J Supercomput (2010) 51: 393–417
DOI 10.1007/s11227-009-0372-8

A framework for facilitating cooperation in multi-agent systems

Toktam Ebadi · Maryam Purvis · Martin Purvis

Published online: 23 December 2009

Abstract This work introduces a multi-agent framework that facilitates cooperation in multi-agent robotic systems. It uses a layered approach based on Coloured Petri Nets for modelling complex, concurrent conversations among agents. In this approach each agent employs a Coloured Petri Net model that allows agents to follow a plan specifying their interactions. It also allows programmers to plan for the concurrent feature of the conversation and make sure that all possible states of the problem space are considered. The framework assists the agents to identify and adapt different strategies for teammates and task selection dynamically. The agents can change their strategies in the course of dynamic environments to improve their performance. We have examined the performance of the agents in this framework by developing some task selection and teammate selection strategies for agents in a disaster scenario.

Keywords Cooperation · Coordination · Multi-agent systems · Coloured Petri Net · Robot

1 Introduction

Agent technology is well-suited for coordinated problem solving in virtual organizations. It can facilitate semi-automated negotiations between distributed elements and can support the more efficient management of distributed systems.

T. Ebadi (✉) · M. Purvis · M. Purvis
Department of Information Science, University of Otago, Dunedin, New Zealand
e-mail: tebadi@infoscience.otago.ac.nz

M. Purvis
e-mail: tehrany@infoscience.otago.ac.nz

M. Purvis
e-mail: mpurvis@infoscience.otago.ac.nz

Open distributed multi-agent systems (MAS) are composed of multiple independent agents that perform dependent tasks. Multi-agent robotic systems are proactive distributed parallel systems in which each agent is an autonomous robot and has limited resources. Typically, agents deployed in open environments may have different expertise and act as self-interested entities to achieve their corresponding goals. In such systems, it is not feasible to predict in advance what resources the agents may require in order to achieve their individual goals. Moreover, it is not possible to specify *a priori* the contexts in which an agent might need to interact with another for its service requirements. In addition, agents may confront different types of domains and must be able to interact with other agents to build a team.

Agent communication languages (ACLs) have been studied in depth to facilitate complex multi-agent systems, and such standards define communicative acts and interaction protocols ranging from a simple query to complex negotiations by auctions or bidding on contracts. For instance, the FIPA *Contract Net Interaction Protocol* [1] specifies the sequences of the messages that the interacting agents use by using the *Contract Net Protocol* for their negotiations. Several formalisms have been proposed to describe such standards. In particular, AUML (the Agent Unified Modeling language) has been used to standardize FIPA interaction protocols [1, 2]. AUML is suitable for human understanding and visualization. AUML is visually suitable for the representation of very simple interaction protocols. However for more general interactions, it is preferable to use a formalism that is more scalable and is also amenable to automated analysis, verification, and monitoring. For this reason, there has been an increasing interest in the employment of Coloured Petri Nets (CPNs) for the representation of agent interaction protocols. Petri Nets have been used for analysing the various aspects of the multi-agent systems [3–5], including validation, testing [6], debugging and monitoring [7] and plan description [8]. CPNs extend the power and flexibility of the Petri Net notation and also offer the strengths of a high-level programming language. The programming language element of CPNs provides the required data types and the manipulations of data values. As a result CPNs are well-suited for simulating, analysing and modelling distributed and concurrent systems [9]. They can express a wide range of interactions in a graphical representation with a well defined semantics. The formal aspects of Petri Nets allow precise modelling and analysis of system behaviour, while the graphical representation of Petri Nets facilitates intuitive understanding of the proposed solution. In addition, the modular and hierarchical aspects of the Petri Net models can help in designing solutions for complex systems [10].

In the application domain, we observe that climate change will lead to the occurrence of more disasters in the near future and thus rapid and effective response to disaster situation is very important. Therefore there is a need for modelling frameworks that simulate real environments in which agents act autonomously in real time. The current work presents a virtual organization of situated self-interested autonomous robots that need to coordinate their activities for performing tasks in dynamic environments. It introduces an implemented multi-threaded multi-agent framework that allows agents to cooperate in dynamic environments where there is no central mechanism to control the system. The framework allows agents to interact with each other and facilitates cooperation when required. In this system, various agents coordinate their activities using standardized interaction protocols, or “conversations”.

The agents in the system employ the standard FIPA agent communication language and interaction protocols.

CPNs are used as a modelling language to model the concurrent conversation activities among agents. The framework allows the agents to identify various task and teammate selection strategies and employ them. In addition, the framework accommodates agent learning. Under various conditions the agents may change their strategies in a dynamic environment in which the environmental constraints (task density or task time constraint) change constantly.

In this paper a disaster application scenario is examined for illustration in which a group of people are trapped in an unsafe area, and a group of robots with different capabilities may be required to save the victims. The robotic agents have different capabilities (and at various quality levels) required for different tasks. The capabilities are designed in such a way that each agent may be expert only in connection with a single capability, so cooperation of a team of agents is required in order to complete a task. Tasks are heterogeneous and have various requirements which can be satisfied by various capabilities of agents. In addition to capabilities, some of the agents ("skilled agents") are assumed to be equipped with some especial devices which can locate the tasks and recognise the task requirements. Due to the presumed high cost of these devices, there is a limited number of such agents equipped with these parts. All the other agents ("helper agents"), without any task-discovery devices, can be recruited by skilled agents to perform the tasks. Skilled agents explore the environment in order to find tasks. When a skilled agent finds a task, it creates a CPN model of the task-cooperation interaction and starts executing its model. The skilled agent sends requests to its neighbouring agents asking for help. When a potential helper agent receives a message from an initiator, it creates its own CPN model for that interaction and communicates, and thereby coordinates its activities, by sending messages to the initiator or other agents if it is required. More details on the CPN models are provided in Sect. 5.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the agent platform and conversation handling module used in this work. Section 4 describes how the agents and the environment are modelled. Section 5 details the CPN models for representing the conversation protocol. Section 6 describes the different strategies that agents may employ, and a learning and adapting mechanism is discussed in Sect. 7. Experimental results are discussed in Sect. 8. Finally, Sect. 9 concludes and outlines future work.

2 Related work

Cost et al. [11, 12] used CPNs for representing Knowledge Query and Manipulation Language (KQML) and FIPA interaction protocols [13]. They studied the use of CPNs in modelling agent communicative interactions. In their work, transitions represent events associated with messages. They tested their model for the simplified KQML Register conversation protocol. Billington et al. [14] showed how CPNs can be effectively employed to model and analyse the *Contract Net protocol* [1]. They presented a model of the protocol implemented in CPN tools and analysed it.

Gutnik et al. [15] proposed a CPN-based representation for overhearing, where an overhearing agent passively tracked many concurrent conversations involving multiple participants based on their exchanged messages. In their work places represent valid joint conversation states and messages, and token colour was used to distinguish concurrent conversations. In their approach the overhearing agent can monitor the messages. In addition to using tokens in state places, this representation used message places, where tokens are placed when a corresponding message is overheard. They showed semantically how their representation covers essential features required to model multi-agent conversations as defined by FIPA conversation standards [13] in an overhearing context. Their representation is only suitable for overhearing scenarios since each place in their CPNs represents a joint conversation. Dang et al. [3] used CPNs to represent concurrent negotiations among agents. They introduced a two-phase commit protocol that supports multi-issue negotiation. In their model each agent can interact with many agents. All of the above studies focused on Petri Net representations of simple interaction protocols and the suitability of CPNs as a model underlying a language for agent conversations. They tested the suitability of their proposed models for simple interactions and did not consider complex interaction protocols. Moreover, their models were primarily conceptual, and they did not present any results to demonstrate an executable model of an implemented framework.

Autonomy is an important feature of any agent system. Generally there are two aspects in the agents' autonomy: the internal and external aspects. The internal aspect refers to self-governance of the agents, and the external aspect implies that agents' behaviour cannot be imposed by any other agent, and thus any interactions between agents must recognise the participants' autonomy [16]. Although using autonomous agents can model more realistic and dynamic situations, many studies in multi-agent systems have not considered agents as autonomous entities [17, 18]. For example, Holvoet [19] proposed Petri Net agents, but their agents are not autonomous: the interaction between their agents (via transition synchronisation) is not filtered by an interface but directly concern transition modelling of the agents' behaviour.

Chainbi et al. [20] implemented agents' behaviours in the form of Cooperative Nets [21], which model the behaviour of objects. In their work the behaviour of an object is static and is defined by one net and thus cannot be adapted or altered at run time. Kohler et al. [22] and Duvigneau et al. [23] used reference nets, a special kind of CPN, to model the structure and behaviour of agents in terms of executable CPN protocols. Their reference nets were executed using the Renew simulation engine [24]. In [22] agents send their messages via synchronous channels. The synchronous channels do not provide support for situations where agents are hosted on different platforms. Duvigneau et al. [23] implemented a FIPA-compliant agent platform for multi-agent systems called CAPA which provides support for agents hosted on different platform to communicate. The work done by Kohler et al. [22] and Duvigneau et al. [23] is closer to our approach, since their models allow agents to run several processes concurrently. However, their work lacks a FIPA-compliant transport protocol, and there is no indication of adaptation to dynamic situations.

In contrast with the work described above, we have simulated a physical system where robotic agents are deployed in a physical environment. Our framework employs FIPA-compliant agents, and the standard FIPA agent communication language

and interaction protocols. It can facilitate semi-automated negotiations between distributed elements and supports more efficient management of distributed systems. The agents in the system coordinate their activities by sending messages. Since each robotic agent runs on a separate robot (or platform), all the messages are asynchronous and are sent via the HTTP transport protocol from one agent to another. The implemented framework allows the agents to deploy complex interaction protocols in situations when agents are required to coordinate their activities with other agents in the system. This feature enables the agents to keep track of their conversations with various agents in complex situations. For example agent A may be involved in a conversation with agent B and decide to help B. At the same time agent C may ask agent A to help it with another task. In this situation, agent A may send a message to agent C saying that it will be available, for example in 10 minutes. In the same way agent A may hold several other conversations over various topics with various agents. Using complex interaction protocols allows agents to follow their separate conversations with other agents efficiently without the need of asking for the past conversation details every time a message is received. Here each agent has various threads for managing its conversations, processing its message queues, etc., in a concurrent fashion.

In our framework agents must operate in the context of all the concurrency issues in real multi-agent environments, where message delivery is not guaranteed. For example, our agents have a timeout mechanism. Assume a situation in which an agent sends a request for help to a number of other agents for an urgent task that needs to be performed quickly. The requesting agent may not want to wait until it receives all the responses back from all of the requested agents, but instead it may wait for a short time and then select from the available agents. Without a timeout mechanism and a multi-threaded agent platform, one cannot measure the performance of the system under such realistic constraints. We are not aware of any other work that has considered such timeout in connection with modelling FIPA-compliant agents. Another feature of our framework is its adaptability. It allows agents to change their behaviours dynamically in order to adapt to different environmental conditions. Agents may decide to deploy an updated version of their previous models or change their roles and deploy a completely different CPN model that matches to new situations. The framework also allows adaptability by introducing various teammate selection strategies, which can be selected according to the circumstances of various situations. These strategies can be dynamically selected by the agents in order to maximise their individual utility. Note that the strategies are put into a separate module which could easily be replaced with a new module that contains user-defined strategies. Moreover, in our framework each agent has its own individual model for each conversation and does not know about the conversation model of the other agents.

3 Agent framework

3.1 OPAL agent and conversation manager

This work employs the OPAL agent platform [25] to support multi-robot cooperation. OPAL is a FIPA-compliant [13] agent platform, which employs micro-agents in its

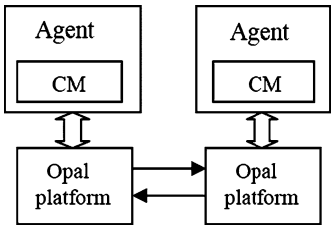

Fig. 1 The interaction between OPAL agents and OPAL platform

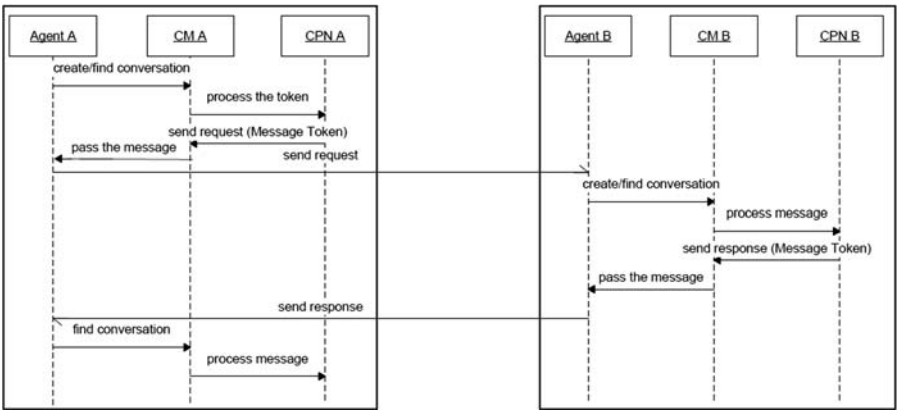

Fig. 2 Agents' components interactions

internals. Micro-agents are non-FIPA Java objects which have agent-like properties and may run in their own thread. A typical OPAL agent could contain numerous micro agents to perform tasks such as dispatch messages, manage conversations, and execute planning.

OPAL has a module called the Conversation Manager (CM), and the CM uses CPNs in handling agent conversations. Figure 1 shows the high-level view of the OPAL system. In OPAL every agent has its own CM to manage its conversations. Each CM is capable of handling multiple conversations for a single agent. Each agent participating in a conversation has a role in that conversation. The role information allows the CM to find an appropriate Petri Net model for handling that conversation.

For creating a new conversation, the initiating agent must know who is participating in the conversation and the role of each participating agent. Each message contains a conversation id that identifies the CPN model of that conversation. When a message is sent from an initiator to a helper, if there is already such a conversation (a conversation with that id) then it finds the CPN model of that conversation and handles the message, but if not, it creates a new conversation with that conversation id and handles the message.

Figure 2 shows the interaction between different agents' components for a simple request and response. All the inter-agent interactions are asynchronous messages, while all the intra-agent interactions are method calls on different modules within the agent. It shows that after finding a task, the CM creates a local instance of a "conversation". It also creates a CPN model (based on the interaction protocol and

the agent's role) and puts the token (task and possible teammates information) into its CPN model. Then the agent executes its CPN which leads to a message to be sent to available neighbouring agents asking for help. When a helper agent receives a message, through its CM, similarly it executes its CPN model based on the new token that was received (the message). This process is the basis of agent communication in this distributed framework where agents reside on different hosts. After the first interaction between two agents (for a particular task), if any of the agents receives a message from the other agent, its CM will find the corresponding "conversation" based on the message conversation-id and continue its conversation.

3.2 Coloured Petri Net

Place transition Petri Nets (PNs) are a well known formalism for modelling concurrent systems. A PN is a directed, connected, bipartite graph in which each node is either a place or transition. Places contain tokens. If there is at least one token in every place connected to a transition, then that transition is enabled. Any enabled transition may fire. If a transition fires then one or more tokens are removed from each input place and put into output places. The total number of tokens generated from the transition to the output places may not necessarily match the number of tokens consumed by the transition. This is due to the fact that in PNs, tokens represent the state of the model and not objects, and thus do not need to be conserved [9, 26].

CPNs [9] differ from PNs significantly, because their tokens are not simply blank markers, but have data states associated with them. A token's *colour* is a schema or type specification. Places are sets of tuples called *multi-sets.* Arcs specify the schema that they carry, and can also specify basic Boolean conditions. Specifically, arcs exiting and entering a place may have an associated function that determines which *multi-sets* are to be removed or deposited. Simple Boolean expressions, called *guards*, are associated with transitions and enforce some constraints on tuple elements. A CPN can be defined by a 7-tuple:

$$(\Sigma, P, T, A, C, G, E)$$

where

- Σ: is a finite set of non-empty type specifications also called colour sets. A token is a value belonging to a type.
- P: is a finite set of places which are sets of tuples, called multi-set.
- T: is a finite set of Transitions.
- A: is a finite set of Arcs, which are directed connectors between places and transitions.
- C: is a colour function.
- G: is a guard function which is a boolean expression. A guard is associated with a transition and enforces some constraints on tuple elements before the transition can be enabled.
- E: is an arc expression function which specifies the schema it carries. It can also specify the basic boolean conditions. Arcs directed to or away from a place may have an associated function that determines what multi-set elements are to be removed or deposited.

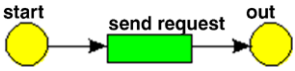

Fig. 3 The CPN model of agent *A*

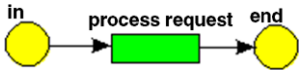

Fig. 4 The CPN model of agent *B*

The distribution of tokens in the places of a CPN/PN is called marking. The marking of a CPN determines the state of the system being modeled. An enabled transition may fire by removing tokens from input places specified by the arc expressions of all the incoming arcs and depositing tokens in output places specified by the arc expressions of the outgoing arcs.

3.3 JFern

JFern [27] is a lightweight Coloured Petri Net framework with a simulator, written in Java. We used JFern as the Petri Net simulator to design the CPN models. The CM requires the JFern engine to run the Petri Net model of each agent. In order for the CM to work with CPN model, some special CPN places must be created in the PN model.

- *start*, *in* and *out*: for the initiator of a conversation
- *in* and *out*: for the helper agents in a conversation.

The *start* place: is only required for the initiator role and is used for initiating the conversation (putting the token which includes the necessary information for the conversation, such as the conversation id and the name of the interaction protocol).

in place: all incoming messages to the agent are handled by the CM and relevant information associated with a message will be encapsulated in a token and inserted directly into this place.

out place: Every token that reaches the output place will have information that it contains sent as an agent message to the receiving agent. The receiving agent will take the received message and insert the relevant information into its *in place* of its appropriate CPN.

Figures 3 and 4 show how a message token is sent from one agent to another. In this example agent *A* sends a "hello" message to agent *B*. The *send request* transition creates a message token and puts the token into *out* place. The CM of agent *A* sends the message to agent *B*. The CM of agent *B* receives the message and puts the token into *in* place of agent *B*. Then the *process message* transition prints the contents of the message, which is "hello".

4 Environment and agent model

We simulated a physical environment divided into several spatial regions. A RFID tag is assumed to be deployed in each region and holds some information with respect to the geographical coordinates of the region. The tasks information are encoded in

RFID tags and are distributed in the environment. There are two types of robots. Skilled robots which have more computational capabilities and are able to decode the task information, and helper robots which have less computation capability and thus cannot retrieve the task details. All the agents are equipped with low-range RFID readers that allow agents to position themselves in the environment by reading the coordinate information from environment tags.

Here agents deploy the FIPA [13] protocols for communication. Moreover, the environment has a task manager agent which produces tasks at certain intervals (which can vary) and removes the tasks when their time has expired. It also rewards agents participating in performing a task.

4.1 Agent's capabilities

Agents are assumed to have different capabilities that are useful in satisfying different task requirements. The capabilities of each agent are fixed and do not change over time. In this work each agent has two capabilities but is expert at one of them. The capability values representing the quality level of the expertise may range from 0 to 1. For example an agent may have the following set as its capabilities.

$$(\alpha = 0.6,\ \beta = 0.0)$$

Such an agent is able to satisfy the α-requirement of a task with maximum level of 0.6.

4.2 Tasks

Tasks are distributed in the environment and have different requirements that should be satisfied by the different capabilities of the agents. A task is represented as a tuple:

$$(r, t, p, w)$$

where r: is the set of requirements. Each requirement is a value between 0 and 1, t: is the time constraint of the task, p: is the task priority. The priority is a value between 0 and 9 and represents the task urgency, w: is the basic reward that a team receives by performing the task. The reward is distributed equally to the agents that participate in completing a task.

For instance, a task may have the following set as its requirements:

$$(\alpha = 0.2,\ \beta = 0.6)$$

Such task requires 0.2 of α-capability of agents for its α dimension and 0.6 of β-capability of agents for its β dimension. The agents participating in performing a task receive the reward if they can perform the task before time expires.

4.3 Agents reward

The agents participating in a task receive a reward that is proportional to the amount of the task that they had completed. All agents participating in a task receive the same

reward calculated using the following equation:

$$R = \left[\sum_{i=1}^{n} \frac{\sum_{k=1}^{m} (A_k)_i}{r_i} \right] \times \frac{w \times p}{m \times n}$$

where $(A_k)_i$: is the capability of the agent A_k for ith requirement, r_i: is the ith requirement of the task, w: is the basic task reward, p: is the task priority, n: is the number of task requirements, m: is the number of agents participating in performing the task.

In the above equation if

$$\frac{\sum_{k=1}^{m} (A_k)_i}{r_i} > 1$$

then

$$\frac{\sum_{k=1}^{m} (A_k)_i}{r_i} \leftarrow 1$$

In some situations agents can be rewarded even if they only perform a portion of the assigned task. The reward formula assures that the agents' rewards are proportional to the completed part of the task. For example consider the following situation. In this case agent A and agent B have teamed up to perform a task. The capabilities of agents A and B are $(\alpha = 0.3, \beta = 0.0)$, $(\alpha = 0.0, \beta = 0.2)$ respectively, the task requirements are $(\alpha = 0.6, \beta = 0.4)$, the basic reward is 8 and the task priority is 1. In such situation the reward of the agents will be:

$$R = \left(\frac{0.3}{0.6} + \frac{0.2}{0.4} \right) \times \frac{8 \times 1}{2 \times 2} = 2$$

In this example the agents have performed half of the task requirements, and they receive half of the reward. The reward is divided by the number of requirements (so that tasks that have more requirements do not generate higher rewards). In addition, the reward must be divided by the number of agents (so the teams that have higher number of participants do not generate higher reward in the system).

In the previous example if the number of teammates is not considered then each agent receives 4 as reward and the total reward of 8 is generated in the system. In order to explain the task reward partitioning further we can think of the following example. Assume that instead of two agents there are three agents, and the respective capabilities are $(\alpha = 0.1, \beta = 0.0)$, $(\alpha = 0.2, \beta = 0.0)$ and $(\alpha = 0.0, \beta = 0.2)$. In this case each agent will receive reward of 4. Since there are now three agents in the team, the total reward generated in the system will be 12. Although in both examples the teams performed half of the requirements, the second example produces a greater reward in the system. The inverse relation of reward with the number of teammates will not allow this happen.

4.4 Agents roles

In this work each agent plays a role in each conversation. A role is the identity of a set of acts executed by an agent. Here we consider scenarios in which cooperation among

agents for performing a task is required and the agent's role in each conversation depends on the type of the agent. There are two roles: Initiator and helper.

- Initiator: It is a skilled agent which is capable of detecting tasks. If an agent detects a task, then it may start a new conversation to find teammates.
- Helper: Agents which do not have the task detection device play the helper role. These agents can be recruited by agents playing the Initiator role.

5 Modelling agent roles

In this work two models are designed based on the roles that the agents could play in a conversation. Figures 5 and 6 show the CPN models for the initiator and helper role, respectively. In these models an environmental system time is used to accommodate the time delay associated with agents' responses in a distributed multi-agent environment. This is to make sure that agents have a timeout mechanism and do not wait indefinitely for responses from other agents when they are not available.

Figure 5 has three different phases. In the first phase the agent sends requests to its neighbours and asks whether they could participate in performing the task. In the second phase the agent tries to form a team based on positive responses that it receives from requested agents. In the third phase the agent sends a *move* message to its selected teammates and a *reject* message to other agents who had responded positively but have not been selected by the initiator.

Phase 1: When an agent finds a task, it creates a token and puts it into the *start* place. The token has the task information (requirements, time constraint, and reward)

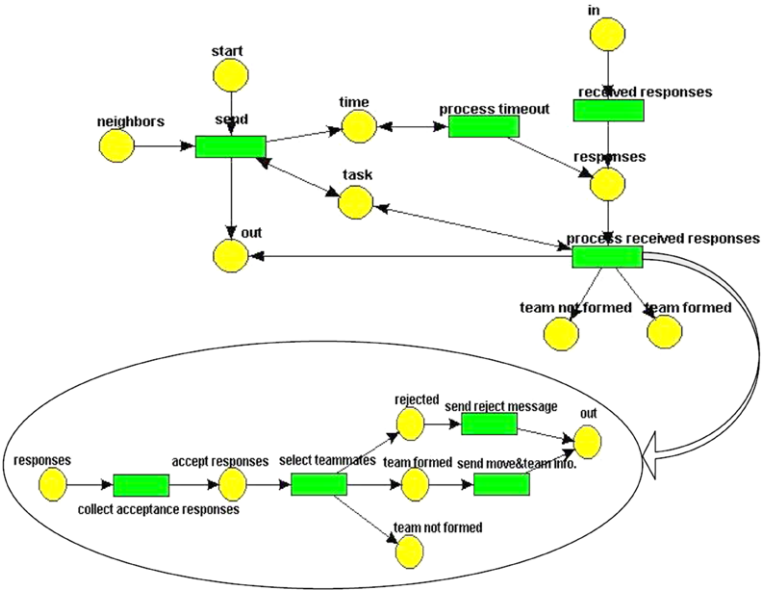

Fig. 5 The Petri Net model for the initiator role

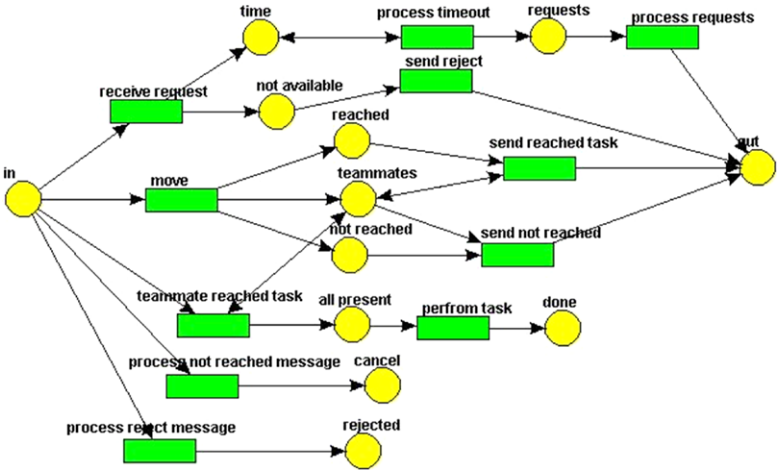

Fig. 6 The Petri Net model for the helper role

and the name of the interaction protocol for the conversation. The names of the helper agents in the neighbourhood are put into *neighbours* place. Then the agent sends help requests to its neighbours. After sending the requests, the sent time is put into the *time* place. The agent waits until the waiting time elapses.

Phase 2: After the waiting time elapses, the agent begins processing all the responses that it has received from other agents and selects its teammates. The guard on the arc that connects the *in* place to the *received responses* transition filters the received messages and only allows messages with the *accept* or *reject* performative [28] to be passed to the *process received responses* transition.

Figure 5 shows a hierarchical view of the initiator Petri Net for the *process received responses* transition. The transition *collect acceptance responses* collects all the positive responses and the transition *select teammates* processes the positive responses.

Phase 3: If the agent could find helper agents with the required capabilities, then a *move* message is sent to the selected teammate that directs the agents to move. This message also contains the details of the team. The agent also sends a reject message to all the agents who have responded positively but have not been selected as teammates, and informs them that they are not selected. If the agent cannot find agents with the required capabilities, then it drops its current task and starts searching for new tasks.

Figure 6 shows the CPN model of the helper role. A helper agent receives requests from various initiators in its neighbourhood (transition *receive request*). If the agent is involved in performing another task, then it sends a reject message to the requester agent (transition *send reject*). However, if the helper is available, then it may want to wait for a certain length of time to receive several requests and then select the best offer. The CPN model of the helper accommodates the waiting time by including the *process timeout* transition. After receiving a request, the helper agent puts the current time as a token into the *time* place. The *process timeout* transition compares the time that the first request was received with the current time.

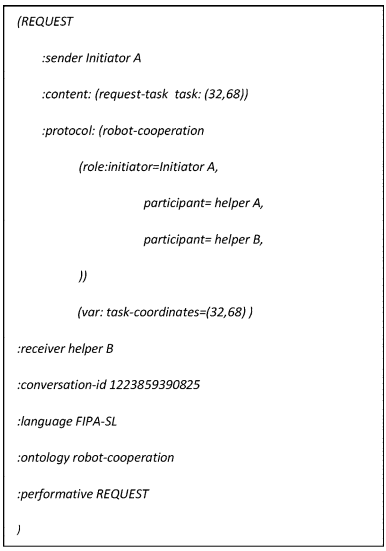

Fig. 7 An example of a request message

If the difference is more than the waiting time for helper agents, then it passes all the received requests to the *process request* transition otherwise, it puts the time token back to the *time* place. When waiting time has elapsed, the *process request* transition processes all the requests and selects the best offer. Then it sends a positive response to the selected initiator and changes its status to *unavailable*. A reject message is also sent to all other initiators that have not been selected.

A helper agent with a positive response to a request may be rejected by the initiator of the conversation if better-suited helpers are available. In this case the rejected agent changes its status to *available* and can participate in other tasks (transition *process reject message*). When a helper agent receives a *move* message from an initiator, it starts moving toward the task. The move message also contains the name of the teammates. After reaching the task location, it sends a message to its teammates informing them of its current location. When all the teammates are present at the task location, then the agents start performing the task. For performing a task, each agent has to spend some time at the task which is proportional to the part of the task that is to be completed by the agent.

If the agent does not reach the task location after a certain period of time, then it sends a message to its teammates informing them about the issue (transition *send not reached*). In this situation all the team members cancel their current contract (transition *process not reached message*).

Figure 7 shows a *request* message from Initiator A to helper B. It shows that in this conversation the initiator is talking to helper A and helper B for a task which is located at the coordinates (32, 68) of the grid environment. The conversation-id allows helper B to retrieve the CPN model of the conversation and continue its conversation with Initiator A or create a CPN model for this conversation if this is the first time that it interacts with Initiator A for the task.

6 Strategies

Various strategies are accommodated and designed for selecting teammates and tasks in the framework. The aim of these strategies is to show the capabilities of the framework in adapting to the dynamic environments in real multi-agent robotic systems. In such systems the skills of robotic agents and their distance from tasks have an impact on agents' performance. These strategies also allow measuring the performance of the system in scenarios when time is critical and agents must perform their tasks within a specified time. Since the framework is separated from the strategies, these strategies can be easily replaced with other strategies without any changes to the framework.

6.1 Teammate selection strategies

In order to examine and demonstrate the capabilities of the developed agent cooperation framework, we designed some specific teammate-selection strategies and performed simulation experiments with them. Some strategies require the agents to complete the tasks (*best possible*). However, some other strategies will allow partial performance of the tasks (*best available*, *nearest available* and *impatient* strategy). Agents which complete only part of the tasks receive a reward which is proportional to the completed part of the task but agents which complete the tasks will receive the full reward for the tasks.

6.1.1 Nearest available strategy

An agent that employs this strategy waits for certain length of time and then selects the agents that lie at the least distance to the task. These agents can perform some parts of a task. Agents with this strategy receive partial rewards proportional to the completed part of the task. The Algorithm 1 shows the Java pseudocode for an agent that employs this strategy.

The initiator agent ranks all the helper agents that have responded positively to its request, based on their distance from the selected task. The distance refers to the number of steps that the agent must take in order to reach to the task location. Please note that *nearest available* strategy may lead to selection of redundant helpers (i.e. the last selected helper may have a high capability so the first selected helpers were not really needed).

Algorithm 2 shows the pseudocode for calculating the agent distance from the task.

6.1.2 Impatient strategy

Agents that employ this strategy wait for a short time and select the first agents that respond to their requests as their teammates. In most cases these agents simply select all the agents that have responded positively.

Algorithm 1 *Nearest available* strategy

```
//retrieve the task requirements
taskαrequirement=task.get(αRequirement);
taskβrequirement=task.get(βRequirement);
availableHelpers =this.getPositiveResponses();
Iterator iterHelper= availableHelpers.Iterator();
//calculate the distance rating for all the available agents
while (iterHelper.hasNext()){
        helper = iterHelper.next();
        dist = calculateDistanceToTask(helper,task);
        if(dist = = 0)
            sortedMap.put(1,helper);
        else
            sortedMap.put(1/dist,helper);
}
//sorts the agents based on their distance ratings
sort(sortedMap);
 foundαCapabilities=0;
foundβCapabilities=0;
Iterator iterSort= sortedMap.values.Iterator();
//select teammates from the sorted collection
while(iterSort.hasNext() &&
    ( foundαCapabilities < taskαrequirement ||
        foundβCapabilities< taskβrequirement)){
selectedHelper = iterSort.next();
teammates.add(selectedHelper);
foundαCapabilities+=selectedHelper.get(αCapability);
foundβCapabilities+= selectedHelper.get(βCapability);
  }
  Return teammates;
```

6.1.3 Best available teammate strategy

Agents that employ best available teammate strategy initially wait for certain length of time. After the waiting time has elapsed, the agent selects helper agents based on their quality. The quality refers to the potential helper agent's capabilities with respect to the task requirement. For each agent A_j the quality is measured based on

Algorithm 2 Calculate distance to task

```
xHelperPosition = helper.get(xPosition);
yHelperPosition = helper.get(yPosition);
xTaskPosition = task.get(xPosition);
yTaskPosition = task.get(yPosition);
//calculate the distance of the agent to the task
distance=|xTaskPosition-xHelperPosition|+
            |yTaskPosition-yHelperPosition|
return distance;
```

the following equation:

$$\forall(r_i \in r) \neq 0 \quad \Rightarrow \quad Q_{A_j} = \sum_{i=1}^{n} \frac{(A_j)_i}{r_i}$$

where r: is set of task requirements, r_i: is the ith requirement of the task, n: is the number of task requirements, $(A_j)_i$: is the capability of agent A_j that corresponds to ith requirement of the task.

The Q_{A_j} indicates the quality of an agent A_j for a particular task. For instance if an agent has the capabilities $(\alpha = 0.2, \beta = 0.0)$ and the task requirements are $(\alpha = 0.3, \beta = 0.1)$, then the quality of the agent is $\frac{0.2}{0.3} + \frac{0.0}{0.1} = 0.66$. If an agent has a higher capability for a particular task requirement, then the ratio (capability/requirement) is considered to be 1, so the agents that have higher quality than the task requirements are not rated higher than 1.

These agents may select teammates that only have some of the expertise required for the task, in which case they receive a partial reward for the part of the task that is completed. Agents with this strategy usually receive a high reward for each task due to selecting high quality teammates. The Algorithm 3 shows the Java pseudocode for selecting the *best available* teammate strategy.

6.1.4 Best possible teammate strategy

An agent that employs this strategy waits for a certain length of time and only selects other agents as its teammates if they can complete the task.

$$\forall r_i \in r \quad \Longrightarrow \quad \sum_{k=1}^{n} (A_k)_i \geq r_i$$

where r_i: is each requirement of the task, r: is the set of task requirements, n: is the number of agents with a positive response to the initiator request, $(A_k)_i$: the capability of agent A_k that corresponds to ith requirement of the task.

Algorithm 3 *Best available* strategy

```
foundαCapabilities=0;
foundβCapabilities=0;
availableHelpers = positiveResponses();
//retrieve task requirements
taskαrequirementValue = task.get(αRequirement);
taskβrequirementValue = task.get(βRequirement);
Iterator iterHelper= availableHelpers.Iterator();
//select teammates based on their quality
while(iterHelper.hasNext() &&
    (foundαCapabilities < taskαrequirement ||
        foundβCapabilities<taskβrequirement)){
            /*calculate the quality of the agents for the task and
            sort them based on their quality*/
            sortedListOfHelper=sortBasedOnQuality(
            availableHelpers);
            selectedHelper = sortedListOfHelpers.get(0);
            teammates.add(selectedHelper);
            helperαCapability= selectedHelper.get(αCapability);
            helperβCapability= selectedHelper.get(βCapability);
            foundαCapabilities += helperαCapability ;
            foundβCapabilities += helperβCapability ;
            // update the task requirements
            taskαrequirementValue=      taskαrequirementValue-
            helperαCapability;
            taskβrequirementValue=      taskβrequirementValue-
            helperβCapability;
            availableHelpers.remove(selectedHelper);
}
Return teammates;
```

6.2 Task selection strategies

The following describes the strategies that agents could adapt for selecting their tasks. These strategies correspond to the disaster scenario where new tasks with various urgency levels may appear at the different locations unpredictably. The aim is to study the effect of these strategies on the performance of the agents with various teammate selection strategies and thereby demonstrate how different selection strategies are

most appropriate for different circumstances. Thus it is advantageous for an agent cooperation framework to offer flexibility in connection with task selection strategies.

6.2.1 High priority task selection strategy

Agents with this strategy select a task with the highest priority within their vicinity (task reading range). If there are several tasks with the highest priority value then the closest one is selected.

6.2.2 Nearest task selection strategy

Agents with this strategy select a task that lies at the least distance away from them.

7 Learning and adaptation

Agents may be required to change their strategies under various circumstances in order to adapt to dynamic conditions in the environment. Therefore, there is a need for a mechanism that allow agents to find a strategy that improves the reward for the agents under varying circumstances. In our experiment a simple learning mechanism is adapted. Each agent keeps track of its performance and assesses its performance at certain intervals. Initially agents start exploring with various possible strategies. Each agent builds a list of different strategies and updates the average reward of each strategy after each interaction. After exploring for a certain duration, each agent makes a decision based on the gained reward for each strategy and selects the strategy that has given the agent the highest reward. After selecting the strategy, then the agent performs with its selected strategy for another round. If the current reward (gained during the current period) is less than the previous reward, then the agent starts exploring its strategies again; otherwise it keeps performing with its current strategy for another period. This simple learning mechanism allows agents to adapt to dynamic conditions in the environment.

8 Experiments

8.1 Experimental setup

Note that while the developed framework was empirically examined here by performing computer simulations of agent activity, the framework is ultimately intended for deployment on real, physical robots. Our multi-threaded simulation environment comes close to reproducing the concurrency conditions of real distributed multi-agent robotic systems.

The experimental agent framework was tested by employing OPAL agents on a simulation grid-type environment where agents are capable of running multiple conversations over various tasks concurrently. The simulation environment is a grid of 100 by 100 cells in which each cell refers to one square of the grid. There are 100

Table 1 Simulation parameters

Parameter	Value
Agents visibility range for other agents	10
Agents visibility range for tasks	5
Task production interval	120000
Waiting time for non-Impatient strategies	20000
Waiting time for Impatient strategy	12000
Waiting time for helpers	500
Time unit	5000
Task priority	$0 <$ random < 10
No. of Initiator agents	16
No. of helpers agents	84

robots with different capabilities. Out of the 100 robots only 16 of them are able to detect the tasks. Task requirements and agents capabilities are chosen randomly. The basic reward for each task is 7. The visibility range of all the agents is fixed to 10 cells, but the visibility range for tasks is set to 5. This allows only a few agents to be within range of a task so that they can see it at any point of time. This reflects the low visibility range for detecting the tasks. In robotic applications, the sensors that are used to read the information from the environment (identify the task) have a short range (i.e. cameras, thermometer). Although the sensory range for task detection is relatively short, it may be possible in the physical world of robots to employ greater-ranged WiFi or Bluetooth technology for inter-agent communications. We simulate these circumstances by making the communication range between agents greater than the task-identification range. Table 1 shows the simulation parameters.

We used the system time in our framework. Time unit refers to the time required to move from one cell of the grid environment to the next adjacent cell. For example if an agent needs to move to a location which is about 8 cells away from its current location, it takes 8 time units for the agent to reach to its destination. In addition there is some time associated with performing a task by an agent. For each unit of the task requirement (0.1), an agent must wait one time unit on the task location. For instance, if there is a helper agent that is assigned to perform 0.3 of the α-requirement of a task, then the agent must spent 3 time units on task location before it can leave the location.

In all the experiments agents and tasks are distributed randomly on the grid environment. All times in this system are in milliseconds and all the simulations were run for 960000 milliseconds (16 minutes). The results presented in the diagrams are averaged over ten runs.

The aim of the first two experiments is to show what strategy is suitable under various environmental conditions. These conditions can vary very much depend on the number of tasks in the area and also the task time. The task time refers to the time given to the agents to perform a task.

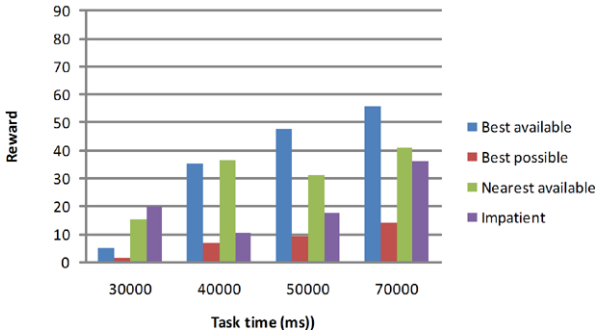

Fig. 8 The effect of agents' strategies on agents reward when task density is low

8.2 The effect of agent's teammate selection strategy on agent's reward when task density is low

The aim of this experiment is to show the effect of various teammate selection strategies on agents' rewards in situations where task density is not very high and under various task time constraints. In this experiment the task manager produces 20 tasks every 120000 milliseconds, and all tasks have the same priority (1). Four groups of agents were deployed in each simulation run, where each group has one strategy, so various strategies compete against each other. All skilled agents wait for 20000 milliseconds to receive their responses, except impatient agents, which wait 12000 milliseconds.

Figure 8 shows that when the time is tight (30000 ms) the *impatient* strategy outperforms other strategies, because agents with this strategy do not wait to receive their response back from most of the requested agents. Since time is tight, these agents have time to move toward their tasks and perform them. However, the other groups that wait for longer period may not be able to reach to the task positions and perform them before time expires. When the time is a bit less tight (40000 ms), the *nearest available* strategy outperforms the other strategies. These agents wait for a certain length of time and select the closest agents as their teammates to work on tasks. Since agents with this strategy select the closest agents, they have a higher chance of performing their tasks on time. The low reward of agents with *impatient* strategy is due to performing only small parts of those tasks (since they do not wait long enough to find agents with the required capabilities). The reward of the *best available* strategy is slightly less than the reward of agents with the *nearest available* strategy under somewhat more relaxed time constraints (4000 ms). These agents select teammates with high quality capabilities, which may be farther away from task position and therefore they may not have enough time to reach the task position on time. When the task time is further relaxed (50000 and 70000 ms), the *best available* strategy starts outperforming other strategies, which is due to their performing greater portions of the tasks by having high quality teammates.

The agents with the *best possible* strategy perform worse under various time constraints. This is the effect of the perfectionist attitudes of these agents. This approach is useful when there is more incentive in fully completing a job. For example, if there were a container full of poisonous and explosive chemicals near to a building which

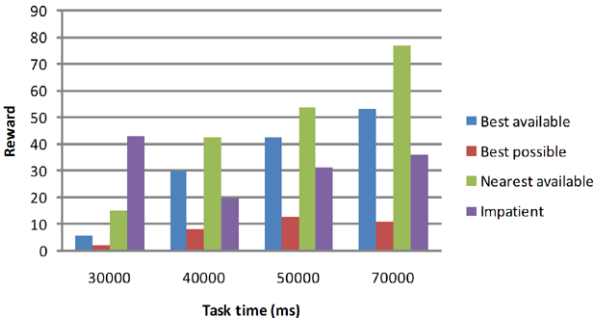

Fig. 9 The effect of agents' strategies on agents reward when task density is high

is on fire and there are people trapped inside the building, then partially removing the explosive material is insufficient.

8.3 The effect of agent's teammate selection strategy on agent's reward when task density is high

These experiments study the effect of agents' strategies on their rewards when the task density is high. The agent manager produces 60 tasks every 120000 ms, and the other parameters are the same as in the previous experiment.

Figure 9 shows that when time is tight, the result is similar to the previous experiment. However, when time is relaxed (50000 and 70000 ms), the *nearest available* strategy outperforms the other strategies. This is due to the fact that agents which select their nearest neighbours reach the task location more quickly. Since the task density is high, these agents will perform more tasks and therefore increase their rewards. The reason that the reward of the *impatient* strategy is lower than *nearest* strategy is that *impatient* agents form a team with a very small number of agents which in most cases do not have the required capabilities. Therefore, they are only capable of performing a small part of each task.

8.4 The effect of agent's task selection strategy on agent's reward

These experiments show the effect of various agents' task selection strategies on agents reward. Two separate runs of simulation were performed for each teammate selection strategy. The task manager produces 60 tasks every 120000 ms and task time is relaxed (70000 ms).

Figure 10 shows that *best available* strategy performs better when agents select *high priority tasks.* These agents perform large portions of high priority tasks and therefore receive higher rewards. The *nearest available* strategy performance improves by selecting closer tasks. Since the task density is high, agents with this strategy can perform more tasks by selecting closer teammates and improve their rewards. The *impatient* strategy performs well by selecting *high priority tasks* which gives the agents a chance to gain a high reward. The performance of the *best possible* strategy is not very different under either of the two task selection strategies.

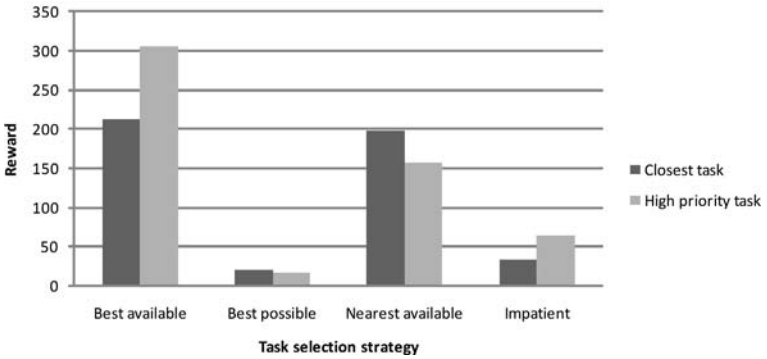

Fig. 10 Effect of agents' task selection strategies on agents reward

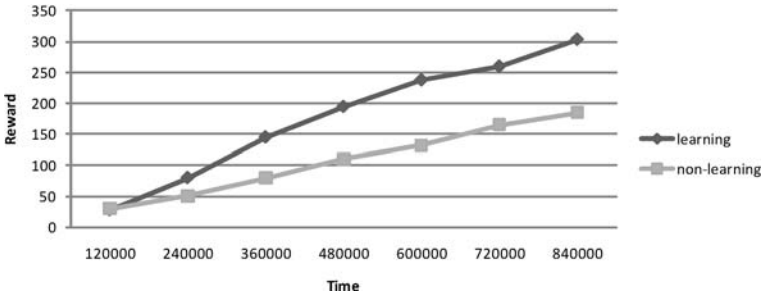

Fig. 11 Effect of agents' learning

8.5 The effect of agent's learning on agent's reward

For this experiment two simulation runs were executed. In one experiment four groups of agents were formed, where each group deploys a single strategy. The strategy of the agents is fixed for this experiment, and agents do not change their strategies during the simulation. In the second experiment all the agents are set to be learning agents and deploy the learning mechanism explained in Sect. 7. A learning agent may change its strategy if the following condition holds.

$$R_c - \alpha < R_p$$

where R_c: the reward that has been achieved during the current performance period, R_p: the reward that has been achieved during the previous performance period, α: 10.

In this experiment the task manager produces 60 tasks every 120000 ms, so task density is high and the task time constraints is very relaxed (70000 ms).

In this situation after the first exploration phase (at 120000 ms), all the learning agents select their strategies. Since the task density is high and task time constraint is relaxed, all the learning agents change their strategies to *nearest available,* which allows agents to perform a high number of tasks and thus improve their reward. Figure 11 compares the total reward of agents in the system when agents learn and do not learn.

9 Conclusion

This paper has introduced a general lightweight framework for enhancing cooperation among autonomous agents. The agent robots communicate by employing the standard FIPA protocols. The framework employed CPNs to model concurrent activities of the agents. The use of CPNs is useful for modelling and managing multiple conversations for individual agents. This facilitates the development and use of agent-based systems in real time situations in which the agents must coordinate with the agents in a concurrent fashion. The framework allowed agents to execute their CPN models by using JFern [27] engine. The simulation of running system was animated by using JFern and can be used for validating the models.

Although we only presented two roles for the agents in this paper but the framework allows the use of multiple roles for various tasks for each agent. In addition, the system allows agents to adapt to new conditions by altering their CPNs. However, due to space constraints this feature is not discussed in this paper. Moreover, the framework can accommodate varying tasks and partner selection strategies. It also accommodates dynamic situations in which the participating agents may change their strategies if they learn from their past experiences.

The performance effects of various teammate and task selection strategies has been empirically evaluated. Our experiments show that under various environmental constraints, different strategies may be preferred. When time constraints were tight in our experimental scenarios, the *impatient* strategy performed best, regardless of the task density. However, when time requirements were relaxed and the task density was low, the *best available* strategy that performed large portions of tasks performed better than other strategies. When time requirements were relaxed but the task density was high, then the *nearest available* strategy outperformed other strategies, due to performing a greater number of tasks. In addition, the effect of the task selection strategy on agents' reward was investigated (when task density was high and the task time constraint was relaxed). Under such constraints, the *best available* and *impatient* strategies performed well by selecting *high priority tasks*. However, the performance of the *nearest available* strategy improved by selecting the closest tasks. The learning experiment shows the adaptation capability of the agents. Overall, the developed agent cooperation framework supports the efficient design, development, and physical deployment of multi-agent systems that engage in cooperative behaviour to perform tasks in spatially-distributed environments.

In future investigations, we will examine situations in which robots alter their strategies in heterogeneous environments. For instance, in some area of the environment the task density may be high, while in the rest of the environment the task density may be low. A richer learning mechanism can also improve the adaptation. The examined strategies could help in different situations such as natural disaster where speed is crucial and agents may require different strategies. All of these experimental strategies could help in the development of agile and adaptive cooperative agent systems that can vary their activities according to the circumstances. We will also examine the effect of helper agents' attitudes and experience toward teammate and task selection. In situations where initiator agents may adapt different strategies, the decision of helper agents on selecting a request can affect the performance of the

helper agents. For instance the helper agents may be interested to work with *nearest available* strategy in areas where task density is very high so that they can perform more tasks and improve their reward. In both of these cases the issue is agent learning and adaptation in dynamic environments.

References

1. FIPA (2003) FIPA specifications. FIPA contract net interaction protocol specification. Version H. Available from: http://www.fipa.org/specs/fipa00029/index.html
2. FIPA (2003) FIPA query interaction protocol specification. Available from: http://www.fipa.org/specs/fipa00027/index.html
3. Dang J, Huhns MN (2006) Concurrent multiple-issue negotiation for Internet-based services. IEEE Internet Comput 10(6):42–49. http://dx.doi.org/10.1109/MIC.2006.118
4. Damas B, Lima P (2004) Stochastic discrete event model of a multi-robot team playing an adversarial game. In: Fifth IFAC/EURON symposium on intelligent autonomous vehicles IAV2004, Lisboa, Portugal, 2004
5. Milutinovic D, Lima P (2002) Petri Net models of robotic tasks. In: IEEE international conference on robotics and automation, Washington, DC, USA, 2002, pp 4059–4064
6. Desel J, Oberweis A, Zimmer T, Zimmermann G (1997) Validation of information system models: Petri Nets and test casegeneration. In: Proceedings of the workshop on challenges in open agent systems, Orlando, Florida, 1997, pp 3401–3406
7. Poutakidis D, Padgham L, Winikoff M (2002) Debugging multi-agent systems using design artifacts: the case of interaction protocols. In: Proceedings of the first international joint conference on autonomous agents and multiagent systems: part 2, Bologna, Italy, 2002. ACM Press, New York
8. Ziparo VA, Locchi L, Nardi D, Palamara H, Costelha H (2008) Petri Net plans. In: Proceedings of seventh international conference on autonomous agents and multiagent systems, Estoril, Portugal, 2008, pp 79–86
9. Jensen K (1992) Coloured Petri Nets: basic concepts analysis methods and practical use, vol 1. Springer, Berlin
10. Nowostawski M, Purvis M, Cranefield S (2001) A layered approach for modelling agent conversations. In: Proceedings of the 2nd international workshop on infrastructure for agents, MAS, and scalable MAS, 5th international conference on autonomous agents, Montreal, 2001, pp 163–170
11. Cost RS, Chen Y, Finin T, Labrou YK, Peng Y (1999) Modeling agent conversations with colored Petri Nets. In: Proceeding of the third international conference on autonomous agents (Agents'99), workshop on agent conversation policies, Seattle, Washington, 1999
12. Cost RS, Chen Y, Finin T, Labrou Y, Peng Y (2000) Using colored Petri Nets for conversation modeling. In: Lecture notes in computer science. Springer, Berlin, pp 178–192
13. FIPA (2002) The foundation for intelligent physical agents. Available from: http://www.fipa.org/repository/index.html
14. Billington J, Gupta AK (2007) Effectiveness of coloured Petri Nets for modelling and analysing the contract net protocol. In: Proceeding eighth workshop and tutorial on practical use of coloured Petri Nets and the CPN tools, Aarhus, Denmark, 2007, pp 49–65
15. Gutnik G, Kaminka GA (2006) Representing conversations for scalable overhearing. J Artif Intell Res 25:349–387
16. Nowostawski M, Purvis M (2007) The concept of autonomy in distributed computation and multi-agent systems. In: International conference on intelligent agent technology. IEEE Computer Society, Los Alamitos, pp 420–423
17. Weyns D, Holvoet T (2004) A colored Petri Net for regional synchronization in situated multi-agent systems. In: Proceedings of first international workshop on Petri Nets and coordination, Bologna, Italy 2004, pp 65–86
18. Costelha H, Lima P (2008) Modelling, analysis and execution of multi-robot tasks using petri nets. In: Proceedings of the 7th international joint conference on autonomous agents and multiagent systems. International Foundation for Autonomous Agents and Multiagent Systems, Richland, pp 1187–1190.
19. Holvoet T (1995) Agents and Petri Nets. Petri Net Newsletter (49), 3–8

20. Chainbi W, Hanachi C, Sibertin-Blanc C (1996) The multi-agent prey/predator problem: a Petri Net solution. In: Symposium on discrete events and manufacturing systems, CESA'96 IMACS multiconference computational engineering in systems applications (CESA), Citeseer, Lille, France, 1996, pp 291–299
21. Sibertin-Blanc C (1994) Cooperative nets. In: Lecture notes in computer science, vol 815. Springer, Berlin, pp 471–490
22. Kohler M, Moldt D, Rolke H (2001) Modelling the structure and behaviour of Petri Net agents. In: Proceedings of the 22nd international conference on application and theory of Petri Nets. Springer, Berlin, pp 224–241
23. Duvigneau M, Moldt D, Rolke H (2003) Concurrent architecture for a multi-agent platform. In: Third international workshop on agent-oriented software engineering, pp 59–72
24. Kummer O, Wienberg F, Duvigneau M (2001) Renew-user guide. Department of Informatics, University of Hamburg, Hamburg
25. Purvis M, Nowostawski M, Cranefield S (2002) A multi-level approach and infrastructure for agent-oriented software development. In: First international conference on autonomous agents and multi agent systems, Bologna, Italy, 2002. ACM Press, New York, pp 88–89
26. DESIGN/CPN tutorial (2009) Version 5.0, Meta Software Corporation
27. Nowostawski M (2000) JFern-Java-based Petri Net framework
28. FIPA (2002) FIPA ACL message structure specification. Available from: http://www.fipa.org/specs/fipa00061/SC00061G.html