstances change.

In the following sections we will explain the DynCNET protocol by gradually extending the CNET procotol to the DynCNET protocol. Section 3.1 describes the CNET protocol. In section 3.2 we will add dynamic task assignment to the CNET protocol. Due to the dynamic task assignment DynCNET can encounter problems with the synchronization of messages. A solution to this problem is added to the protocol in section 3.3. Another synchronization

problems occurs due to the mobility of the AGV's. The protocol is extended with a solution for this problem in section 3.4.

3.1. CNET

The CNET protocol specifies the interaction between agents for competitive negotiation through the use of contracts. In essence, CNET allows tasks to be distributed among a group of agents [4].

The CNET protocol works as follows:

1. The Transport Agent sends out a call for proposals.
2. The AGV Agents within the scope of the tasks answer with a proposal.
3. The Transport Agent selects a winner and notifies the corresponding AGV Agent.
4. When the selected AGV arrives at the load the AGV Agent notifies the Transport Agent that the task is started.

The protocol is shown in Figure 3.

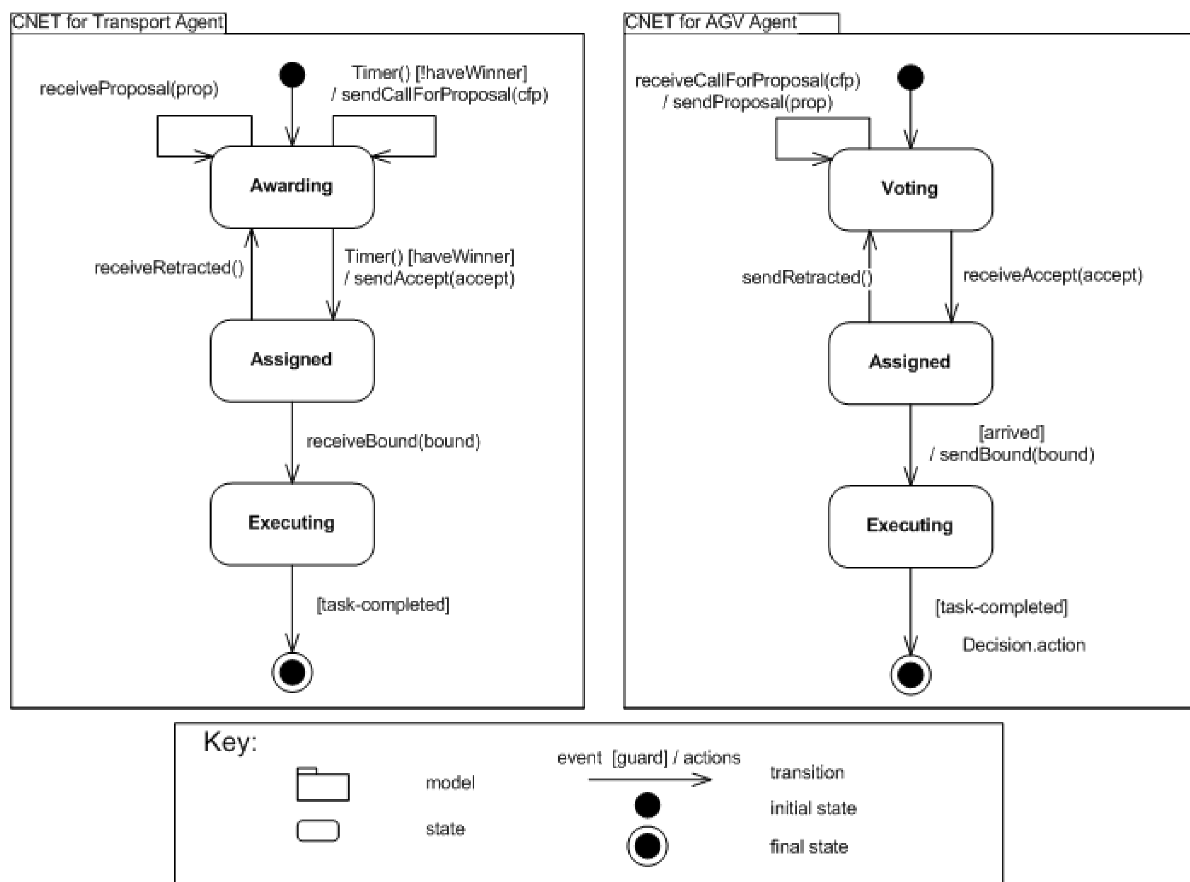


FIGURE 3: CNET PROTOCOL

3.2. DynCNET

The DynCNET protocol can be viewed upon from two different perspectives: The Transport Agent and the AGV Agent. The next sections discuss the DynCNET protocol from both perspectives.

3.2.1. DynCNET protocol for a Transport Agent

The DynCNET protocol for a Transport Agent will be explained using Figure 4. The Transport Agent will reside in the Awarding state when no AGV has been selected for the execution of its task. In this state, call for proposals will be sent to the AGV Agents within the scope of the task. Next, a winner is chosen based on the proposals arriving from the AGV's. By sending a provisional accept message to the winner the Transport Agent transitions to the assigned state. This means that there is a provisional agreement between the Transport Agent and the AGV agent about performing the transport task. When the AGV picks up the load, the AGV Agent notifies the Transport Agent that the task has started by means of a bound message. On receiving this message, the Transport Agent will transition to the Executing state. At this point the DynCNET protocol only defers from the CNET protocol in that it sends out a provisional accept message instead of an accept message.

Consider the scenario in Figure 5 in which AGV B has a provisional agreement with task 2. While AGV B drives towards the transport location for task 2, AGV A drops its current load at transport location 1 and becomes available for task 2. This is an opportunity for task 2 to switch to another, better suited AGV. DynCNET is able to exploit such opportunities. When the Transport Agent resides in the Assigned state it will keep sending out call for proposals to discover and exploit opportunities. When a better proposals arrives the Transport Agent will transition to the SwitchParticipant state in which an abort message is send to the currently assigned AGV. After the abort message has been send, a provisional accept is send to the new winner.

Table 2 defines relations between the interface definitions from the Transport Agent Architecture (Table 1) and the State Machine describing the DynCNET protocol for a Transport Agent (Figure 4). Two types of relations are defined: EventInterfaceMatches and ActionInterfaceMatches. An EventInterfaceMatch relates the event of a transition to a method in an interface if calling the method results in triggering the transition and thus perform the transition. In Table 2 all receive methods are EventInterfaceMatches which will be called upon the Communication component from the Transport Base Local Virtual Environment. An ActionInterfaceMatch relates an action in a transition to an interface method if performing the transition results in calling the method. All send methods in Table 2 are ActionInterfaceMatches which will be called upon the Transport Base Local Virtual Environment component if the corresponding transition is performed.

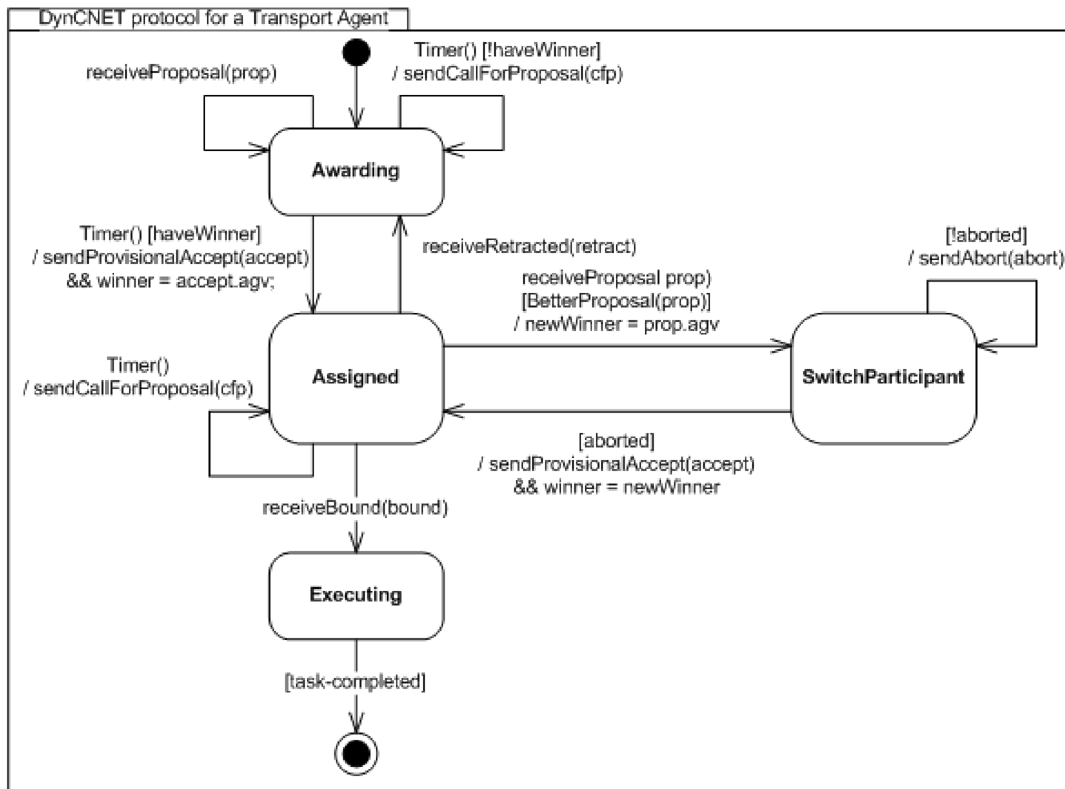


FIGURE 4: DYNCNET PROTOCOL FOR A TRANSPORT AGENT (USING FIGURE 3 KEY)

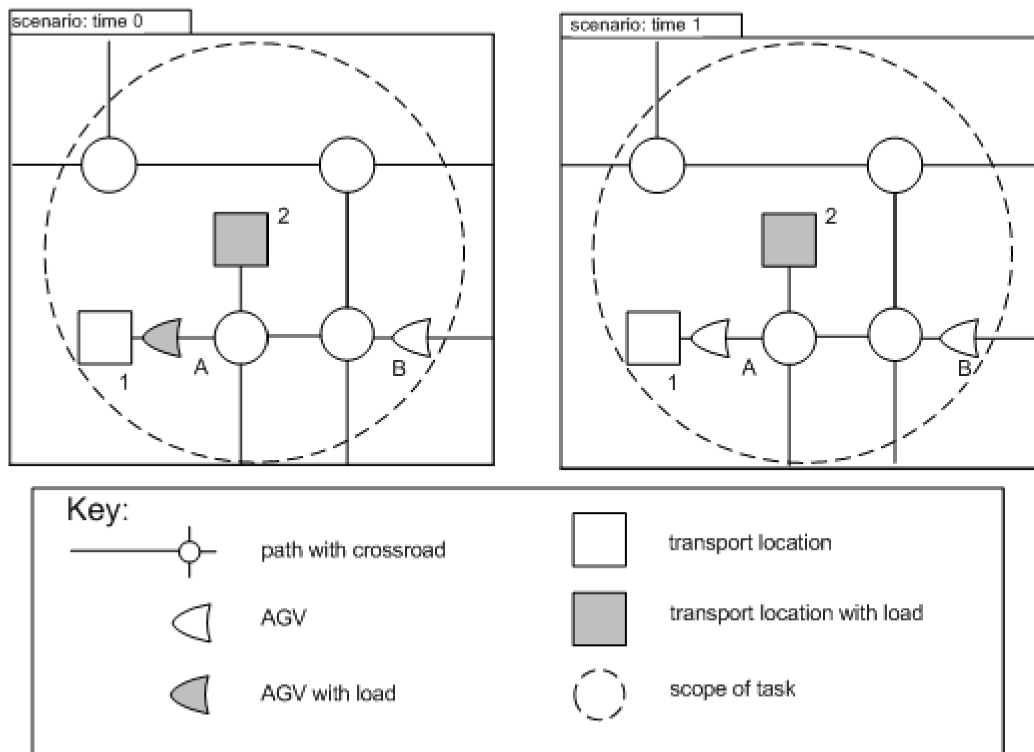


FIGURE 5: SCENARIO WITH AGV A BECOMING AVAILABLE FOR TASK 2 WHEN IT DROPS ITS LOAD AT TRANSPORT LOCATION 1.

- ActionInterfaceMatch (1)
  - Action: AwardingToAwarding.sendCallForProposal(cfp)
  - Method: out: void sendCallForProposal(CFP cfp)
- ActionInterfaceMatch (2)
  - Action: AwardingToAssigned.sendProvisionalAccept(accept)
  - Method: out: void sendProvisionalAccept(Accept accept)
- ActionInterfaceMatch (3)
  - Action: AssignedToAssigned.sendCallForProposal(cfp)
  - Method: out: void sendCallForProposal(CFP cfp)
- ActionInterfaceMatch (4)
  - Action: SwitchParticipantToSwitchParticipant.sendAbort(abort)
  - Method: out: void sendAbort(Abort abort)
- ActionInterfaceMatch (5)
  - Event:SwitchParticipantToAssigned.sendProvisionalAccept(accept)
  - Method: out: void sendProvisionalAccept(Accept accept)
- EventInterfaceMatch (6)
  - Event:AssignedToAwarding.receiveRetracted(retract)
  - Method: in: void receiveRetracted(Retract retract)
- EventInterfaceMatch (7)
  - Event:AssignedToExecuting.receiveBound(bound)
  - Method: in: void receiveBound(Bound bound)
- EventInterfaceMatch (8)
  - Event:AwardingToAwarding.receiveProposal(prop)
  - Method: in:void receiveProposal(Prop prop)
- EventInterfaceMatch (9)
  - Event:AssignedToSwitchParticipant.receiveProposal(iprop)
  - Method: in: void receiveProposal(Prop prop)

TABLE 2: INTERFACEMATCHES FOR THE DYNCNET PROTOCOL FOR A TRANSPORT AGENT

3.2.2. DynCNET protocol for an AGV Agent

Figure 6 presents the DynCNET protocol for an AGV Agent. When no task is assigned to the AGV Agent, the AGV Agent resides in the Voting state. In the voting state it receives call for proposals which are answered with proposals to perform the requested task. If the AGV is chosen it receives a provisional accept and transition to the Intentional state. This means that there is a provisional agreement between the Transport Agent and the AGV agent about performing the transport task. Once the AGV arrives at the pickup location, the AGV Agent sends a bound to the corresponding Transport Agent.

Consider the scenario in Figure 7 in which AGV A has a provisional agreement with task 1. When AGV A drives towards the transport location for task 1, task 2 enters the system. This is an opportunity for AGV A to switch to a better suited task. DynCNET enables AGV's to switch tasks and thus exploit such opportunities. When the AGV resides in the Intentional state, the AGV Agent keeps replying to call for proposals. By doing this it can find more suitable task. If the AGV Agent receives a provisional accept for such a task it transitions to the Switch Initiator state. The AGV Agent retracts itself from its current task. Once the AGV agent is retracted from its task, it can switch to its new task and transition back to the intentional state.

Table 3 summarizes the relations between the interface definitions from the Transport Agent Architecture (Table 1) and the State Machine describing the DynCNET protocol for an AGV Agent (Figure 6). As with the Transport Agent all receive method are EventInterfaceMatches and all send methods are ActionInterfaceMatches.

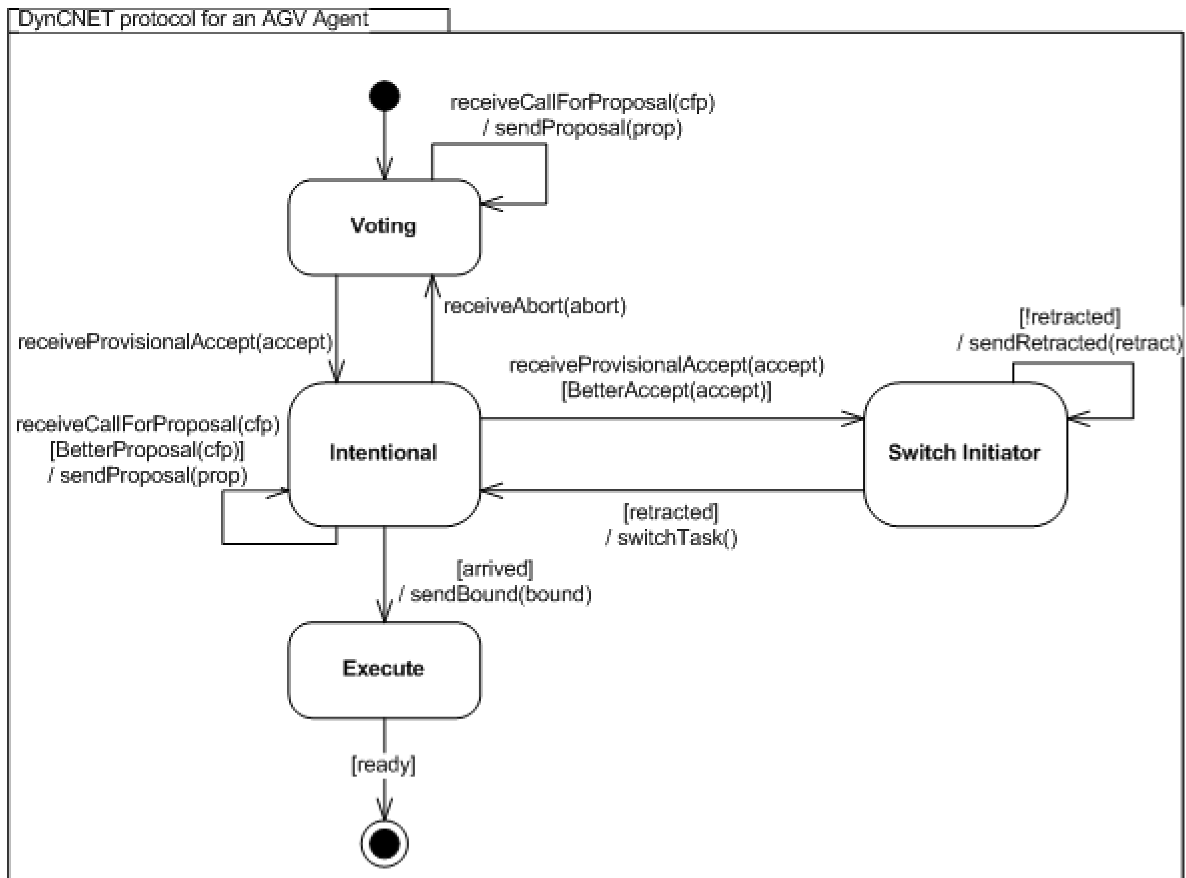


FIGURE 6: DYNCNET PROTOCOL FOR AN AGV AGENT (USING FIGURE 3 KEY)

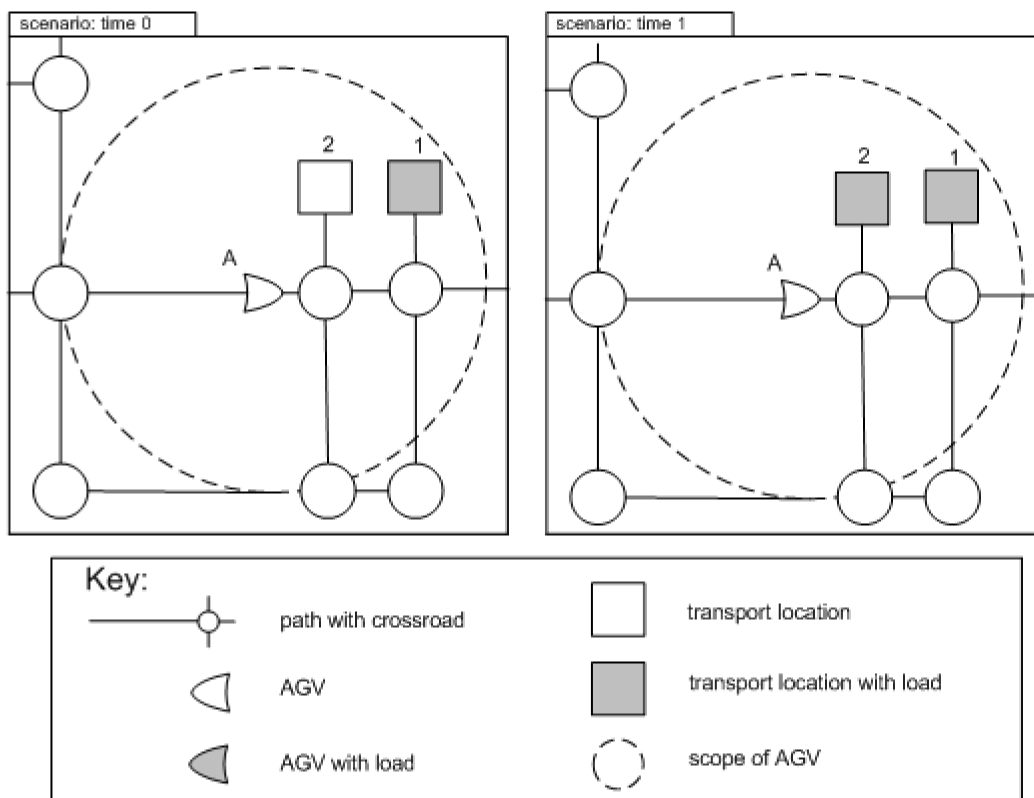


FIGURE 7: SCENARIO WITH TASK 2 BECOMING AVAILABLE FOR AGV A

- EventInterfaceMatch (1)
  - Event: VotingToVoting.receiveCallForProposal(cfp)
  - Method: in: void receiveCallForProposal(Cfp cfp)
- EventInterfaceMatch (2)
  - Event: VotingToIntentional.receiveProvisionalAccept(accept)
  - Method: in: void receiveProvisionalAccept(Accept accept)
- EventInterfaceMatch (3)
  - Event: IntentionalToIntentional.receiveCallForProposal(cfp)
  - Method: in: void receiveCallForProposal(Cfp cfp)
- EventInterfaceMatch (4)
  - Event: IntentionalToSwitchInitiator.receiveProvisionalAccept(accept)
  - Method: in: void receiveProvisionalAccept(Accept accept)
- EventInterfaceMatch (5)
  - Event: IntentionalToVoting.receiveAbort(abort)
  - Method: in: void receiveAbort(Abort abort)
- ActionInterfaceMatch (6)
  - Action: VotingToVoting.sendProposal(prop)
  - Method: out: void sendProposal(Prop prop)
- ActionInterfaceMatch (7)
  - Action: IntentionalToIntentional.sendProposal(prop)
  - Method: out: void sendProposal(Prop prop)
- ActionInterfaceMatch (8)
  - Action: SwitchInitiatorToSwitchInitiator.sendRetracted(retract)
  - Method: out: void sendRetracted(Retract retract)
- ActionInterfaceMatch (9)
  - Action: IntentionalToExecute.sendBound(bound)
  - Method: void sendBound(Bound bound)

TABLE 3: INTERFACEMATCHES FOR THE DYNCNET PROTOCOL FOR AN AGV AGENT

3.3. Synchronization of abort messages

The previous section described the basic DynCNET protocol for the Transport and AGV Agent. To keep the protocol clear and simple, synchronization issues were not taken into account. The synchronization of abort messages will be discussed in this section.

When a Transport Agent receives a better proposal from an AGV, the Transport Agent assigns the task to this AGV. But first an abort message is send to the currently assigned AGV. However due to the distributed environment it's possible that the currently assigned AGV has already started executing the task while the Transport Agent hasn't received a bound message yet, i.e. because of network delays. Figure 8 shows the assignment of a tasks to a better suited AGV while the currently assigned AGV is already executing the task.

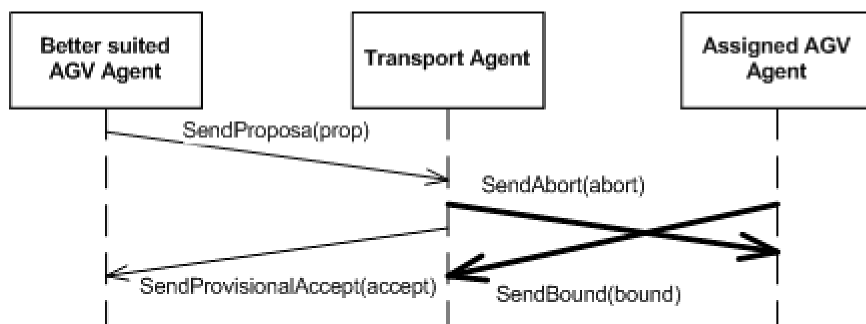


FIGURE 8: MESSAGE SYNCHRONIZATION PROBLEM

Figure 9 is used to solve this synchronization issue. When a better proposal arrives at the Transport Agent, the Transport Agent will transition to the aborting state. In this state an abort message is sent to the currently assigned AGV. After sending the abort message the Transport Agent will wait in the WaitingToAbort state for the AGV's answer. In case the AGV is in the Intentional state and thus hasn't started executing the task yet, an AcceptAbort message is send to the Transport Agent while the AGV Agent transitions back to the Voting state. When the Transport Agent receives the AcceptAbort message a provisional accept is send to the new winner. In case the AGV is in the Execute state and thus has started executing the task, a RefuseAbort message is send to the Transport Agent. When the Transport Agent receives the RefuseAbort message, it transitions to the Executing state

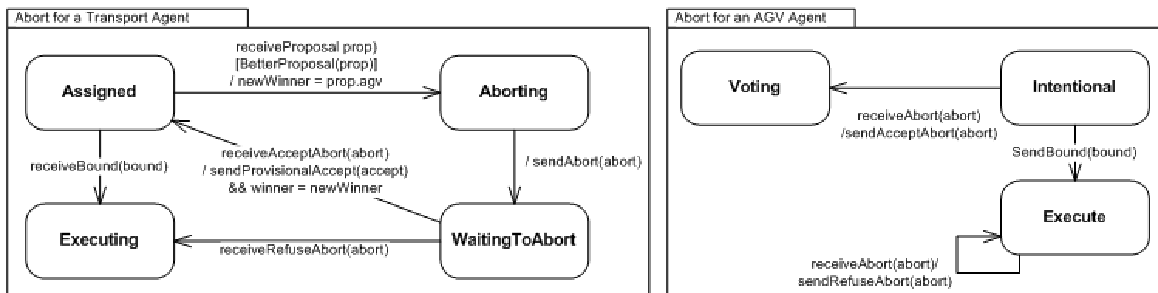


FIGURE 9: SYNCHRONIZATION FOR ABORT (USING FIGURE 3 KEY)

3.3.1. DynCNET protocol for a Transport Agent with abort synchronization

Table 4 describes the relations between the interface definitions from the Transport Agent Architecture (Figure 2) and the State Machine describing the abort synchronization for a Transport Agent (Figure 9).

- ActionInterfaceMatch (a)
  - Action: AbortingToWaitingtoAbort.sendAbort(abort)
  - Method: out: void sendAbort(Abort abort)
- EventInterfaceMatch (b)
  - Action: WaitingToAbortToAssigned.receiveAcceptAbort(abort)
  - Method: in: void receiveAcceptAbort(Abort abort)
- ActionInterfaceMatch (c)
  - Action: WaitingToAbortToAssigned.sendProvisionalAccept(accept)
  - Method: out: void sendProvisionalAccept(Accept accept)
- EventInterfaceMatch (d)
  - Event:WaitingToAbortToExecuting.receiveRefuseAbort(abort)
  - Method: in: void receiveRefuseAbort(abort)
- EventInterfaceMatch (e)
  - Event: AssignedToExecuting.receiveBound(bound)
  - Method: in: void receiveBound(Bound bound)
- EventInterfaceMatch (f)
  - Event: AssignedToAborting.receiveProposal(prop)
  - Method: in: void receiveProposal(Prop prop)

TABLE 4: INTERFACEMATCHES FOR ABORT SYNCHRONIZATION FOR A TRANSPORT AGENT

The synchronization of abort messages for a Transport Agent is joined with the DynCNET protocol for a Transport Agent. The composition of the state machines is done by using relations. Therefore relations need to be defined between both state machines. Figure 10

describes these relations. Two types of relations are defined: Unification and SubElement relations. Unification relations define that two elements from different state machines are the same. SubElement relations define that some elements from a state machine are subelements of an element from another state machine. By using these relations the state machines can be united to form one model. These relations can also be used to join the previously defined InterfaceMatches for both State Machines. Figure 11 shows the composition of the DynCNET protocol and the solution for the synchronization problem. Table 5 summarizes the Event- and ActionInterfaceMatches generated from the original InterfaceMatches in Table 2 and Table 4 and the State Machine relations from Figure 10.

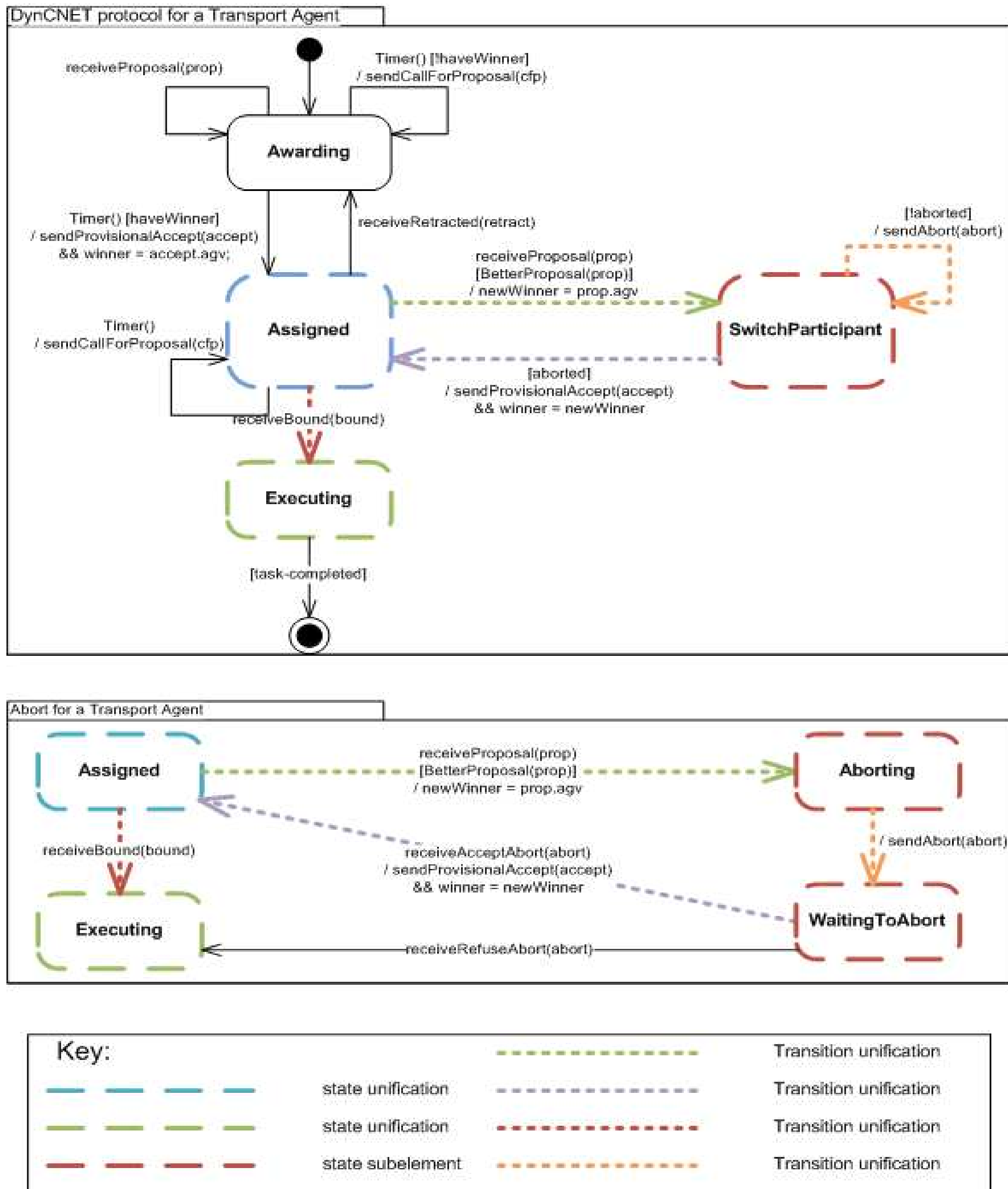


FIGURE 10: RELATIONS BETWEEN DYNCNET PROTOCOL FOR A TRANSPORT AGENT AND ABORT SYNCHRONIZATION FOR A TRANSPORT AGENT

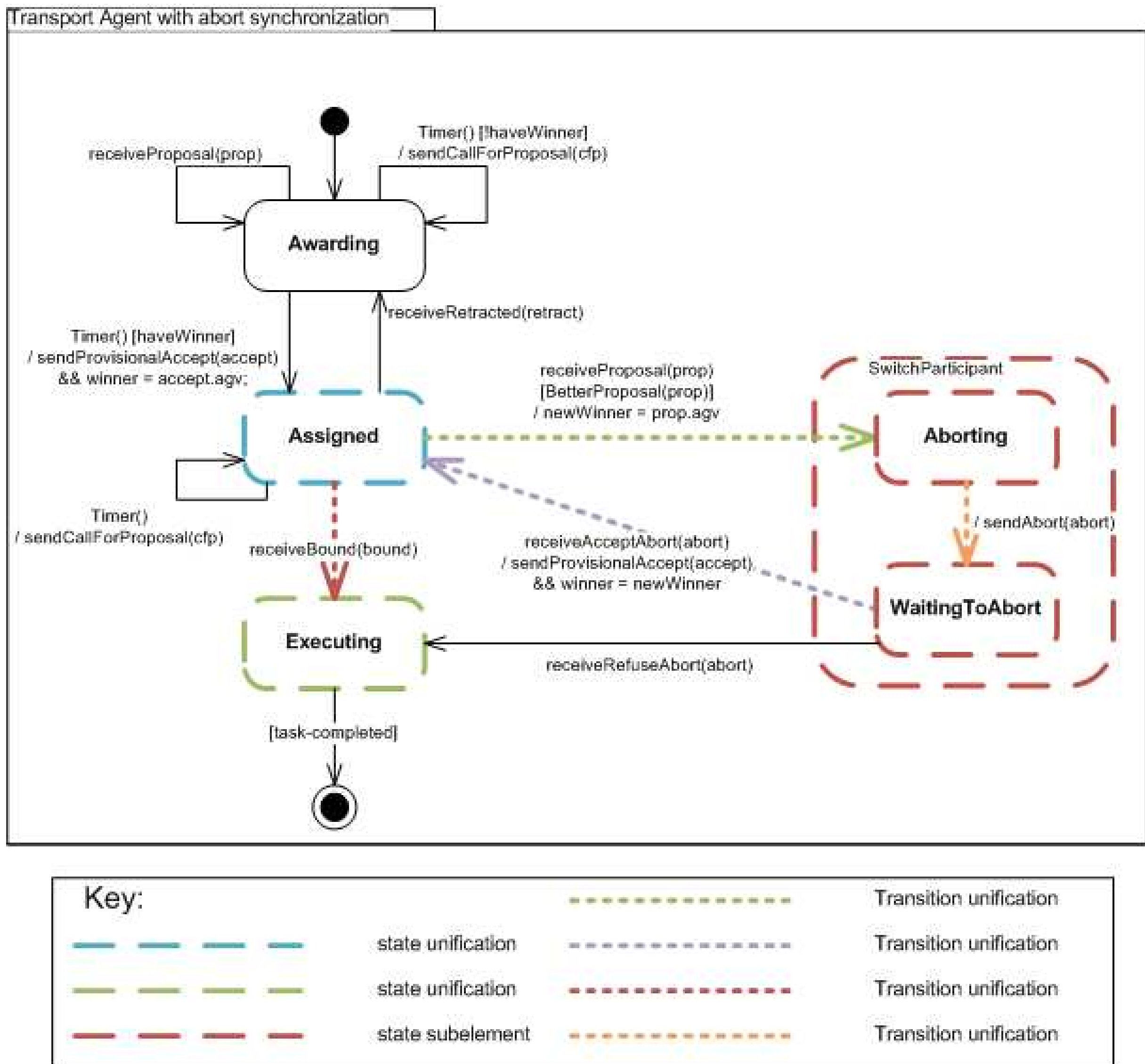


FIGURE 11: TRANSPORT AGENT WITH SYNCHRONIZATION OF ABORT MESSAGES

- ActionInterfaceMatch (1)
  - Event: AwardingToAwarding.sendCallForProposal(cfp)
  - Method: out: void sendCallForProposal(CFP cfp)
- ActionInterfaceMatch (2)
  - Event: AwardingToAssigned.sendProvisionalAccept(accept)
  - out: void sendProvisionalAccept(Accept accept)
- ActionInterfaceMatch (3)
  - Event: AssignedToAssigned.sendCallForProposal(cfp)
  - Method: out: void sendCallForProposal(CFP cfp)
- ActionInterfaceMatch (4 + a)
  - Event: AbortingToWaitingtoAbort.sendAbort(abort)
  - Method: out: void sendAbort(Abort abort)
- EventInterfaceMatch (b)
  - Event: WaitingToAbortToAssigned.receiveAcceptAbort(abort)
  - Method: in: void receiveAcceptAbort(Abort abort)
- ActionInterfaceMatch (5 + c)

- Action: WaitingToAbortToAssigned.sendProvisionalAccept(accept)
  - Method: out: void sendProvisionalAccept(Accept accept)
- EventInterfaceMatch (d)
  - Event: WaitingToAbortToExecuting.receiveRefuseAbort(abort)
  - Method: in: void receiveRefuseAbort(Abort abot)
- EventInterfaceMatch (6)
  - Event: AssignedToAwarding.receiveRetracted(retract)
  - Method: in: void receiveRetracted(Retract retract)
- EventInterfaceMatch (7 + e)
  - Event: AssignedToExecuting.receiveBound(bound)
  - Method: in: void receiveBound(Bound bound)
- EventInterfaceMatch (8)
  - Event: AwardingToAwarding.receiveProposal(Prop prop)
  - Method: in: void receiveProposal(Prop prop)
- EventInterfaceMatch (9 + f)
  - Event: AssignedToAborting.receiveProposal(prop)
  - Method: in: void receiveProposal(Prop prop)

TABLE 5: INTERFACEMATCHES FOR A TRANSPORT AGENT WITH ABORT SYNCHRONIZATION

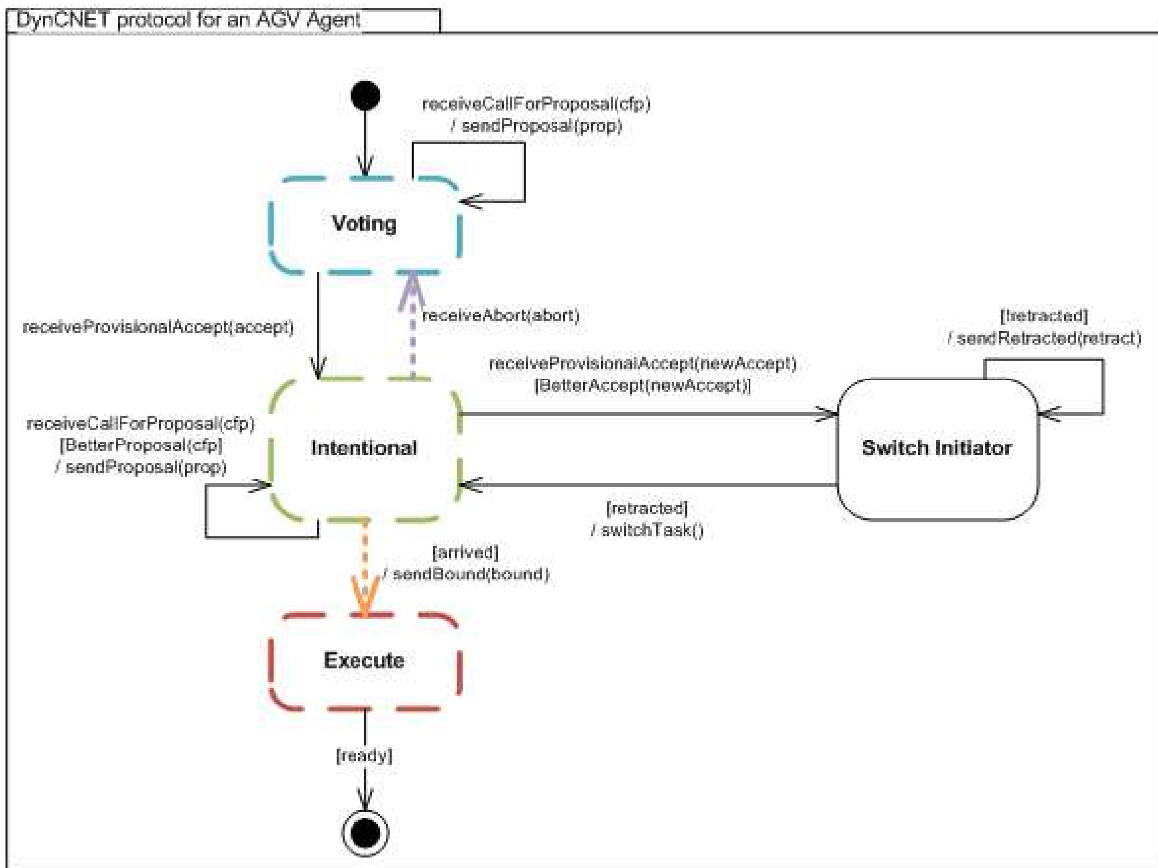
3.3.2. DynCNET protocol for an AGV Agent with abort synchronization

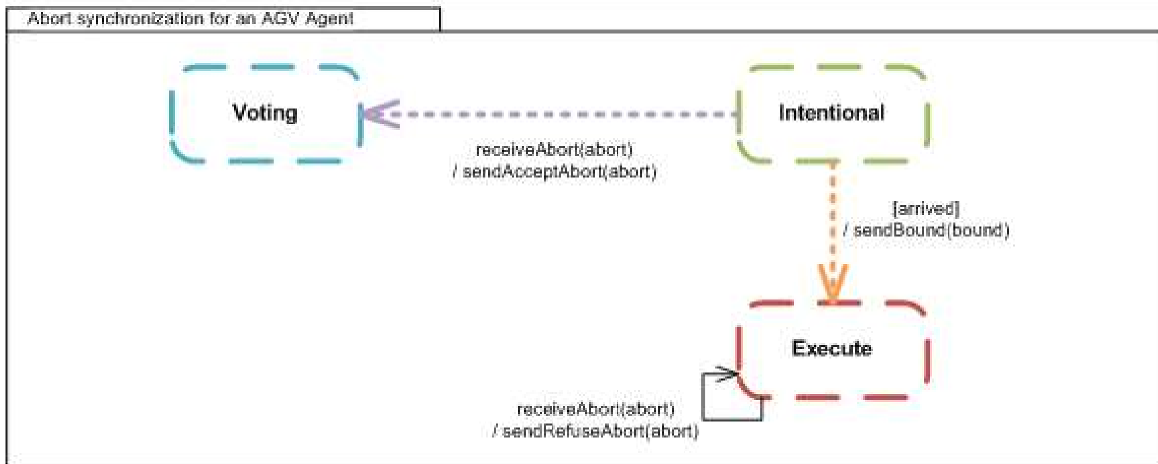
Table 4 describes the relations between the interface definitions from the Transport Agent Architecture (Figure 2) and the State Machine describing the abort synchronization for an AGV Agent (Figure 9).

- ActionInterfaceMatch (a)
  - Action: IntentionalToVoting.receiveAbort(abort)
  - Method: in: void receiveAbort(Abort abort)
- ActionInterfaceMatch (b)
  - Action: ExecuteToExecute.receiveAbort(abort)
  - Method: in: void receiveAbort(Abort abort)
- EventInterfaceMatch (c)
  - Event: IntentionalToVoting.sendAcceptAbort(abort)
  - Method: out: void sendAcceptAbort(Abort abort)
- EventInterfaceMatch (d)
  - Event: ExecuteToExecute.sendRefuseAbort(abort)
  - Method: out: void sendRefuseAbort(Abort abort)
- EventInterfaceMatch (e)
  - Event: IntentionalToExecute.sendbound(bound)
  - Method: out: void sendBound(Bound bound)

TABLE 6: INTERFACEMATCHES FOR ABORT SYNCHRONIZATION FOR AN AGV AGENT

Next, the synchronization of abort messages for an AGV Agent is joined with the DynCNET protocol for an AGV Agent. Figure 12 describes the relations between both state machines. The composition is shown in Figure 13. InterfaceMatches for the composition have been generated from the original InterfaceMatches in Table 3 and Table 6 and the state machine relations from Figure 12. These InterfaceMatches are presented in Table 7.





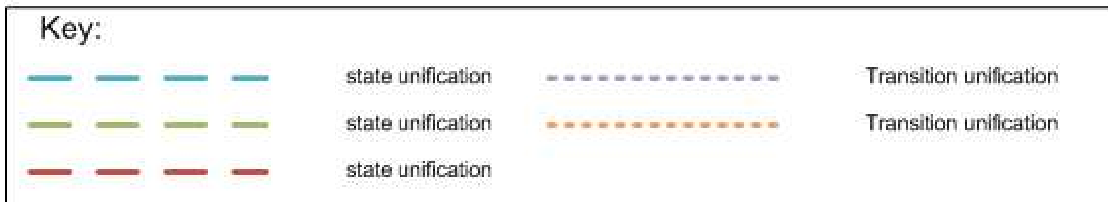


FIGURE 12: RELATIONS BETWEEN DYNCNET PROTOCOL FOR AN AGV AGENT AND ABORT SYNCHRONIZATION FOR AN AGV AGENT

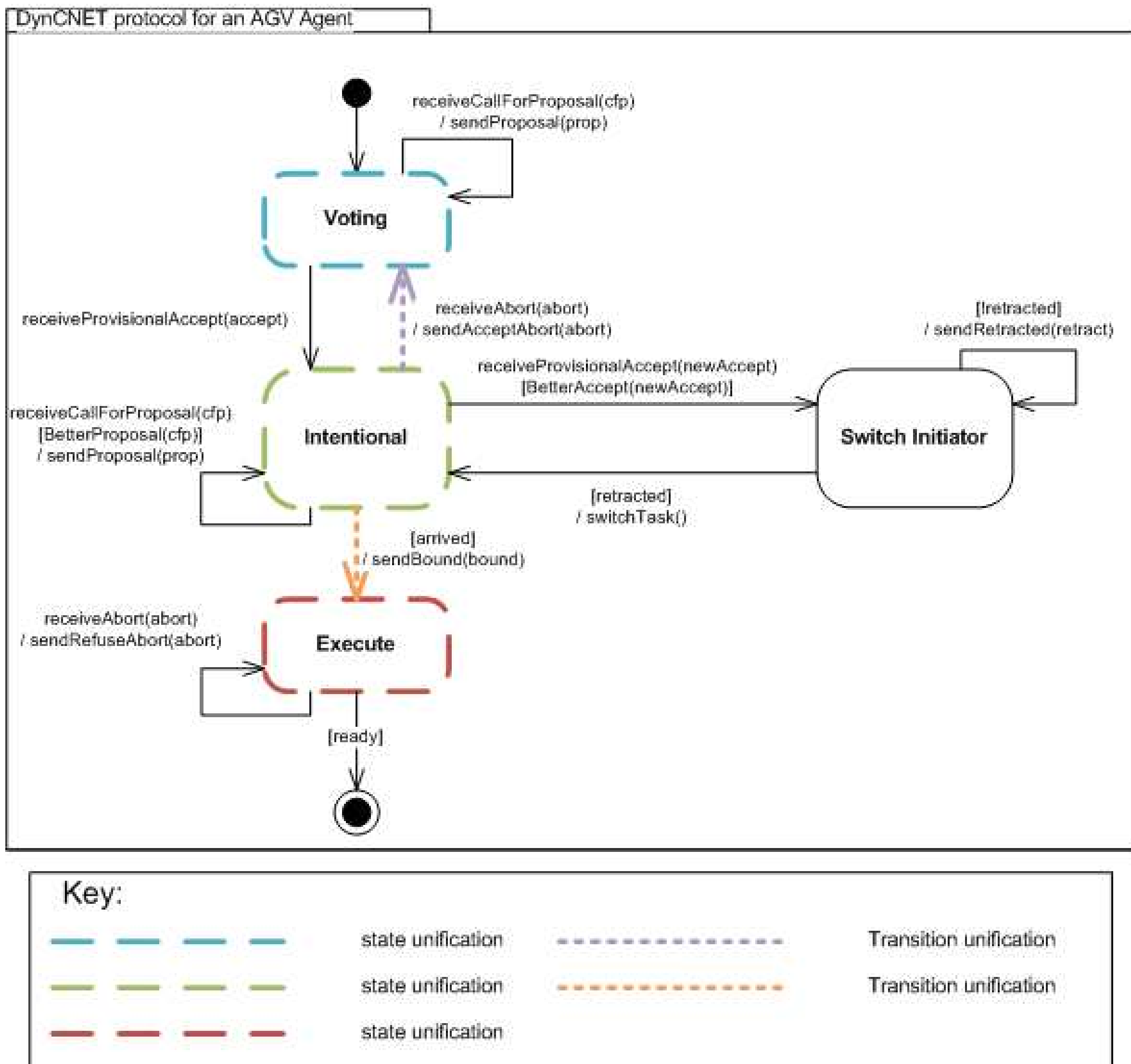


FIGURE 13: AGV AGENT WITH ABORT SYNCHRONIZATION

- EventInterfaceMatch (1)
  - Event: VotingToVoting.receiveCallForProposal(cfp)
  - Method: in: void receiveCallForProposal(Cfp cfp)
- EventInterfaceMatch (2)
  - Event: VotingToIntentional.receiveProvisionalAccept(accept)
  - Method: in: void receiveProvisionalAccept(Accept accept)
- EventInterfaceMatch (3)
  - Event: IntentionalToIntentional.receiveCallForProposal(cfp)
  - Method: in: void receiveCallForProposal(Cfp cfp)
- EventInterfaceMatch (4)
  - Event: IntentionalToSwitchInitiator.receiveProvisionalAccept(newAccept)
  - Method: in: void receiveProvisionalAccept(Accept accept)
- EventInterfaceMatch (5 + a)
  - Event: IntentionalToVoting.receiveAbort(abort)
  - Method: in: void receiveAbort(Abort abort)
- ActionInterfaceMatch (6)
  - Action: VotingToVoting.sendProposal(prop)
  - Method: out: void sendProposal(Prop prop)
- ActionInterfaceMatch (7)

- Action: IntentionalToIntentional.sendProposal(prop)
  - Method: out: void sendProposal(Prop prop)
- ActionInterfaceMatch (8)
  - Action: SwitchInitiatorToSwitchInitiator.sendRetracted(retract)
  - Method: out: void sendRetracted(Retract retract)
- ActionInterfaceMatch (9 + e)
  - Action: IntentionalToExecute.sendBound(bound)
  - Method: void sendBound(Bound bound)
- EventInterfaceMatch (b)
  - Action: ExecuteToExecute.receiveAbort(abort)
  - Method: in: void receiveAbort(Abort abort)
- ActionInterfaceMatch (c)
  - Event: IntentionalToVoting.sendAcceptAbort(abort)
  - Method: out: void sendAcceptAbort(Abort abort)
- ActionInterfaceMatch (d)
  - Event: ExecuteToExecute.sendRefuseAbort(abort)
  - Method: out: void sendRefuseAbort(Abort abort)

TABLE 7: INTERFACEMATCHES FOR AN AGV AGENT WITH ABORT SYNCHRONIZATION

3.4. Synchronization of scope handling

A second synchronization issue occurs when AGV Agents leave the scope of their assigned task. When an AGV leaves the scope of its task, the Transport Agent has to assign the task to a new AGV.

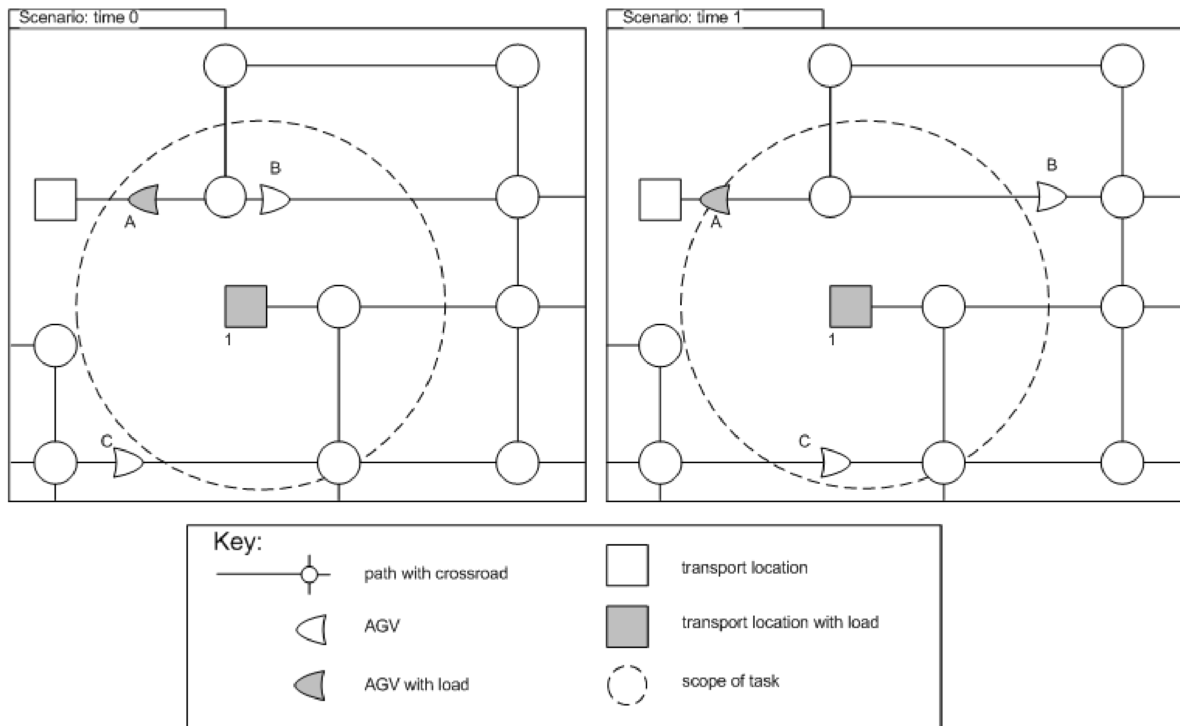


FIGURE 14: SCOPE HANDLING SYNCHRONIZATION PROBLEM WITH TASK 1 ALLOCATED TO AGV B

Consider the scenario in Figure 14 in which AGV B has a provisional agreement to execute task 1. While B drives towards the pickup location it leaves the scope of its task. When this happens the task will have to be reassigned to a new AGV, e.g. AGV C.

Figure 15 describes a solution to this issue. This synchronization problem can only occur in a few situations:

1. When a Transport Agent resides in the Assigned state the AGV can leave the scope of the task. In this case the Transport Agent can easily transition back to the awarding state and look for another AGV.
2. A more difficult situation occurs when the Transport Agent receives a better proposal from an AGV and this AGV goes out of scope. We need to consider two cases:
    a. No abort message has been send to the originally assigned AGV: in this case the Transport Agent can transition from the aborting state to the Assigned state.
    b. An abort message has been send to the original assigned AGV: In this case the Transport Agent transitions to the ParticipantOutOfScope state. When an acceptAbort message arrives the Transport Agent transitions to the Awarding state because the task is currently not assigned anymore to an AGV.

Table 8 describes the relations between the interface definitions from the Transport Agent Architecture (Table 1) and the state machine for the scope handling solution (Figure 15).

The solution for the synchronization with scope handling problem is joined with our DynCNET protocol for a Transport Agent with abort synchronization. The relations for this composition are defined in Figure 16. The final composition is shown in Figure 17. Table 9 gives the InterfaceMatches for a Transport Agent with abort and scope handling synchronization (Figure 17). These InterfaceMatches were generated from the original InterfaceMatches in Table 5 and Table 8 and the state machine relations in Figure 16.

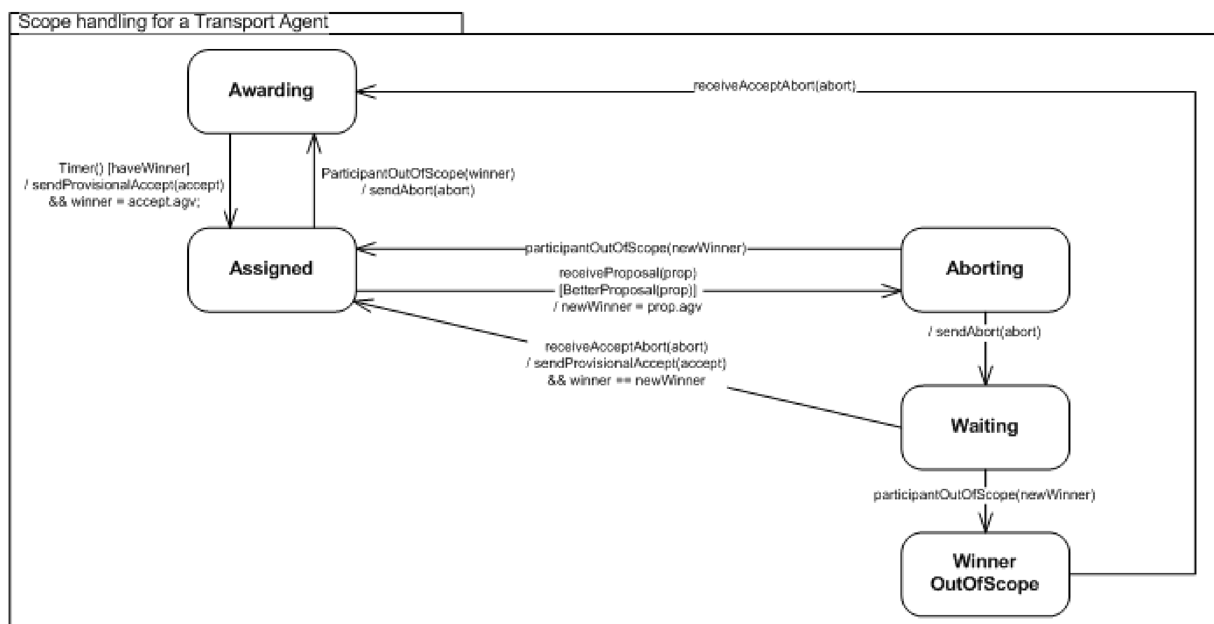
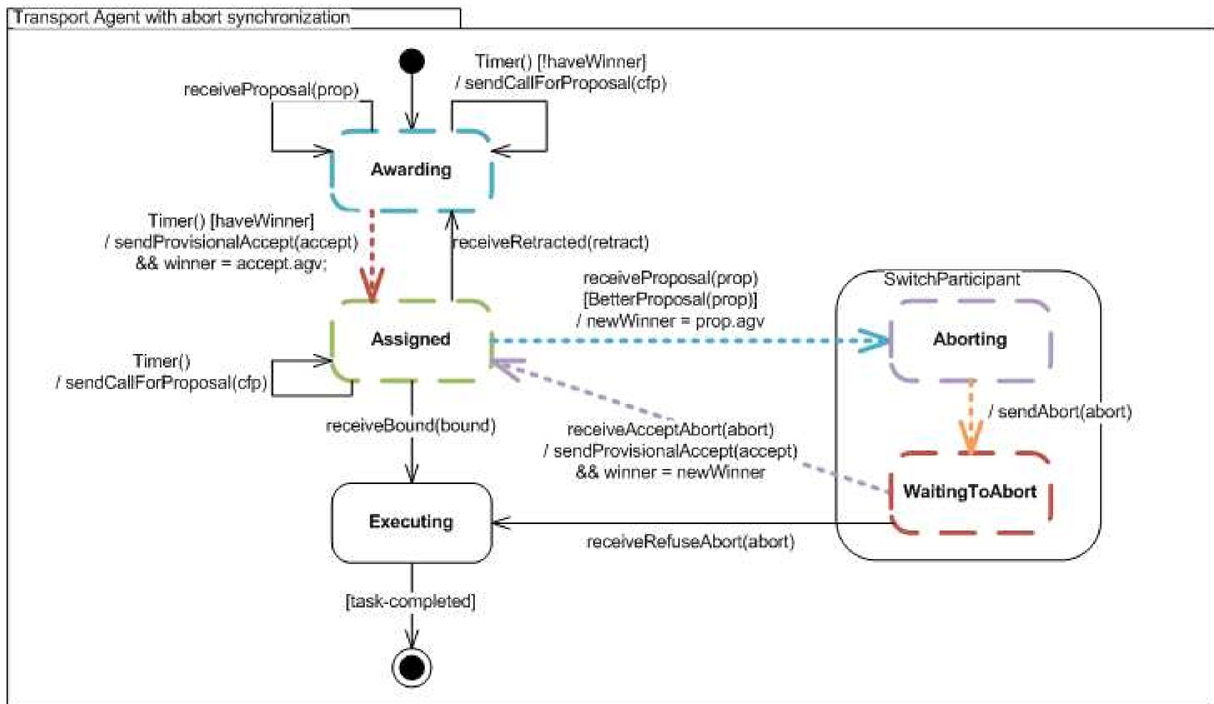
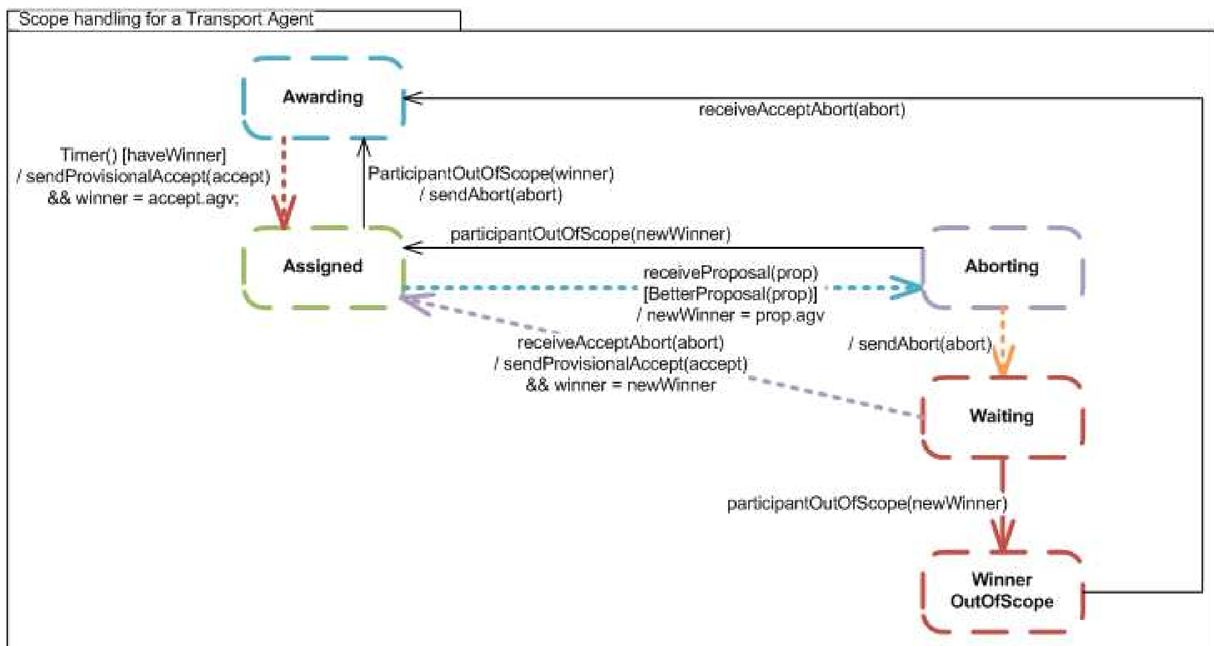


FIGURE 15: SCOPE HANDLING FOR A TRANSPORT AGENT (USING FIGURE 3 KEY)

- ActionInterfaceMatch (I)
  - Action: WaitingToAssigned.sendProvisionalAccept(accept)
  - Method: out: void sendProvisionalAccept(Accept accept)
- ActionInterfaceMatch (II)
  - Action: AssignedToAwarding.sendAbort(abort)
  - Method: out: void sendAbort(Abort abort)
- EventInterfaceMatch (III)
  - Event: AssignedToAwarding.participantOutIfScope(winner)
  - Method: in: void ParticipantOutOfScope(Id id)
- EventInterfaceMatch (IV)
  - Event: AbortingToAssigned.participantOutOfScope(newWinner)
  - Method: in: void ParticipantOutOfScope(Id id)
- EventInterfaceMatch (V)
  - Event: WaitingToWinnerOutOfScope.participantOutOfScope(newWinner)
  - Method: in: void ParticipantOutOfScope(Id id)
- EventInterfaceMatch (VI)
  - Event: AssignedToAborting.receiveProposal(prop)
  - Method: in: void receiveProposal(Prop prop)
- EventInterfaceMatch (VII)
  - Event: WaitingToAssigned.receiveAcceptAbort(abort)
  - Method: in: void receiveAcceptAbort(Abort abort)
- EventInterfaceMatch (VIII)
  - Event: ParctipantOutOfScopeToAwarding.receiveAcceptAbort(abort)
  - Method: in: void receiveAcceptAbort(Abort abort)
- ActionInterfaceMatch (IX)
  - Action: AbortingToWaiting.sendAbort(abort)
  - Method: out: void sendAbort(Abort abort)
- ActionInterfaceMatch(X)
  - Action: AwardingToAssigned.sendProvisional(accept)
  - Method: out: void sendProvisionalAccept(Accept accept)

TABLE 8: INTERFACEMATCHES FOR SCOPE HANDLING FOR A TRANSPORT AGENT





Key:

(blue dashed)	state unification	(blue dotted)	Transition unification
(green dashed)	state unification	(purple dotted)	Transition unification
(purple dashed)	state unification	(red dotted)	Transition unification
(red dashed)	state subelement	(orange dotted)	Transition unification

FIGURE 16: RELATIONS BETWEEN TRANSPORT AGENT WITH ABORT SYNCHRONIZATION AND SCOPE HANDLING FOR A TRANSPORT AGENT

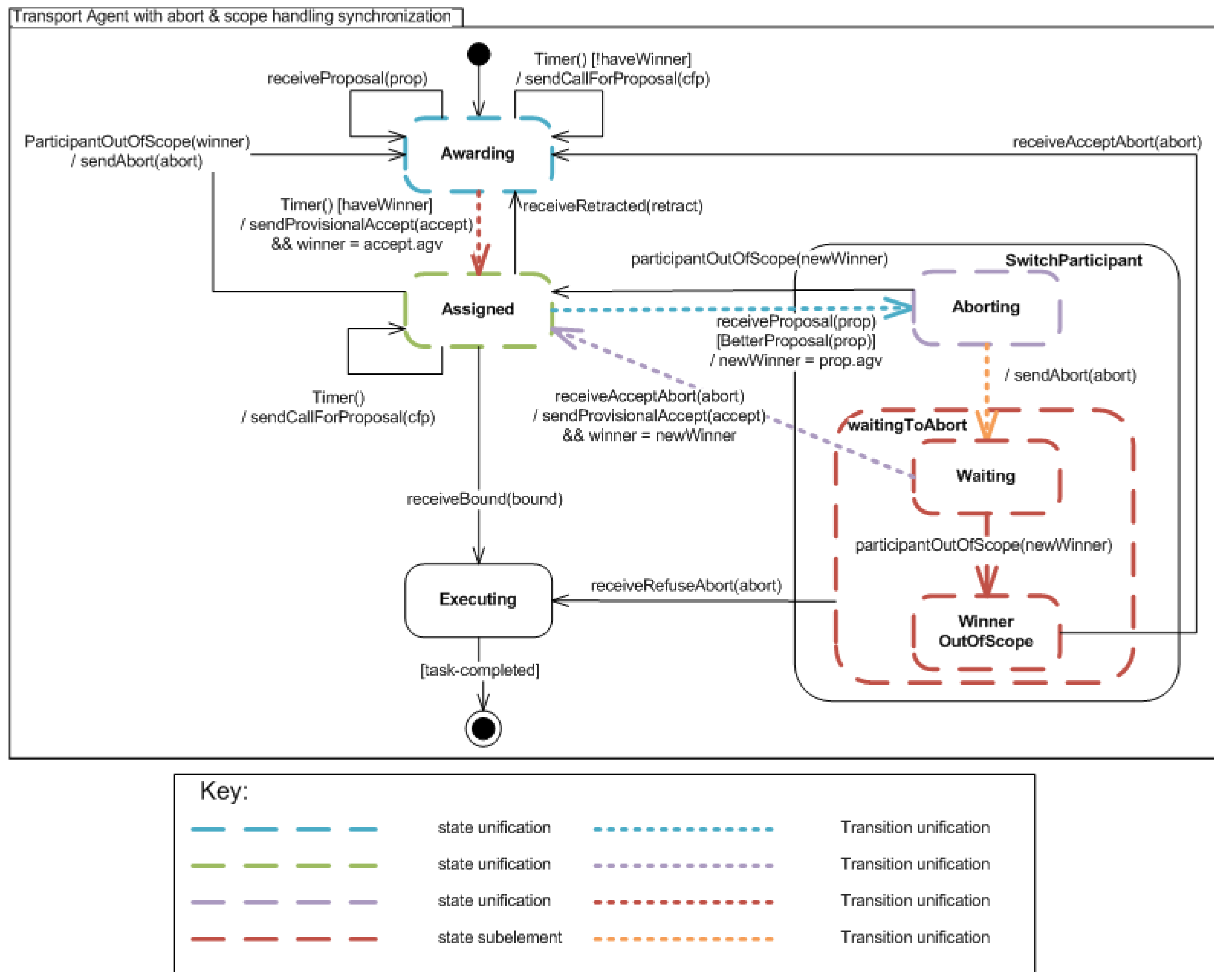


FIGURE 17: TRANSPORT AGENT WITH ABORT AND SCOPE HANDLING SYNCHRONIZATION

- ActionInterfaceMatch (1)
  - Action: AwardingToAwarding.sendCallForProposal(cfp)
  - Method: out: void sendCallForProposal(CFP cfp)
- ActionInterfaceMatch (2 + I)
  - Action: AwardingToAssigned.sendProvisionalAccept(accept)
  - Method: out: void sendProvisionalAccept(Accept accept)
- ActionInterfaceMatch (3)
  - Action: AssignedToAssigned.sendCallForProposal(cfp)
  - Method: out: void sendCallForProposal(CFP cfp)
- ActionInterfaceMatch (4+a+IX)
  - Action: AbortingToWaiting.sendAbort(abort)
  - Method: out: void sendAbort(Abort abort)
- EventInterfaceMatch (b + VII)
  - Event: WaitingToAssigned.receiveAcceptAbort(abort)
  - Method: in: void receiveAcceptAbort(Abort abort)
- EventInterfaceMatch (VIII)
  - Event: WinnerOutOfScopeToAwarding.receiveAcceptAbort(abort)
  - Method: in: void receiveAcceptAbort(Abort abort)
- ActionInterfaceMatch (5+c)
  - Action: WaitingToAssigned.sendProvisionalAccept(accept)
  - Method: out: void sendProvisionalAccept(Accept accept)
- EventInterfaceMatch (d)
  - Event: WaitingToAbortToExecuting.receiveRefuseAbort(abort)

- Method: in: void receiveRefuseAbort(Abort abort)
- EventInterfaceMatch (6)
  - Event: AssignedToAwarding.receiveRetracted(retract)
  - Method: in: void receiveRetracted(Retract retract)
- EventInterfaceMatch (7+e)
  - Event: AssignedToExecuting.receiveBound(bound)
  - Method: in: void receiveBound(Bound bound)
- EventInterfaceMatch (8)
  - Event: AwardingToAwarding.receiveProposal(prop)
  - Method: in: void receiveProposal(Prop prop)
- EventInterfaceMatch (9+f+VI)
  - Event: AssignedToAborting.receiveProposal(prop)
  - Method: in: void receiveProposal(Prop prop)
- EventInterfaceMatch (III)
  - Event: AssignedToAwarding.participantOutOfScope(winner)
  - Method: in: void ParticipantOutOfScope(Id id)
- EventInterfaceMatch (IV)
  - Event: AbortingToAssigned.participantOutOfScope(newWinner)
  - Method: in: void ParticipantOutOfScope(Id id)
- EventInterfaceMatch (V)
  - Event: WaitingToWinnerOutOfScope.participantOutOfScope(newWinner)
  - Method: in: void ParticipantOutOfScope(Id id)
- ActionInterfaceMatch (II)
  - Action: AssignedToAwarding.sendAbort(abort)
  - Method: out: void sendAbort(Abort abort)

TABLE 9: INTERFACEMATCHES FOR A TRANSPORT AGENT WITH ABORT AND SCOPE HANDLING SYNCHRONIZATION

4. Conclusion

This paper described the DynCNET protocol, a dynamic task assignment protocol for Multi-Agent Systems. The DynCNET protocol was build step by step. In the first step, we presented the CNET protocol. In a second step, we extended the CNET protocol with dynamic task assignment. Due to dynamic task assignment AGV's can switch to more suitable tasks when such tasks enter the system and Transport Agents can switch to more suitable AGV's when such AGV's become available.

This dynamism also comes with disadvantages like the synchronization of abort messages. Due to the distributed environment it is possible that a task is assigned to a better suited AGV while the currently assigned AGV is already executing the task. In the third step, a solution was proposed for the synchronization of abort messages and joined with the DynCNET protocol.

In the final step, a solution for AGV's leaving the scope of their task was proposed and joined with the rest of the DynCNET protocol. The final result is the DynCNET protocol with support for synchronization of abort messages and scope handling shown in Figure 17 (Transport Agent) and Figure 13 (AGV Agent).

References

[1] Weyns D., Boucké N. and Holvoet T. A Field-Based Versus a protocol-based Approach for Adaptive Task Assignment. *Journal on Autonomous Agents and Multi-Agent Systems*,2008
[2] Demarsin Bart, DynCNET: A protocol for flexible transport assignment in AGV transportation systems. Master Thesis, KULeuven, 2006.
[3] Reid G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver*. IEEE TRANSACTIONS ON COMPUTERS, 1104-1113 VOL. C-29, NO. 12, DECEMBER 1980*.
[4] Zafeer Alibhai. What is Contract Net Interaction Protocol?. IRMS Laboratory, 2003.